Your guide to an exciting new experience in personal computing! Learn how to use the ATARI ASSEMBLER CARTRIDGE and discover the rewards of programming in Assembly Language for the 400 or 800.
The Atari Assembler
The Atari Assembler

Don Inman    Kurt Inman

Reston Publishing Company, Inc.
A Prentice-Hall Company
Reston, Virginia
Contents

LIST OF ILLUSTRATIONS ix
PREFACE xi

Chapter 1 INTRODUCTION 1
Computer Architecture, 4
Review of BASIC, 7
Graphics Keywords, 8

Chapter 2 MACHINE LANGUAGE FROM BASIC 12
Binary Number Patterns, 12
Hexadecimal Notation, 14
Hexadecimal-to-Decimal Conversion, 16
How the Machine Language Program Works, 21
Summary, 24
Exercises, 25
Answers, 26

Chapter 3 MEMORY USE 28
Atari Memory Map, 28
How BASIC Finds the Machine Language Program, 30
Passing Variables to Machine Language Subroutine, 33
Using the One-Variable Program, 35
Passing More Than One Variable, 36
A Machine Language Loop, 40
New Instructions Used, 43
Tracing Through the Subroutine, 45
Summary, 48
Exercises, 49
Answers, 50
<table>
<thead>
<tr>
<th>Chapter</th>
<th>CONTENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>GETTING STARTED WITH THE ASSEMBLER 52</td>
</tr>
<tr>
<td></td>
<td>The Writer/Editor, 53</td>
</tr>
<tr>
<td></td>
<td>The Assembler Program, 57</td>
</tr>
<tr>
<td></td>
<td>Executing the Machine Language Program—The Debugger, 62</td>
</tr>
<tr>
<td></td>
<td>Summary, 70</td>
</tr>
<tr>
<td></td>
<td>Exercises, 71</td>
</tr>
<tr>
<td></td>
<td>Answers, 72</td>
</tr>
<tr>
<td>5</td>
<td>SPECIAL-PURPOSE REGISTERS AND ADDRESSING MODES 73</td>
</tr>
<tr>
<td></td>
<td>The Accumulator, 74</td>
</tr>
<tr>
<td></td>
<td>The X and Y Registers, 74</td>
</tr>
<tr>
<td></td>
<td>The Processor Status Register, 82</td>
</tr>
<tr>
<td></td>
<td>The Stack Pointer Register, 87</td>
</tr>
<tr>
<td></td>
<td>Addressing, Modes, 88</td>
</tr>
<tr>
<td></td>
<td>Summary, 93</td>
</tr>
<tr>
<td></td>
<td>Exercises, 93</td>
</tr>
<tr>
<td></td>
<td>Answers, 94</td>
</tr>
<tr>
<td>6</td>
<td>BRANCHING OUT 96</td>
</tr>
<tr>
<td></td>
<td>Examples Using Forward Branches, 98</td>
</tr>
<tr>
<td></td>
<td>Examples Using Backward Branches, 99</td>
</tr>
<tr>
<td></td>
<td>Using the Carry Flag, 102</td>
</tr>
<tr>
<td></td>
<td>Using the Zero Flag, 108</td>
</tr>
<tr>
<td></td>
<td>Using the Negative Flag, 111</td>
</tr>
<tr>
<td></td>
<td>The Overflow Flag, 116</td>
</tr>
<tr>
<td></td>
<td>Summary, 116</td>
</tr>
<tr>
<td></td>
<td>Exercises, 117</td>
</tr>
<tr>
<td></td>
<td>Answers, 118</td>
</tr>
<tr>
<td>7</td>
<td>ASSEMBLER REVIEW 120</td>
</tr>
<tr>
<td></td>
<td>Source Program Format, 120</td>
</tr>
<tr>
<td></td>
<td>Methods to Use Operands, 123</td>
</tr>
<tr>
<td></td>
<td>The Assembler Writer/Editor Mode, 124</td>
</tr>
<tr>
<td></td>
<td>The Debug Mode, 131</td>
</tr>
<tr>
<td></td>
<td>Exercises, 139</td>
</tr>
<tr>
<td></td>
<td>Answers, 142</td>
</tr>
<tr>
<td>Chapter 8</td>
<td>DESIGNING A PROGRAM</td>
</tr>
<tr>
<td>-----------</td>
<td>---------------------</td>
</tr>
<tr>
<td></td>
<td>Absolute Indexed Addressing, 146</td>
</tr>
<tr>
<td></td>
<td>Using the Add Five Pairs of Numbers Program, 148</td>
</tr>
<tr>
<td></td>
<td>Using the Add Ten Program, 153</td>
</tr>
<tr>
<td></td>
<td>A Variation of the Add Ten Program, 155</td>
</tr>
<tr>
<td></td>
<td>Yet Another Variation, 158</td>
</tr>
<tr>
<td></td>
<td>Summary, 159</td>
</tr>
<tr>
<td></td>
<td>Exercises, 160</td>
</tr>
<tr>
<td></td>
<td>Answers, 162</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chapter 9</th>
<th>ADDITION AND SUBTRACTION</th>
<th>164</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Two-Byte Addition, 165</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Two Programs in Memory, 170</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Two-Byte Subtraction, 172</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Negative Numbers, 174</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Multiple-Byte Addition and Subtraction, 178</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Decimal Arithmetic, 179</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Summary, 184</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Exercises, 185</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Answers, 186</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chapter 10</th>
<th>SHIFT AND ROTATE</th>
<th>188</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Arithmetic Shift Left, 191</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Logical Shift Right, 198</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Rotate Left, 202</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Rotate Right, 205</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Summary, 209</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Exercises, 209</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Answers, 211</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chapter 11</th>
<th>MULTIPLICATION, DIVISION, AND SUBROUTINES</th>
<th>213</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Eight-Bit Multiplication, 214</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Using the 8-Bit Multiplication Program, 218</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Eight-Bit Division, 221</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Subroutines, 225</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Using a Subroutine, 228</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Summary, 232</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Exercises, 232</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Answers, 234</td>
<td></td>
</tr>
<tr>
<td>Chapter 12</td>
<td>PROGRAMMING PRACTICE</td>
<td>236</td>
</tr>
<tr>
<td>-----------</td>
<td>----------------------</td>
<td>-----</td>
</tr>
<tr>
<td></td>
<td>Using a Logic Function, 237</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Entering the Subroutine, 241</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Program to Sound Off, 245</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Play Notes Program, 248</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Program to Shape Sound, 249</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Program to Print on the Screen, 252</td>
<td></td>
</tr>
<tr>
<td></td>
<td>You're On Your Own, 256</td>
<td></td>
</tr>
<tr>
<td>Appendix A</td>
<td>6502 INSTRUCTIONS—FLAGS AFFECTED</td>
<td>257</td>
</tr>
<tr>
<td>Appendix B</td>
<td>6502 INSTRUCTIONS—ADDRESSING MODES</td>
<td>259</td>
</tr>
<tr>
<td>Appendix C</td>
<td>FREQUENCY VALUES FOR THREE-OCTAVE SCALE</td>
<td>261</td>
</tr>
<tr>
<td>Appendix D</td>
<td>ATARI ASSEMBLER ERROR CODES</td>
<td>263</td>
</tr>
<tr>
<td>Appendix E</td>
<td>ATARI OPERATING SYSTEM ERRORS</td>
<td>264</td>
</tr>
<tr>
<td>Appendix F</td>
<td>ATASCII CHARACTER SET</td>
<td>265</td>
</tr>
<tr>
<td></td>
<td>INDEX</td>
<td>268</td>
</tr>
</tbody>
</table>
# Illustrations

<table>
<thead>
<tr>
<th>Figure No.</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-1</td>
<td>Language Cartridges</td>
</tr>
<tr>
<td>1-2</td>
<td>Steps in Assembly Language Programming</td>
</tr>
<tr>
<td>1-3</td>
<td>Machine Language Subroutine from BASIC</td>
</tr>
<tr>
<td>1-4</td>
<td>Atari Building Blocks</td>
</tr>
<tr>
<td>1-5</td>
<td>6502 Functional Elements</td>
</tr>
<tr>
<td>1-6</td>
<td>8 Line Data Bus</td>
</tr>
<tr>
<td>1-7</td>
<td>Program Counter</td>
</tr>
<tr>
<td>1-8</td>
<td>Last In, First Out Stack</td>
</tr>
<tr>
<td>1-9</td>
<td>Processor Status Register</td>
</tr>
<tr>
<td>2-1</td>
<td>Program Flow Including Subroutine</td>
</tr>
<tr>
<td>2-2</td>
<td>Decimal-Binary-Hexadecimal Equivalents</td>
</tr>
<tr>
<td>2-3</td>
<td>Eight-bit Conversion</td>
</tr>
<tr>
<td>2-4</td>
<td>Hex Digit-to-Decimal Conversion</td>
</tr>
<tr>
<td>2-5</td>
<td>Hex Code Conversion Exercise</td>
</tr>
<tr>
<td>2-6</td>
<td>Flowchart for Addition Subroutine</td>
</tr>
<tr>
<td>2-7</td>
<td>Decimal Equivalents of Hex Codes</td>
</tr>
<tr>
<td>2-8</td>
<td>Machine Language Subroutine</td>
</tr>
<tr>
<td>3-1</td>
<td>Memory Map</td>
</tr>
<tr>
<td>3-2</td>
<td>Memory Used for Machine Language Subroutine</td>
</tr>
<tr>
<td>3-3</td>
<td>Hex Code Storage</td>
</tr>
<tr>
<td>3-4</td>
<td>Machine Language Exercise</td>
</tr>
<tr>
<td>4-1</td>
<td>Our Atari 800 System</td>
</tr>
<tr>
<td>4-2</td>
<td>Use Flow of Assembler Cartridge</td>
</tr>
<tr>
<td>4-3</td>
<td>Buffer Memory</td>
</tr>
<tr>
<td>4-4</td>
<td>Machine Language Program Storage</td>
</tr>
<tr>
<td>4-5</td>
<td>Program Trace</td>
</tr>
<tr>
<td>5-1</td>
<td>Processor Status Register</td>
</tr>
<tr>
<td>5-2</td>
<td>Effect of Instructions on Flags</td>
</tr>
<tr>
<td>6-1</td>
<td>Forward Branches</td>
</tr>
<tr>
<td>6-2</td>
<td>Backward Branches</td>
</tr>
<tr>
<td>6-3</td>
<td>Status Flags for Branches</td>
</tr>
<tr>
<td>6-4</td>
<td>Signed Number Wheel</td>
</tr>
<tr>
<td>7-1</td>
<td>Statement Fields</td>
</tr>
<tr>
<td>7-2</td>
<td>Buffer Memory</td>
</tr>
<tr>
<td>Figure No.</td>
<td>Page</td>
</tr>
<tr>
<td>-----------</td>
<td>------</td>
</tr>
<tr>
<td>7-3</td>
<td>126</td>
</tr>
<tr>
<td>8-1</td>
<td>145</td>
</tr>
<tr>
<td>8-2</td>
<td>145</td>
</tr>
<tr>
<td>8-3</td>
<td>145</td>
</tr>
<tr>
<td>8-4</td>
<td>147</td>
</tr>
<tr>
<td>8-5</td>
<td>152</td>
</tr>
<tr>
<td>8-6</td>
<td>153</td>
</tr>
<tr>
<td>8-7</td>
<td>157</td>
</tr>
<tr>
<td>9-1</td>
<td>166</td>
</tr>
<tr>
<td>9-2</td>
<td>166</td>
</tr>
<tr>
<td>9-3</td>
<td>167</td>
</tr>
<tr>
<td>9-4</td>
<td>167</td>
</tr>
<tr>
<td>9-5</td>
<td>169</td>
</tr>
<tr>
<td>9-6</td>
<td>170</td>
</tr>
<tr>
<td>9-7</td>
<td>174</td>
</tr>
<tr>
<td>9-8</td>
<td>177</td>
</tr>
<tr>
<td>9-9</td>
<td>179</td>
</tr>
<tr>
<td>9-10</td>
<td>183</td>
</tr>
<tr>
<td>10-1</td>
<td>189</td>
</tr>
<tr>
<td>10-2</td>
<td>192</td>
</tr>
<tr>
<td>10-3</td>
<td>195</td>
</tr>
<tr>
<td>10-4</td>
<td>197</td>
</tr>
<tr>
<td>10-5</td>
<td>198</td>
</tr>
<tr>
<td>10-6</td>
<td>198</td>
</tr>
<tr>
<td>10-7</td>
<td>202</td>
</tr>
<tr>
<td>10-8</td>
<td>202</td>
</tr>
<tr>
<td>10-9</td>
<td>205</td>
</tr>
<tr>
<td>10-10</td>
<td>205</td>
</tr>
<tr>
<td>11-1</td>
<td>214</td>
</tr>
<tr>
<td>11-2</td>
<td>215</td>
</tr>
<tr>
<td>11-3</td>
<td>218</td>
</tr>
<tr>
<td>11-4</td>
<td>219</td>
</tr>
<tr>
<td>11-5</td>
<td>221</td>
</tr>
<tr>
<td>11-6</td>
<td>222</td>
</tr>
<tr>
<td>11-7</td>
<td>224</td>
</tr>
<tr>
<td>11-8</td>
<td>225</td>
</tr>
<tr>
<td>11-9</td>
<td>226</td>
</tr>
<tr>
<td>11-10</td>
<td>230</td>
</tr>
<tr>
<td>11-11</td>
<td>231</td>
</tr>
<tr>
<td>12-1</td>
<td>239</td>
</tr>
<tr>
<td>12-2</td>
<td>245</td>
</tr>
<tr>
<td>12-3</td>
<td>247</td>
</tr>
<tr>
<td>12-4</td>
<td>252</td>
</tr>
<tr>
<td>12-5</td>
<td>254</td>
</tr>
</tbody>
</table>
Due to the spectacular growth of the use of personal microcomputers such as the Atari 400 and 800 models, more and more people are acquiring their own computers. These new users naturally want to make the most of the capabilities of this versatile tool. After conquering BASIC language, an assembler is the next logical step.

The Atari Assembler Cartridge provides the ideal tool. It is powerful, and yet simple to understand and use. It provides all the features necessary to take the drudgery out of hand assembling machine language programs.

We have written this book with two main objectives in mind:

1. to provide simple, detailed directions for using the Atari Assembler Cartridge;
2. to provide fundamental information on programming in assembly language.

The book is written for the beginning assembly language programmer who has some knowledge of BASIC language programming. Steps for using the assembler are given in detail. Sketches of the video screen are shown at intermediate and final stages for entry and execution of programs. Machine language instructions are explained both verbally and pictorially. Every effort has been made to make the book easy to read and understand.

We believe the approach to assembly language programming through BASIC is sound. We assume you have a knowledge of BASIC, and we try to ease you gently into assembly language using your existing knowledge of BASIC.

Chapter exercises and answers are given to reinforce what you have learned. Although the book is not meant to cover the Atari Assembler Cartridge or the 6502 instruction set completely, it provides enough detail to allow you to use both at an intelligent level. From that point, you are encouraged to use what you have learned to explore assembly language to the depth that you desire.
Chapter 1

Introduction

This book is primarily concerned with the operation and use of the Atari Assembler Cartridge. The Assembler Cartridge plugs into the left slot of the Atari computer replacing the BASIC language cartridge.

Through the Assembler Cartridge programs, an assembly language program is written and edited. It is assembled into machine language codes, and the machine language program is then executed to produce the desired results. The steps in this procedure are outlined in Figure 1-2.

1. Enter assembly language program
2. Assemble into a machine language program
3. Execute the machine language program

Figure 1-2. Steps in Assembly Language Programming
INTRODUCTION

Our discussion of the Assembler Cartridge and its use will begin in Chapter 4. Instead of jumping right into the cold waters of assembly language programming, you should try a more gentle approach. In this chapter and the next, you will learn about the Atari's organization (or architecture) and go through a quick review of its BASIC language, with which we assume you are familiar. You will learn to create and execute machine language programs from within a BASIC program of your own in Chapters 2 and 3 (see Figure 1-3). We are using this approach because most users of the Model 400 and 800 Atari computers will probably do most of their programming in the Atari BASIC language supplied in cartridge form with their computers. This will allow you to learn the fundamentals of machine language programming through a language with which you are already familiar.

Figure 1-3. Machine Language Subroutine from BASIC

BASIC language instructions must be translated or interpreted (by the BASIC language cartridge) before the computer can understand what is to be done. Because of this extra step for translation, instructions in BASIC cannot be executed as quickly as instructions written in machine language. A program written in BASIC also uses more memory space than the equivalent program written in machine language.

It is possible to write most of a program in BASIC and write parts that must be done quickly and efficiently in machine language. The machine language parts of the program are accessed as subroutines from the BASIC program. These subroutines can be "understood" directly by the computer, and no time is wasted for interpretation. Chapters 2 and 3 will thoroughly explain this method and provide sample demonstrations and step-by-step instructions.

Although the computer can execute machine language instructions faster than it can execute BASIC instructions, it takes the programmer longer to write a machine language program.

There are several disadvantages to machine language:

1. Each instruction has its own numeric code which the computer understands. The programmer must either memorize these codes (Heaven forbid!) or look them up in a table each time they are used:
Examples:

<table>
<thead>
<tr>
<th>Hex Code</th>
<th>Operation Performed</th>
</tr>
</thead>
<tbody>
<tr>
<td>A9</td>
<td>Load the accumulator with the number following the hex code.</td>
</tr>
<tr>
<td>AD</td>
<td>Load the accumulator with the number contained in the memory location that follows the hex code.</td>
</tr>
<tr>
<td>8D</td>
<td>Store the value in the accumulator into the memory location that follows the hex code.</td>
</tr>
</tbody>
</table>

2. Each machine language Operation Code (Op Code) must be placed in memory in the correct sequence for the successful operation of the program.

3. When branches are made to change the sequential operation of a program, the programmer must calculate and include in the instruction exactly how many memory locations must be skipped to arrive at the correct instruction.

4. The detail work involved in machine language programming is tremendous, and the chance for programming errors is very high.

Assembly language eliminates many of the drawbacks of machine language programming. Compare the following list to the disadvantages of machine language.

1. Each assembly language instruction has its own mnemonic code which is an abbreviation of the operation to be performed.

Examples:

<table>
<thead>
<tr>
<th>Mnemonic Code</th>
<th>Operand</th>
<th>Operation Performed</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDA</td>
<td>#14</td>
<td>Load the accumulator with the number 14.</td>
</tr>
<tr>
<td>LDA</td>
<td>$1100</td>
<td>Load the accumulator with the value contained in memory location 1100.</td>
</tr>
<tr>
<td>STA</td>
<td>$1105</td>
<td>Store the value contained in the accumulator in memory location 1105.</td>
</tr>
</tbody>
</table>

These mnemonic codes are much easier for a programmer to work with than the numeric codes of machine language.

2. The assembly language program has line numbers similar to BASIC, and the program is automatically placed into the correct sequence of memory locations by the assembler.

3. Through the use of labels (combinations of words, letters and/or numbers) branches are made to labeled instructions, and no calculations are required (the assembler does it for you).
4. The assembler creates the machine language program for you from the mnemonic codes, eliminating all the time-consuming detail work. The chance for programming errors is much less than for programs constructed by hand in machine language.

The Atari Assembler Cartridge will be introduced in Chapter 4 after you learn a little more about how the computer works in its own language.

**COMPUTER ARCHITECTURE**

We assume that you are familiar with your Atari BASIC programming language and now wish to investigate the computer’s capabilities through its own language.

Programming in the machine’s language requires that you become familiar with the fundamental building blocks (or architecture) of the Atari computer. The two building blocks that you will be most concerned with are the Central Processing Unit and the computer’s memory.

You will need an understanding of how the computer’s memory is used for storage and retrieval of information. You will get to know the binary codes used to “instruct” the computer. These codes are referred to as the *instruction set* of the Central Processing Unit.

It sounds very complicated, but we will move slowly. One thing will be introduced at a time, and we will repeat previously discussed information at frequent intervals.

The Atari 400 and 800 model computers use a Central Processing Unit (CPU) called the 6502. The 6502 is a member of a family of microprocessors developed by MOS Technology, Inc., the MCS650X product family. This CPU can process, or execute, a set of instructions (listed in Appendix A) that are identified by their machine language codes. The codes are composed of binary numbers that are 8 binary digits (bits) long.

Binary digits come in two denominations. A binary digit is either a 1 (one) or a 0 (zero). A machine language code which can be recognized by the 6502 CPU is made up of 8 of these binary digits.
Examples:

<table>
<thead>
<tr>
<th>Binary code</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>10101001</td>
<td>Load the accumulator with the number following this instruction.</td>
</tr>
<tr>
<td>10101101</td>
<td>Load the accumulator from the memory location that follows this instruction.</td>
</tr>
<tr>
<td>10001101</td>
<td>Store the value that is in the accumulator in the memory location that follows this instruction.</td>
</tr>
</tbody>
</table>

Note that each code is exactly 8 binary digits (8 bits) long. The block of 8 bits is called a *byte*. The accumulator, mentioned in the instruction, is a special storage location called a *register*. It can hold one byte of data at a time.

The accumulator (often referred to as register A) is probably the busiest register (a special temporary storage location) in the system. All operations between memory locations must be communicated through the accumulator or one of the other registers. The accumulator is used as a temporary storage when moving data from one memory location to another. Operations on data are performed in the accumulator. Therefore, it is used in many of the machine language and assembly language instructions. It "accumulates" the results of successive operations on data as the instructions request.

It is convenient for a programmer to think of the 6502 microprocessor as consisting of several functional elements. Figure 1-5 shows the registers (special storage locations within the CPU) that are used repeatedly as the computer performs its many and varied chores. Some of these elements are used for specific purposes, and others are used for general purposes as needed by the programmer.

![Functional Elements](image)

*Figure 1-5. 6502 Functional Elements*

Data is transferred between memory and the processor's internal registers over 8 bidirectional data lines called the *data bus*. Each bit of a data byte passes along its own line. However, the transfer of the whole byte (all 8 bits) of data takes place at the same time. Even though each bit travels along a separate path (line) of the data bus, the byte is transferred as one complete unit. Thus the structure of the 6502 microprocessor is said to be a byte-oriented structure. The 8 lines together are referred to as a bus.
The X and Y registers are also used for temporary data storage. In addition, they have the ability to be incremented and decremented by a programmed instruction. When the X or Y register is incremented, the value in the register is increased by one. When X or Y is decremented, the value in the register is decreased by one. Therefore, the X and Y registers can be used as counters (or pointers) to store data into successive memory locations or to load data from successive memory locations. This ability to point to successive memory locations is called indexing (hence their name: index registers). They can also be used as counters to determine conditions for ending a series of repeated operations (a loop). The 6502 instruction set includes several special instructions to load the index registers with predetermined values to facilitate the execution of loops in a manner similar to BASIC language FOR-NEXT loops.

The 16-bit program counter acts as a program address pointer to assure that instructions are executed in the desired order. Instructions of a program are stored in consecutive locations in memory. They are made up of machine language operation codes and/or address bytes and numbers to be operated on. To control the desired sequence of operations of a program, the program counter is used as a pointer to designate the position in memory where the microprocessor will obtain each successive instruction. The program counter is incremented, after each instruction is "fetched," to point at the next instruction to be performed.

The stack is a special area of memory that is used to save data according to the sequence in which it is stored. It acts like a Last-In, First-Out file system (Figure 1-8). The 16-bit stack pointer keeps track of where data has been stored
within the stack. It contains an address designating the current location of the top of the stack. The stack pointer and the program counter are large enough (16 bits) to hold a full-length address (0 through 65535).

Individual bits of the 8-bit Processor Status Register are used to keep track of specific effects that instructions have on the "status" of the computer. The presence or absence of an effect is shown by whether a particular bit has been set to one or reset to zero. These individual bits are also called flags. Flags used by the 6502 microprocessor are Carry, Zero, Interrupt, Decimal, Break, Overflow, and Negative. These conditions (or effects) are discussed in more detail as they are needed in understanding the instructions. Your first encounter will be with the Carry flag in Chapter 2.

![Figure 1-8. Last-In, First-Out Stack](image)

**REVIEW OF BASIC**

In Chapter 2, we will introduce the use of machine language programs that are entered through the use of BASIC language programs. The machine language program will be a subroutine of the BASIC program. Atari's BASIC includes a function that is used to transfer from the BASIC program to such a machine language subroutine. It is known as the **USR function**.

**Example:**

120 X = USR(1000)

Go to the machine language subroutine at memory location 1000
The balance of this chapter is devoted to the review of some of the BASIC statements, commands, functions, and keywords used by the Atari 400/800 computers. Special attention should be given to the USR function as well as the POKE, PEEK, and ADR keywords. If you feel the review unnecessary, proceed to Chapter 2.

Atari BASIC uses statements and keywords that are different from other BASIC language versions due to the computer's special graphics and sound capabilities. These are briefly described here. For more complete descriptions refer to the Atari BASIC Reference Manual.

Graphics Keywords

COLOR Primarily selects the appropriate color register, but specific uses depend on the graphics mode in use.

DRAWTO Draws a line from a PLOTTed point to the point specified.

FILL Not really a keyword but an I/O operation that fills an area on the screen between plotted points and lines with a specified color.

GET Used to input the code byte for the character displayed at the cursor location.

GRAPHICS Controls which graphics mode is to be used.

LOCATE Stores the color number that controls a particular point on the screen in the specified variable.

PLOT Plots a point in the graphics window based on the X,Y coordinates specified.

POSITION Places the cursor at a particular location on the screen.

PUT Used to output data to the screen for display.

SETCOLOR Sets the color register specified by loading it with the hue and luminance data specified.

Sound and Game Keywords

PADDLE Returns the status of the specified controller (a number between 1 and 228 depending on paddle position).

PTRIG Returns a number representing the status of the trigger button of a controller (0 if button pressed, otherwise 1).

SOUND Plays a note through one of four specified voices, with the pitch, distortion, and volume specified.

STICK Same function as PADDLE, but for joystick controllers.

STRIG Same function as PTRIG, but for joystick controllers.
Special Keywords

Four keywords that are of special importance to us are: ADR, PEEK, POKE, and USR. These words provide a link between BASIC and machine language programs.

ADR  This function returns the memory address of a specified string. Knowing the address enables the programmer to pass data to USR subroutines. A two-step process is used to obtain the address of a matrix of numeric values. First use a DIM statement for a string of length 1. On a separate BASIC program line, ask for the address of the string variable.

Example:

200 DIM A$(1)  

210 PRINT ADR(A$)

The address of the matrix will be one more than the address of the string A$.

The ADR function must be used with care. If your program (made up of the data matrix) is not addressed correctly, the computer may go off to Never-Never Land (the wrong place) and not return. If this happens, you can turn the computer off, wait about 5 seconds, and power up again. Your program will have gone bye-bye in the meantime. Start all over again.

PEEK  This function allows the user to PEEK into a specified memory location. The user can then make use of the information that is “seen” there in his BASIC program.

Examples:

200 PRINT "THE LEFT MARGIN IS SET AT"; PEEK(82)

   This memory location always contains the position at which printing on the screen begins for each line (ordinarily 2).

210 PRINT " THE RIGHT MARGIN IS SET AT"; PEEK(83)

   This memory location contains the rightmost position at which a character is printed on the screen (ordinarily 39).

Any memory address (given in decimal form) in your Atari may be PEEKed at without disturbing its contents. This applies to either ROM (Read Only Memory) or RAM (Random Access Memory). RAM is the kind of memory
that may be either written into or read from. ROM can only be read from. We will often use this instruction in Chapter 2 to PEEK at machine language programs that we have entered in memory.

POKE This function is the opposite of PEEK. You can use it to insert or modify the contents of RAM memory. We will use it in Chapter 2 to POKE in machine language programs.

Examples:
150 POKE 82,8  
160 POKE 83,30

This would perform the following function.

```
0123456789012345678901234567890123456789
PRINTED MATTER ACCEPTED
ONLY FROM POSITION 8-30

Screen positions
(0-39)
```

POKE, like PEEK, can be used in either the direct or deferred program mode. You cannot POKE data into ROM memory. Data can only be read from ROM, not written. Since a POKE statement actually changes data stored in memory, great care must be taken when using it. If incorrect data is POKEd into memory or if data is POKEd into the wrong memory location, disastrous results may occur. When you want to use POKE, refer to the Atari memory map in Appendix G of the Atari BASIC Reference Manual.

One precaution before indiscriminately using POKE. PEEK first and write down the value in that memory location into which you wish to POKE. Then, if the POKE doesn’t produce the desired result, you can POKE the original value back where it was.

USR This function “calls” a 6502 machine language subroutine and returns with the result of the subroutine’s execution. It will be used in Chapter 2 to execute the machine language programs that are POKEd into memory. The format used for this function is:

```
USR(X,Y,Z,...)
```

An integer or arithmetic expression that evaluates to an integer. It is the decimal address of the beginning of the machine language program.
Definitions

The following definitions were condensed from the Atari BASIC Reference Manual for a quick review.

ARRAY  A list of places where items can be filed away for future use.
ARRAY VARIABLE  A name for an array consisting of one or more elements.
BASIC statement  Similar to a sentence in English; requests the computer to do something.
BREAK KEY  Pressed to stop a program's execution.
CONCATENATION  The joining of two or more strings.
CONSTANT  A number or string without a variable name.
DEFERRED PROGRAMS  Those stored in memory (with line numbers) for future execution.
DIRECT PROGRAMS  Those executed immediately as each line is entered from the keyboard.
EXPRESSION  May consist of any combination of legal variables, constants, operators, and functions used together.
FUNCTION  A computation built into the computer's operating system so that it can be used by the user in his programs.
KEYWORD  Any reserved word used by BASIC that cannot be used for other purposes.
LOGICAL LINE  Each numbered line in a BASIC program; consists of one to three physical video screen lines terminated by a RETURN.
NESTED LOOP  One loop (inner loop) executed within another (outer loop).
OPERATOR  Symbols used to perform operations such as arithmetic, comparison or logic.
RETURN KEY  Pressed to enter each program line.
STRINGS  A group of characters strung together with quotes.
VARIABLES  The name for a number or string.

For a complete list and discussion of Atari BASIC, refer to the Atari BASIC Reference Manual. With this brief review let's move right on to Chapter 2.
Chapter 2

Machine Language from BASIC

Short machine language programs can be written and executed from BASIC through the use of the USR function. In this way, it is possible to execute part of a program in BASIC, "jump" to a machine language subprogram and then return to BASIC again. Figure 2-1 shows the flow of such a program.

![Program Flow Including Subroutine](image)

The computer can only understand instruction codes written in fixed-length binary numbers. These binary instruction codes are called *machine language*.

BINARY NUMBER PATTERNS

The computer interprets these bits as being one of two numeric symbols, 0 or 1. The pattern of 1's and 0's makes a meaningful word, or a complete idea, to the computer. Therefore, we need to learn these words if we are to communicate directly with it.
An example of a pattern of 8 computer bits (a pattern with a size and shape that the computer can understand) is shown.

```
OFF  ON  OFF  ON  ON  OFF  ON  ON
```

The computer would recognize this pattern as a unique number code and would respond by taking a specific action or using the number as a specific piece of data.

The Central Processing Unit (CPU) of the Atari computer was originally manufactured by MOS Technology, Inc. At the present time, two other companies (Synertek and Rockwell) also manufacture the CPU. This unit is named the 6502 microprocessor. It is called a Central Processing Unit because all instructions and numerical values are routed there for processing.

The 6502 microprocessor (and hence, the Atari computer), like many other microprocessors, only understands instructions which are coded in blocks of eight binary digits, called bytes. Therefore, the biggest hurdle to machine language programming is to learn to work with information in binary form.

```
0 1 1 1 1 1 0 1
```

← block of 8 bits or 1 byte

```
1
```

← one bit

```
1 0
```

← two bits (not worth much these days)

```
1 1 0 1
```

← four bits (sometimes referred to as a nybble)

```
0 1 1 1 1 1 1 0 1
```

← eight bits (commonly called a byte)

The Atari uses words that are eight bits in length, that is, it can digest words whose size is one byte. All instructions and numerical values must be sent to its Central Processing Unit in this byte size. A typical instruction, shown below, loads the computer's accumulator with the one byte of data following the instruction in a machine language program.
Load Accumulator from the L Register

<table>
<thead>
<tr>
<th>MNEMONIC CODE (Abbreviation)</th>
<th>BINARY CODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDA</td>
<td>1 0 1 0 1 0 0 1</td>
</tr>
</tbody>
</table>

Don’t let the computer terminology throw you. The accumulator is similar to a memory location that is used in special ways which we will discuss later on. We are just introducing it here to show the format of an instruction.

The BASIC Language Cartridge, provided with your Atari, automatically interprets BASIC language statements and decimal numbers so that the computer can understand them. Therefore, when you are programming in BASIC, you enter decimal numbers and the results are displayed as decimal numbers even though the computer is working with binary numbers.

**HEXADECIMAL NOTATION**

Machine language codes are usually given in a third number system called *hexadecimal*. The hexadecimal system has a base of 16 and is used as a shorthand to represent binary numbers. Figure 2-2 shows a chart that demonstrates equivalent decimal, binary, and hexadecimal numbers.

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Binary</th>
<th>Hexadecimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>0100</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>0101</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>0110</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>0111</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>1000</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>1001</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>1010</td>
<td>A</td>
</tr>
<tr>
<td>11</td>
<td>1011</td>
<td>B</td>
</tr>
<tr>
<td>12</td>
<td>1100</td>
<td>C</td>
</tr>
<tr>
<td>13</td>
<td>1101</td>
<td>D</td>
</tr>
<tr>
<td>14</td>
<td>1110</td>
<td>E</td>
</tr>
<tr>
<td>15</td>
<td>1111</td>
<td>F</td>
</tr>
</tbody>
</table>

*Figure 2-2. Decimal-Binary Hexadecimal Equivalents*
We will often refer to this shorthand as hex. Four binary digits may be represented by one hex digit. Thus, our 8-bit instruction may be represented by a 2-digit hex number by breaking the byte (8 bits) into two parts.

Binary Number

```
0 1 1 1 1 1 0 1
```

0111 = 7 hex

Therefore:

```
0 1 1 1 1 1 0 1
```

1101 = D hex

Each place value in the binary system is a power of two, just as each place value in the decimal system is a power of ten. Two is called the base of the binary system, and ten is called the base of the decimal system. If we look at the place values of the binary numbers 0000 through 1111, we can attach more meaning to them.

<table>
<thead>
<tr>
<th>Binary Places</th>
<th>Decimal Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^3$</td>
<td>$2^2$</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>$2^3$</td>
<td>$2^2$</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>$2^3$</td>
<td>$2^2$</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Using combinations of these place values, we may obtain any decimal value from 0 through 16 or any hex value from 0 through F.

Examples:

- 0101 = $2^2 + 2^0 = 4+1 = 5$ decimal and also 5 hex
- 1001 = $2^3 + 2^0 = 8+1 = 9$ decimal and also 9 hex
- 1100 = $2^3 + 2^2 = 8+4 = 12$ decimal which is C hex
- 1011 = $2^3 + 2^1 + 2^0 = 8+2+1 = 11$ decimal which is B hex

Let's now take a closer look at how we may express any 8-bit binary number by two hex digits. We saw earlier that the highest hex digit (F) corresponds to the four-bit binary value 1111. The next higher binary value is 10000. The one is in the $2^4$ place which equals 16. Therefore, we have one 16 and nothing else. This can be expressed by the hex value 10, which means one 16 and no 1's. There is a direct relationship between upper 4 bits of an 8-bit binary number and the sixteen's place digit of a hex number.
76

MACHINE
LANGUAGE
FROM BASIC

Next look at the binary place values of the complete 8-bit number.

<table>
<thead>
<tr>
<th>Binary Places</th>
<th>Hex value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^7$</td>
<td>$2^6$</td>
</tr>
<tr>
<td>0 0 0 0 0 0 0 0</td>
<td>1</td>
</tr>
<tr>
<td>0 0 0 0 0 0 1 0</td>
<td>2</td>
</tr>
<tr>
<td>0 0 0 0 0 1 0 0</td>
<td>4</td>
</tr>
<tr>
<td>0 0 0 0 1 0 0 0</td>
<td>8</td>
</tr>
</tbody>
</table>

$2^4 = 16$

$2^5 = 2*16 = 32$

$2^6 = 4*16 = 64$

$2^7 = 8*16 = 128$

<table>
<thead>
<tr>
<th>Binary Places</th>
<th>Decimal Equivalent</th>
<th>Hex Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^7$</td>
<td>$2^6$</td>
<td>$2^5$</td>
</tr>
<tr>
<td>0 0 0 0 0 0 0 0</td>
<td>0+0+0+0+0+0+0+1 = 1</td>
<td>1</td>
</tr>
<tr>
<td>0 0 0 0 0 0 1 0</td>
<td>0+0+0+0+0+0+2+0 = 2</td>
<td>2</td>
</tr>
<tr>
<td>0 0 0 0 0 1 0 0</td>
<td>0+0+0+0+4+0+0+0 = 4</td>
<td>4</td>
</tr>
<tr>
<td>0 0 0 0 1 0 0 0</td>
<td>0+0+0+8+0+0+0+0 = 8</td>
<td>8</td>
</tr>
<tr>
<td>0 0 0 1 0 0 0 0</td>
<td>0+0+16+0+0+0+0+0 = 16</td>
<td>10</td>
</tr>
<tr>
<td>0 0 1 0 0 0 0 0</td>
<td>0+32+0+0+0+0+0+0 = 32</td>
<td>20</td>
</tr>
<tr>
<td>0 1 0 0 0 0 0 0</td>
<td>64+0+0+0+0+0+0+0 = 64</td>
<td>40</td>
</tr>
<tr>
<td>1 0 0 0 0 0 0 0</td>
<td>128+0+0+0+0+0+0+0 = 128</td>
<td>80</td>
</tr>
</tbody>
</table>

Figure 2-3. Eight-bit Conversion

Using combinations of all eight bits, you may obtain any decimal value from 0 through 255, or any hex value from 0 through FF. If we break an 8-bit binary number into two four-bit parts, each part may be represented by one hex digit.

Examples:

<table>
<thead>
<tr>
<th>Binary</th>
<th>Split binary</th>
<th>Hex</th>
<th>Decimal Equivalent</th>
<th>Hex Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>01101101</td>
<td>0110 1101</td>
<td>6D</td>
<td>64+32+8+4+1 = 109 in decimal</td>
<td>109 decimal</td>
</tr>
<tr>
<td>11000101</td>
<td>1100 0101</td>
<td>C5</td>
<td>128+64+4+1 = 197 in decimal</td>
<td>197 decimal</td>
</tr>
<tr>
<td>10101100</td>
<td>1010 1100</td>
<td>AC</td>
<td>128+32+8+4 = 172 in decimal</td>
<td>172 decimal</td>
</tr>
</tbody>
</table>

HEXADECIMAL-TO-DECIMAL CONVERSION

Hex codes made up of one digit are quite easily converted to decimal values. If you haven’t memorized the equivalents by now, here is a table of the values again.
A two-digit hex code is also easily converted to its decimal equivalent if we keep in mind two things:

1. The decimal equivalent of each hex digit (as in the previous table).
2. The place value assigned to each hex digit.

Place values in the decimal number system are powers of ten. The least significant place in a decimal integer is valued at $10^0$. This is called the one's place ($10^0=1$). The next place to the left is valued at $10^1$ and is called the ten's place ($10^1=10$).

Examples:

\[
89 = (8 \times 10^1) + (9 \times 10^0)
\]

or

\[
(8 \times 10) + (9 \times 1) = 80 + 9
\]

digit place digit place
value value

\[
23 = (2 \times 10^1) + (3 \times 10^0)
\]

or

\[
(2 \times 10) + (3 \times 1) = 20 + 3
\]

The hexadecimal place values are powers of sixteen (hexadecimal = 6 and 10). The least significant place in a hex integer is valued at $16^0$. This is the one's place ($16^0=1$). The next place to the left is valued at $16^1$ and is called the sixteen's place ($16^1=16$).
Examples:
23 hex = \((2 \times 16^1) + (3 \times 16^0)\)
\[= 2 \times 16 + 3 \times 1 = 32 + 3 = 35\] decimal

or
\[= 2 \times 16 + 3 \times 1 = 35\] decimal

89 hex = \((8 \times 16^1) + (9 \times 16^0)\)
\[= 8 \times 16 + 9 \times 1 = 128 + 9 = 137\] decimal

or
\[= 8 \times 16 + 9 \times 1 = 137\] decimal

A9 hex = \((10 \times 16^1) + (9 \times 16^0)\)
\[= 10 \times 16 + 9 \times 1 = 160 + 9 = 169\] decimal

or
\[= 10 \times 16 + 9 \times 1 = 169\] decimal

See if you can fill in the decimal equivalents of the following hex codes.

<table>
<thead>
<tr>
<th>Hex Code</th>
<th>Decimal Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>68</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td></td>
</tr>
<tr>
<td>A9</td>
<td></td>
</tr>
<tr>
<td>3F</td>
<td></td>
</tr>
<tr>
<td>69</td>
<td></td>
</tr>
<tr>
<td>41</td>
<td></td>
</tr>
<tr>
<td>8D</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td></td>
</tr>
<tr>
<td>60</td>
<td></td>
</tr>
</tbody>
</table>

*Figure 2-5. Hex Code Conversion Exercise*

If you have converted the hex codes of Figure 2-5 correctly, you are ready to enter your first machine language program. The codes are machine language instructions and data. Remember though, you are entering them from a BASIC program. Therefore, you must use the decimal form for entry. Check your answers now by looking at Figure 2-7.

Before the discussion of the machine language subroutine, let's take a look at the BASIC program used to enter and execute the subroutine.
BASIC PROGRAM—ADD TWO NUMBERS

100 REM INITIALIZE STORAGE ADDRESS
110 CLR: DIM E$(1), E(10)

120 REM POKE IN SUBROUTINE
130 FOR Y = 1 TO 10
140 INPUT N
150 POKE ADR(E$)+Y,N
160 NEXT Y

170 REM CALL SUBROUTINE
180 X=USR(ADR(E$)+1)

190 REM PRINT RESULTS
200 GR.0: PRINT "THE SUM IS ";
210 PRINT PEEK(6144)
220 END

Notice line 110. After the DIM statement for E$, it is necessary to dimension an array (E in this program) to the number of codes that are being used in the machine language program. This saves the necessary memory locations to receive the codes POKEEd in by the FOR-NEXT loop in lines 130 through 160.

A flow diagram for the BASIC program and its machine language subroutine is shown in Figure 2-6.

Figure 2-6. Flowchart for Addition Subroutine
When you type RUN, the machine language subroutine is POKEd into memory. Each time through the input loop, enter one decimal equivalent of one hex code following the question mark (input prompt). The inputs are the decimal values that you put in Figure 2-5.

<table>
<thead>
<tr>
<th>Hex Code</th>
<th>Decimal Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>68</td>
<td>104</td>
</tr>
<tr>
<td>18</td>
<td>24</td>
</tr>
<tr>
<td>A9</td>
<td>169</td>
</tr>
<tr>
<td>3F</td>
<td>63</td>
</tr>
<tr>
<td>69</td>
<td>105</td>
</tr>
<tr>
<td>41</td>
<td>65</td>
</tr>
<tr>
<td>8D</td>
<td>141</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>18</td>
<td>24</td>
</tr>
<tr>
<td>60</td>
<td>96</td>
</tr>
</tbody>
</table>

*Figure 2-7. Decimal Equivalents of Hex Codes*

After all 10 codes have been entered, the machine language subroutine is executed. On return from the subroutine the result is printed.

Enter the program and RUN it. This is the way it should look on the screen just before you press RETURN after the ninth input.

```
?104
?24
?169
?63
?105
?65
?141
?0
?24
?96
```

The 10th input

After you press RETURN the tenth time, the screen is cleared and your result appears.

```
THE SUM IS 128
READY
```
Congratulations! You have just completed your first machine language program. When your BASIC program executed line 180, the USR function caused the execution of the machine language subroutine.

Figure 2-8 shows each hex code, the mnemonic code that will be used when you use Assembler language, and an explanation of the purpose of each code.

<table>
<thead>
<tr>
<th>Hex Code</th>
<th>Mnemonic Code</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>68</td>
<td>PLA</td>
<td>Pull one byte off the stack and place it in the accumulator</td>
</tr>
<tr>
<td>18</td>
<td>CLC</td>
<td>Clear the Carry flag</td>
</tr>
<tr>
<td>A9</td>
<td>LDA</td>
<td>Load the accumulator with the value immediately following</td>
</tr>
<tr>
<td>3F</td>
<td></td>
<td>Value loaded (3F hex = 63 decimal)</td>
</tr>
<tr>
<td>69</td>
<td>ADC</td>
<td>Add the hex value that follows to the value in the accumulator</td>
</tr>
<tr>
<td>41</td>
<td></td>
<td>Value added (41 hex = 65 decimal)</td>
</tr>
<tr>
<td>8D</td>
<td>STA</td>
<td>Store the value in the accumulator into the memory location that follows</td>
</tr>
<tr>
<td>00</td>
<td></td>
<td>Least Significant Byte of memory</td>
</tr>
<tr>
<td>18</td>
<td></td>
<td>Most Significant Byte of memory</td>
</tr>
<tr>
<td>60</td>
<td>RTS</td>
<td>Return from subroutine</td>
</tr>
</tbody>
</table>

*Figure 2-8. Machine Language Subroutine*

Notice that some codes consist of single bytes (68, 18, 60), some consist of two bytes (A9 and 3F, 69 and 41), and one consists of three bytes (8D and 00 and 18). Data and memory addresses accompany the appropriate instruction to form multi-byte instructions.

**HOW THE MACHINE LANGUAGE PROGRAM WORKS**

So far, we have said little about the the individual machine language codes. These codes have little or no meaning to you at this time, but each does have a precise meaning to the computer. Let's consider them in the following groups.

1. 68
2. 18
1. The first instruction (68) is the machine language code that tells the computer to pull one byte of data off the stack and place it in the accumulator.

The stack is a special area of memory where the computer keeps certain numbers and addresses in a specific order for later use. Each new item going onto the stack is placed on top of the stack. All other items are "pushed down" one place when a new item goes on top.

Example:

```
New item

3 items on the stack
```

```
New item is pushed on from the top.
```

When the USR function "calls" a machine language program from BASIC, the BASIC address to which the machine language is to return when done is placed on the stack in two separate bytes—low-address byte first, then high-address byte on top.

```
Top yy ← High byte of return address
Bottom xx ← Low byte of return address
```

The Stack

Following this, the computer puts any variables being passed to the machine language subroutine. The last item placed on the stack at this time is the number of variables that were passed. In our example, we didn't pass any variables. Therefore the stack looked like this when the machine language subroutine was called.

```
Top 0 ← Number of variables passed
yy ← High byte of return address
Bottom xx ← Low byte of return address
```

In order to have the return address available when the subroutine has been completed, we had to remove that zero on top of the stack. So, we used the instruc-
tion (68) to pull the top value off the stack. It was placed in the accumulator by the instruction. The stack now has the return address on top ready for use.

<table>
<thead>
<tr>
<th>Top</th>
<th>yy</th>
<th>— High byte of return address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bottom</td>
<td>xx</td>
<td>— Low byte of return address</td>
</tr>
</tbody>
</table>

The Stack Now

<table>
<thead>
<tr>
<th>0</th>
</tr>
</thead>
</table>

The Accumulator Now

The number (0) in the accumulator will not be used, but we had to get it off the stack in order to access the return address bytes when needed.

2. The second instruction (18) clears the Carry bit of the Processor Status Register (resets it to zero). The add instruction (fourth instruction) will automatically add the Carry bit when two numbers are summed. Since you do not want a carry to be added in, the instruction is used to ensure that the Carry bit is zero. Later, you will be doing some multiple-byte additions with larger numbers. At that time, you will see that the Carry bit must be used when adding all but the Least Significant Byte.

3. The third instruction consists of two hex codes (or two bytes):
   \[ \text{A9 3F} \]
   A9 is the hex representation for the instruction, “Load the accumulator with the value that immediately follows.” The 3F is the number that is loaded.

4. The fourth instruction consists of the two hex codes (two bytes):
   \[ \text{69 41} \]
   69 is the hex representation for the instruction “Add the number that follows to the value in the accumulator.” The result will replace the value in the accumulator. The hex value 41 is the number that is added.

5. The fifth instruction consists of three hex codes (three bytes):
   \[ \text{8D 00 18} \]
   8D is the hex representation of the instruction, “Store the value that is in the accumulator into the memory location that follows.” The memory location used is 1800 hex (6144 decimal). Notice that the memory location is too large to be expressed as one byte. Therefore, two hex codes are used. Notice also the order in which the high-address byte (18) and the low-address byte (0) are given. The low-address byte is first, followed by the high-address byte. This convention is followed when instructions contain a two-byte address.

6. The last instruction consists of a single hex code (one byte):
   \[ \text{60} \]
   60 is the hex representation of the instruction, “Return from subroutine.” This single instruction tells the computer to return to the place where it came from.
The computer takes the top two bytes off the stack in order to find where it should return in the BASIC program and obtain the next BASIC statement.

A machine language program is executed in sequential order by memory locations just as a BASIC program is executed by line numbers. There are exceptions like the BASIC GOTO and GOSUB statements. Machine language programs can perform similar tricks with JUMP and BRANCH instructions, as you will see later.

Although this book is mainly concerned with Assembly Language programming and the Atari Assembler Cartridge, you should realize that the computer actually works with machine language codes. The assembler language is merely an easier method to remember and enter machine language codes, as you will see in later chapters.

We have used six machine language instructions in this first program. Since machine language instructions are coded numbers, they are hard to remember. However, each one of them has a mnemonic code (an abbreviation for the meaning of the instruction) that is used in assembly language programming.

### INSTRUCTIONS USED SO FAR

<table>
<thead>
<tr>
<th>Machine Language Code</th>
<th>Assembler Mnemonic</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>68</td>
<td>PLA</td>
<td>Pull one byte off the stack and place in the accumulator</td>
</tr>
<tr>
<td>18</td>
<td>CLC</td>
<td>Clear the Carry bit</td>
</tr>
<tr>
<td>A9</td>
<td>LDA</td>
<td>Load the accumulator with the hex value that follows</td>
</tr>
<tr>
<td>69</td>
<td>ADC</td>
<td>Add (with carry) the hex value that follows to the accumulator</td>
</tr>
<tr>
<td>8D</td>
<td>STA</td>
<td>Store the value in the accumulator into the memory location that follows</td>
</tr>
<tr>
<td>60</td>
<td>RTS</td>
<td>Return from the subroutine</td>
</tr>
</tbody>
</table>

### SUMMARY

In this chapter, you have learned:

- To access a machine language subroutine from a BASIC program by the USR function of BASIC;
• That machine language instruction codes are binary numbers that can be entered in hexadecimal form;
• That some instruction codes occupy one byte, some two bytes, and some three bytes;
• How the "push down" stack is used to store numbers and addresses temporarily;
• How the numbers are pulled off the stack;
• That every machine language subroutine must have a Return from Subroutine instruction to send control back to the main program; and
• The following machine language instructions:

68 PLA Pull one byte off the stack and place it in the accumulator
A9 LDA Load the accumulator with the value immediately following
69 ADC Add the hex value that follows to the value in the accumulator
8D STA Store the value in the accumulator into the memory location that follows
60 RTS Return from Subroutine
18 CLC Clear the Carry flag

EXERCISES

1. Hexadecimal codes are a short way to write __________ instruction codes which the computer can understand.
   (decimal, binary)

2. What kind of numbers are used in BASIC language programs?
   (binary, hex, decimal)
3. Fill in the hexadecimal and decimal equivalents of the following binary numbers.
   (a) 1101 = hex = decimal
   (b) 0110 = hex = decimal
   (c) 10101001 = hex = decimal
4. When using a machine language subroutine from BASIC, you enter instruction codes in \((\text{hex, decimal})\) form, and the BASIC interpreter converts them to \((\text{hex, decimal})\) form.
5. What is the three-letter abbreviation for the function used in Atari’s BASIC to call a machine language subroutine? ___ ___ ___
6. If you want to call a machine language program from BASIC and had dimensioned the variable \(E\$\) to 1 \((100 \text{ DIM } E\$(1))\) for this purpose, what is the BASIC line that would call the subroutine?
   500 ________________
7. What is the last instruction that every machine language subroutine must execute to get back to the BASIC program?
   ________________
8. Suppose you wish to store the decimal value, 72, in the memory location 1925 hex. The number 72 is now in the accumulator. The machine language instruction to store the value would consist of three bytes in the following order. Fill in the blanks.
   8D
   __
   __
9. In exercise 8, what hex value would be loaded into memory location 1925?
   ________________

**ANSWERS**

1. Binary
2. Decimal
3. (a) 1101 = D hex = 13 decimal
   (b) 0110 = 6 hex = 6 decimal
   (c) 10101001 = A9 hex = 169 decimal
4. Hex
   Decimal
5. USR
6. 500 X = USR(ADR(E$)+1)

   This may be some other variable

7. 60 or RTS or Return from Subroutine

8. 8D
   25 (Don’t forget the reverse order)
   19

9. 48 (4*16+8 = 72)
Memory Use

When you use BASIC language and machine language programs together, you must use separate portions of memory for each language. Atari's BASIC interpreter (in the BASIC ROM Cartridge) takes care of allocating the memory locations used in the BASIC language part of your program. You must make sure that your machine language part uses a different area of memory.

ATARI MEMORY MAP

Certain areas of the computer's 65536 memory locations are used by the Atari Operating System (8K of ROM) and by the ROM cartridges (such as BASIC or Assembler) that may be plugged into the computer console. We cannot POKE machine language programs into these areas. The Operating System and ROM cartridges also use some of the system's RAM space. We must not POKE into those areas as we might destroy our own program or cause some other programming disaster.

Figure 3-1 shows a map of the Atari's memory. Look closely and you can see areas of memory called Free RAM. These areas may be safely used for your programs. However, as stated above, you must keep BASIC and machine language programs separated.

Figure 3-1. Memory Map
The technique which we have been using so far is to DIMension a string variable \texttt{E$} to one \{1\}. This saves one memory location immediately following the area used by our BASIC program. The address of this location will vary from program to program depending on the length of the BASIC program, but the machine language memory locations will always follow the BASIC program. We will demonstrate this with three short BASIC programs which are of different lengths. They all end by printing the address of the memory location saved for \texttt{E$}.

**DEMO PROGRAM #1**

10  CLR: DIM \texttt{E$(1)}
20  PRINT ADR(\texttt{E$})

Enter and RUN Demo Program #1. You will see that one location was saved for \texttt{E$} at address 11182.

```
RUN
11182
READY
```

**DEMO PROGRAM #2**

10  CLR: DIM \texttt{E$(1)}
20  FOR X = 1 TO 10  ← A very short time delay
30  NEXT X
40  PRINT ADR(\texttt{E$})

Enter and RUN Demo Program #2. You will see that the location saved for \texttt{E$} is now at address 11221. Since Demo Program #2 is longer than Demo Program #1, the location saved for \texttt{E$} has a higher address.

```
RUN
11221
READY
```
DEMO PROGRAM #3

10 CLR: DIM E$(1)
20 FOR X = 1 TO 10
30 PRINT X;
40 NEXT X
50 PRINT
60 PRINT ADR(E$)

Enter and RUN Demo Program #3. You will see that the address saved for E$ is higher yet because Demo Program #3 is longer than the other two.

```
12345678910
11235

READY
```

We have made use of the memory location saved for E$ to find a safe place in RAM (out of the way of our BASIC program and other reserved areas) for our machine language subroutines. We POKEd the machine language subroutine into a memory area that started at the next higher location than that saved for E$. We will continue to use this method.

HOW BASIC FINDS THE MACHINE LANGUAGE PROGRAM

You may wonder how your BASIC program knows where to find the machine language subroutine. Let's take another look at the program used in Chapter 2 that added two numbers.

First section

```
100 REM INITIALIZE STORAGE ADDRESS
110 CLR: DIM E$(1), E(10)
```

Save one memory location for E$       Save 10 locations for machine language subroutine
Second section

120 REM POKE IN SUBROUTINE
130 FOR Y = 1 TO 10
140 INPUT N
150 POKE ADR(E$)+Y,N
160 NEXT Y

POKE 10 hex codes into successive memory locations (E$+1 through E$+10)

Third section

170 REM CALL SUBROUTINE
180 X = USR(ADR(E$)+1)

Jump to the machine language subroutine starting in memory at address (E$)+1

---

Figure 3-2. Memory Used for Machine Language Subroutine

The fourth section of the BASIC program prints the results of the machine language program.

Fourth section

190 REM PRINT RESULTS
200 GR,O: PRINT "THE SUM IS";
210 PRINT PEEK (6144) ← Print result that the machine language program has placed in memory location 6144 (1800 hex)
220 END

The computer saved one location for E$ in an area of memory that is out of the way of your BASIC program. At lines 130 through 160, the program POKEs the machine language codes into the memory just above the space that was saved for E$. Your BASIC program “calls” the machine language program at line 180 by the USR function. The USR function specifies the address where the machine language program can be found. Thus the USR function works like the BASIC statement: GOSUB
The Atari also has the capability of passing one or more quantities to a machine language program through the USR function.

Example:

100 A=5
110 X = USR(ADR(E$)+1,A)

Pass the value of A to the machine language program

The value of variable A is placed on the stack as described in Chapter 2. The value is placed there as a two-byte hex number.

Let's review that process again:

1. The address to which the computer is to return in your BASIC program is placed on the stack.

\[
\begin{array}{c|c}
\text{High byte of return address} & \rightarrow \text{YY} \\
\text{Low byte of return address} & \rightarrow \text{XX}
\end{array}
\]
PASSING VARIABLES TO MACHINE LANGUAGE SUBROUTINE

2. The value of variable A is placed on top of the stack. If A = 5,

| 00 | ← High byte of A                      |
| 05 | ← Low byte of A                      |
| yy | ← High byte of return address       |
| xx | ← Low byte of return address        |

3. The number of variables passed is placed on top of the stack.

| 01 | ← Number of variables passed         |
| 00 | ← High byte of A                     |
| 05 | ← Low byte of A                      |
| yy | ← High byte of return address       |
| xx | ← Low byte of return address        |

Your machine language program may then pull the values off the stack and use them.

PASSING VARIABLES TO MACHINE LANGUAGE SUBROUTINE

Following is an addition program that will demonstrate this methods. It allows you to input a value that is passed to the machine language subroutine. The value of five is added to your input by the machine language subroutine. The result is placed in memory so that the BASIC program can find it.

BASIC PROGRAM—ONE VARIABLE PASSED

100 CLR: DIME$(1), E(10)

200 FOR Y = 1 TO 10
210   INPUT N
220   POKE ADR(E$)+Y,N
230   NEXT Y

300   GR.O: PRINT “INPUT A”
310   INPUT A
320   X = USR(ADR(E$)+1,A)
330   PRINT PEEK(6144)
340   END

Input machine language subroutine (10 bytes)
Input a positive decimal integer less than 251
Call subroutine
Print sum of your number and 5
When the BASIC program is executed, the value of A will be placed on the stack, and your machine language program will pull it off and use it. The result will be printed when you come back to your BASIC program.

The following machine language subroutine shows the contents of the accumulator, memory storage location, and stack as the subroutine is executed.

### MACHINE LANGUAGE SUBROUTINE

<table>
<thead>
<tr>
<th>Address</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>E$+1</td>
<td>68</td>
<td>Pull top value off stack and put into accumulator</td>
</tr>
<tr>
<td>E$+2</td>
<td>18</td>
<td>Clear the Carry bit</td>
</tr>
<tr>
<td>E$+3</td>
<td>68</td>
<td>Pull High byte of A off stack and put into accumulator; not used</td>
</tr>
<tr>
<td>E$+4</td>
<td>68</td>
<td>Pull Low byte of A off stack and put into accumulator</td>
</tr>
<tr>
<td>E$+5</td>
<td>69</td>
<td>Add the number that follows</td>
</tr>
<tr>
<td>E$+6</td>
<td>05</td>
<td>Five is added</td>
</tr>
<tr>
<td>E$+7</td>
<td>8D</td>
<td>Store accumulator into memory that follows</td>
</tr>
<tr>
<td>E$+8</td>
<td>00</td>
<td>Low byte of memory</td>
</tr>
</tbody>
</table>

#### Original Stack

<table>
<thead>
<tr>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
</tr>
<tr>
<td>ha</td>
</tr>
<tr>
<td>la</td>
</tr>
<tr>
<td>yy</td>
</tr>
<tr>
<td>xx</td>
</tr>
</tbody>
</table>

#### New Stack

<table>
<thead>
<tr>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ha</td>
</tr>
<tr>
<td>la</td>
</tr>
<tr>
<td>yy</td>
</tr>
<tr>
<td>xx</td>
</tr>
</tbody>
</table>

No. of variables high and low byte of A return address

No. of variables high and low byte of A return address
USING THE ONE-VARIABLE PROGRAM

To enter and use the program, follow these steps.

1. Enter the BASIC PROGRAM—ONE VARIABLE PASSED.
2. RUN the BASIC program and ENTER these hex codes (one at a time in response to the ? prompt).

?104
?24
?104
?104
?105
?5
?141
?0
?24
?96

3. After the last entry, the screen will clear and the INPUT prompt will appear on the screen.

INPUT A?

Try typing 95 and press the RETURN key.

INPUT A?95
100

READY

95+5 = 100

You may try other numbers by typing GOTO 300 in response to the READY prompt. This will repeat step 3. Do not type RUN at this time or you will have to enter all the hex codes again as in step 2.
PASSING MORE THAN ONE VARIABLE

More than one variable can be passed from your BASIC program to your machine language subroutine by the USR function. The next program demonstrates this. It passes two variables, A and B. These two values are then summed by the machine language subroutine. Notice the changes in the BASIC program and the subroutine.

**BASIC PROGRAM—TWO VARIABLES PASSED**

100  CLR: DIM E$(1), E(16)  
200  FOR Y = 1 TO 16
210    INPUT N
220    POKE ADR(E$)+Y,N
230    NEXT Y
300  GR. O: PRINT "INPUT A";
310  INPUT A  
320  PRINT "INPUT B";
330  INPUT B
340  X=USR(ADR(E$)+1,B,A)  
350  PRINT PEEK(6144)
360  PRINT "PRESS RETURN TO REPEAT";
370  INPUT A$:
380  GOTO 300

When line 340 is executed, the USR function causes the stack to be filled like this:

```
TOP    2  ← Number of variables passed
   ha   ← High byte of A
   la   ← Low byte of A
   hb   ← High byte of B
   lb   ← Low byte of B
   yy   ← High byte of return address
   xx   ← Low byte of return address
```

BOTTOM
Remember, when data items are pulled off the stack, they come off the top. The last item put on the stack is the first one taken off the stack. You may notice that two bytes are used for each number that was pushed on the stack.

When a two-byte number is used, one byte is referred to as the Least Significant Byte (LSB). The other is referred to as the Most Significant Byte (MSB).

Example:

<table>
<thead>
<tr>
<th>MSB (Most Significant Byte)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2^7</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>LSB (Least Significant Byte)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2^7</td>
</tr>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

Don't confuse the Most and Least Significant Bytes with the Most and Least Significant Bits. Each byte has an MSB (Most Significant Bit) and a LSB (Least Significant Bit).

To use a two-byte number, you consider the Most Significant Byte as an extension of the Least Significant Byte. The place values of the Least Significant were assigned powers of two from 0 through 7.
The places values of the Most Significant Byte are assigned the next higher powers of two (8 through 15).

The decimal value resulting from the combined bytes (considered as one number) is:

\[
\begin{array}{cccccccc}
2^15 & 2^14 & 2^13 & 2^12 & 2^11 & 2^10 & 2^9 & 2^8 \\
1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\
\end{array}
\]

In decimal: \(32768+2048+256+64+4+2 = 35142\)

Split into 4-bit parts:

- \(1000\)
- \(1001\)
- \(0100\)
- \(0110\)

This binary value is equivalent to this hex value

HEX digits: 8 9 4 6

Hex format: MSB | LSB = \[
\begin{array}{cccc}
16^3 & 16^2 & 16^1 & 16^0 \\
8 & 9 & 4 & 6 \\
\end{array}
\]

\[
\begin{align*}
8*4096 &= 32768 \\
+9*256 &= 2304 \\
+4*16 &= 64 \\
+6*1 &= 6 \\
35142 \quad (\text{decimal})
\end{align*}
\]

The use of two-byte numbers allows us to use larger inputs (up to 65535). However, our machine language program is only designed to add single-byte integers (0-255). Therefore, the High bytes of the variable A and the variable B are both zero. If you input numbers whose sum is larger than 255, you will get an incorrect result.

The machine language program is longer this time, consisting of 16 hex codes (16 bytes).
MACHINE LANGUAGE SUBROUTINE

<table>
<thead>
<tr>
<th>Address</th>
<th>Hex Code</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>E$+1</td>
<td>68</td>
<td>Pull top value off stack and store in accumulator</td>
</tr>
<tr>
<td>E$+2</td>
<td>18</td>
<td>Clear the Carry flag to prepare for addition of Low bytes of numbers</td>
</tr>
<tr>
<td>E$+3</td>
<td>68</td>
<td>Pull High byte of A and put in accumulator</td>
</tr>
<tr>
<td>E$+4</td>
<td>68</td>
<td>Pull Low byte of A and put in accumulator</td>
</tr>
<tr>
<td>E$+5</td>
<td>8D</td>
<td>Store low byte of A into memory location that follows</td>
</tr>
<tr>
<td>E$+6</td>
<td>01</td>
<td>Low byte of memory used for storage</td>
</tr>
<tr>
<td>E$+7</td>
<td>18</td>
<td>High byte of memory used for storage</td>
</tr>
<tr>
<td>E$+8</td>
<td>68</td>
<td>Pull High byte of B and put in accumulator</td>
</tr>
<tr>
<td>E$+9</td>
<td>68</td>
<td>Pull Low byte of B and put in accumulator</td>
</tr>
<tr>
<td>E$+10</td>
<td>6D</td>
<td>Add with carry the value in the accumulator and the value in the memory location that follows</td>
</tr>
<tr>
<td>E$+11</td>
<td>01</td>
<td>Low byte of memory</td>
</tr>
<tr>
<td>E$+12</td>
<td>18</td>
<td>High byte of memory</td>
</tr>
<tr>
<td>E$+13</td>
<td>8D</td>
<td>Store the result in the memory location that follows</td>
</tr>
<tr>
<td>E$+14</td>
<td>00</td>
<td>Low byte of memory used for storage</td>
</tr>
<tr>
<td>E$+15</td>
<td>18</td>
<td>High byte of memory used for storage</td>
</tr>
<tr>
<td>E$+16</td>
<td>60</td>
<td>Return from subroutine</td>
</tr>
</tbody>
</table>

Enter the BASIC program now and RUN it so that you can enter the machine language codes in the usual way. After the 16 hex codes have been entered, the screen clears and you will see the following.

```
INPUT A?•

Try 135

INPUT A?135
INPUT B?•

Try 99

INPUT A?135
INPUT B?99
234
PRESS RETURN TO REPEAT?•
```
A new instruction is used at E$+10$ of the Two Variable program. It is ADC (ADd with Carry) with a hex code of 6D. We used the hex code 69 for addition in earlier programs. Why the difference? Well, some of the functions, such as addition, can be performed in several modes. This will be more thoroughly discussed in Chapter 5. The 69 instruction adds the value immediately following it to the value in the accumulator. The 6D instruction adds the value that is contained in the memory location that follows the instruction to the value in the accumulator. Some instructions can be performed in only one mode, but others can be performed in more than one mode. Even though the instruction may perform the same function, each mode of the instruction performs the function in a different way. Therefore, each mode has a different hex code (called an Operation Code or Op Code for short) for each mode.

So far you have only seen machine language routines that proceed straight through consecutive memory locations. All of the arithmetic operations were performed in the accumulator, called the A register. The 6502 microprocessor has other registers which are used in different ways.

Since the computer is able to perform repetitive operations, such as a FOR-NEXT loop, in BASIC, you probably realize that similar operations can be done in machine language. In Chapter 1, a brief discussion of the X and Y index registers was given. Let’s take a look now at how they can be used.

**A MACHINE LANGUAGE LOOP**

You have no doubt used FOR-NEXT loops many times in BASIC. In the next machine language subroutine, we'll utilize a similar looping technique using the X index register as a counter. Here is a comparison of the FOR-NEXT loop and a description of a machine language loop that would do about the same thing.

<table>
<thead>
<tr>
<th>BASIC</th>
<th>MACHINE LANGUAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>FOR X = 0 TO 9</td>
<td>Load X register with zero.</td>
</tr>
<tr>
<td>PRINT X</td>
<td>Store the value in the X register into memory.</td>
</tr>
<tr>
<td>NEXT X</td>
<td>Increment the X register.</td>
</tr>
<tr>
<td></td>
<td>Compare the X register with 10.</td>
</tr>
<tr>
<td></td>
<td>Branch back if X is not equal to 10.</td>
</tr>
<tr>
<td></td>
<td>Return if X does equal 10.</td>
</tr>
</tbody>
</table>

The machine language program will be more detailed than the BASIC program. Each step must be implemented through exact machine language instructions.
Here is how the BASIC program and machine language subroutine will be accomplished.

FLOW OF BASIC PROGRAM

Clear arrays and set memory
READ in machine language instructions
POKE instructions into memory
Clear the screen and call the subroutine

FLOW OF SUBROUTINE

Pull top byte off stack
Set loop counter to zero

Store the count in memory
Increment the counter

Is \( X = 0A \) ?

Yes
Return to BASIC program

No

PRINT the counts used in the machine language program

END

The hex codes for the machine language instructions are given in Appendixes A and B. A more complete description of the instructions can be found in the MCS6500 Microcomputer Family Programming Manual.* The instructions are listed by their mnemonic codes. The operation codes are given in hexadecimal form. Therefore, you need to convert them to decimal form in order to POKE them into memory from your BASIC program. Calculate the decimal equivalents for the machine language subroutine that follows. Fill in the blanks in the table.

*Published by MOS Technology, Inc., 950 Rittenhouse Road, Norristown, PA 19401.
MACHINE LANGUAGE SUBROUTINE

<table>
<thead>
<tr>
<th>Address</th>
<th>Hex Code</th>
<th>Decimal equivalent</th>
<th>Mnemonic code</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>E$+1</td>
<td>68</td>
<td></td>
<td>PLA</td>
<td>Pull byte off stack</td>
</tr>
<tr>
<td>E$+2</td>
<td>A2</td>
<td></td>
<td>LDX 0</td>
<td>Load the X register with zero</td>
</tr>
<tr>
<td>E$+3</td>
<td>00</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E$+4</td>
<td>8A</td>
<td></td>
<td>TXA</td>
<td>Transfer the value in X register to the accumulator</td>
</tr>
<tr>
<td>E$+5</td>
<td>9D</td>
<td></td>
<td>STA 1800,X</td>
<td></td>
</tr>
<tr>
<td>E$+6</td>
<td>00</td>
<td></td>
<td></td>
<td>Store the value in accumulator into memory</td>
</tr>
<tr>
<td>E$+7</td>
<td>18</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E$+8</td>
<td>E8</td>
<td></td>
<td>INX</td>
<td>Increment the count in the X register</td>
</tr>
<tr>
<td>E$+9</td>
<td>E0</td>
<td></td>
<td>CPX 0A</td>
<td>Compare the value in X with 10 (decimal)</td>
</tr>
<tr>
<td>E$+10</td>
<td>0A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E$+11</td>
<td>D0</td>
<td></td>
<td>BNE F7</td>
<td>Branch back if X = 10 (decimal) to address E$+4</td>
</tr>
<tr>
<td>E$+12</td>
<td>F7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E$+13</td>
<td>60</td>
<td></td>
<td>RTS</td>
<td>Return from Subroutine</td>
</tr>
</tbody>
</table>

Figure 3-4. Machine Language Exercise

This subroutine uses several new instructions.
NEW INSTRUCTIONS USED

Six new instructions appeared in the machine language subroutine. At E$+2 the hex code A2 was used. This is the operation code for the instruction, "Load the X register with the value that follows it." In our subroutine, we are loading the X register with zero. Therefore the instruction at E$+2 and the data (0) at E$+3 go together.

At E$+4, the hex code 8A was used. This is the operation code for the instruction, "Transfer the value that is in the X register to the accumulator." We are going to store this value into memory, but to do this we must first place it into the accumulator. The accumulator is quite often used as a temporary storage location in passing data from one place in the computer to another.

At E$+5, the hex code 9D is used. This is the operation code for the instruction, "Store the value that is in the accumulator into memory." This is a different mode of the store accumulator instruction. This mode adds together the memory location following the instruction to the value contained in the X register. The value in the accumulator is stored into this calculated address. The X register is used to index the memory location. The same instruction can be used several times to store values in different memory locations by changing the value in the X register.

At E$+8, the hex code E8 is used. This is the operation code for the instruction, "Increment the value that is in the X register." The value is increased (incremented) by one. Thus, each time the computer passes through the loop, a new memory location will be used in the instruction at E$+5.

At E$+9, the hex code E0 is used. This is the operation code for the instruction, "Compare the value that is in the X register to the value that follows the E0 operation code." The value OA (10 decimal) follows this instruction. Thus the value in the X register is compared with 0A (hex). This supplies the computer with a condition (X=OA or X≠OA) so that it can make a branching decision at the next instruction.

At E$+11, the hex code D0 is used. This is the Operation Code for the instruction, "Branch if the previous condition is not equal." Otherwise go on to the next sequential instruction. For values of X less than 0A, the computer will branch back to the instruction at location E$+4. When X reaches OA, the computer will go on to the instruction at E$+13. The computer goes through this loop 10 times (for X values of 0 through 9) and then returns to the main program. The value F7 used at location E$+12 is a signed hexadecimal equivalent of the decimal value -9. In other words, the computer must branch back nine steps if the given condition is true (X≠OA). The size and direction of branches will be discussed in Chapter 5.

We'll trace through the machine language subroutine step by step later in this chapter, but let's first take a look at the BASIC program which loads, calls, and prints out the results of the machine language subroutine.
BASIC PROGRAM

100 REM INITIALIZE MEMORY
110 CLR: DIM E$(1), E(13)

120 REM ENTER MACHINE CODES
130 FOR N = 1 TO 13
140 READ C ← Read codes from DATA
150 POKE ADR(E$)+N,C ← Place in subroutine
160 NEXT N

170 REM CALL SUBROUTINE
180 X=USR(ADR(E$)+1)

190 REM PRINT RESULTS
200 GR.0 ← Clear screen
210 FOR N = 0 TO 9
220 PRINT PEEK(6144+N) ← Print the counts used
230 NEXT N for machine language loop
240 END

250 REM DECIMAL DATA
260 DATA 104, 162, 0, 138, 157, 0, 24, 232, 22, 4, 10, 208, 247, 96
        Machine codes in decimal form (from Figure 3-4)

The FOR-NEXT loop at lines 130 through 160 READs in the decimal equivalents of the hex codes and POKEs them into the correct memory locations for the machine language subroutine. Remember, that even though BASIC needs the data in decimal form, the codes will be inserted into memory in binary form. The first few bytes of the program will actually be stored like this:

\[
\begin{align*}
E$+1 & \begin{array} {c} 0 \ 1 \ 1 \ 0 \ 1 \ 0 \ 0 \ 0 \\ 6 \ 8 \end{array} = 64+32+8 = 104 \text{ decimal} \\
E$+2 & \begin{array} {c} 1 \ 0 \ 1 \ 0 \ 0 \ 0 \ 1 \ 0 \\ A \ 2 \end{array} = 128+32+2 = 162 \text{ decimal} \\
E$+3 & \begin{array} {c} 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \\ 0 \ 0 \end{array} = 0 \text{ decimal}
\end{align*}
\]
TRACING THROUGH THE SUBROUTINE

The best way to understand a program is to put yourself in the place of the computer and perform each instruction as it occurs in the program. Let's now trace through the operations performed by the computer in our subroutine. To do this, we will have to keep track of what is happening in register X, the accumulator, and in the memory storage locations. We might also show what is contained on the stack.

As the machine language subroutine is called, the stack contains:

```
00 ← Number of variables passed
yy             ← Return address
xx
```

When the first machine language program is executed, the stack changes.

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>X</th>
<th>Accumulator</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>E$+2</td>
<td>Load X</td>
<td>0</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>E$+3</td>
<td>with zero</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>E$+4</td>
<td>Transfer X to A</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>E$+5</td>
<td>Store accumulator</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>in (1800+X)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E$+7</td>
<td></td>
<td>0</td>
<td>0</td>
<td>1800→0</td>
</tr>
<tr>
<td>E$+8</td>
<td>Increment X</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>E$+9</td>
<td>Compare X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E$+10</td>
<td>with 0A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(X = 1, not 0A)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E$+11</td>
<td>Branch back to</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>E$+12</td>
<td>E$+4 if X ≠ 0A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E$+4</td>
<td>Transfer X to A</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Address</td>
<td>Instruction</td>
<td>X</td>
<td>Accumulator</td>
<td>Memory</td>
</tr>
<tr>
<td>---------</td>
<td>------------------------------------------------</td>
<td>-----</td>
<td>-------------</td>
<td>--------</td>
</tr>
<tr>
<td>E$+5</td>
<td>Store Accumulator in (1800+X)</td>
<td>1</td>
<td>1</td>
<td>1801</td>
</tr>
<tr>
<td>E$+6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E$+7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E$+8</td>
<td>Increment X</td>
<td>2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>E$+9</td>
<td>Compare X with 0A</td>
<td>2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>E$+10</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(X=2, not 0A)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E$+11</td>
<td>Branch back to E$+4 if X\neq 0A</td>
<td>2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>E$+12</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E$+4</td>
<td>Transfer X to A</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>E$+5</td>
<td>Store Accumulator in (1800+X)</td>
<td>2</td>
<td>2</td>
<td>1802</td>
</tr>
<tr>
<td>E$+6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E$+7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E$+8</td>
<td>Increment X</td>
<td>3</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>E$+9</td>
<td>Compare X with 0A</td>
<td>3</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>E$+10</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(X=3, not 0A)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E$+11</td>
<td>Branch back to E$+4 if X\neq 0A</td>
<td>3</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>E$+12</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E$+4</td>
<td>Transfer X to A</td>
<td>3</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>E$+5</td>
<td>Store Accumulator in (1800+X)</td>
<td>3</td>
<td>3</td>
<td>1803</td>
</tr>
<tr>
<td>E$+6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E$+7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

This process continues with the value in X being increased by one. That value is transferred to the accumulator and then put into the memory location whose address is 1800+X.

The X register is used as a counter for the loop and also to index the memory storage (store into successive memory locations from 1800 upwards). We continue the trace after X has been incremented to 9 and the branch has been taken back to E$+4.
TRACING THROUGH THE SUBROUTINE

Address Instruction X Accumulator Memory
E$+4 Transfer X to A 9 9
E$+5 Store Accumulator in (1800+X) 9 9 1800→
E$+6
E$+7
E$+8 Increment X A 9
E$+9 Compare X A 9
E$+10 with 0A
(X=0A, at last)
E$+11 Branch back to A 9
E$+12 to E$+4 if X ≠0A
E$+13 Return from Subroutine

The return address is taken from the stack. The stack is now empty. Thus, when the subroutine is completed, the values assigned to the X register (0 through 9) have been stored into memory.

DATA STORED BY SUBROUTINE

<table>
<thead>
<tr>
<th>Hex Address</th>
<th>Decimal Address</th>
<th>Value Stored</th>
</tr>
</thead>
<tbody>
<tr>
<td>1800</td>
<td>6144</td>
<td>0</td>
</tr>
<tr>
<td>1801</td>
<td>6145</td>
<td>1</td>
</tr>
<tr>
<td>1802</td>
<td>6146</td>
<td>2</td>
</tr>
<tr>
<td>1803</td>
<td>6147</td>
<td>3</td>
</tr>
<tr>
<td>1804</td>
<td>6148</td>
<td>4</td>
</tr>
<tr>
<td>1805</td>
<td>6149</td>
<td>5</td>
</tr>
<tr>
<td>1806</td>
<td>6150</td>
<td>6</td>
</tr>
<tr>
<td>1807</td>
<td>6151</td>
<td>7</td>
</tr>
<tr>
<td>1808</td>
<td>6152</td>
<td>8</td>
</tr>
<tr>
<td>1809</td>
<td>6153</td>
<td>9</td>
</tr>
</tbody>
</table>

Notice that the last value placed in the X register (0A) was not saved in memory. When the subroutine returns to the BASIC program the values stored in the above memory locations are printed out on the screen.
SUMMARY

In this chapter, you have learned:

- How to convert a hexadecimal number to its decimal equivalent;
- How memory is allocated in the Atari computer;
- How to set aside memory locations for a machine language subroutine;
- How to use the X index register to count the number of times that a loop is executed in a machine language program;
- How to test the X register to determine when to exit a loop;
- How to trace a machine language program by performing the instructions yourself;
- That some instructions can be used in more than one way (more than one mode);
- That the USR function can pass values from your BASIC program to your machine language program;

\[
\begin{align*}
1080 & \quad A=5: B=6 \\
1090 & \quad X=\text{USR}(\text{ADR}(E$)+1,A,B
\end{align*}
\]

- That numbers passed by the USR function are pushed onto the stack as two-byte numbers;
- The following new machine language instructions:

A2   LDX   Load the X register with the value that follows.
8A   TXA   Transfer the value in the X register to the accumulator.
9D  STA  Store the accumulator’s contents into the memory location that follows, indexed by the X register.

E8  INX  Increment the value in the X register.

E0  CPX  Compare the value in the X register to the value that follows.

D0  BNE  Branch forward, or back, the number of steps given if condition is not equal to zero.

EXERCISES

1. Convert the following hexadecimal numbers to their decimal equivalents.
   A2 hex = ____________ decimal
   3B hex = ____________ decimal
   9E hex = ____________ decimal

2. It is not safe (or at least caution must be used) to POKE ____________ BASIC, machine language instructions in the area of memory below your ____________ BASIC, machine language program instructions.

3. If your BASIC program dimensions the string variable E$ to one, you are saving space in the memory area above your BASIC program. At what address would it be safe to start a machine language program called from your BASIC program? ____________

4. Does the location saved for E$ depend upon the length of your BASIC program? ____________

5. Data assigned to variables can be passed from a BASIC program to a machine language subroutine. Even though no variables are passed to the sub-
routine, the subroutine must perform some operation on the stack before a return can be made from the subroutine. What is that operation?

6. Suppose that the X register contains the value 7, and the accumulator holds the value 1A. The computer executes the instruction:
   E$+7  9D  STA 1800, X
   E$+8  00
   E$+9  18
   In what memory location is the value 1A stored?

7. The BASIC program of page 44 and the subroutine of page 42 are to be executed. The only change made is to line 210 of the BASIC program. It is changed to:
   210 FOR N = 3 TO 5
   Show what will be printed on the screen when the program is executed.
   ← Answer here

8. What changes would be made to the machine language subroutine on page 42 if you wanted the loop to count to 20 decimal instead of 10 decimal?
   address         hex code

9. What changes would you make to the BASIC program on page 44 to print out the 20 values stored by the changed machine language program (exercise 8)?

   260 DATA

ANSWERS

1. A2 hex = 162 decimal
   3B hex = 59 decimal
   9E hex = 158 decimal

2. Machine
   BASIC

3. E$+1 or ADR(E$)+1

4. Yes (Space is saved for E$ at the end of your BASIC program.)
5. One byte must be pulled off the top of the stack (the number that tells how many variables were passed, zero in this case) before the return address is available.

6. Location 1807 (1800+7) the value in X

7. 
3
4
5

READY

8. E$+10 14 (16+4 = 20)

9. Line 210 FOR N = 0 TO 19
Line 260 DATA 104,162,0,138,157,0,24,232,22,4,20,208,247,95
change these values
Getting Started
With the Assembler

Only the 400 or 800 computer and the Assembler Cartridge are necessary to write, assemble, and execute assembly language programs. However, some storage device (an Atari 410 Program Recorder or Atari 810 Disk Drive) is recommended. Otherwise, you will have to enter your program from the keyboard each time that you want to use it. The Atari 820 printer is an optional addition. It will allow you to make permanent records of your programs in a form that is easy to read.

In writing this book, the authors used the system elements shown below in solid lines. Those elements in broken lines are optional.

Communication between elements of the Atari 400/800 system is managed by the Atari Operating System contained in a 10K ROM (10,000-byte Read Only Memory) using some associated RAM (Random Access Memory). The Assembler taps into the Operating System as needed.

The Atari Assembler Cartridge contains three separate programs:

1. The Writer/Editor
2. The Assembler
3. The Debugger

The Writer/Editor Program, just as the name implies, is used to write and edit your assembly language programs. Assembly language is a shorthand that
uses English like abbreviations to represent instructions to the computer. It also uses numbers in decimal or hexadecimal form to provide data for the programs.

The Assembler Program translates the abbreviations provided by the Writer/Editor into machine language codes and data that the computer can understand. It also takes care of assigning the instructions and data to their proper memory locations.

The Debugger Program is used to execute, test, or trace the operation of the machine language program that the Assembler produced.

You can see that the three programs are used in a logical order. If the Debugger produces a faulty program, you may return to the Writer/Editor Program for changes. The process is then repeated until satisfactory results are obtained. Here is a diagram of the order in which the three programs are used.

![Flowchart](Image)

*Figure 4-2. Use Flow of Assembler Cartridge*

**THE WRITER/EDITOR**

Before we get too involved in the technicalities of assembly and machine language, let's power up the computer and try a short sample program.

Place the Assembler Cartridge in the left slot of the computer, make sure
that your TV set is connected and turned on, and turn on the computer. This is what you should see on the screen:

```
EDIT
```

← This message tells you the Writer/Editor Program is being used.

Cursor indicates that it's your turn to type.

A small amount of memory space (108 locations) is used to store the characters typed in for the line that you are currently typing. This area or memory is called the Current Line Buffer. The buffer is written over as each new line is typed in. It is located in an area of 384 locations (180 HEX) which is reserved for the Assembler Cartridge.

An area of memory just above that saved for the Assembler is used for the Edit Text Buffer. It stores all the input from the keyboard as you are writing the Assembly Language Program. As you enter more and more from the keyboard, the Edit Text Buffer fills up. No upper bound is specified. However, it is possible to exceed the memory limits of your computer.

```
End of 8 K memory → 1C1F (Hex)
```

![Buffer Memory](image)

To find the location of the Current Line Buffer and the Edit Text Buffer, type the command:

```
SIZE (and press the RETURN key)
```

The display:

```
Memory location of Current Line Buffer
0700
EDIT SIZE
0700 0880 1C1F
```

Ready for more

The memory address of the location where the next character typed will appear (increases as you type)
The highest memory location available in your computer
The numbers displayed will differ according to the amount of memory in your system. Remember, our system is the Atari 800 with 8K or RAM and no disk.

You can find out, at any time, how long your program is by using the SIZE command. Subtract the first number (700 in our case) + 180 from the second number. The result is the approximate number of characters in your program.

Before we enter our program, let's take a look at the format used for writing assembly language programs. The program to be assembled is called a Source Program. The Source Program consists of statements just as a BASIC program does. The statements are numbered just as BASIC lines are numbered. Every statement must start with a line number in the range of 0 through 65535. After each statement is entered using the Writer/Editor Program of the Assembler Cartridge, it is terminated (or entered) by pressing the RETURN key.

The format used for each statement is divided into five fields.

Field 1
Statement Number

Field 2
Label (Optional)

Field 3
Op Code

Field 4
Operand (Not always required—depends on the instruction)

Field 5
Comment (Optional—like a REMARK in BASIC)

Exactly one space between fields 1 and 2 if field 2 is used—otherwise 2 spaces

Examples:

1. No label used

Statement Number

20

Op Code

CLC

No label—so two spaces

This instruction—Clear the Carry bit

2. No label used

Statement Number

30

Op Code

LDA

Operand (In Immediate Addressing Mode—#)

#0

No label—so two spaces

This instruction—Put zero in the accumulator
3. Label used

<table>
<thead>
<tr>
<th>Statement Number</th>
<th>Label</th>
<th>Op Code</th>
<th>Operand (In Immediate mode)</th>
</tr>
</thead>
<tbody>
<tr>
<td>40</td>
<td>LOOP</td>
<td>ADC</td>
<td>#1</td>
</tr>
</tbody>
</table>

Label used—so one space

This instruction—Add one to the value in accumulator

4. No label used

<table>
<thead>
<tr>
<th>Statement Number</th>
<th>Op Code</th>
<th>Operand (In Immediate mode)</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>CMP</td>
<td>#3</td>
</tr>
</tbody>
</table>

No label—so 2 spaces

This instruction—Compare the value in accumulator with 3

5. No label—operand gives branch location

<table>
<thead>
<tr>
<th>Statement Number</th>
<th>Op Code</th>
<th>Operand (a label this time)</th>
</tr>
</thead>
<tbody>
<tr>
<td>60</td>
<td>BNE</td>
<td>LOOP</td>
</tr>
</tbody>
</table>

No label—so 2 spaces

This instruction—If the 2 values are *not* equal, go back to the instruction with the LOOP label

Now, let's write an assembly language program using the above statements. When we left the computer, it was in the Edit mode.

```
EDIT
SIZE
0700    0800    1C1F
```

We must first tell the assembler where to start our machine language program, called the *Object Program*, in memory. We do this with a special command.
The starting memory location of the machine language program must be high enough above the Edit Text Buffer (0880 in our case) so that the two programs do not overlap.

Type: 10 *=$1000

Line 2 spaces Starting address ($ indicates a hexadecimal value)

number

Then the balance of the program is entered. Remember to press the RETURN key at the end of each line (just like you do in BASIC).

20 CLC
30 LDA #0
40 LOOP ADC #1
50 CMP #3
60 BNE LOOP

The program must conclude with an END statement (called a PSEUDO OPERATOR or Directive). Every program should have one and only one END Directive.

70 END

one space

THE ASSEMBLER PROGRAM

After the program has been entered using the Writer/Editor Program, the next step is performed by the Assembler Program. The assembler will convert the assembly language instructions to machine language codes and assign memory locations to the machine language codes.

The video screen at this time (before going to the Assembler Program) shows:

EDIT
10 *=$1000
20 CLC
30 LDA #0
40 LOOP ADC #1
50 CMP #3
60 BNE LOOP
70 END

Let's check to see how much memory has been used by our assembly language program. Remember, when we typed SIZE earlier, we saw:
GETTING STARTED WITH THE ASSEMBLER

To see how much memory has been used, once again type: SIZE.

```
60  BNE LOOP
70  END
SIZE
0700 08D4 1C1F
```

Our program has used 8D4-880 or 54 (hex) memory locations:

- 1C1F Top of our memory
- ? End of machine language program
- 1000 Start of machine language program
- 08D4 Top of Edit Text Buffer
- 0880 Beginning Edit Text Buffer

Notice that there is an unused area between the assembly language program (Edit Text Buffer) and the machine language program. That is good. When longer assembly language programs are used, care must be taken that machine language and assembly language programs do not overlap. We can now proceed to the Assembler Program.

You now type: ASM (and press the RETURN key)
Each machine code is displayed followed by the assembly language instructions that created it.

```
60  END
ASM
0000 10 *= $1000
1000 18 20 CLC
1001 A900 30 LDA #0
1003 6901 40 LOOP ADC #1
```
The assembler created this machine language program.

(Object Program)

The assembly language Source Program that you entered is shown on the right side of the screen. The machine language Object Program that the assembler produced is shown on the left side of the screen with the memory locations that were assigned to each machine language instruction. Let's examine the programs line by line to see how the instructions match up.

<table>
<thead>
<tr>
<th>Machine Language</th>
<th>Assembly Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASM 0000</td>
<td>10 *= $1000</td>
</tr>
<tr>
<td></td>
<td>Line 10 assigned the starting address of the machine language program to 1000 (hex).</td>
</tr>
<tr>
<td>1000 18</td>
<td>20 CLC</td>
</tr>
<tr>
<td></td>
<td>The instruction CLC is assembled into the machine language code: 18. The instruction 18 is placed in memory location 1000.</td>
</tr>
<tr>
<td>1001 A900</td>
<td>30 LDA #0</td>
</tr>
<tr>
<td></td>
<td>The instruction LDA #0 is assembled into the machine language code: A900. The instruction A9 (from LDA #) is placed in memory location 1001, and the data (00) is placed in location 1002.</td>
</tr>
<tr>
<td>1003 6901</td>
<td>40 LOOP ADC #1</td>
</tr>
</tbody>
</table>
|                  | The instruction ADC #1 is assembled into the machine language code: 6901. The instruction 69 (from ADC #) is placed in memory location 1003, and the data (01) in location 1004.
The instruction \texttt{CMP} \#3 is assembled into the machine language code: \texttt{C903}. The instruction \texttt{C9} (from \texttt{CMP} \#) is placed in memory location 1005, and the data (03) in location 1006.

The instruction \texttt{BNE} \texttt{LOOP} is assembled into the machine language code: \texttt{DOFA}. The instruction \texttt{D0} (from \texttt{BNE}) is placed in location 1007, and the length of the branch (\texttt{FA}) is placed in location 1008. The branch if taken goes back to memory location 1003 where the \texttt{LOOP} began.

Line 70 tells the assembler where to stop. No machine code is created.

We have used five assembly language instructions in this program. These instructions consist of a \textit{mnemonic} code that is an abbreviation of its English language meaning and usually an operand. These meanings are given below.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Operand</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLC</td>
<td></td>
<td>Clear the Carry bit of the Processor Status Register (set it to zero).</td>
</tr>
<tr>
<td>LDA</td>
<td>#0</td>
<td>Load Accumulator with the immediate operand (#0)</td>
</tr>
<tr>
<td>ADC</td>
<td>#1</td>
<td>Add to accumulator with Carry the immediate operand (#1).</td>
</tr>
</tbody>
</table>
Add 1

CMP #3 = CoMPare the value in accumulator with the immediate operand (#3)

BNE LOOP = Branch on result Not Equal to the instruction with a label of LOOP

Is Accumulator = 3?

The machine language program has been assembled and is stored in the computer's memory as follows:

<table>
<thead>
<tr>
<th>Memory Location</th>
<th>Machine Code</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>18</td>
<td>Instruction CLC</td>
</tr>
<tr>
<td>1001</td>
<td>A9</td>
<td>Instruction LDA #</td>
</tr>
<tr>
<td>1002</td>
<td>00</td>
<td>Data loaded</td>
</tr>
<tr>
<td>1003</td>
<td>69</td>
<td>Instruction ADC #</td>
</tr>
<tr>
<td>1004</td>
<td>01</td>
<td>Data to be added</td>
</tr>
<tr>
<td>1005</td>
<td>C9</td>
<td>Instruction CMP #</td>
</tr>
<tr>
<td>1006</td>
<td>03</td>
<td>Data to be compared</td>
</tr>
<tr>
<td>1007</td>
<td>D0</td>
<td>Instruction BNE</td>
</tr>
<tr>
<td>1008</td>
<td>FA</td>
<td>Data telling where to branch</td>
</tr>
</tbody>
</table>

Figure 4-4. Machine Language Program Storage
EXECUTING THE MACHINE LANGUAGE PROGRAM—THE DEBUGGER

You have used the Writer/Editor Program to write the assembly language program and the Assembler Program to assemble it into machine language codes and allocate memory locations. It's now time to use the Debugger Program to execute (or run) the machine language program.

After the program was assembled, you probably noticed that the computer returned to the Writer/Editor Program and displayed the word EDIT followed on the next line by the cursor.

You enter the Debugger Program by typing: BUG

The computer responds:

The computer is now waiting for a DEBUGGER command. There are several, but we'll only look at two of them right now to avoid confusion. The first has the form:

GXXXX G for GOTO

XXXX represents the hexadecimal digits of the memory location of the beginning of our program

If you type: G1000 (and press the RETURN key), the program will execute immediately. You will see the following on the display:
What does all this mean?

1. The number 1009 tells you that the program stopped at this memory location. It executed all the instructions from location 1000 through 1008.

2. The only other value on the line that we are presently concerned with is: A=03. A stands for Accumulator. The value 03 shows that the accumulator holds a value of 3 when the program ended. This is just what we wanted. Our program put an original value of 0 in the accumulator (LDA #0). It then added a 1 each time through the loop until the accumulator contained a value of 3. Then the computer stopped.

Let's watch it count, step by step, as the program is executed. Notice that the DEBUG message was displayed at the end of the previous program execution. This means that we are still in the Debugger Program and can try out our second DEBUGGER command. We can perform a step-by-step trace of the program by typing:

T for TRACE

T1000 (and press return)
Starting memory location 1000

This is what we then see on the display:
64 GETTING STARTED WITH THE ASSEMBLER

```
1000 18  CLC  A=00 X=00 Y=00 P=32 S=00
1001 A9 00  LDA  #$00  A=00 X=00 Y=00 P=32 S=00
1003 69 01  ADC  #$01  ← One added to accumulator
       A=01 X=00 Y=00 P=30 S=00
1005 C9 03  CMP  #$03  A=01 X=00 Y=00 P=BO S=00
1007 D0 FA  BNE  $1003  A=01 X=00 Y=00 P=BO S=00
1003 69 01  ADC  #$01  ← Branch back and add 1 more
       A=02 X=00 Y=00 P=30 S=00
1005 C9 03  CMP  #$03  A=02 X=00 Y=00 P=BO S=00
1007 D0 FA  BNE  $1003  A=02 X=00 Y=00 P=BO S=00
1003 69 01  ADC  #$01  ← Branch back and add 1 more
       A=03 X=00 Y=00 P=30 S=00
1005 C9 03  CMP  #$03  A=03 X=00 Y=00 P=33 S=00  Now comparison is equal—so no branch
1007 D0 FA  BNE  $1003  A=03 X=00 Y=00 P=33 S=00
1009 00  BRK  BRK for break—program stops
       A=03 X=00 Y=00 P=33 S=00
DEBUG
```

Figure 4-5. Program Trace

Notice that the Accumulator (A) increases by one each time the ADC #$01 instruction is executed at location 1003. The value in the accumulator is shown following each instruction line as: A=nn (where nn is a two-digit hex value). The values that you see in the other registers may differ from that shown here. Do not worry about them for the time being. You should only be concerned with the accumulator at this time. Registers are covered in Chapter 5.

If you want to see how fast the computer can count to 255 (FF HEX), let's modify the compare value at line 50 to:

```
50 CMP #$FF
```

The $ denotes a hex number (when not used, a decimal value is assumed.)

With this modification, the computer will not stop when the value in the accumulator equals 3 but will go right on to FF. We could enter a whole new pro-
gram to accomplish this change, but it's easier to alter the program that we already have.

We left the computer in the Debugger Program after tracing our last program. We need to get back to the Writer/Editor Program to change line 50.

Type: X (and press RETURN)

```
1009 00           BRK
   A = 03  X = 00  Y = 00  P = 33  S = 00
DEBUG
   X
EDIT
```

To take a look at the previous Assembler Program, guess what command is used.

Type: LIST (and press RETURN)

```
10 *= $1000
20 CLC
30 LDA #0
40 LOOP ADC #1
50 CMP #3
60 BNE LOOP
70 END
```

All that you have to do to change line 50 is to type in the new line (with the line number of course).

Type: 50 CMP #$FF   (Don't forget the $ sign.)
GETTING STARTED WITH THE ASSEMBLER

To verify that the change has been made, LIST the program again.

```
70   END
50  CMP #$FF
```

LIST

```
10  *=$1000
20  CLC
30  LDA #0
40  LOOP ADC #1
50   CMP #$FF
    ← Yes, the change was made.
60  BNE LOOP
70  END
```

Another way to replace the value to be compared would be by the Editor's REPLACE statement. We could have accessed the Edit mode and typed:

```
REP/#3/#$FF/
```

Replace the old value the new value

The result would have been the same.

Even though the change has been made to the assembly language program, it has not been made to the machine language program. We must assemble the modified program.
Remember, we must use the Debugger Program to execute the program. So type: BUG

Now run the program by typing: G1000

Wow, that was quick! How do we know the computer didn’t just print the final result instead of counting all the way? Let’s slow it down by tracing the program.

Keep your eye on the column that prints the value in the accumulator (A). The results will go by pretty fast as the screen fills up and scrolls upward. It will take between 2 and 3 minutes for the accumulator to reach FF.
Type: T1000 (and press RETURN)

The screen scrolls merrily on its way. After some time it finally stops at the end of the program.

```
1007 DO FA BNE $1003
   A=FF X=00 Y=00 P=33 S=00
1009 00 BRK
   A=FF X=00 Y=00 P=33 S=00
DEBUG
```

Yes, it really did count up to FF.

It did take awhile for the computer to trace through the program. That's because it had to look up and print out all those values in the registers. It works much faster when you leave it alone and don't ask it to perform any extra tasks.

We have found out that we can increase the value in the accumulator by adding one to it. We can also subtract values from the accumulator. Why not start with a value of FF and subtract one from it each time through the loop? To do this we need two new instructions.

**SBC** (Subtract from accumulator with borrow; borrow is the opposite of Carry, but they call it SBC)

**SEC** (set the Carry bit. This will replace the Clear Carry instruction.)

We can use our old program to model the new one.

### ADD PROGRAM

<table>
<thead>
<tr>
<th>Line</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>*=1000</td>
<td>add a value</td>
</tr>
<tr>
<td>20</td>
<td>CLC</td>
<td>clear carry</td>
</tr>
<tr>
<td>30</td>
<td>LDA #0</td>
<td>start at</td>
</tr>
<tr>
<td>40</td>
<td>LOOP ADC #1</td>
<td>add and subtract</td>
</tr>
<tr>
<td>50</td>
<td>CMP #$FF</td>
<td>go until</td>
</tr>
<tr>
<td>60</td>
<td>BNE LOOP</td>
<td>jump if not</td>
</tr>
<tr>
<td>70</td>
<td>END</td>
<td>end program</td>
</tr>
</tbody>
</table>

### SUBTRACT PROGRAM

<table>
<thead>
<tr>
<th>Line</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>*=1000</td>
<td>add a value</td>
</tr>
<tr>
<td>20</td>
<td>SEC</td>
<td>set carry</td>
</tr>
<tr>
<td>30</td>
<td>LDA #$FF</td>
<td>load value</td>
</tr>
<tr>
<td>40</td>
<td>LOOP SBC #1</td>
<td>subtract</td>
</tr>
<tr>
<td>50</td>
<td>CMP #0</td>
<td>compare value</td>
</tr>
<tr>
<td>60</td>
<td>BNE LOOP</td>
<td>jump if not</td>
</tr>
<tr>
<td>70</td>
<td>END</td>
<td>end program</td>
</tr>
</tbody>
</table>

If you still have the Add Program in memory (check by entering the Writer/Editor Program and LISTing the program), you can change lines 20, 30, 40, and 50. Then assemble the new program. Otherwise, enter the complete new Subtract Program, and then assemble it.

After assembling the new program, enter the Debugger Program and execute it by typing:
You can verify the action by using the trace feature. Use the BREAK key to stop the program at a convenient point. As shown by the following display, we stopped ours shortly after it started.

Type: T1000

If you let the program execute until the accumulator counts down to zero, you'll find out that the computer can count backwards just as fast as forwards. Keep your eye on the Accumulator as the data scrolls by.

Let's take a break now to sum up what you have learned so far. Several small doses of assembly language are easier to take than a few very large ones. A more complete coverage of the Assembler features is summarized in Chapter 7.
The Assembler Cartridge consists of 3 programs:

1. **Writer/Editor**—Used to write and edit assembly language programs.

2. **Assembler**—Used to translate assembly instructions into machine code and to assign memory locations for the machine language program.

3. **Debugger**—Used to execute machine language programs.

**Key Words Used**

**Writer/Editor**

1. **EDIT**—a prompt to let you know you are in the **Writer/Editor Program**.

2. **SIZE**—Displays memory locations of Current Line Buffer, Edit Text Buffer, and your machine’s highest available memory.

3. **LIST**—Lists the assembly language program on the video screen.

4. **BUG**—A command that transfers control from the **Writer/Editor Program** to the **Debugger Program**.

5. **ASM**—A command that transfers control from the **Writer/Editor Program** to the **Assembler Program**.

**Debugger**

1. **DEBUG**—A prompt to let you know that you are in the **Debugger Program**.

2. **GXXXX**—A command which executes the current machine language program beginning at memory location XXXX (each X is a hex digit).

3. **TXXXX**—A command to trace each step of the current machine language program as it is executing from memory location XXXX (each X is again a hex digit).

4. **X**—A command that transfers control from the **Debugger Program** to the **Writer/Editor Program**.

**Instructions Used:**

**Assembly Language Instructions** *(for Source Program)*  |  **Machine Language Code Produced** *(for Object Program)*  |  **Comments**
---|---|---
LDA#0  |  A9 00  |  Load accumulator with zero—Immediate Addressing mode
ADC #1  |  69 01  |  Add 1 to the accumulator—Immediate Addressing mode
CMP #3  |  C9 03  |  Compare value in accumulator with 3—Immediate Addressing mode
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Machine Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BNE LOOP</td>
<td>D0 FA</td>
<td>Branch on result not equal to zero back to the instruction labeled LOOP—Relative Addressing mode</td>
</tr>
<tr>
<td>END</td>
<td></td>
<td>A Directive—no machine code is generated.</td>
</tr>
<tr>
<td>SBC #1</td>
<td>E9 01</td>
<td>Subtract 1 from the value in the accumulator—Immediate Addressing mode.</td>
</tr>
<tr>
<td>CLC</td>
<td>18</td>
<td>Clear the Carry bit</td>
</tr>
<tr>
<td>SEC</td>
<td>38</td>
<td>Set the Carry bit to 1</td>
</tr>
</tbody>
</table>

**EXERCISES**

1. Assembly language programs are written using the Program of the Assembler Cartridge.
2. The Program of the Assembler Cartridge is used to translate assembler language programs to machine code.
3. Machine language programs are executed from the Program of the Assembler Cartridge.
4. The Source Program is assembled into the Program.
5. The function of LDA #0 is to load the accumulator with zero. Give the function of the following assembly language instructions.
   (a) LDA #$FF
   (b) ADC #10
   (c) CMP #$1E

Exercises 6 through 10 refer to this assembled program.

```
ASM
0000 10 *= $1000
1000 18   CLC
1001 A9 01 LDA #1
1003 69 02 LOOP ADC #2
1005 C9 03 CMP #9
1007 D0 FA BNE LOOP
70 END
```

Machine language program

Assembly language program
6. The accumulator is first loaded with what number? ______________
7. What number is added each time through the loop? ______________
8. What will be the number in the accumulator when the program has passed through the loop the last time? ______________
9. If a trace of the machine language program was executed, the trace would display accumulator values that are ______________ from ____ (odd, even) through ____.
10. Lines 30 and 50 in the assembly language program could be changed so that the display of a trace would display accumulator values that were even from 0 through 8. Show the two modified assembly language lines that would make this change.
   30 ______________
   50 ______________

ANSWERS

1. Writer/Editor
2. Assembler
3. Debugger
4. Object
5. (a) Load the accumulator with FF (hex)
   (b) Add 10 to the accumulator (10 is decimal—no $ sign)
   (c) Compare the value in the accumulator with 1E (hex)
6. 1
7. 2
8. 9
9. odd, 1, 9
10. 30 LDA #0
    50 CMP #8
The 6502 microprocessor used in the Atari 400/800 computers has several special-purpose registers. These registers hold 8 bits (1 byte) of data just like memory locations but are used for special purposes. You have seen displays of the data contained in these registers in the trace of the program in Chapter 4.

Example:

```
1004 C9 03 CMP #03
```

A=03 X=00 Y=00 P=33 S=00

Accumulator X register Y register Processor Status Register Stack Pointer Register

We will discuss these special registers in the order listed below.

1. The Accumulator—Most operations on data are performed here. You can tell from the short program in Chapter 4 that the accumulator is a very busy place.

2. The X register—This register is used as a scratch pad or as an index in certain addressing modes.

3. The Y register—This register is used in the same way as the X register.

4. The Processor Status Register—This register contains a record of the microprocessor's status as each instruction is executed. Each bit of this register holds one item of status information.

5. The Stack Pointer Register—The data in this register is the memory location of the top of the stack. The stack is a special block of memory at addresses 01FF down through 0100 (hex).
THE ACCUMULATOR

The programs in Chapters 3 and 4 showed how the accumulator can be loaded with a number in the Immediate Addressing mode. One program also added a number to this value (using the Immediate Addressing mode) and then compared the result with the value 3 (again in the Immediate Addressing mode). All of those operations were performed using only the accumulator and data held in certain memory locations of the program.

A branch instruction (BNE in the Relative Addressing mode) was also used in the program. To determine whether to take the branch or not, the computer examined the Processor Status Register. It looked at one bit (the zero bit) of the Processor Status register.

THE X AND Y REGISTERS

Let's now develop a short program to demonstrate some action taking place in the X register. It will be very similar to the programs used in Chapters 3 and 4.

1. LDX #0 Load the X register (LDX) with zero using the Immediate Addressing mode (#0).
2. INX Increment the X register (INX) using the Implied Addressing mode (no operand used).
3. CPX #3 Compare the value in the X register (CPX) with 3 using the Immediate Addressing mode (#3).
4. BNE LOOP Branch if the result (X register—3) is not equal to zero (BNE) back to the instruction that you have labeled LOOP (INX in this program).

The program is written using the Writer/Editor Program of the Assembler cartridge as before. When you are in the Edit mode, type NEW to erase any old program that may still be in the Edit Buffer. This does not erase the memory contents (the previously assembled machine language program). Enter the program as shown.

```
EDIT
10  *=1000
20  LDX #0
30  LOOP INX
40  CPX #3
50  BNE LOOP
60  END
```

← You enter this.
Then you use the **Assembler Program** to assemble the machine language program.

- **ASM**
  - **0000** 10 * = $1000
  - **1000 A200** 20 LDX #0
  - **1002 E8** 30 LOOP INX
  - **1003 E003** 40 CPX #3
  - **1005 DOFB** 50 BNE LOOP

**END**

Now enter the **Debugger Program** and execute the run.

- **DEBUG**
  - **G1000**
  - **1007**

**A=00 X=03 Y=00 P=33 S=00**

You type these.

Yes, the X register ends with the value 3.
This time, trace the program step by step. Type: T1000

```
1000 A2 00      LDX #$00
    A=00 X=00 Y=00 P=32 S=00
1002 E8         INX
    A=00 X=01 Y=00 P=30 S=00
1003 E0 03      CPX #$03
    A=00 X=01 Y=00 P=B0 S=00
1005 D0 FB      BNE $1002
    A=00 X=01 Y=00 P=B0 S=00
1002 E8         INX
    A=00 X=02 Y=00 P=30 S=00
1003 E0 03      CPX #$03
    A=00 X=02 Y=00 P=B0 S=00
1005 D0 FB      BNE $1002
    A=00 X=02 Y=00 P=B0 S=00
1002 E8         INX
    A=00 X=03 Y=00 P=30 S=00
1003 E0 03      CPX #$03
    A=00 X=03 Y=00 P=33 S=00
1005 D0 FB      BNE $1002
    A=00 X=03 Y=00 P=33 S=00
1007 00         BRK
    A=00 X=03 Y=00 P=33 S=00
```

Notice that the value in the X register increases each time that the computer executes INX at location 1002. Notice also that the program stops at location 1007 with the instruction BRK (BREAK).

The computer uses the BREAK instruction to stop itself at the end of a program. We'll discuss the BRK instruction more thoroughly in Chapter 9.

Next, let's modify the program to count in both the accumulator and the X register. We'll use one new instruction:

TAX

No, it's not really tax time. This is just a mnemonic code that stands for: Transfer Accumulator to X register. When this instruction is executed, the data
currently in the accumulator is transferred to (or copied into) the X register. The data also stays in the accumulator.

![Diagram of Accumulator and X Register]

Copy accumulator into X register (TAX)

Type: X and press the RETURN key to get back to the Writer/Editor mode. Now you can erase the old program by typing:

NEW

Wow! This is just like BASIC. The old program is erased and you are ready to write the new program.

PROGRAM TO COUNT IN ACCUMULATOR
AND IN THE X REGISTER

<table>
<thead>
<tr>
<th>EDIT</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>*=1000</td>
</tr>
<tr>
<td>20</td>
<td>CLC</td>
</tr>
<tr>
<td>30</td>
<td>LDA #0</td>
</tr>
<tr>
<td>40</td>
<td>TAX</td>
</tr>
<tr>
<td>50</td>
<td>LOOP ADC #1</td>
</tr>
<tr>
<td>60</td>
<td>INX</td>
</tr>
<tr>
<td>70</td>
<td>CPX #3</td>
</tr>
<tr>
<td>80</td>
<td>BNE LOOP</td>
</tr>
<tr>
<td>90</td>
<td>END</td>
</tr>
</tbody>
</table>

New Instruction, Transfer content of Accumulator to the X register
Add 1 to accumulator
Increment X register
Then assemble it:

```
80  BNE LOOP
90  END
```

```
ASM
0000  10  *=  $1000
1000  18  20  CLC
1001  A900  30  LDA #0
1003  AA  40  TAX
1004  6901  50 LOOP  ADC #1
1006  E8  60  INX
1007  E003  70  CPX #3
1009  D0F9  80  BNE LOOP
```

90  END

Now enter the Debugger Program.

```
DEBUG
```

To see what happens as a program is executed, the trace feature of the assembler is a tremendous aid. Let's use it again to watch our program in action. Type: T1000 and press the RETURN key.
What happened? The first part of the program rolled by so fast that we couldn't read it. The video screen can only hold 24 lines of information at one time. Now what?

Looking in the Atari Assembler Manual, I see a DEBUGGER command:

SXXXX Single-step operation

Let's try that. The instructions in the manual say to type in $S$ followed by the address of the first instruction. Then type $S$ and press the RETURN key. Do this last operation repeatedly to see each step. Here goes.

The value 3 is left over from the last program. NEW does not change the machine language memory or registers only the Text Edit Buffer.
The computer stops here, but it's only waiting for you to type an $S$ and press RETURN. So...

```
<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>A</th>
<th>X</th>
<th>Y</th>
<th>P</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>1001</td>
<td>LDA #$00</td>
<td>00</td>
<td>03</td>
<td>00</td>
<td>32</td>
<td>00</td>
</tr>
<tr>
<td></td>
<td>DEBUG</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1003</td>
<td>TAX</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>32</td>
<td>00</td>
</tr>
<tr>
<td></td>
<td>DEBUG</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1004</td>
<td>ADC #$01</td>
<td>01</td>
<td>00</td>
<td>00</td>
<td>30</td>
<td>00</td>
</tr>
<tr>
<td></td>
<td>DEBUG</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1006</td>
<td>INX</td>
<td>01</td>
<td>01</td>
<td>00</td>
<td>30</td>
<td>00</td>
</tr>
<tr>
<td></td>
<td>DEBUG</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1007</td>
<td>CPX #$03</td>
<td>01</td>
<td>01</td>
<td>00</td>
<td>30</td>
<td>00</td>
</tr>
<tr>
<td></td>
<td>DEBUG</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1009</td>
<td>BNE $1004</td>
<td>01</td>
<td>01</td>
<td>00</td>
<td>30</td>
<td>00</td>
</tr>
<tr>
<td></td>
<td>DEBUG</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1004</td>
<td>ADC #$01</td>
<td>02</td>
<td>01</td>
<td>00</td>
<td>30</td>
<td>00</td>
</tr>
<tr>
<td></td>
<td>DEBUG</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1006</td>
<td>INX</td>
<td>02</td>
<td>02</td>
<td>00</td>
<td>30</td>
<td>00</td>
</tr>
</tbody>
</table>
```

- Accumulator cleared to zero
- X register cleared to zero
- One added to accumulator
- X register incremented
- One added again to accumulator
- X register incremented again
this continues until X=03

1007 E0 03 CPX #$03 ← Is X=3?
A=03 X=03 Y=00 P=33 S=00

DEBUG
S
1009 D0 F9 BNE $1003 ← Do not branch this time
A=03 X=03 Y=00 P=33 S=00

DEBUG
S
100B 00 BRK
A=03 X=03 Y=00 P=33 S=00

DEBUG

We have now used the A (accumulator) and the X registers. Since the Y register is used in the same way as the X register, we will pass it over for the time being. You no doubt noticed that the value in the Y register did not change in each of the programs that we used.

Before we leave this program, let's go back to the Writer/Editor Program. You do this by typing an X and pressing the RETURN key.

DEBUG
X

EDIT

Remember, you can list assembly language programs in the assembler's Writer/Editor Program just like you LIST in BASIC.
This feature of the Writer/Editor Program can be very helpful when editing a program to make changes due to programming errors or to make modifications for any other reason.

THE PROCESSOR STATUS REGISTER

This register holds seven bits of information about the status of the microprocessor. The eighth bit of the register is not used. The individual information bits are called flags. These flags are always in one of two conditions: SET to one (1) or RESET to zero (0). The flags are called: Carry, Zero result, Interrupt disable, Decimal mode, Break command, Overflow, and Negative result. Here is the position that each flag occupies in the register.

<table>
<thead>
<tr>
<th>Bit number</th>
<th>B_7</th>
<th>B_6</th>
<th>B_5</th>
<th>B_4</th>
<th>B_3</th>
<th>B_2</th>
<th>B_1</th>
<th>B_0</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>V</td>
<td>B</td>
<td>D</td>
<td>I</td>
<td>Z</td>
<td>C</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 5-1. Processor Status Register

where:  
N = Negative result  
V = Overflow  
B = Break command  
D = Decimal mode  
I = Interrupt disable  
Z = Zero result  
C = Carry
B₀, Carry bit  This bit is modified as a result of certain arithmetic and logic operations. It can also be SET or RESET by programmed instruction. We will be using it later.

B₁, Zero bit  This flag is automatically SET when any data movement or arithmetic operation has a result equal to zero. It was tested by the BNE (Branch on result not equal) in the programs of Chapter 4 and Chapter 5. See page 59 for an example.

B₂, Interrupt disable bit  This flag controls the effect of the interrupt request pin of the microprocessor. We will not be concerned with this bit for now.

B₃, Decimal mode bit  This flag controls whether or not the addition and subtraction operations are performed by the computer as binary or decimal. More on this later.

B₄, Break command bit  This bit is SET only by the microprocessor. It is used during interrupts. It appeared in our display at the end of the programs when we had executed a trace or single-step execution.

B₅, Expansion bit  This bit is not used at present. It is reserved for future expansion of the 6502 microprocessor.

B₆, Overflow bit  This bit is used to indicate that an overflow has occurred in a signed binary arithmetic operation. More on this later when we get to signed number arithmetic operations.

B₇, Negative bit  This bit tells whether the result of some arithmetic operation is negative or not. We'll discuss this in detail when we get to signed number arithmetic operations.

It may seem like we are putting you off on many of the flags in the Processor Status Register. We are! We'll introduce each of the flags as they are used by our programs. The first one that you have encountered is the Zero flag.

If you look back to the trace of the program in Figure 4-5, the first and second execution of the CMP #3 instruction at location 1005 shows:

```
First execution → 1005 C9 03   CMP #$03
          A=01 X=00 Y=00 P=B0 S=00
```

```
Second execution → 1005 C9 03   CMP #$03
          A=02 X=00 Y=00 P=B0 S=00
```

Notice that the Processor Status Register (P) holds the value B₀.
This means that when the value in the accumulator was compared to 3, the result (1-3 or 2-3) was not equal to zero.

Therefore, the BNE (Branch on result not equal to zero) was taken. However, when this same instruction was executed the third time, the display showed:

Third execution → 1005 C9 03
A=03 X=00 Y=00 P=33 S=00

P has changed

The Processor Status Register now holds 33.

The value in the accumulator (03) is equal to 3. The result (03-03) does equal zero.

The branch (BNE) is not taken, and the execution of the program stops. You can see that by looking at certain bits in the Processor Status Register, the computer can make decisions (such as the order in which instructions are executed) depending on certain conditions. This type of decision making adds great power to the capability of both the computer and the programmer.

Some instructions affect the status bits of the Processor Status Register and others do not. Here is a table of the instructions we have covered so far and the flags that they affect in the Processor Status Register.
THE PROCESSOR STATUS REGISTER

Instruction | Status Bits Affected
--- | ---
PLA | Z,C
LDA | N,Z
ADC | N,V,Z,C
STA | none
RTS | none
CLC | C
LDX | N,Z
TXA | N,Z
INX | N,Z
CPX | N,Z,C
BNE | none
CMP | N,Z,C
SBC | N,V,Z,C
TAX | N,Z
SEC | C

Figure 5-2. Effect of Instructions on Flag Bits

The following table shows the status flags that are affected by each 6502 instruction. X means that the flag is affected. The result will depend upon the condition, or status, resulting from the operation performed by the instruction. A one indicates that the flag is set. A zero indicates that a flag is reset.

<table>
<thead>
<tr>
<th>Mnemonic Code</th>
<th>Operation Performed</th>
<th>Status Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>N V B D I Z C</td>
</tr>
<tr>
<td>ADC</td>
<td>Add memory to accumulator with carry</td>
<td>X X X X X X</td>
</tr>
<tr>
<td>AND</td>
<td>AND memory with accumulator</td>
<td>X X</td>
</tr>
<tr>
<td>ASL</td>
<td>Shift left one bit (memory or accum.)</td>
<td>X X</td>
</tr>
<tr>
<td>BCC</td>
<td>Branch on carry clear (If C=0)</td>
<td>X X</td>
</tr>
<tr>
<td>BCS</td>
<td>Branch on carry set (If C=1)</td>
<td>X X</td>
</tr>
<tr>
<td>BEQ</td>
<td>Branch on result zero (If Z=1)</td>
<td>X X</td>
</tr>
<tr>
<td>BIT</td>
<td>Test bits in accumulator with memory</td>
<td>X X</td>
</tr>
<tr>
<td>BMI</td>
<td>Branch on result minus (If N=1)</td>
<td>X X</td>
</tr>
<tr>
<td>BNE</td>
<td>Branch on result not zero (If Z=0)</td>
<td>X X</td>
</tr>
<tr>
<td>BPL</td>
<td>Branch on result plus (If N=0)</td>
<td>X X</td>
</tr>
<tr>
<td>BRK</td>
<td>Force Break</td>
<td>X X</td>
</tr>
<tr>
<td>BVC</td>
<td>Branch on overflow clear (If V=0)</td>
<td>X X</td>
</tr>
<tr>
<td>BVS</td>
<td>Branch on overflow set (If V=1)</td>
<td>X X</td>
</tr>
<tr>
<td>CLC</td>
<td>Clear carry flag</td>
<td>X X</td>
</tr>
<tr>
<td>CLD</td>
<td>Clear decimal mode</td>
<td>X X</td>
</tr>
<tr>
<td>Mnemonic</td>
<td>Operation Performed</td>
<td>Status Flags</td>
</tr>
<tr>
<td>----------</td>
<td>---------------------</td>
<td>--------------</td>
</tr>
<tr>
<td>Code</td>
<td></td>
<td>N</td>
</tr>
<tr>
<td>CLI</td>
<td>Clear interrupt disable flag</td>
<td>0</td>
</tr>
<tr>
<td>CLV</td>
<td>Clear overflow flag</td>
<td>X</td>
</tr>
<tr>
<td>CMP</td>
<td>Compare memory and accumulator</td>
<td>X</td>
</tr>
<tr>
<td>CPX</td>
<td>Compare memory and index X</td>
<td>X</td>
</tr>
<tr>
<td>CPY</td>
<td>Compare memory and index Y</td>
<td>X</td>
</tr>
<tr>
<td>DEC</td>
<td>Decrement memory by one</td>
<td>X</td>
</tr>
<tr>
<td>DEX</td>
<td>Decrement index X by one</td>
<td>X</td>
</tr>
<tr>
<td>DEY</td>
<td>Decrement index Y by one</td>
<td>X</td>
</tr>
<tr>
<td>EOR</td>
<td>Exclusive OR memory with accumulator</td>
<td>X</td>
</tr>
<tr>
<td>INC</td>
<td>Increment memory by one</td>
<td>X</td>
</tr>
<tr>
<td>INX</td>
<td>Increment index X by one</td>
<td>X</td>
</tr>
<tr>
<td>INY</td>
<td>Increment index Y by one</td>
<td>X</td>
</tr>
<tr>
<td>JMP</td>
<td>Jump to new location</td>
<td>X</td>
</tr>
<tr>
<td>JSR</td>
<td>Jump to new location save rtn. add.</td>
<td>X</td>
</tr>
<tr>
<td>LDA</td>
<td>Load accumulator from memory</td>
<td>X</td>
</tr>
<tr>
<td>LDX</td>
<td>Load index X from memory</td>
<td>X</td>
</tr>
<tr>
<td>LDY</td>
<td>Load index Y from memory</td>
<td>X</td>
</tr>
<tr>
<td>LSR</td>
<td>Shift right one bit (memory or accum.)</td>
<td>0</td>
</tr>
<tr>
<td>NOP</td>
<td>No operation</td>
<td>X</td>
</tr>
<tr>
<td>ORA</td>
<td>OR memory with accumulator</td>
<td>X</td>
</tr>
<tr>
<td>PHA</td>
<td>Push accumulator on stack</td>
<td>X</td>
</tr>
<tr>
<td>PHP</td>
<td>Push processor status on stack</td>
<td>X</td>
</tr>
<tr>
<td>PLA</td>
<td>Pull accumulator from stack</td>
<td>X</td>
</tr>
<tr>
<td>PLP</td>
<td>Pull processor status from stack</td>
<td>X</td>
</tr>
<tr>
<td>ROL</td>
<td>Rotate one bit left (mem. or accum.)</td>
<td>X</td>
</tr>
<tr>
<td>ROR</td>
<td>Rotate one bit right (mem. or accum.)</td>
<td>X</td>
</tr>
<tr>
<td>RTI</td>
<td>Return from interrupt</td>
<td>X</td>
</tr>
<tr>
<td>RTS</td>
<td>Return from subroutine</td>
<td>X</td>
</tr>
<tr>
<td>SBC</td>
<td>Subtract memory and borrow from accum.</td>
<td>X</td>
</tr>
<tr>
<td>SEC</td>
<td>Set carry flag</td>
<td>X</td>
</tr>
<tr>
<td>SED</td>
<td>Set decimal mode</td>
<td></td>
</tr>
<tr>
<td>SEI</td>
<td>Set interrupt disable flag</td>
<td></td>
</tr>
<tr>
<td>STA</td>
<td>Store accumulator in memory</td>
<td>X</td>
</tr>
<tr>
<td>STX</td>
<td>Store index X in memory</td>
<td>X</td>
</tr>
<tr>
<td>STY</td>
<td>Store index Y in memory</td>
<td>X</td>
</tr>
<tr>
<td>TAX</td>
<td>Transfer accumulator to index X</td>
<td>X</td>
</tr>
<tr>
<td>TAY</td>
<td>Transfer accumulator to index Y</td>
<td>X</td>
</tr>
<tr>
<td>TSX</td>
<td>Transfer stack pointer to index X</td>
<td>X</td>
</tr>
<tr>
<td>TXA</td>
<td>Transfer index X to accumulator</td>
<td>X</td>
</tr>
<tr>
<td>TXS</td>
<td>Transfer index X to stack pointer</td>
<td>X</td>
</tr>
<tr>
<td>TYA</td>
<td>Transfer index Y to accumulator</td>
<td>X</td>
</tr>
</tbody>
</table>
Flag abbreviations are:

- N  Negative result flag
- V  Overflow flag
- Expansion flag (not labeled)
- B  Break command flag
- D  Decimal mode flag
- I  Interrupt disable flag
- Z  Zero result flag
- C  Carry flag

The Flags are up!

THE STACK POINTER REGISTER

Temporary storage is sometimes required to save current values of a register while it is being used for a second task. When that task is finished, the original value is replaced in the register. The microprocessor uses an area of RAM called a stack for this purpose. Values that are saved in this area are arranged like a stack of cards with each card having a value. Each value saved is placed on the top of the stack. When the values are retrieved, they are taken from the top of the stack also. The last value put on the stack is always the first value taken off.

The Stack Pointer Register is a 16-bit register that keeps track of the memory address of the top of the stack.

Example:
Suppose the top of the stack is originally at address 01FF. If two data values are placed on the stack, the Stack Pointer Register would then contain the value 01FD.

<table>
<thead>
<tr>
<th>Stack Pointer</th>
<th>originally</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>01FF</td>
</tr>
</tbody>
</table>

| 01FD | ? | ← Next value here |
| 01FE | 65 | ← 2nd value placed on |
| 01FF | 10 | ← 1st value placed on |
Stack Pointer

now

[01FD]

the next location where data is to be placed

The process is reversed when data is taken off the stack. The stack pointer would be automatically incremented to point at 01FE and the value 65 returned. When ready to remove the value 03, the stack pointer would again be incremented and the 03 returned. The Stack Pointer Register would then contain 01FF again.

There are four machine language instructions that the programmer may use to move data on or off the stack. Data may be saved and retrieved from the Accumulator or the Processor Status Register.

1. PHA (Push Accumulator on stack)
   Op Code = 48
   Implied mode

2. PHP (Push Processor status on stack)
   Op Code = 08
   Implied mode

3. PLA (Pull Accumulator from stack)
   Op Code = 68
   Implied mode

4. PLP (Pull Processor status from stack)
   Op Code = 28
   Implied mode

ADDRESSING MODES

There are several different addressing modes used by the 6502 microprocessor. Some instructions are only used in one addressing mode. Others may be used in more than one mode. So far, we have only used three modes: the Immediate, the Implied, and the Relative.

The addressing modes may be classified into two types: Indexed and Non-Indexed. Let's first consider the Non-Indexed Addressing modes. They are the simplest to use and understand.

The Implied Addressing Mode

Instructions using Implied Addressing are one byte long. That byte contains the Op Code, which designates an operation that is internal to the microprocessor, and no operand is involved. Examples that we have used in this chapter are INX and TAX.
ADDRESSING MODES

TAX (Transfer Accumulator to X register)
Op Code AA
Status flags effected: N and Z

The Op Code totally defines the operation of Implied Addressing Instructions. Therefore, only one byte is needed to describe the operation. Instructions used in this mode are not used in any other mode.

The Immediate Addressing Mode

Instructions using the Immediate Addressing mode employ two bytes to describe the operation. The first byte contains the Op Code specifying the operation and the addressing mode. The second byte contains a constant value known at the time a program is being written. Putting these values directly in the program saves the programmer from loading them into memory and retrieving them when needed.

You have used the following Immediate Addressing mode instructions in Chapters 3, 4, and 5:

CPX (Compare to X register)
Op Code EO (If Immediate Addressing)
Second byte OA (Value X is compared to)
Status flags affected: N,Z and C

LDX (Load the X register)
Op Code A2 (If Immediate Addressing)
Second byte 00 (Value loaded in X)
Status flags affected: Z and C

LDA (Load the Accumulator)
Op Code A9 (If Immediate Addressing)
Second byte 00 (Value loaded)
Status flags affected: N and Z

ADC (Add to Accumulator with carry)
Op Code 69 (If Immediate Addressing)
Second byte 01 (Value added)
Status flags affected: N,Z,C and V

CMP (Compare to Accumulator)
Op Code C9 (If Immediate Addressing)
Second byte 03 (Value accumulator is compared to)
Status flags affected: N,Z and C

Immediate addressing is the simplest way to manipulate constants. All of the instructions used in this mode may also be used in other modes. The Op Code of the instruction will be different for each mode that it is used in so that the computer will know which mode is desired.
The Relative Addressing Mode

All of the branch instructions use this, and only this, mode. They are two-byte instructions. The Op Code occupies the first byte. The second byte contains a signed number that specifies the length of the branch (if the branch is taken). The second byte is ignored if the conditions existing do not require the branch to be taken. At the time the decision (to branch or not) is taken, the program counter is pointing at the next instruction. You have used one instruction in the Relative Addressing Mode, BNE.

\[
\begin{array}{ll}
\text{page 60} & \text{BNE (Branch on result not equal to zero)} \\
\text{Op Code} & \text{DO} \\
\text{Second byte} & \text{FA (equivalent to -6: back 6 locations)} \\
& \text{Status flags affected: none}
\end{array}
\]

Branches are made, or not made, depending on the status of certain flags of the Processor Status Register (N,V,Z or C). Only branch instructions use this mode.

The Absolute Addressing Mode

Absolute Addressing Instructions contain three bytes: one for the Op Code and two for the address operand. The low-order address byte occupies the second byte, and the high-order address byte occupies the third byte. The programmer can therefore specify a full 16-bit address to access any memory location. It is considered to be the normal mode for addressing. You have used this mode in Chapter 2 with a STORE instruction (page 21). This mode will be used again in the next chapter.

Example:

\[
\begin{array}{ll}
\text{LDA (Load Accumulator)} \\
\text{Op Code AD (If Absolute Addressing)} \\
\text{Second byte F3 (low-order address)} \\
\text{Third byte 10 (high-order address)} \\
& \text{Status flags affected: N and Z}
\end{array}
\]

This example would load the value that is contained in memory location 10F3 into the accumulator. The value held in 10F3 remains the same.

The Zero Page Addressing Mode

Zero Page instructions are two bytes long. The first byte contains the Op Code, and the second byte contains the low-order byte of the address operand. The high-order address byte is assumed by the microprocessor to be zero (therefore, its name). The zero page of memory runs from 0000 through 00FF. Each
consecutive block of 256 locations is called a page of memory. The advantage of zero page instructions is the saving of one byte in the instruction. The time necessary to execute the instruction is therefore shorter than that of Absolute Addressing. The user should organize his use of memory so that the most frequently accessed memory is in the zero page. This cannot always be done because manufacturers of computers quite often use this memory in their operating system. We have not used any instructions in this mode yet.

Example:

LDA (Load the Accumulator)
Op Code  A5 (If Zero Page Addressing)
Second byte  80 (low-order address byte)
Status flags affected: N and Z

This example would load the accumulator with the contents of memory location 0080.

We will hold off a discussion of Indexed Addressing modes until Chapter 8, where they are first used.
<table>
<thead>
<tr>
<th>Mnemonic Code</th>
<th>Op Codes</th>
<th>Accumulator</th>
<th>Immediate</th>
<th>Zero Page</th>
<th>Zero Page, X</th>
<th>Zero Page, Y</th>
<th>Absolute</th>
<th>Absolute, X</th>
<th>Absolute, Y</th>
<th>Implied</th>
<th>Relative</th>
<th>Indexed Indirect</th>
<th>Indexed Indexed</th>
<th>Indirect</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMP</td>
<td></td>
<td>C9 C5 D5</td>
<td>D9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPX</td>
<td></td>
<td>E0 E4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPY</td>
<td></td>
<td>C0 C4</td>
<td>C6 D6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DEC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DEX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DEY</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EOR</td>
<td></td>
<td>49 45 55</td>
<td>59</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INY</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JMP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>6C</td>
</tr>
<tr>
<td>JSR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDA</td>
<td></td>
<td>A9 A5 B5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>A1 B1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDX</td>
<td></td>
<td>A2 A6</td>
<td>B6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDY</td>
<td></td>
<td>A0 A4 B4</td>
<td>AC BC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LSR</td>
<td></td>
<td>4A 46 56</td>
<td>5E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NOP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ORA</td>
<td></td>
<td>09 05 15</td>
<td>19</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>48</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PHA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PHP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PLA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PLP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ROL</td>
<td></td>
<td>2A 26 36</td>
<td>3E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ROR</td>
<td></td>
<td>6A 66 76</td>
<td>7E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RTI</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RTS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SBC</td>
<td></td>
<td>E9 E5 F5</td>
<td>ED FD F9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>E1 F1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SEC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SED</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>38</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SEI</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>STA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>81 91</td>
</tr>
<tr>
<td>STX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>STY</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TAX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TAY</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TSX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TXA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TXS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TYA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
SUMMARY

The flexibility of the 6502 microprocessor used in the Atari 400/800 computers is achieved through a variety of addressing modes and special-purpose registers.

The registers discussed in this chapter include:

- Accumulator (A register)—Most operations performed on data take place in this register.
- X Index Register—Used as a scratch pad or as an index in some addressing modes.
- Y Index Register—same function as X Index Register.
- Processor Status Register (P register)—Contains the microprocessor's status as each instruction is executed.
- Stack Pointer Register (S register)—Contains the memory location of the top of the stack.

The addressing modes discussed include:

- Implied—One-byte instructions that need no operand.
- Immediate—The first byte specifies the operation; the operand (second byte) is a one-byte data item.
- Relative—Used for branch instructions; first byte specifies the kind of branch; the one byte operand tells how far and in what direction to branch.
- Absolute—A three-byte instruction; first byte specifies the operation; the other two bytes give the operand that consists of a memory address to be used.
- Zero Page—Similar to Absolute Addressing, but only a single-byte operand is necessary for the memory address to be used.

EXERCISES

1. How many bits of information can be contained in the accumulator? _____ bits
2. Name the two index registers. 
   ____________ and ____________
3. The status flags are contained in which register?
   ______________  ____________________ register
4. The register into which data is loaded, stored from, and in which arithmetic operations are performed is called: ________________________________
Question 5 through 8 refer to the following program:

```
10  *= $1050
20  LDX #0
30  LOOP INX
40  CPX #2
50  BNE LOOP
60  END
```

5. Which index register is used in the program? 
6. Which status flag is used to determine if a branch is taken? 
7. What values will appear in the index register as the program is executed? 
8. What will be the starting address of the machine language program when it is assembled? 
9. The assembler cartridge's Writer/Editor command to display the assembly language program on the video screen is: 

   (NEW, LIST, RUN)

10. The bits of the Processor Status Register are labeled as shown:

```
   | N | V | B | D | I | Z | C |
```

   (a) Bit B_1 contains the _______ flag.
   (b) Bit B_7 contains the _______ flag.
   (c) The Carry flag is contained in bit _______.

11. When using the stack, the Stack Pointer Register contains the memory address of the top of the stack. The last item placed on the stack will be the _______ item removed from the stack.

   (first, last)

12. How many bytes are used by an Implied Address instruction? 

13. What addressing mode is used by Branch instructions? 

14. Which instruction is shorter (fewer bytes):

   Absolute or Zero Page? 

**ANSWERS**

1. 8 bits 
2. X and Y 
3. Processor Status Register 
4. The accumulator (or A register) 
5. X
6. Zero (or Zero flag)
7. 0,1,2 (or 00000000,00000001,00000010 binary)
8. 1050 (hex)
9. LIST
10. (a) Z (or Zero flag)
    (b) N (or Negative flag)
    (c) B₀
11. First
12. 1 (one)
13. Relative
14. Zero Page
In Chapters 3, 4, and 5 we have used a branch instruction (BNE) to perform a program loop. Branch instructions are two-byte instructions which use the Relative Addressing mode. That is, the address to which the branch is to be taken is calculated relative to the current position of the program counter. The program counter (see page 000) is used as a pointer to designate where the microprocessor will obtain the instruction that is to be executed following the instruction that is currently being executed. It is always one instruction ahead of the computer execution.

Example:

<table>
<thead>
<tr>
<th>Memory</th>
<th>Op Code</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1008</td>
<td>D0</td>
<td>BNE</td>
</tr>
<tr>
<td>1009</td>
<td>F9</td>
<td></td>
</tr>
<tr>
<td>100A</td>
<td>00</td>
<td>BRK</td>
</tr>
</tbody>
</table>

Therefore, the branch is taken relative to the program counter (or one memory location beyond the second byte of the branch instruction). The operand (second byte—F9 in this example) used with the branch instruction tells how far and in what direction the branch should be taken.

When used as the operand in a branch instruction such as BNE, all hex values from 01 through 7F cause a branch forward from the current position of the program counter. The following instruction would cause a branch forward from memory location 100A (where the program counter points as the BNE instruction is executed) to hexadecimal memory location 1012 (100A+8).

```
1008  D0  BNE  08
1009  08
```
An example of the above as used in a section of a program follows:

```
1006 E0 CPX 03  Compare the value of the
1007 03 X register with 03 hex.
```

```
1008 D0 BNE 08 Branch, if Y is not equal to
1009 08 03, forward 8 steps.
```

Program counter starts here

```
100A
100B
100C
100D
100E
100F
1010
1011
1012
1013
1014
```

Branch forward 8 steps if X not=03

All hex values from 80 through FF are used by branch instructions as backward (or negative) branches. In the “Program to Count in Accumulator” (Chapter 5) the instruction used is:

```
1008 D0 BNE LOOP Branch, if X is not equal to 03, backward
1009 F9 7 steps.
```

The branch is made backward (or in the negative direction) since F9 is between 80 and FF. Counting back 7 steps from location 100A puts the branch destination at 1003, the start of the loop.

```
PROGRAM TO COUNT IN ACCUMULATOR
1000 A9 LDA #0
1001 00
1002 AA TAX
1003 69 ADC #1
1004 01
1005 E8 INX
1006 E0 CPX #3
1007 03
1008 D0 BNE
1009 F9
100A 00 BRK
```

Branch back 7

Program Counter here when BNE is executed
We will not go into the method used by the computer to determine the values of negative numbers. Instead, we will provide tables to determine the operand used with the branch. The assembler takes care of these details if you provide it with appropriate labels.

Remember that forward branches will use an operand in the range of 1 through 7F, and backward branches will use an operand in the range of 80 through FF.

EXAMPLES USING FORWARD BRANCHES

1. 100E DO BNE 07  
   100F 07  
   1010  
   1017

Program counter starts at 1010  
Branch desired to 1017 (7 steps)

Look up in table:
Steps Branch
Forward Operand
(Decimal) (Hex)
7 07 ← Operand

If condition tested is not equal to zero,  
branch forward to 1017 (1010+7 steps).

2. 1010 DO BNE 1F  
   1009 1F  
   1010  
   102F

Program counter starts at 1010  
Branch desired to 102F (31 decimal steps)

Look up in table:
Steps Branch
Forward Operand
(Decimal) (Hex)
31 1F ← Operand

If condition tested is not equal to zero,  
branch forward to 102F (1010+1F hex steps).

3. 100E DO BNE 77  
   100F 77  
   1010  
   1087

Program counter starts at 1010  
Branch desired to 1087 (119 decimal steps)

Look up in table:
Steps Branch
Forward Operand
(Decimal) (Hex)
119 77 ← Operand

If condition tested is not equal to zero,  
branch forward to 1087 (1010+77 hex steps).
4. 100E D0 BNE 5B  Program counter starts at 1010
    100F 5B  Branch desired to 106B (91 decimal steps)
    1010  .  Look up in table:
    .  .  Steps  Branch
    106B  .  Forward  Operand
            (Decimal)  (Hex)
            91  5B  ← Operand

If condition tested is not equal to zero,
branch forward to 106B (1010+5B hex steps).

EXAMPLES USING BACKWARD BRANCHES

1. 100B  .  Program counter starts at 1012
    .  Branch desired to 100B (-7 steps)
    1010 D0 BNE F9  Look up in table:
    1011 F9  Backward  Operand
    1012  .  (Decimal)  (Hex)
            7  F9  ← Operand

If condition tested is not equal to zero,
branch backward to 100B (1012-7 steps).

2. OFF3  .  Program counter starts at 1012
    .  Branch desired to FF3 (-31 decimal steps)
    1010 D0 BNE E1  Look up in table:
    1011 E1  Backward  Operand
    1012  .  (Decimal)  (Hex)
            31  E1  ← Operand

If condition tested is not equal to zero,
branch backward to FF3 (1012-1F hex steps).

3. OF9B  .  Program counter starts at 1012
    .  Branch desired to F9B (-119 decimal steps)
    1010 D0 BNE 89  Look up in table:
    1011 89  Backward  Operand
    1012  .  (Decimal)  (Hex)
            119  89  ← Operand

If condition tested is not equal to zero,
branch backward to F9B (1012-77 hex steps).
<table>
<thead>
<tr>
<th>Steps Forward (Decimal)</th>
<th>Branch Operand (Hex)</th>
<th>Steps Forward (Decimal)</th>
<th>Branch Operand (Hex)</th>
<th>Steps Forward (Decimal)</th>
<th>Branch Operand (Hex)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>01</td>
<td>49</td>
<td>31</td>
<td>97</td>
<td>61</td>
</tr>
<tr>
<td>2</td>
<td>02</td>
<td>50</td>
<td>32</td>
<td>98</td>
<td>62</td>
</tr>
<tr>
<td>3</td>
<td>03</td>
<td>51</td>
<td>33</td>
<td>99</td>
<td>63</td>
</tr>
<tr>
<td>4</td>
<td>04</td>
<td>52</td>
<td>34</td>
<td>100</td>
<td>64</td>
</tr>
<tr>
<td>5</td>
<td>05</td>
<td>53</td>
<td>35</td>
<td>101</td>
<td>65</td>
</tr>
<tr>
<td>6</td>
<td>06</td>
<td>54</td>
<td>36</td>
<td>102</td>
<td>66</td>
</tr>
<tr>
<td>7</td>
<td>07</td>
<td>55</td>
<td>37</td>
<td>103</td>
<td>67</td>
</tr>
<tr>
<td>8</td>
<td>08</td>
<td>56</td>
<td>38</td>
<td>104</td>
<td>68</td>
</tr>
<tr>
<td>9</td>
<td>09</td>
<td>57</td>
<td>39</td>
<td>105</td>
<td>69</td>
</tr>
<tr>
<td>10</td>
<td>0A</td>
<td>58</td>
<td>3A</td>
<td>106</td>
<td>6A</td>
</tr>
<tr>
<td>11</td>
<td>0B</td>
<td>59</td>
<td>3B</td>
<td>107</td>
<td>6B</td>
</tr>
<tr>
<td>12</td>
<td>0C</td>
<td>60</td>
<td>3C</td>
<td>108</td>
<td>6C</td>
</tr>
<tr>
<td>13</td>
<td>0D</td>
<td>61</td>
<td>3D</td>
<td>109</td>
<td>6D</td>
</tr>
<tr>
<td>14</td>
<td>0E</td>
<td>62</td>
<td>3E</td>
<td>110</td>
<td>6E</td>
</tr>
<tr>
<td>15</td>
<td>0F</td>
<td>63</td>
<td>3F</td>
<td>111</td>
<td>6F</td>
</tr>
<tr>
<td>16</td>
<td>10</td>
<td>64</td>
<td>40</td>
<td>112</td>
<td>70</td>
</tr>
<tr>
<td>17</td>
<td>11</td>
<td>65</td>
<td>41</td>
<td>113</td>
<td>71</td>
</tr>
<tr>
<td>18</td>
<td>12</td>
<td>66</td>
<td>42</td>
<td>114</td>
<td>72</td>
</tr>
<tr>
<td>19</td>
<td>13</td>
<td>67</td>
<td>43</td>
<td>115</td>
<td>73</td>
</tr>
<tr>
<td>20</td>
<td>14</td>
<td>68</td>
<td>44</td>
<td>116</td>
<td>74</td>
</tr>
<tr>
<td>21</td>
<td>15</td>
<td>69</td>
<td>45</td>
<td>117</td>
<td>75</td>
</tr>
<tr>
<td>22</td>
<td>16</td>
<td>70</td>
<td>46</td>
<td>118</td>
<td>76</td>
</tr>
<tr>
<td>23</td>
<td>17</td>
<td>71</td>
<td>47</td>
<td>119</td>
<td>77</td>
</tr>
<tr>
<td>24</td>
<td>18</td>
<td>72</td>
<td>48</td>
<td>120</td>
<td>78</td>
</tr>
<tr>
<td>25</td>
<td>19</td>
<td>73</td>
<td>49</td>
<td>121</td>
<td>79</td>
</tr>
<tr>
<td>26</td>
<td>1A</td>
<td>74</td>
<td>4A</td>
<td>122</td>
<td>7A</td>
</tr>
<tr>
<td>27</td>
<td>1B</td>
<td>75</td>
<td>4B</td>
<td>123</td>
<td>7B</td>
</tr>
<tr>
<td>28</td>
<td>1C</td>
<td>76</td>
<td>4C</td>
<td>124</td>
<td>7C</td>
</tr>
<tr>
<td>29</td>
<td>1D</td>
<td>77</td>
<td>4D</td>
<td>125</td>
<td>7D</td>
</tr>
<tr>
<td>30</td>
<td>1E</td>
<td>78</td>
<td>4E</td>
<td>126</td>
<td>7E</td>
</tr>
<tr>
<td>31</td>
<td>1F</td>
<td>79</td>
<td>4F</td>
<td>127</td>
<td>7F</td>
</tr>
<tr>
<td>32</td>
<td>20</td>
<td>80</td>
<td>50</td>
<td></td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>21</td>
<td>81</td>
<td>51</td>
<td></td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>22</td>
<td>82</td>
<td>52</td>
<td></td>
<td></td>
</tr>
<tr>
<td>35</td>
<td>23</td>
<td>83</td>
<td>53</td>
<td></td>
<td></td>
</tr>
<tr>
<td>36</td>
<td>24</td>
<td>84</td>
<td>54</td>
<td></td>
<td></td>
</tr>
<tr>
<td>37</td>
<td>25</td>
<td>85</td>
<td>55</td>
<td></td>
<td></td>
</tr>
<tr>
<td>38</td>
<td>26</td>
<td>86</td>
<td>56</td>
<td></td>
<td></td>
</tr>
<tr>
<td>39</td>
<td>27</td>
<td>87</td>
<td>57</td>
<td></td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>28</td>
<td>88</td>
<td>58</td>
<td></td>
<td></td>
</tr>
<tr>
<td>41</td>
<td>29</td>
<td>89</td>
<td>59</td>
<td></td>
<td></td>
</tr>
<tr>
<td>42</td>
<td>2A</td>
<td>90</td>
<td>5A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>43</td>
<td>2B</td>
<td>91</td>
<td>5B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>44</td>
<td>2C</td>
<td>92</td>
<td>5C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>45</td>
<td>2D</td>
<td>93</td>
<td>5D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>46</td>
<td>2E</td>
<td>94</td>
<td>5E</td>
<td></td>
<td></td>
</tr>
<tr>
<td>47</td>
<td>2F</td>
<td>95</td>
<td>5F</td>
<td></td>
<td></td>
</tr>
<tr>
<td>48</td>
<td>30</td>
<td>96</td>
<td>60</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table to Determine Backward Branches

<table>
<thead>
<tr>
<th>Steps Backward (Decimal)</th>
<th>Branch Operand (Hex)</th>
<th>Steps Backward (Decimal)</th>
<th>Branch Operand (Hex)</th>
<th>Steps Backward (Decimal)</th>
<th>Branch Operand (Hex)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>FF</td>
<td>49</td>
<td>CF</td>
<td>97</td>
<td>9F</td>
</tr>
<tr>
<td>2</td>
<td>FE</td>
<td>50</td>
<td>CE</td>
<td>98</td>
<td>9E</td>
</tr>
<tr>
<td>3</td>
<td>FD</td>
<td>51</td>
<td>CD</td>
<td>99</td>
<td>9D</td>
</tr>
<tr>
<td>4</td>
<td>FC</td>
<td>52</td>
<td>CC</td>
<td>100</td>
<td>9C</td>
</tr>
<tr>
<td>5</td>
<td>FB</td>
<td>53</td>
<td>CB</td>
<td>101</td>
<td>9B</td>
</tr>
<tr>
<td>6</td>
<td>FA</td>
<td>54</td>
<td>CA</td>
<td>102</td>
<td>9A</td>
</tr>
<tr>
<td>7</td>
<td>F9</td>
<td>55</td>
<td>C9</td>
<td>103</td>
<td>99</td>
</tr>
<tr>
<td>8</td>
<td>F8</td>
<td>56</td>
<td>C8</td>
<td>104</td>
<td>98</td>
</tr>
<tr>
<td>9</td>
<td>F7</td>
<td>57</td>
<td>C7</td>
<td>105</td>
<td>97</td>
</tr>
<tr>
<td>10</td>
<td>F6</td>
<td>58</td>
<td>C6</td>
<td>106</td>
<td>96</td>
</tr>
<tr>
<td>11</td>
<td>F5</td>
<td>59</td>
<td>C5</td>
<td>107</td>
<td>95</td>
</tr>
<tr>
<td>12</td>
<td>F4</td>
<td>60</td>
<td>C4</td>
<td>108</td>
<td>94</td>
</tr>
<tr>
<td>13</td>
<td>F3</td>
<td>61</td>
<td>C3</td>
<td>109</td>
<td>93</td>
</tr>
<tr>
<td>14</td>
<td>F2</td>
<td>62</td>
<td>C2</td>
<td>110</td>
<td>92</td>
</tr>
<tr>
<td>15</td>
<td>F1</td>
<td>63</td>
<td>C1</td>
<td>111</td>
<td>91</td>
</tr>
<tr>
<td>16</td>
<td>F0</td>
<td>64</td>
<td>C0</td>
<td>112</td>
<td>90</td>
</tr>
<tr>
<td>17</td>
<td>EF</td>
<td>65</td>
<td>BF</td>
<td>113</td>
<td>8F</td>
</tr>
<tr>
<td>18</td>
<td>EE</td>
<td>66</td>
<td>BE</td>
<td>114</td>
<td>8E</td>
</tr>
<tr>
<td>19</td>
<td>ED</td>
<td>67</td>
<td>BD</td>
<td>115</td>
<td>8D</td>
</tr>
<tr>
<td>20</td>
<td>EC</td>
<td>68</td>
<td>BC</td>
<td>116</td>
<td>8C</td>
</tr>
<tr>
<td>21</td>
<td>EB</td>
<td>69</td>
<td>BB</td>
<td>117</td>
<td>8B</td>
</tr>
<tr>
<td>22</td>
<td>EA</td>
<td>70</td>
<td>BA</td>
<td>118</td>
<td>8A</td>
</tr>
<tr>
<td>23</td>
<td>E9</td>
<td>71</td>
<td>B9</td>
<td>119</td>
<td>89</td>
</tr>
<tr>
<td>24</td>
<td>E8</td>
<td>72</td>
<td>B8</td>
<td>120</td>
<td>88</td>
</tr>
<tr>
<td>25</td>
<td>E7</td>
<td>73</td>
<td>B7</td>
<td>121</td>
<td>87</td>
</tr>
<tr>
<td>26</td>
<td>E6</td>
<td>74</td>
<td>B6</td>
<td>122</td>
<td>86</td>
</tr>
<tr>
<td>27</td>
<td>E5</td>
<td>75</td>
<td>B5</td>
<td>123</td>
<td>85</td>
</tr>
<tr>
<td>28</td>
<td>E4</td>
<td>76</td>
<td>B4</td>
<td>124</td>
<td>84</td>
</tr>
<tr>
<td>29</td>
<td>E3</td>
<td>77</td>
<td>B3</td>
<td>125</td>
<td>83</td>
</tr>
<tr>
<td>30</td>
<td>E2</td>
<td>78</td>
<td>B2</td>
<td>126</td>
<td>82</td>
</tr>
<tr>
<td>31</td>
<td>E1</td>
<td>79</td>
<td>B1</td>
<td>127</td>
<td>81</td>
</tr>
<tr>
<td>32</td>
<td>E0</td>
<td>80</td>
<td>B0</td>
<td>128</td>
<td>80</td>
</tr>
<tr>
<td>33</td>
<td>DF</td>
<td>81</td>
<td>AF</td>
<td></td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>DE</td>
<td>82</td>
<td>AE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>35</td>
<td>DD</td>
<td>83</td>
<td>AD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>36</td>
<td>DC</td>
<td>84</td>
<td>AC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>37</td>
<td>DB</td>
<td>85</td>
<td>AB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>38</td>
<td>DA</td>
<td>86</td>
<td>AA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>39</td>
<td>D9</td>
<td>87</td>
<td>A9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>D8</td>
<td>88</td>
<td>A8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>41</td>
<td>D7</td>
<td>89</td>
<td>A7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>42</td>
<td>D6</td>
<td>90</td>
<td>A6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>43</td>
<td>D5</td>
<td>91</td>
<td>A5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>44</td>
<td>D4</td>
<td>92</td>
<td>A4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>45</td>
<td>D3</td>
<td>93</td>
<td>A3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>46</td>
<td>D2</td>
<td>94</td>
<td>A2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>47</td>
<td>D1</td>
<td>95</td>
<td>A1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>48</td>
<td>D0</td>
<td>96</td>
<td>A0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 6-2. Backward Branches
4. Program counter starts at 1012
Branch desired to FB7 (-91 decimal steps)

- 0FB7

Look up in table:

<table>
<thead>
<tr>
<th>Steps Backward (Decimal)</th>
<th>Branch Operand (Hex)</th>
</tr>
</thead>
<tbody>
<tr>
<td>91</td>
<td>A5</td>
</tr>
</tbody>
</table>

If condition tested is not equal to zero, branch backward to FB7 (1012-5B hex steps).

NOTE—All of the above calculations were performed with hex numbers. Hexadecimal subtraction will be discussed in Chapter 9. The table in Figure 6-2 will give you the necessary operand for backward branches.

All branch instructions, listed in the following table, use the Relative Addressing mode. The status flags that determine the condition on which the branch is taken, or not taken, are given in the table.

<table>
<thead>
<tr>
<th>Mnemonic Code</th>
<th>Instruction</th>
<th>Status Flag</th>
<th>Conditions for Branch</th>
</tr>
</thead>
<tbody>
<tr>
<td>BCC</td>
<td>Branch on Carry Clear</td>
<td>C</td>
<td>0</td>
</tr>
<tr>
<td>BCS</td>
<td>Branch on Carry Set</td>
<td>C</td>
<td>1</td>
</tr>
<tr>
<td>BEQ</td>
<td>Branch on result zero</td>
<td>Z</td>
<td>1</td>
</tr>
<tr>
<td>BMI</td>
<td>Branch on result minus</td>
<td>N</td>
<td>1</td>
</tr>
<tr>
<td>BNE</td>
<td>Branch on result not zero</td>
<td>Z</td>
<td>0</td>
</tr>
<tr>
<td>BPL</td>
<td>Branch on result positive</td>
<td>N</td>
<td>0</td>
</tr>
<tr>
<td>BVC</td>
<td>Branch on overflow clear</td>
<td>V</td>
<td>0</td>
</tr>
<tr>
<td>BVS</td>
<td>Branch on overflow set</td>
<td>V</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure 6-3. Status Flags for Branches

In order to demonstrate some of the branch instructions, we will write some short demonstration programs that perform arithmetic operations.

**USING THE CARRY FLAG**

Suppose you were using the machine language subroutine on page 39, and you input the decimal numbers 123 and 133. The result of their sum would be displayed as zero because the result is too large to be contained in one byte. An extra bit is needed to express the true result.
The sum of that problem:

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>01111011</td>
</tr>
<tr>
<td>133</td>
<td>10000101</td>
</tr>
<tr>
<td>Sum</td>
<td>00000000</td>
</tr>
</tbody>
</table>

extra 8 bits displayed

When an extra bit (as in this example) occurs from the addition of two 8-bit numbers, the computer automatically sets the carry flag (C=1). You can verify this by using the Atari Assembler Cartridge to enter the following program.

```assembly
EDIT
10  *=$1000
20  CLC
30  LDA #$7B
40  ADC #$85
50  END
```

Then assemble the program by typing: ASM

```assembly
ASM
0000  10  *=$1000
1000  18  20  CLC
1001  A97B 30  LDA #$7B
1003  6985 40  ADC #$85
      50  END
```

Notice the symbols (# and $) used with the operands in lines 30 and 40 of the Source Program (the Assembly Language program). The # symbol tells the
computer that the LDA and ADC instructions are to be used in the Immediate mode. The $ symbol tells the computer that the numbers in the operand are to be treated as hexadecimal values. The symbols are a necessary part of the assembly language instruction.

Enter the Debugger Program and trace the program. In this way, you can watch the changes in the Processor Status Register (P) where the status flags are stored. You will want to keep your eye on the Carry bit (flag) in the Processor Status Register before and after the add instruction is used.

```
EDIT
BUG ← Type BUG to enter the DEBUGGER
DEBUG
T1000 ← Type T1000 to trace the program beginning at memory location 1000
```

<table>
<thead>
<tr>
<th>Location</th>
<th>Instruction</th>
<th>Accumulator</th>
<th>X</th>
<th>Y</th>
<th>Processor Status Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>CLC</td>
<td>A=00</td>
<td>X=00</td>
<td>Y=00</td>
<td>P=BO S=00</td>
</tr>
<tr>
<td>1001</td>
<td>LDA #$7B</td>
<td>A=7B</td>
<td>X=00</td>
<td>Y=00</td>
<td>P=30 S=00</td>
</tr>
<tr>
<td>1003</td>
<td>ADC #$85</td>
<td>A=00</td>
<td>X=00</td>
<td>Y=00</td>
<td>P=33 S=00</td>
</tr>
<tr>
<td>1005</td>
<td>BRK</td>
<td>A=00</td>
<td>X=00</td>
<td>Y=00</td>
<td>P=33 S=00</td>
</tr>
</tbody>
</table>

Notice that at step 1003 the accumulator changed to zero, and the Processor Status Register changed from 30 to 33. The accumulator tells us that the sum is zero, but consider what the P register shows.

---

**Before step 1003**

```
[0 0 1 1 0 0 0 0]  
\[\underbrace{3 \ 0}\]  
```

**After step 1003**

```
[0 0 1 1 0 0 1 1]  
\[\underbrace{3 \ 3}\]  
```
We can consider the result to be a combination of the carry bit and the value in the accumulator.

In Hexadecimal

\[
\begin{align*}
1 + 0\text{ }0 &= 100 \\
\text{carry bit} &\rightarrow \text{in accumulator}
\end{align*}
\]

In decimal this would be \(256 + 0 = 256\)

By expanding our program, we can store the results in memory by making use of the Branch on Carry Clear (BCC) instruction. Use the NEW command to clear the EDIT Buffer.

Using the Writer/Editor Program of the Assembly Cartridge:

<table>
<thead>
<tr>
<th>EDIT</th>
</tr>
</thead>
<tbody>
<tr>
<td>10 *= $1000</td>
</tr>
<tr>
<td>20 CLC</td>
</tr>
<tr>
<td>30 LDA #$00</td>
</tr>
<tr>
<td>40 STA $1050</td>
</tr>
<tr>
<td>50 LDA #$7D</td>
</tr>
<tr>
<td>60 ADC #$A4</td>
</tr>
<tr>
<td>70 STA $1051</td>
</tr>
<tr>
<td>80 BCC END</td>
</tr>
<tr>
<td>90 INC $1050</td>
</tr>
<tr>
<td>100 END</td>
</tr>
</tbody>
</table>

The program adds two hex numbers and stores the low 8 bits from the accumulator into memory location 1051. If there was a carry, memory location 1050 will be incremented at line 90. If no carry is made, the instruction (BCC) at line 80 will cause the computer to skip over the INC instruction at line 90 and go to the end of the program. Thus, memory location 1050 will contain the extra bit when a carry has been made but will be zero if no carry was made. The combination of memory locations 1050 and 1051 will provide the complete result. Assemble the program and execute it. Then look at the results:

Assembly

<table>
<thead>
<tr>
<th>Type: ASM</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 10 *= $1000</td>
</tr>
<tr>
<td>1000 18 20 CLC</td>
</tr>
<tr>
<td>1001 A900 30 LDA #$00</td>
</tr>
</tbody>
</table>
Notice that the STA operand and the INC operand take two bytes for the address involved. The $ sign once again indicates a hexadecimal value. The Branch on Carry Clear operand shows a forward branch of 3 steps from 100F to 1012 where the program stops.

This is the first time you have used the INC instruction. It is used to increment by one the value in the specified memory location. It is similar to the INX instruction (increment the X register) that you used earlier.

**Example:**

**INC (Increment memory)**
- Op code EE (If Absolute mode addressing)
- Second byte 50 (Least significant address byte)
- Third byte 10 (Most significant address byte)
- Status flags affected: Z and C

DEBUG and Execute

**Type: BUG**

```
BUG
DEBUG
```

**Type: T1000 (to trace the program)**
Notice that a branch was not made at 100D (the Carry flag was not clear, it was set). Therefore the memory location 1050 was incremented at line 100F.

To check the results, you will want to display the memory locations 1050 and 1051 to verify that the answer is correct.

Type: D1050,1051

the hex result = ($0X16^3$) + ($1X16^2$) + ($2X16$) + 1
  = 0 + 256 + 32 + 1
  = 289 decimal
We could use the Branch on Carry Set (BCS) instruction in the previous program instead of the Branch on Carry Clear.

<table>
<thead>
<tr>
<th>Using BCC</th>
<th>Using BCS</th>
</tr>
</thead>
<tbody>
<tr>
<td>010 *=1000</td>
<td>010 *=1000</td>
</tr>
<tr>
<td>020 CLC</td>
<td>020 CLC</td>
</tr>
<tr>
<td>030 LDA #0</td>
<td>030 LDA #0</td>
</tr>
<tr>
<td>040 STA $1050</td>
<td>040 STA $1050</td>
</tr>
<tr>
<td>050 LDA #7D</td>
<td>050 LDA #7D</td>
</tr>
<tr>
<td>060 ADC #$A4</td>
<td>060 ADC #$A4</td>
</tr>
<tr>
<td>070 STA $1051</td>
<td>070 STA $1051</td>
</tr>
<tr>
<td>080 BCC END</td>
<td>080 BCS SET</td>
</tr>
<tr>
<td>090 INC $1050</td>
<td>090 JMP END</td>
</tr>
<tr>
<td>100 END</td>
<td>100 SET INC $1050</td>
</tr>
<tr>
<td></td>
<td>110 END</td>
</tr>
</tbody>
</table>

The result of these two programs would be the same. The program using BCS would jump to the END at line 90 if the Carry flag had not been set. It would branch from line 80 to line 100 if a carry had been set.

All BRANCH instructions are dependent on some condition expressed by the status flags in the Processor Status Register.

All JUMP instructions are unconditional.

**USING THE ZERO FLAG**

In Chapter 4, you used a program which counted from 1 through FF. The counting loop was accomplished by comparing the value in the accumulator with FF and using the Branch on result Not Equal zero (BNE). This was the assembler program:

```
10 *=1000
20 CLC
30 LDA #0
40 LOOP ADC #1
50 CMP #$FF
60 BNE LOOP
70 END
```
The companion instruction to BNE is BEQ (Branch on result EQual to zero). The counting program could be written to use the BEQ instruction in the following way:

<table>
<thead>
<tr>
<th>Line</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>*$1000</td>
<td>Load accumulator with zero</td>
</tr>
<tr>
<td>20</td>
<td>CLC</td>
<td>Clear carry flag</td>
</tr>
<tr>
<td>30</td>
<td>LDA #0</td>
<td>Load accumulator with zero</td>
</tr>
<tr>
<td>40</td>
<td>LOOP ADC #1</td>
<td>Loop</td>
</tr>
<tr>
<td>50</td>
<td>CMP #$FF</td>
<td>Compare with zero</td>
</tr>
<tr>
<td>60</td>
<td>BEQ END</td>
<td>Branch on result zero</td>
</tr>
<tr>
<td>70</td>
<td>JMP LOOP</td>
<td>Jump to loop</td>
</tr>
<tr>
<td>80</td>
<td>END</td>
<td>Program ends</td>
</tr>
</tbody>
</table>

Use the assembler to enter and trace each program. You can see from the flowcharts that the first program (using BNE) is more straightforward. However, you do have the option of using either instruction (BNE or BEQ). In some programs the BEQ instruction would be the best choice.

If you time the traces of the two programs you will find that the first program is a little faster. You can see by the printout of the traces that one more instruction must be executed each time the second program passes through the counting loop.

Trace of BNE program:

Video Display at the end of the trace:

```
1007 D0 FA BNE $1003
  A=FD X=00 Y=00 P=B0 S=00
1003 69 01 ADC #$01
  A=FE X=00 Y=00 P=B0 S=00
1005 C9 FF CMP #$FF
  A=FE X=00 Y=00 P=B0 S=00
1007 D0 FA BNE $1003
  A=FE X=00 Y=00 P=B0 S=00
1003 69 01 ADC #$01
  A=FF X=00 Y=00 P=B0 S=00
1005 C9 FF CMP #$FF A=FF P=33
  Branch is not taken
1007 D0 FA BNE $1003
  A=FF X=00 Y=00 P=33 S=00
1009 00 BRK
  Program ends
```

```
3 instructions in this loop
A=FE P=B0
Branch is taken back to 1003
A=FF P=33
Branch is not taken
Program ends
DEBUG
```
This trace took approximately 2.3 minutes. Notice the Processor Status Register after the Compare instruction:

First, when $A = FE$

```
 0 1 1 1 0 0 0 0
```

- Zero Flag = 0
- Result $\neq 0$
- Branch taken

Second, when $A = FF$

```
 0 0 1 1 0 0 1 1
```

- Zero Flag = 1
- Result = zero
- Branch not taken

Trace of BEQ Program:

Video Display at the end of the trace:

```
1009 4C 03 10 JMP $1003
  A=FD X=00 Y=00 P=B0 S=00
1003 69 01 ADC #$01
  A=FE X=00 Y=00 P=B0 S=00
1005 C9 FF CMP #$FF
  A=FE X=00 Y=00 P=B0 S=00
  First A=FE
1007 F0 05 BEQ END
  A=FE X=00 Y=00 P=B0 S=00
1009 4C 03 10 JMP $1003
  A=FE X=00 Y=00 P=B0 S=00
1003 69 01 ADC #$01
  A=FF X=00 Y=00 P=B0 S=00
1005 C9 FF CMP #$FF
  A=FF X=00 Y=00 P=33 S=00
  Second A=FF
1007 F0 05 BEQ END
  A=FF X=00 Y=00 P=33 S=00
100C 00 BRK
  A=FF X=00 Y=00 P=33 S=00
```

4 instructions in this loop
This trace took approximately 3.1 minutes.

Once again, notice the Processor Status Register after the Compare instruction:

First, when A = FE

\[
\begin{array}{c}
1 & 0 & 1 & 1 & 0 & 0 & 0 \\
\end{array}
\]

\[\text{Zero Flag} = 0 \quad \text{Branch is taken}\]

Second, when A = FF

\[
\begin{array}{c}
0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\
\end{array}
\]

\[\text{Zero Flag} = 1 \quad \text{Branch is not taken}\]

The BEQ program took a longer time to trace than the BNE program. This difference in timing would not be noticed if the program is run in the normal manner. However, in the trace mode, the time for printing the extra step is noticeable.

**USING THE NEGATIVE FLAG**

Once again, let's use a variation of the counting program to see how some branch instructions decide whether a number is positive or negative. We'll first use the Branch on result Plus (BPL) instruction. The N (negative) flag is reset to zero for positive numbers and is set to 1 for negative numbers when certain instructions are executed. If the Negative flag has been reset to zero, the BPL instruction will cause a branch to be taken (because the result is positive). If the Negative flag has been set to 1 (result negative), the branch will not be taken.

**BRANCH ON RESULT PLUS DEMONSTRATION**

Using the assembler cartridge, enter the following program in the Writer/Editor mode. Type NEW to clear the last program:

10  *= $1000
20  CLC
30  LDA #0
40  LOOP ADC #1
50  BPL LOOP
60  END

Assemble the demonstration program. Then enter the Debugger Mode and trace the program.
The end of the trace looks like this:

1003 69 01 ADC #$01
   A = 7F X = 00 Y = 00 P = 30 S = 00
1005 10 FA BPL $1003 ← Branch back to 1003
   A = 7F X = 00 Y = 00 P = 30 S = 00
1003 69 01 ADC #$01
   A = 80 X = 00 Y = 00 P = F0 S = 00 ← Notice change in P
1005 10 FA BPL $1003 when A = 80
   A = 80 X = 00 Y = 00 P = F0 S = 00
1007 00 BRK ← Branch not taken
   A = 80 X = 00 Y = 00 P = F0 S = 00
DEBUG

Notice that the branch loop was executed until the value in the accumulator reached 80. This was the first negative value reached.

| 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |

Process Status Register

N set to 1
USING THE NEGATIVE FLAG

Negative values can be more dramatically seen by starting with zero in the accumulator and subtracting one each time through a loop.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Result</th>
<th>Signed Decimal Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 - 1</td>
<td>FF</td>
<td>-1</td>
</tr>
<tr>
<td>FF - 1</td>
<td>FE</td>
<td>-2</td>
</tr>
<tr>
<td>FE - 1</td>
<td>FD</td>
<td>-3</td>
</tr>
<tr>
<td>FD - 1</td>
<td>FC</td>
<td>-4</td>
</tr>
<tr>
<td>.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>etc</td>
<td></td>
<td></td>
</tr>
<tr>
<td>.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>83 - 1</td>
<td>82</td>
<td>-126</td>
</tr>
<tr>
<td>82 - 1</td>
<td>81</td>
<td>-127</td>
</tr>
<tr>
<td>81 - 1</td>
<td>80</td>
<td>-128</td>
</tr>
</tbody>
</table>

You might think of 8-bit signed numbers as locations on a large number wheel rather than the usual number line. Then they would look like this.

![Signed Number Wheel, Hex Values Inside; Decimal Equivalents Outside](image-url)

*Figure 6-4. Signed Number Wheel, Hex Values Inside; Decimal Equivalents Outside*
To demonstrate the subtraction of one from the accumulator, we'll use the Branch on result Minus (BMI) to create a loop. The carry bit is used in subtraction in the borrowing process. When using addition, we cleared the Carry bit before performing the operation. In subtraction, we must set the carry bit before performing the operation. Here is the program as entered during the assembler's Writer/Editor mode.

```
10  *= $1000
20  LDA #0
30  LOOP SEC
40  SBC #1
50  BMI LOOP
60  END
```

Assemble the program. Then enter the Debug mode and single step through the first part of the program.

```
<table>
<thead>
<tr>
<th>ASM</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>1000</td>
</tr>
<tr>
<td>1002</td>
</tr>
<tr>
<td>1003</td>
</tr>
<tr>
<td>1005</td>
</tr>
<tr>
<td>60 END</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>EDIT</th>
</tr>
</thead>
<tbody>
<tr>
<td>BUG</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DEBUG</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DEBUG</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1000</td>
</tr>
<tr>
<td>1000</td>
</tr>
<tr>
<td>A900</td>
</tr>
<tr>
<td>LDA #0</td>
</tr>
<tr>
<td>A=00</td>
</tr>
</tbody>
</table>
```
You can see that this will go on for some time before we finally reach a non-negative number. You can also see that the computer considers FF and FE as negative values and therefore branches back to subtract again. It will branch back until the accumulator reaches 7F. By looking at the Signed Number Wheel in Figure 6-4, you will find that 80 is interpreted as a negative value (-128 decimal). However, 7F is interpreted as positive (+127 decimal). Therefore, the branch will not be taken at that point.

Instead of plodding on through all the negative integers up to and including -128, change to the Trace mode to finish up the program. After the last DE-
BUG prompt, type T1000 and press RETURN. Now the computer will trace through the program, and you can see the END result.

As seen on the display:

```
1005  30 FB   BMI $1002
    A=80 X=00 Y=00 P=B1 S=00 ← 80 still considered
    38       SEC        negative—branch
    A=80 X=00 Y=00 P=B1 S=00
1003  E9 01   SBC #$01
    A=7F X=00 Y=00 P=71 S=00 ← Negative flag off (P = 71)
1005  30 FB   BMI $1002
    A=7F X=00 Y=00 P=71 S=00 ← 7F considered positive—
    00       BRK              no branch taken
1007  00
```

**THE OVERFLOW FLAG**

Two branch instructions BVC (Branch on oVerflow Clear) and BVS (Branch on oVerflow Set) can also change the normal sequential operation of a machine language program. If you are not performing arithmetic with signed numbers, the Overflow flag may be completely ignored. We will discuss arithmetic with signed numbers in Chapter 9.

![Processor Status Register](image)

**SUMMARY**

This chapter discussed branch instructions—what they are and how they are used. You learned that:

- All branch instructions are made in the Relative Addressing mode;
- Branch instructions cause a branch when a specified condition is true;
- The Program Counter is moved relative to its current position when a branch is taken;
- The second byte of a branch instruction tells how far and in what direction (forward or backward) the Program Counter is moved;
• Forward branches are taken when the second byte of the branch instruction is in the range of 1 through 7F;
• Backward branches are taken when the second byte of the branch instruction is in the range of 80 through FF;
• The following status flags are used to determine the condition for branches:
  Negative
  Overflow
  Zero
  Carry
• The condition is true when the specified flag is set to one;
• The condition is false when the specified flag is reset to zero;
• Hexadecimal numbers can be interpreted as negative values if they are in the range of 80 through FF and as positive values when they are in the range of 1 through 7F;
• All branch instructions consider zero a positive value.

EXERCISES

1. When the following instruction is being executed, at what memory location is the Program Counter pointing?

(1010, 1011, or 1012)

<table>
<thead>
<tr>
<th>Memory</th>
<th>Op Code</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1010</td>
<td>D0</td>
<td>BNE</td>
</tr>
<tr>
<td>1011</td>
<td>F9</td>
<td></td>
</tr>
</tbody>
</table>

2. What status flag will determine whether the branch in Exercise 1 is taken or not? ______________ flag

3. Which direction (forward or backward) will the branch in Exercise 1 be taken? ______________

4. Using the tables in Figures 6-1 and 6-2, give the hex operand that should be used to make the following branches:
   (a) forward 38 decimal steps ______________
   (b) backward 100 decimal steps ______________
   (c) forward 93 decimal steps ______________
   (d) backward 42 decimal steps ______________

5. Give the two-byte branch instruction which would branch backward 11 decimal steps if the result of an operation were positive.
<table>
<thead>
<tr>
<th>Op Code</th>
<th>Mnemonic</th>
</tr>
</thead>
</table>

6. If two hex values are added and the result is greater than FF, what flag will be set?

(Negative, Zero, or Carry)

7. What hex value would be in the accumulator following the execution of these two instructions? 

\[ \text{LDA} \ #\#7A \\
\text{ADC} \ #\#87 \]

8. Tell the contents (0 or 1) of the following flags after the two instructions in Exercise 7 have been executed.

- Zero flag 
- Negative flag 
- Carry flag 

9. Tell which mode of the assembler is used for the following: (Modes are: EDIT, ASM, OR DEBUG)

- (a) to LIST a source program
- (b) to run an object program
- (c) to display a memory location used to store a result in a machine language program
- (d) to trace an object program

10. Tell what the following mnemonic codes represent:

- (a) BCC
- (b) BEQ
- (c) BPL
- (d) BCS
- (e) BNE
- (f) BMI

**ANSWERS**

1. 1012 (always one instruction ahead)
2. Zero flag
3. Backward
4. (a) 26  
   (b) 9C  
   (c) 5D  
   (d) D6
5. Op Code  Mnemonic
   10   BPL
   F5

6. Carry flag for sure (Possibly N,Z)

7. 01

8. Zero flag 0
   Negative flag 0
   Carry flag 1

9. (a) EDIT
    (b) DEBUG
    (c) DEBUG
    (d) DEBUG

10. (a) Branch on Carry Clear
    (b) Branch on result EQual zero
    (c) Branch on result PLus
    (d) Branch on Carry Set
    (e) Branch on result Not Equal zero
    (f) Branch on result MInus
Assembler Review

This chapter summarizes what you have learned about the Atari Assembler and adds some new features that have not been discussed so far.

An assembly language program is made up of statements containing statement numbers, labels, Op code mnemonics, operands, and comments. The program, which is written in the Writer/Editor mode of the Assembler Cartridge, is called the source program. You assemble the source program in the Assemble mode. This produces a machine language program, called the object program. The assembler places this program in memory. The object program is then executed in the Debug mode to produce the desired result.

SOURCE PROGRAM FORMAT

The source program is made up of statements that begin with a line number and is terminated by pressing the RETURN key. Statements are broken up into the following fields, some of which are optional.

Figure 7-1. Statement Fields

Every statement must start with a statement number that is in the range of 0 through 65,535. You should number your statements in such a way that new ones may be inserted at a later time in case you want to alter your original program. Multiples of 10 work well for this purpose (i.e., 10, 20, 30, etc.). The Writer/Editor has convenient commands for automatically numbering or renumbering programs (see page 129). A statement may be up to 107 characters in length.
The label field is optional. If it is used, it follows the line number with exactly one space between the last digit of the line number and the first letter of the label. A label must start with a letter and contain only letters and numbers. It can be as short as two characters or as long as the limitation of the statement length. If a label is not used, two spaces or a tab is used between the statement number and the Op Code Mnemonic field (which follows the label field).

Examples:

With label

\[
20 \text{ LOOP}
\]

Statement number

one label space

With no label

\[
20 \quad \text{Op Code}
\]

two spaces, or

\[
20 \quad \text{Op Code}
\]

a tab gives 8 spaces

The Op Code Mnemonic must be one of those given in Appendix A or B. This field starts at least two spaces beyond the statement number (if no label is used) or one space after the label. A mnemonic placed in the wrong field will not be identified as an error in the Writer/Editor mode, but will be identified as Error -6 when you assemble the program (See Appendix D for Error codes.)

Examples:

With label

\[
20 \text{ LOOP CLC}
\]

one space Op Code for Clear Carry

With no label

\[
50 \quad \text{INY}
\]

two spaces Op Code

\[
50 \quad \text{INY}
\]

8 spaces from a tab

The field of the operand starts at least one space, or a tab, after the mnemonic field. Some Op Code mnemonics require an operand, while others do not. The form of the operand that is appropriate for each mnemonic code that uses one is given in Appendix A.
Examples:

For hex operand

30 \text{LDA} \ #20

\text{two spaces} \quad \text{Op Code} \quad \text{Operand hex}

Immediate Addressing mode

30 \text{LDA} \ \$1100

Load accumulator—from memory location $1100$

For decimal operand

30 \text{LDA} \ #32

Immediate Addressing

10 \text{LDA} \ 4352

Absolute Addressing

(\text{no} \# \text{sign})

The last field, \textit{comment}, appears on the listing of a source program but is not assembled. Therefore, it does not appear in the object program. There are two ways to enter comments.

- One way is to enter it in the comment field which occupies the remainder of the statement following the operand (if there is one) or the mnemonic code (if there is no operand). At least one space separates it from the last field used. There may not be enough room to put the entire comment in its field. In that case, you can extend your comment into the next line.

Examples:

20 \text{CLC GET READY FOR ADDITION}

\underline{comment}

30 \text{BNE LOOP IF THE COUNT HAS NOT REACHED ZERO GO BACK TO LOOP}

\textit{Extend comment to new line (no line number)}

- A second way is to put the comment on its own line. In that case, one space follows the statement number, and a semicolon is used before the comment.
METHODS TO USE OPERANDS

Line numbers, Op Code Mnemonics, and comments are straightforward and easy to use. The operands are sometimes tricky and deserve some more explanation.

Operands may be in the form of hexadecimal numbers, decimal numbers, or letters. The programmer has the option to use whichever form is desired.

A number used as an operand is interpreted as a decimal number by the assembler unless it is preceded by a $ sign. If a $ precedes the number, it is interpreted as a hexadecimal number.

Examples:

40 LDA 4500 \( \leftarrow \) 4500 is interpreted as a decimal number
50 STA $1100 \( \leftarrow \$1100 \) is interpreted as a hexadecimal number

If a group of letters has been used as a label or have been assigned a numeric value, the letters may be used as an operand.

Examples:

20 ABC=$33
30 DEF=51
40 LOOP CLC

\* LOOP was used as a label previously

80 BNE LOOP
90 CMP DEF \( \leftarrow \) Decimal value 51

\*

90 CMP $ABC \( \leftarrow \) Hexadecimal value 33

In addition to providing data, the operand also tells the computer what addressing mode to use for the operation specified by the Op Code Mnemonic.
Examples:

20  LDA #12  The # sign indicates the Immediate Addressing mode
30  CPY $1212  Absolute Addressing mode (no # sign)
40  STA $1250, Y  Absolute Indexed mode (note: ,Y added)
50  ADC ($2B,X)  Indexed Indirect mode (note: parentheses)
60  CMP ($3C),Y  Indirect Indexed mode (note: placement of parentheses)
70  INC $3C  Zero Page mode
80  INC $20,X  Indexed Zero Page mode

THE ASSEMBLER WRITER/EDITOR MODE

The Writer/Editor mode (referred to as the Edit mode for convenience) controls the communication between you, through the keyboard, and the video screen. When you power up your Atari with the Assembler Cartridge installed, you should see the Edit mode prompt on the screen. If you do not see the Edit prompt at this time, some error or malfunction has occurred. Check your Operator's Manual to make sure you have followed all procedures correctly.

Any entries from the keyboard will now appear in successive locations across the screen starting at the cursor ( ) position. The screen is 38 locations wide. If the end of a line is reached, there is an automatic "wrap around" so that your entries continue on the next line.

We have discussed some of the following commands used in the Edit mode. Others are new to you.

EDIT

10 ;THIS PROGRAM WILL DEMONSTRATE ADDITION

38 locations, then wrap around
1. **SIZE** is used to find the location of the Current Line Buffer and Edit Text Buffer. Three numbers are displayed when the **SIZE** command is executed.

![Diagram of Buffer Memory]

The middle number in the example (0880) increases as the Edit Text Buffer is filled by your program. Therefore, the **SIZE** command can be used to find out how much memory is being used as you enter the **Source Program**. The machine language **Object Program** should be placed at memory locations higher than that occupied by your **Source Program**.

2. **LOMEM** is used to change the location of the Buffer Memory area. You have not used this command yet and may never use it. If you wanted to use 256 memory locations between 0700 and 0800 in your assembled program, you would use the **LOMEM** command like this:

```
EDIT
LOMEM 800
EDIT
```

Your memory would now be organized for use as shown in Figure 7-3.
Now give the SIZE command again.

```
EDIT
LOMEM 800

EDIT
SIZE
0800 0980 1C1F

EDIT
```

The location of the buffer areas cannot be changed after you have started writing a program. If you use the LOMEM command, *it must be the first command after you power up*.

3. LIST is used to display, on the video screen, the program that is currently in the Edit Text Buffer. It can be used in several forms.

(a) LIST to display the whole program

```
EDIT
LIST

10  *=1000
20   CLC
30   LDA #0
40   LOOP ADC #1
50   CMP #$FF
60   BNE LOOP
70   END

EDIT
```
(b) LIST 40 to display only line 40

```
EDIT
LIST 40
40 LOOP ADC #1
EDIT
```

(c) LIST 50,70 to display all lines 50 through 70

```
EDIT
LIST 50,70
50 CMP #$FF
60 BNE LOOP
70 END
EDIT
```

4. NEW is used to clear the Edit Text Buffer. It does not clear the memory occupied by a previously assembled machine language program—only the Edit Text Buffer. After this command has been executed, you cannot restore any Source Program that existed previously in the Edit Text Buffer.

If the program in 3a under LIST is in the Edit Text Buffer, and you type NEW, the Edit Text Buffer will be cleared.

```
EDIT
NEW  You typed NEW
EDIT
LIST  then LIST
EDIT  Nothing is there
```

5. DEL is used to delete statements from the Edit Text Buffer. You have not used this command. It allows you to delete one line or several successive lines.
Suppose the program of 3a is still in the Edit Text Buffer, and you want to delete line 20.

```plaintext
EDIT
DEL 20

EDIT
LIST

10  *=$1000
30  LDA #0
40  LOOP ADC #1
50  CMP #$FF
60  BNE LOOP
70  END

EDIT

Line 20 is gone
```

Now, take out lines 40, 50, and 60.

```plaintext
EDIT
DEL 40,60

EDIT
LIST

10  *=$1000
30  LDA #0
70  END
```

Not much left now

6. REP is used to replace a specified string in the Edit Text Buffer with a different specified string. Thus, a program change can be quickly made by one of the several forms of this command. Suppose you once again have the program of 3a in the Edit Text Buffer.

(a) To replace the first occurrence of the string “LDA #0” with “LDA #5”:
(b) To replace all occurrences of the string "LOOP" with the string "CIRCLE":

```
EDIT
REP/LOOP/CIRCLE/,A
```

Lines 40 and 60 would be changed to:

```
40  CIRCLE ADC #1
50  BNE CIRCLE
```

Other forms of this command are explained in the Atari Assembler Manual.

7. NUM is used to number the statements automatically in an assembly language program. It can be used in several forms.

(a) To increment the statement numbers by 10:

```
EDIT
NUM
10  •
```

Prints a 10 and skips one space

Type in the first line and press RETURN

```
EDIT
NUM
10  *=$1000
20  •
```

Type another space, *=$1000 and press RETURN

Prints 20 and skips one space
(b) To increment by a value different than 10:

```
EDIT
NUM 3

3 ← Now starts at 3
```

Type in first line and press RETURN

```
EDIT
NUM 3

3 *=1000 ← Type space then *=1000 and press RETURN
6 • Prints 6 and skips one space
```

(c) To force a new line and change the increment (or the starting line):

```
EDIT
NUM ← Regular NUM

10 *=1000
20 CLC
NUM 50,3 ← New NUM form

50 LDA #0 ← New line number, 50
53 LOOP ADC #1 ← New increment, 3
56 • Press RETURN to cancel the NUM command.
```

8. REN is used to renumber statements in the Edit Text Buffer.

(a) To renumber statements in increments of 10 starting with 10, type:

```
REN then press RETURN
```

(b) To renumber statements in increments of 5 starting with line number 10, type:

```
REN 5 and press RETURN
```
(c) To renumber all statements in increments of 2 starting with line number 20, type:
REN 20,2 and press RETURN

9. ASM is used to transfer from the **Writer/Editor Program** to the **Assembler Program**.

10. **BUG** is used to transfer from the **Writer/Editor Program** to the **Debug Program**.

There are also commands to save and retrieve programs or specific blocks of memory. Their use is described in the Atari Assembler Manual and will not be repeated here. Each of these commands

**LIST, ENTER, ASM, SAVE, and LOAD**

are used in several forms depending on whether you wish to use the video screen, printer, cassette recorder, or disk drive.

### THE DEBUG MODE

The Debug mode, which is entered from the Writer/Editor mode by typing "**BUG**", allows you to alter or execute the **Object Program** that has been assembled. It also allows you to enter and change data tables that the **Object Program** uses. The prompt shown on the video display when you are in this mode is the word, **DEBUG**.

```
EDIT  
BUG       You type BUG to enter DEBUG
DEBUG
```

The command to return to the Writer/Editor mode from the Debug mode is the letter, **X**.

```
EDIT  
BUG                        ← Type BUG to enter Debug mode
DEBUG
X                        ← Type X to leave Debug
EDIT                      ← Now you're back in the
     ■ Writer/Editor mode
```

To demonstrate the commands used in the Debug mode, enter the following program.
Now assemble the program.

```
0000 10 *= $1000
1000 A000 20 LDY #0
1002 A900 30 LDA #0
1004 18 40 LOOP CLC
1005 C8 50 INY
1006 6901 60 ADC #1
1008 C003 70 CPY #3
100A D0F8 80 BNE LOOP
90 END
```
Now, go to the Debug mode.

There are three ways to execute a program. The method that you choose will depend on whether you want to display the steps as they are executed or not.

1. To execute the program without a display, type: G1000 and press the RETURN key.

2. To TRACE a program, type T1000 and press RETURN. The display will show each instruction as it is executed along with the contents of the registers after the instruction has been executed.
3. To SINGLE STEP through a program, type S1000 and press RETURN. The display will show the results of the first instruction. To continue the single step procedure, type the letter S and press the RETURN key for each successive instruction that you want executed.

```
DEBUG
S1000
1000 A0 00 LDY #$00
A=03 X=00 Y=00 P=33 S=00
DEBUG
S
1002 A9 00 LDA #$00
A=00 X=00 Y=00 P=33 S=00
DEBUG
S
1004 18 CLC
A=00 X=00 Y=00 P=32 S=00
```

4. To DISPLAY the contents of memory, type the letter D followed by the address of the memory location that you want to examine.

(a) To display a single memory location.

```
DEBUG
D1005,1005
```

(b) To display up to 8 consecutive locations

```
DEBUG
D1000
1000 A0 00 A9 00 18 C8 69 01
```

(c) To display more than 8 consecutive locations, type D, the starting address, a comma, and the ending address.
(d) Notice that in b and c above, memory was displayed with a block of 8 locations per full line. This will result when the requested starting location ends in 0 or 8 (i.e., 1000, 1008, 1010, 1018, etc.) However, if the starting address does not end in 0 or 8, a block containing fewer than 8 values will be displayed on the first line.

Examples:

5. You can LIST a block of memory with the contents DISASSEMBLED into assembly language mnemonics. Type L followed by the starting address or the starting and ending address, just as you do for the display command.

(a) For only one instruction:
(b) For several instructions:

```
DEBUG
L1004,0
1004  18  CLC
```

Complete instructions starting at 1004 (one byte)

```
LDY #$00
LDA #$00
CLC
INY
```

DEBUG

(c) Since the DEBUGGER starts disassembling from the specified starting location, you must be sure that the specified starting location corresponds to the first byte of some instruction in your program. Otherwise the DEBUGGER is confused, or interprets the code as something different than intended.

(1) Confusion:

```
DEBUG
L1009,0
1009,03  ???
```

A bad address

There is no legal instruction whose code is 03

DEBUG

(2) Misleading address:

```
DEBUG
L1001,0
1001  00  BRK
```

A legal instruction but not intended mnemonic
6. As shown in earlier programs, the contents of memory can be changed.

(a) Changing one address:

```
DEBUG
C1003<05 ← Changes contents of 1003 to 5
```

(b) Changing successive locations. The comma increments the location to be changed.

```
DEBUG
C1000<A9,00,A0 ← Puts A9 in 1000, 00 in 1001, A0 in 1002
DEBUG
C1007<02,,06 ← Puts 02 in 1007, 06 in 1009 but does not change 1008
```

7. The contents of a block of memory can be MOVED to a different memory area.

Example:

<table>
<thead>
<tr>
<th>Original Memory Location</th>
<th>Memory Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>1100</td>
<td>40</td>
</tr>
<tr>
<td>1101</td>
<td>41</td>
</tr>
<tr>
<td>1102</td>
<td>42</td>
</tr>
<tr>
<td>1103</td>
<td>43</td>
</tr>
</tbody>
</table>

The move command is then given.

```
DEBUG
M1150<1100,1103 ← Block to be moved
DEBUG
```

Beginning address to which data is to be moved
Memory now looks like this:

<table>
<thead>
<tr>
<th>Memory Location</th>
<th>Memory Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>1100</td>
<td>40</td>
</tr>
<tr>
<td>1101</td>
<td>41</td>
</tr>
<tr>
<td>1102</td>
<td>42</td>
</tr>
<tr>
<td>1103</td>
<td>43</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>1150</td>
<td>40</td>
</tr>
<tr>
<td>1151</td>
<td>41</td>
</tr>
<tr>
<td>1152</td>
<td>42</td>
</tr>
<tr>
<td>1153</td>
<td>43</td>
</tr>
</tbody>
</table>

8. To VERIFY that data has been moved as desired or that two blocks of memory contain the same data, you can compare two blocks of memory.

Example:
Using the results of item 7.

\[
\text{DEBUG} \quad \text{V1100<1150,1153} \quad \text{Block to be compared to 1100-1103}
\]

\[
\text{DEBUG} \quad \text{If the two blocks match, nothing is displayed}
\]

If 1151 contains 31, not 41:

\[
\text{DEBUG} \quad \text{V1100 1150,1153} \quad \text{Mismatch is displayed}
\]

\[
1151 \quad 31 \quad 1101 \quad 41
\]

Other commands and variations of the ones shown in this chapter are given in the Atari Assembler Manual. You should explore the manual thoroughly and experiment with all commands so that you can use all the facilities of the Assembler Cartridge.
Since this entire chapter is a summary of the Assembler commands, no chapter summary is given. Go right on to the chapter exercises.

EXERCISES

1. The five statement fields of a Source Program are given below. Tell which are always used and which are sometimes used in a Source Program statement.
   (a) Statement number ____________________________________
   (b) Label ____________________________________________
   (c) Op Code Mnemonic __________________________________
   (d) Operand __________________________________________
   (e) Comment __________________________________________

2. The assembler will interpret some numbers as decimal and others as hexadecimal depending on the specified operand. Give the interpretation of the following:
   (a) LDA #$13 ____________________________
   (b) LDY #14 ____________________________

3. The SIZE command is given, and the computer displays:

```
EDIT
SIZE
0700   0880   3C1F
```

The computer is then turned off. It is then turned on and the following commands are given. Fill in the display as shown after the final SIZE command.

```
EDIT
LOMEM 900
EDIT
SIZE
```

```
Exercises 4 through 10 refer to the following program:

```
EDIT
LIST
10   *=$1000
20   CLC
30   LDA #0
40   LOOP ADC #5
50   CMP #$50
60   BNE LOOP
70   END
```

4. Give the value in the accumulator at the end of each of the first five passes through the loop.
   1st ________________  4th ________________
   2nd ________________  5th ________________
   3rd ________________

5. (a) Will the accumulator ever contain the hexadecimal value 50?____________
    (b) If so, how many times will the loop be executed? ________________

6. What would be the command used to list all lines from 30 through 60, inclusive? ________________

7. Give the command that would delete line 20. ________________

8. Give one command that will change the word LOOP in lines 40 and 60 to the word ROUND. ________________

9. Give a command that would renumber the program starting with statement number 100 and numbering each successive statement in increments of 5.

10. Give the command to enter the Debugger mode and show the prompt that results.

```
EDIT
________________________
```

The following exercises assume that the program preceding Exercise 4 has been assembled, and that you are now in the Debugger mode. The assembled Object Program is now stored in the following memory locations.
EXERCISES

<table>
<thead>
<tr>
<th>Memory</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>18</td>
</tr>
<tr>
<td>1001</td>
<td>A9</td>
</tr>
<tr>
<td>1002</td>
<td>00</td>
</tr>
<tr>
<td>1003</td>
<td>69</td>
</tr>
<tr>
<td>1004</td>
<td>05</td>
</tr>
<tr>
<td>1005</td>
<td>C9</td>
</tr>
<tr>
<td>1006</td>
<td>50</td>
</tr>
<tr>
<td>1007</td>
<td>D0</td>
</tr>
<tr>
<td>1008</td>
<td>FA</td>
</tr>
<tr>
<td>1009</td>
<td>00</td>
</tr>
</tbody>
</table>

11. What is the command to:
   (a) execute the program ______________________
   (b) trace the program ______________________
   (c) single step the first instruction __________

12. If the command: D1000 is executed, show the displayed result.

   DEBUG
   D1000

   DEBUG
   •

13. Show the display resulting from the command: L1000,0

   DEBUG
   L1000,0

   DEBUG
   •

14. If you wanted to change the program so that it would add ten (hex 0A) each time through the loop instead of 5, show how to change one memory location to accomplish the change.
15. Show a command to move the program so that the first instruction will be in memory location 1120.
10. 
- EDIT
- BUG
- DEBUG

11. (a) G1000
   (b) T1000
   (c) S1000

12. 
- DEBUG
- D1000
- 1000 18 A9 00 69 05 C9 50 D0
- DEBUG

13. 
- DEBUG
- L1000,0
- 1000 18 CLC
- DEBUG

14. 
- DEBUG
- C1004<0A
- DEBUG

15. 
- DEBUG
- M1120<1000,1009
- DEBUG
Designing a Program

A program is usually written to solve a specific problem. The method of solving the problem should be determined before any attempt is made to write the program to solve it. Since assembly language programming is not interactive like BASIC, a detailed plan is needed for each assembler program.

Sample Problem: Add five pairs of two-digit numbers and store the results.

1. In analyzing the stated problem, you can see that some area of memory is needed to hold the five pairs of numbers that are to be added. These will be referred to as data tables.

2. You must also provide for five results. Since you will be adding two-digit numbers, the sum may be greater than two digits. Therefore, it will be necessary to provide for two memory locations for each sum. This group of memory locations will be called the result table.

3. The addition will be performed on two pairs of numbers at a time with the data being accessed from the data tables and the results stored in a different table.

4. You must allocate the necessary memory locations for the program as well as the data and result tables.

NOTICE that you do not know at this time how long the assembled assembled program will be. Therefore, you must estimate where the starting location of the data will be. We allowed lots of room for the program.

Now that you have stated the problem in realistic terms, you should think through the program in functional blocks.
Think about Block A. You could load the data into the correct memory locations using the Assembler Cartridge in the Debug mode. You could also load the data tables from the program itself. The latter method consumes time and memory space, so we will use the first method. It is more direct, is easily changed (if needed), keeps the program simple, and will enable you to use another Assembler Cartridge capability. Therefore, your program will be reduced to Block B.

Let's now expand Block B into functional parts.

---

**Figure 8-1. Memory Blocks Used**

1113

- Block of 10 memory locations for results

110A

- Block of 5 memory locations for 2nd numbers

1109

- Block of 5 memory locations for 1st numbers

1105

- Block of ? memory locations for the assembled program

1104

1000

**Figure 8-2. Functional Blocks**

A. LOAD the data into memory

B. ADD the data and STORE the results in memory

**Figure 8-3. Block B Parts**

Block B

1. Load one 2-digit number into the accumulator.
2. Add the second 2-digit number.
3. Store the result.
4. Go back and repeat until all 5 sums are performed.
The Absolute Indexed Addressing mode will be the most efficient way to load, add and store the numbers. You can use both X and Y registers to index the instructions. The program might consist of the following.

ADD FIVE PAIRS OF NUMBERS

LDY #0  Load the Y register with an initial value of zero. This will index the load and add instructions as well as serving as a loop counter.
LDX #0  Load the X register with a value of zero. X will serve to index the pair of stored results.
LOOP CLC  Start of loop. Clear Carry bit in preparation for adding a pair of numbers.
LDA $1100,Y Load the accumulator from memory location 1100+Y.
ADC $1105,Y Add with carry the number in memory location 1105+Y.
STA $110B,X Store the least significant byte of sum in memory location 110B+X.
BCC SKIP If there was no carry, branch to the instruction labeled SKIP.
INC $110A,X If there was a carry, increment memory location 110A+X.
CPY #5 See if Y = 5
BNE LOOP If Y ≠ 5, branch back to instruction labeled LOOP for next pair of numbers.
END  End of the program

ABSOLUTE INDEXED ADDRESSING

The load, add, increment, and store instructions are all used in the Absolute Indexed Addressing mode. In this mode, a base address is specified. The value contained in either the X or Y register is added to the base address to obtain the memory address actually used.

Examples:
Suppose the Y register holds the value 3, the X register holds 6, and the following values are in memory.
Here is what happens when you execute the following instructions.

LDA $1100,Y would load the accumulator from memory location 1100+3, or 1103

ACCUMULATOR

ADC $1105,Y would add the value 78 to the value in memory location 1105+3, or 1108 (value is 01)

ACCUMULATOR

STA $110B,X would store the value in the accumulator into memory location 110B+6, or 1111

MEMORY 1111

The value for X and Y is originally loaded as zero. The Y value is incremented once each time through the loop, and the X register is incremented twice each time. Thus, a new value is loaded, a new value is added, and the result is stored in a new pair of locations.

<table>
<thead>
<tr>
<th>Y=</th>
<th>X=</th>
<th>Loaded value from</th>
<th>Added value from</th>
<th>Result stored in</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1100</td>
<td>1105</td>
<td>110A and 110B</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>1101</td>
<td>1106</td>
<td>110C and 110D</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>1102</td>
<td>1107</td>
<td>110E and 110F</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>1103</td>
<td>1108</td>
<td>1110 and 1111</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>1104</td>
<td>1109</td>
<td>1112 and 1113</td>
</tr>
<tr>
<td>5</td>
<td>A</td>
<td>END OF PROGRAM</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 8-4. How Memory is Used

Now you are ready to use the computer with the Assembler Cartridge.
USING THE ADD FIVE PAIRS OF NUMBERS PROGRAM

With the Assembler Cartridge in the left slot of the Atari, you might first input the program from the Edit mode.

```
EDIT
10 *=1000
20 LDY #0
30 LDX #0
40 LOOP CLC
50 LDA $1100,Y
60 ADC $1105,Y
70 STA $110B,X
80 BCC SKIP
90 INC $110A,X
100 SKIP INX
110 INX
120 INY
130 CPY #5
140 BNE LOOP
150 END
```

You then assemble the program.

```
150 END
ASM
0000 10 *= $1000
1000 A000 20 LDY $0
1002 A200 30 LDX #0
1004 18 40 LOOP CLC
1005 B90011 50 LDA $1100,Y
1008 790511 60 ADC $1105,Y
```
Next, you must place the data in the appropriate tables. Notice that you are putting zeros in the result table. This must be done to make sure the locations used for the high-order bytes of the result are filled with zeros originally. Since you are using the INCrement instruction to place the carry bit into the high-order byte of the result, you must make sure that a zero is in the memory to begin with. You must be in the Debug mode to enter the data.

The DEBUG Command for changing values in memory is:

CXXXX<yy  where the X's represent hexadecimal digits in the memory address to be changed. The y's represent the one- or two-digit hex number to be placed in the specified memory location.
To change a number of consecutive memory locations the command takes the following form.

\[ \text{CXXXX<aa,bb,cc,dd} \quad \text{where XXXX is the first memory location to be changed. aa,bb,etc., represent hex values to be inserted into consecutive memory locations starting at XXXX. Up to 16 data values may be inserted at one time. Then a new starting address is needed.} \]

\[ \text{DEBUG} \]
\[ \text{C1100<12,34,56,78,9A,BC,DE,F0,123,0,0} \quad \text{Type in 16 codes} \]
\[ \text{DEBUG} \]
\[ \text{C1110<0,0,0,0} \quad \text{Then press RETURN} \]
\[ \text{DEBUG} \quad \text{Press RETURN again after entering four more zeros} \]

There, the data is all in place, and the program has been assembled. As one final check, use the DISPLAY Command in the Debug mode to look at the program and the data.

Display the program.

\[ \text{DEBUG} \]
\[ \text{D1000,1019} \quad \text{Program} \]
\[ 1000 \text{ A0 00 A2 00 18 B9 00 11} \quad \text{Eight values displayed on each line} \]
\[ 1008 \text{ 79 05 11 9D 0B 11 90 03} \]
\[ 1010 \text{ FE 0A 11 E8 E8 C8 C0 05} \]
\[ 1018 \text{ D0 EA} \]

Check the hex codes against those on page 148. If they agree, then check the data.
USING THE ADD FIVE PAIRS OF NUMBERS PROGRAM

---

DEBUG
D1100,1113

1100 12 34 56 78 9A BC DE F0
1108 01 23 00 00 00 00 00 00 00 00

DEBUG

OK, now you can run the program.

DEBUG
G1000
101A

A=BD X=0A Y=05 P=33 S=00

DEBUG

Finally, check the results by displaying the result table.

DEBUG
D110A,110B

110A 00 CE ↔ 12+BC = 00CE
DEBUG
D110C,110D

110C 01 12 ↔ 34+DE = 0112
DEBUG
D110E,110F

110E 01 46 ↔ 56+F0 = 0146
DEBUG
D1110,1111

1110 00 79 ↔ 78+01 = 0079
DEBUG
D1112,1113

1112 00 BD ↔ 9A+23 = BD

---

data tables
Suppose the sample problem changes. Instead of adding the ten numbers in pairs, you might want to add all ten numbers together.

**Sample Problem 2: Add ten two-digit numbers and store the result.**

Once again, expand Block B of the Functional Blocks of Figure 8-2. This time we want to:

1. Load the first 2-digit number into the accumulator.
2. Add the temporary total (originally set to 0) and store the new temporary total.
3. Load the next 2-digit number.
4. Repeat steps 2 and 3 until all ten numbers have been added.

**Figure 8-5. Block B Parts—Add Ten Program**

Notice the differences between the solutions to Sample Problem 1 and Sample Problem 2.

---

### Solution to Sample Problem 2

<table>
<thead>
<tr>
<th>LDY</th>
<th>#0</th>
<th>LOOP</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>CLC</td>
</tr>
<tr>
<td>LDA</td>
<td>$1100,Y</td>
<td></td>
</tr>
<tr>
<td>ADC</td>
<td>$110B</td>
<td></td>
</tr>
<tr>
<td>STA</td>
<td>$110B</td>
<td></td>
</tr>
<tr>
<td>BCC</td>
<td>SKIP</td>
<td></td>
</tr>
<tr>
<td>INC</td>
<td>$110A</td>
<td></td>
</tr>
<tr>
<td>INY</td>
<td></td>
<td>SKIP</td>
</tr>
<tr>
<td>CPY</td>
<td>#0A</td>
<td></td>
</tr>
<tr>
<td>BNE</td>
<td>LOOP</td>
<td>END</td>
</tr>
</tbody>
</table>

### Solution to Sample Problem 1

<table>
<thead>
<tr>
<th>LDY</th>
<th>#0</th>
<th>LOOP</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>CLC</td>
</tr>
<tr>
<td>LDA</td>
<td>$1100,Y</td>
<td></td>
</tr>
<tr>
<td>ADC</td>
<td>$1105,Y</td>
<td></td>
</tr>
<tr>
<td>STA</td>
<td>$110B,X</td>
<td></td>
</tr>
<tr>
<td>BCC</td>
<td>SKIP</td>
<td></td>
</tr>
<tr>
<td>INC</td>
<td>$110A,X</td>
<td></td>
</tr>
<tr>
<td>INX</td>
<td></td>
<td>SKIP</td>
</tr>
<tr>
<td>CPY</td>
<td>#05</td>
<td>BNE</td>
</tr>
<tr>
<td>BNE</td>
<td>LOOP</td>
<td>END</td>
</tr>
</tbody>
</table>

You can see that the solution to Sample Problem 2 uses fewer instructions than the solution to Sample Problem 1. The temporary and final sum is stored in
USING THE ADD TEN PROGRAM

memory locations 110A and 110B. The temporary sum is added to each two-digit number each time the loop is executed, and the new sum is stored back in the same location. For that reason, the X register is not needed in the second solution.

This is how the memory locations are used as each two-digit number is added.

<table>
<thead>
<tr>
<th>Loop Number</th>
<th>Accumulator Loaded</th>
<th>Temporary Sum</th>
<th>110A Stored</th>
<th>110B Stored</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>12</td>
<td>00+12</td>
<td>00</td>
<td>12</td>
</tr>
<tr>
<td>2</td>
<td>34</td>
<td>12+34</td>
<td>00</td>
<td>46</td>
</tr>
<tr>
<td>3</td>
<td>56</td>
<td>46+56</td>
<td>00</td>
<td>9C</td>
</tr>
<tr>
<td>4</td>
<td>78</td>
<td>9C+78</td>
<td>01</td>
<td>14</td>
</tr>
<tr>
<td>5</td>
<td>9A</td>
<td>14+9A</td>
<td>01</td>
<td>AE</td>
</tr>
<tr>
<td>6</td>
<td>BC</td>
<td>AE+BC</td>
<td>02</td>
<td>6A</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
</tbody>
</table>

Figure 8-6. Memory Use in Add Ten Program

USING THE ADD TEN PROGRAM

The steps to use the program follow previous directions. We will not give the steps in detail, but we will show the results.

In the Edit mode:

1. Enter the Program.
2. Assemble the Program.

In the Debug mode:

3. Load the data.
4. Run the program.
5. Display results.
1. Enter the Program.

```
10  *=$1000
20  LDY #0
30  LOOP CLC
40  LDA $1100,Y
50  ADC $110B
60  STA $110B
70  BCC SKIP
80  INC $110A
90  SKIP INY
100 CPY #10
110 BNE LOOP
120 END
```

Note that we can enter the number of steps (10) as:

- #10 for decimal values
- #$10 for hexadecimal values

This indicates hex.

2. Assemble the program.

```
ASM
0000  10  *=  $1000
1000 A000  20  LDY #0
1002  18  30  LOOP CLC
1003 B90011  40  LDA $1100,Y
1006 6D0B11  50  ADC $110B
1009 8D0B11  60  STA $110B
100C 9003  70  BCC SKIP
100E EE0A11  80  INC $110A
1011 C8  90  SKIP INY
1012 C00A  0100  CPY #10
1014 D0EC  0110  BNE LOOP
```

#10 in decimal

Assembled as a hex value, OA
3. Load the data in the Debug mode.

```
BUG
DEBUG
C1100<12,34,56,78,9A,BC,DE,F0,1,23,0,0,
```

Results

Data

4. Run the program.

```
DEBUG
G1000
1016     A=5C X=00 Y=0A P=33 S=00
```

5. Display the result.

```
DEBUG
D110A,110B
110A  04  5C
```

Result in locations 110A and 110B

---

A VARIATION OF THE ADD TEN PROGRAM

There is usually more than one way to program a solution to a problem, and a good programmer is continually seeking ways to make his programs more efficient. Let's look at a slight variation of the last program that eliminates one two-byte instruction.

THE TWO ADD TEN PROGRAMS

<table>
<thead>
<tr>
<th>First Program</th>
<th>Second Program</th>
</tr>
</thead>
<tbody>
<tr>
<td>10  *=S1000</td>
<td>10  *=S1000</td>
</tr>
<tr>
<td>20  LDY #0</td>
<td>20  LDY #10</td>
</tr>
<tr>
<td>30  LOOP CLC</td>
<td>30  LOOP CLC</td>
</tr>
<tr>
<td>40  LDA $1100,Y</td>
<td>40  LDA $1100,Y</td>
</tr>
<tr>
<td>50  ADC $110B</td>
<td>50  ADC $110C</td>
</tr>
<tr>
<td>60  STA $110B</td>
<td>60  STA $110C</td>
</tr>
<tr>
<td>70  BCC SKIP</td>
<td>70  BCC SKIP</td>
</tr>
<tr>
<td>80  INC $110A</td>
<td>80  INC $110B</td>
</tr>
<tr>
<td>90  SKIP INY</td>
<td>90  SKIP DEY</td>
</tr>
<tr>
<td>100 CPY #10</td>
<td>100 BNE LOOP</td>
</tr>
<tr>
<td>110 BNE LOOP</td>
<td>110 END</td>
</tr>
<tr>
<td>120 END</td>
<td>*** omitted</td>
</tr>
</tbody>
</table>

*** denotes program changes
The second program counts downward from ten to zero as the loop is executed over and over. Because of this, the results and data tables are placed in slightly different memory locations. Each value is offset by one location.

One instruction, CPY #10, is omitted from the second program. This is possible because the zero bit of the Processor Status Register is set when the Y register is decremented to zero after the last time through the loop. Therefore, the BNE instruction can be used directly following the DEY instruction with no comparison necessary. Since you were counting upwards from 0 through 10 in the first program, the value in the Y register had to be compared to ten before the BNE instruction could be used.

A comparison of the assembled programs shows that the second program is two bytes shorter. It ends at memory location 1013, whereas the first program ended at memory location 1015.

The data tables are accessed similarly for the two programs, but the memory addresses are offset in the second program due to the way that the Y register is decremented.

First Program

<table>
<thead>
<tr>
<th>Location</th>
<th>Instruction</th>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>10</td>
<td>$1000</td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>A000</td>
<td>#0</td>
<td></td>
</tr>
<tr>
<td>1002</td>
<td>18</td>
<td>LOOP</td>
<td>CLC</td>
</tr>
<tr>
<td>1003</td>
<td>B90011</td>
<td>LDA $1100,Y</td>
<td></td>
</tr>
<tr>
<td>1006</td>
<td>6D0B11</td>
<td>ADC $110B</td>
<td></td>
</tr>
<tr>
<td>1009</td>
<td>8D0B11</td>
<td>STA $110B</td>
<td></td>
</tr>
<tr>
<td>100C</td>
<td>9003</td>
<td>BCC SKIP</td>
<td></td>
</tr>
<tr>
<td>100E</td>
<td>EE0A11</td>
<td>INC $110A</td>
<td></td>
</tr>
<tr>
<td>1011</td>
<td>C8</td>
<td>SKIP INY</td>
<td></td>
</tr>
<tr>
<td>1012</td>
<td>C00A</td>
<td>CPY #10</td>
<td></td>
</tr>
<tr>
<td>1014</td>
<td>D0ED</td>
<td>BNE LOOP</td>
<td></td>
</tr>
<tr>
<td>0120</td>
<td>END</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

* denotes $1000

---

This instruction unnecessary in second program
A VARIATION OF THE ADD TEN PROGRAM

Second Program

<table>
<thead>
<tr>
<th>Address</th>
<th>Opcode</th>
<th>Immediate</th>
<th>Memory</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>10</td>
<td>*$</td>
<td>1000</td>
<td></td>
<td>initial value 10 instead of zero</td>
</tr>
<tr>
<td>1000</td>
<td>A00A</td>
<td>20</td>
<td>10</td>
<td>LDY #10</td>
<td>different memory locations used</td>
</tr>
<tr>
<td>1002</td>
<td>18</td>
<td>30 LOOP</td>
<td>CLC</td>
<td></td>
<td>counting down</td>
</tr>
<tr>
<td>1003</td>
<td>B90011</td>
<td>40</td>
<td>LDA</td>
<td>$1100,Y</td>
<td></td>
</tr>
<tr>
<td>1006</td>
<td>6D0C11</td>
<td>50</td>
<td>ADC</td>
<td>$110C</td>
<td></td>
</tr>
<tr>
<td>1009</td>
<td>8D0C11</td>
<td>60</td>
<td>STA</td>
<td>$110C</td>
<td></td>
</tr>
<tr>
<td>100C</td>
<td>9003</td>
<td>70</td>
<td>BCC</td>
<td>SKIP</td>
<td></td>
</tr>
<tr>
<td>100E</td>
<td>EE0B11</td>
<td>80</td>
<td>INC</td>
<td>$110B</td>
<td></td>
</tr>
<tr>
<td>1011</td>
<td>88</td>
<td>90 SKIP</td>
<td>DEY</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1012</td>
<td>D0EE</td>
<td>0100</td>
<td>BNE</td>
<td>LOOP</td>
<td></td>
</tr>
</tbody>
</table>

0110 END

First Program Tables

<table>
<thead>
<tr>
<th>Data</th>
<th>Memory</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>1100</td>
<td>12</td>
</tr>
<tr>
<td>34</td>
<td>1101</td>
<td>34</td>
</tr>
<tr>
<td>56</td>
<td>1102</td>
<td>56</td>
</tr>
<tr>
<td>78</td>
<td>1103</td>
<td>78</td>
</tr>
<tr>
<td>9A</td>
<td>1104</td>
<td>9A</td>
</tr>
<tr>
<td>BC</td>
<td>1105</td>
<td>BC</td>
</tr>
<tr>
<td>DE</td>
<td>1106</td>
<td>DE</td>
</tr>
<tr>
<td>F0</td>
<td>1107</td>
<td>F0</td>
</tr>
<tr>
<td>01</td>
<td>1108</td>
<td>01</td>
</tr>
<tr>
<td>23</td>
<td>1109</td>
<td>23</td>
</tr>
<tr>
<td>110A</td>
<td>Low-byte result</td>
<td></td>
</tr>
<tr>
<td>110B</td>
<td>High-byte result</td>
<td></td>
</tr>
<tr>
<td>110C</td>
<td>High-byte result</td>
<td></td>
</tr>
</tbody>
</table>

Data accessed in this order

Second Program Tables

<table>
<thead>
<tr>
<th>Data</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Not used</td>
<td>1100</td>
</tr>
<tr>
<td>12</td>
<td>1101</td>
</tr>
<tr>
<td>34</td>
<td>1102</td>
</tr>
<tr>
<td>56</td>
<td>1103</td>
</tr>
<tr>
<td>78</td>
<td>1104</td>
</tr>
<tr>
<td>9A</td>
<td>1105</td>
</tr>
<tr>
<td>BC</td>
<td>1106</td>
</tr>
<tr>
<td>DE</td>
<td>1107</td>
</tr>
<tr>
<td>F0</td>
<td>1108</td>
</tr>
<tr>
<td>01</td>
<td>1109</td>
</tr>
<tr>
<td>23</td>
<td>110A</td>
</tr>
<tr>
<td>110B</td>
<td>Low-byte result</td>
</tr>
<tr>
<td>110C</td>
<td>High-byte result</td>
</tr>
</tbody>
</table>

Data accessed in reverse order

Figure 8-7. Data Tables
Naturally, executing the second program produces the same result as the first even though the numbers are added in the reverse order. One new instruction was introduced in the second program.

**DEY (DEcrement the Y register)**

Op Code 88

It is a one-byte instruction used only in the Implied Addressing mode. It works just the opposite as INY (Increment Y). The value in the Y register is decreased (or decremented) by one when the instruction is executed.

Some programmers might object to the changes in the locations of the data tables. This change was done so that the branch instruction could make direct use of the condition of the zero bit following the DEY instruction. If the data was located as in the first program, the following sequence would result on the last pass through the loop.

<table>
<thead>
<tr>
<th>Old Y</th>
<th>Instruction Executed</th>
<th>Present Sum</th>
<th>Memory Last Accessed</th>
<th>New Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SKIP DEY</td>
<td>04 4A</td>
<td>1101</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>BNE LOOP</td>
<td>04 4A</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Since the zero bit is set, the branch would not be taken back for the last number. Therefore, the final sum (044A) would be incorrect.

**YET ANOTHER VARIATION**

Another variation could be used to keep the data tables in their original position, but it would require that you change the order of the instructions and move the CPY instruction back into the program.
SUMMARY

You can see that you have a considerable amount of freedom in designing a program. Any program is a good program if it solves the problem for which it was designed. Usually, good programs can be improved for efficiency and quickness of execution. Programming provides for large amounts of individualism and creativity. Enjoy it!

SUMMARY

- In this chapter you've learned how to design an assembly language program to solve a specific problem. The suggested steps are:
  
  1. Analyze the problem by deciding how the solution will be accomplished generally.
  2. Decide what memory locations will be used for data and results.
  3. Think through your solution in functional blocks.
  4. Expand each block into steps that the computer can handle.
  5. Write the program in terms of the expanded functional blocks.

- You learned how to handle data with tables accessed by the Absolute Indexed Addressing mode to load, add, and store data. Instructions used in this mode were:
  
  1. LDA $1100, Y  Load data into the accumulator from memory (location 1100+the value in the Y register)
  2. ADC $1105, Y  Add the data from memory (location 1105+the value of Y register) to the data in the accumulator
3. STA $110B,X  Store the accumulator’s contents into memory (location 110B+the value in the X register)
4. INC $110A,X  Increment the content of memory (location 110A+the value in the X register)

- You learned to enter data with the change memory command in the Debug mode.

Examples:
C1100<12,34,56,78,9A,BC,DE,F0,1,23,0,0
,0,0,0,0  16 items entered consecutively starting at memory location 1100
C1110<0,0,0,0  4 more items entered consecutively starting at memory location 1110

- You learned that an index register can count down as well as up.
  Down → DEY Decrement the Y register
  Up →INY Increment the Y register

- You learned to change a program to solve a new problem and to make variations of existing programs.
- You learned that there may be several ways to program a solution to a given problem.

EXERCISES

1. In which mode (EDIT, ASM, or DEBUG) were the data entries loaded into memory for the programs in this chapter? __________________________

2. What addressing mode is used for the following instruction?
  STA $1500,Y
  __________________________

3. If the accumulator holds a value of 10, and the Y register holds 5, into what memory location will the 10 be stored if the instruction in Exercise 2 is executed? __________________________
EXERCISES

4. The following values are in the computer as shown.

<table>
<thead>
<tr>
<th>Accumulator</th>
<th>Memory</th>
<th>Data</th>
<th>Y Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1100</td>
<td></td>
<td>D5</td>
<td></td>
</tr>
<tr>
<td>1101</td>
<td></td>
<td>33</td>
<td></td>
</tr>
<tr>
<td>1102</td>
<td></td>
<td>22</td>
<td></td>
</tr>
<tr>
<td>1103</td>
<td></td>
<td>24</td>
<td></td>
</tr>
</tbody>
</table>

Then the following sequence of instructions is executed. Fill in the data in the appropriate blanks below.

LDA $1100,Y
ADC $1101,Y
INY
STA $1100,Y

<table>
<thead>
<tr>
<th>Accumulator</th>
<th>Memory</th>
<th>Data</th>
<th>Y Register</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1100</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1101</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1102</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1103</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

5. If the instructions of Exercise 4 were executed again using your results for Exercise 4, what would the new values be?

<table>
<thead>
<tr>
<th>Accumulator</th>
<th>Memory</th>
<th>Data</th>
<th>Y Register</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1100</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1101</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1102</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1103</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

6. Suppose you are in the Debug mode, and you execute the command:

C1100<A4,F3,C5,19

Show what would be in the following memory locations.

<table>
<thead>
<tr>
<th>Memory</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1100</td>
<td></td>
</tr>
<tr>
<td>1101</td>
<td></td>
</tr>
<tr>
<td>1102</td>
<td></td>
</tr>
<tr>
<td>1103</td>
<td></td>
</tr>
<tr>
<td>1104</td>
<td></td>
</tr>
</tbody>
</table>

Hint: One of the memory locations is not changed by the command. Place an XX in that location.
7. Give one DEBUG command to display all four memory locations of Exercise 5.

8. Fill in the blanks in the table below for the following program.

```
10    *=1000
20    LDY #0
30    LOOP CLC
40    LDA $1101,Y
50    ADC $1100,Y
60    INY
70    STA $1100,Y
80    CPY #4
90    BNE LOOP
100   END
```

<table>
<thead>
<tr>
<th>End of Loop</th>
<th>Y Register</th>
<th>Accumulator</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>00</td>
<td>01 02 03 04 05</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

9. Show the hex and decimal values for the following Assembler notations.

- #24 _______hex = _______decimal
- #$24 _______hex = _______decimal

**ANSWERS**

1. DEBUG
2. Absolute Indexed Addressing
3. 1505 (hex)
4. Accumulator | Memory | Data | Y Register
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>55</td>
<td>1101</td>
<td>D5</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>1101</td>
<td>33</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1102</td>
<td>55</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1103</td>
<td>24</td>
<td></td>
</tr>
</tbody>
</table>
5. Accumulator

<table>
<thead>
<tr>
<th>Memory</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1100</td>
<td>D5</td>
</tr>
<tr>
<td>1101</td>
<td>33</td>
</tr>
<tr>
<td>1102</td>
<td>55</td>
</tr>
<tr>
<td>1103</td>
<td>79</td>
</tr>
</tbody>
</table>

6. Memory Data

<table>
<thead>
<tr>
<th>Memory</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1100</td>
<td>A4</td>
</tr>
<tr>
<td>1101</td>
<td>F3</td>
</tr>
<tr>
<td>1102</td>
<td>C5</td>
</tr>
<tr>
<td>1103</td>
<td>19</td>
</tr>
<tr>
<td>1104</td>
<td>XX</td>
</tr>
</tbody>
</table>

7. D1100,1103

8.

<table>
<thead>
<tr>
<th>At End of loop #</th>
<th>Y Register</th>
<th>Accumulator</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>00</td>
<td>01 02 03 04 05</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>03</td>
<td>01 03 03 04 05</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>06</td>
<td>01 03 06 04 05</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>0A</td>
<td>01 03 06 0A 05</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>0F</td>
<td>01 03 06 0A OF</td>
</tr>
</tbody>
</table>

9. #24  18 hex = 24 decimal
    #$24  24 hex = 36 decimal
Chapter 9

Addition and Subtraction

You are now familiar with the way that computers add and subtract numbers. Since the size of registers and memory is limited to 8 binary digits, addition and subtraction must be performed with pieces of data of the same size. Keep in mind that the largest decimal integer that can be expressed in 8 bits (one byte) is 255.

You learned in Chapter 6 to test for the Carry bit to see if the sum of two 8-bit numbers was larger than the accumulator could hold.

Example:

<table>
<thead>
<tr>
<th>hex</th>
<th>binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>7D</td>
<td>01111101</td>
</tr>
<tr>
<td>+A4</td>
<td>+10100100</td>
</tr>
<tr>
<td>121</td>
<td>00100001</td>
</tr>
</tbody>
</table>

extra bit
sets the Carry flag

In this way you can tell when the sum of two 8-bit numbers is larger than can be held in a register or memory location. However at some time you will want to add numbers that are larger than 255. To accomplish this, the computer must handle the number in more than one byte. The computer can add a byte from each of two numbers, store the result, and then add the second bytes of the two numbers. Then you can display the two parts as one complete number.
Example:

<table>
<thead>
<tr>
<th>hex</th>
<th>binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>6E61</td>
<td>01101110011000001</td>
</tr>
<tr>
<td>+219C</td>
<td>+0010000110011100</td>
</tr>
<tr>
<td>8FFD</td>
<td></td>
</tr>
</tbody>
</table>

add these bytes first and store

then add these bytes and store in different location

The complete result is 8FFD

TWO-BYTE ADDITION

A paper-and-pencil addition of two-byte numbers will help us decide how to write a program to perform the operation on the computer. Suppose we want to add these two-byte hex numbers.

55A4 = 01010101 10100100
+3CB3 = 00111100 10110011

The binary addition by bytes →

1 01010111

Extra Least Significant Byte (LSB) is in the Carry bit

Then MSB 01010101 10100100
+00111100 10110011
+ 1 01010111

Most Significant Byte (MSB) of result

← from LSB

10010010
Notice that in this example a carry results from the addition of the Least Significant Bytes. The ADC (Add with Carry) instruction will automatically add in this carry bit to the sum of the Most Significant Bytes. Therefore, it appears that the two-byte numbers are summed by adding first, the Least Significant Bytes and second, the Most Significant Bytes.

1. Add and store the least significant bytes
2. Add and store the most significant bytes

*Figure 9-1. Functional Blocks for Addition*

It looks like a very straightforward program. Set up a block of memory to store the bytes that are to be added and the bytes of the result of the addition. Since there are two bytes for each number, set aside 6 bytes.

The first two-byte number to be added is stored in memory location 1100 (Least Significant Byte) and 1101 (Most Significant Byte). The second number is stored in memory location 1102 (Least Significant Byte) and 1103 (Most Significant Byte). These values must all be loaded before the program is executed. You know how to do this in the Debugger mode.
Break down the functional blocks into detailed steps and use the mnemonic instructions that you have become familiar with.

Block 1
- LDA $1000
- ADC $1002
- STA $1004

- Absolute addressing mode from memory 1000 (LSB of 1st number)
- Add LSB of second number
- Store accumulator into appropriate memory location

Block 2
- LDA $1001
- ADC $1003
- STA $1004

- Load MSB of first number
- Add MSB of second number
- Store accumulator into appropriate memory location

Figure 9-3. Detailed Steps for Functional Blocks

One thing that you must remember about the ADC instruction is that the Carry bit (if set) is added to the sum. When you add the Least Significant Bytes, you must be sure that the carry bit is off (reset to zero). Therefore, a CLC (Clear Carry bit) is needed before the Least Significant Bytes are added.

If you draw a flowchart of the operations that must be performed it will help you write the program step by step.

Figure 9-4. Flowchart for Two-Byte Addition
If a program is short, like this one, and you have a detailed flowchart, as in Figure 9-4, you can write the program directly from the flowchart in the Writer/Editor mode.

```
EDIT
NUM 100,10

100  *= $1000
110  CLC
120  LDA $1100
130  ADC $1102
140  STA $1105
150  LDA $1101
160  ADC $1103
170  STA $1104
180  END
190  

Notice that LOA, ADC, and STA are all used in the Absolute Addressing mode this time. You can use that mode since you know exactly where each piece of data is in memory or where you wish to put it in memory.

Once the program has been entered, assemble it.

```
0000   100   *=   $1000
1000   18    110   CLC
1001   AD0011 120   LDA $1100
1004   6D0211 130   ADC $1102
1007   8D0511 140   STA $1105
100A   AD0111 150   LDA $1101
100D   6D0311 160   ADC $1103
Now, the program is assembled. It resides in memory locations 1000 through 1012. Don’t forget the data. Use the data shown in the paper-and-pencil example so that you can check the results:

\[
\begin{align*}
55A4 \\
+3CB3 \\
\hline
9257
\end{align*}
\]

Place the four pieces of data in the four specified memory locations.

<table>
<thead>
<tr>
<th>MEMORY</th>
<th>Data</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1100</td>
<td>A4</td>
<td>LSB 1st number</td>
</tr>
<tr>
<td>1101</td>
<td>55</td>
<td>MSB 1st number</td>
</tr>
<tr>
<td>1102</td>
<td>B3</td>
<td>LSB 2nd number</td>
</tr>
<tr>
<td>1103</td>
<td>3C</td>
<td>MSB 2nd number</td>
</tr>
</tbody>
</table>

*Figure 9-5. Data Used in Example*

You are now ready to go to the Debugger mode to enter the data into the memory locations shown in Figure 9-5. Remember how to do it with the Change Memory command?

```
EDIT
BUG
DEBUG
C1100<A4,55,B3,3C
```

All four pieces in consecutive memory locations

Now execute the program and display memory locations 1104 and 1005 to see the sum.
Before leaving the Two-Byte Addition Program, try adding the pairs of numbers given in Figure 9-6. Use the Change Memory command to put the numbers in memory. Place your answers in the appropriate boxes in Figure 9-6. Check your results against the answers given in the answers to chapter exercises.

<table>
<thead>
<tr>
<th>First Number</th>
<th>Second Number</th>
<th>Sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>000A</td>
<td>000B</td>
<td></td>
</tr>
<tr>
<td>13C5</td>
<td>0F24</td>
<td></td>
</tr>
<tr>
<td>6666</td>
<td>333E</td>
<td></td>
</tr>
<tr>
<td>37AB</td>
<td>A09D</td>
<td></td>
</tr>
<tr>
<td>E111</td>
<td>2000</td>
<td></td>
</tr>
</tbody>
</table>

Figure 9-6. Two-Byte Addition Exercises

Beware of the last exercise in Figure 9-6. Can you tell what result will be displayed after the program has been executed?

TWO PROGRAMS IN MEMORY

Suppose you wish to have an addition program and a subtraction program in the computer at the same time. Only three changes are necessary to convert the addition program now in the computer, to a two-byte subtraction program. It would be time consuming to rewrite the program, but you can quickly copy the addition program to a different block of memory with the DEBUGGER's
Move command. Then you can modify the original addition program to perform subtraction instead.

You have probably noticed that a BRK instruction (Op Code 00) is inserted at the end of each program when it is assembled. The END statement of the Source Program creates the BRK statement. The BRK instruction should be included as you move the original program.

If you don’t remember how to use the Move command, see Chapter 6 or follow this procedure:

```
DEBUG
M1020<1000,1013
DEBUG
```

← Answer from addition

← Move
Verify that the two programs are the same before going on to be sure that the move was made correctly.

\[
\begin{array}{c}
\text{M1020}<1000,1013 \\
\text{DEBUG} \\
\text{V1020}<1000,1013 \\
\text{DEBUG} \\
\end{array}
\]

No mismatches displayed therefore no mistakes

Now, you're ready to change the original program so that it will subtract instead of add.

TWO-BYTE SUBTRACTION

Subtraction of two-byte numbers is performed in a similar manner. The Two-Byte Addition Program can be modified by three minor changes:

- At memory location 1000:
  change 18 (CLC) to 38 (SEC) → Set the carry bit instead of clearing it
- At memory locations 1004 and 100D:
  change 6D (ADC) to ED (SBC) → Subtract with borrow instead of add with carry

The change at 1000 sets the carry bit in preparation for the subtraction just as it did the One-Byte Subtraction Program of Chapter 4. The add with carry instruction (ADC) is replaced by the subtract with borrow (SBC). The subtract instruction is used in the Absolute Addressing mode. The address which contains the number to be subtracted follows the Op Code, as it followed the ADC instruction in the addition program. Thus, the Least Significant Byte of the value to be subtracted is contained in address 1102. The Most Significant Byte of the value to be subtracted is contained in address 1103.

The changes:

\[
\begin{array}{c}
\text{V1020}<1000,1013 \\
\text{DEBUG} \\
\text{C1000}<38 \\
\text{C1004}<\text{ED} \\
\text{C100D}<\text{ED} \\
\end{array}
\]
You now have two machine language programs in the computer. The **Two-Byte Addition Program** was assembled starting at location 1000 but was moved to 1020 through 1032. The **Two-Byte Subtraction Program** is now in locations 1000 through 1013. You can now use either program by choosing the correct memory address for the G command which executes the program starting at the location that you specify.

Before you execute either program, put the original numbers (55A4 and 3CB3) back into memory locations 1100 through 1103.

```
C1100<A4,55,B3,3C ← Be sure you use the
                     correct order
```

You can now use either program.

```
G1000 ← First use subtraction

1013 A=18 X=00 Y=00 P=32 S=00
DEBUG D1104,1105

1104 18 F1 ← Answer: 18F1 for subtraction

DEBUG
```

```
1104 18 F1
DEBUG G1020 ← Now addition

1033 A=92 X=00 Y=00 P=32 S=00
DEBUG D1104,1105 ← Display result

1104 92 57 Answer: 9257 for addition
DEBUG
```
Check the subtraction with paper and pencil to make sure that the subtraction result is correct. We hope you like binary subtraction. There are lots of borrows.

\[
\begin{align*}
55A4 & = \begin{array}{cccccc}
0 & 1 & 0 & 1 & 1 & 0101 1010 0100 \\
\end{array} \\
-3CB3 & = \begin{array}{cccccc}
0 & 0 & 1 & 1 & 0011 1100 1011 0011 \\
\end{array} \\
\hline
0001 & 1000 1111 0001 0001 \\
\hline
= & 1 & 8 & F & 1 \\
\end{align*}
\]

binary \hspace{1cm} hex \hspace{1cm} Yes, it checks

Try the subtraction exercises in Figure 9-7. Watch the result of the last two exercises in the table. Can you explain the results?

<table>
<thead>
<tr>
<th>First Number</th>
<th>Last Number</th>
<th>Difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>FFFF</td>
<td>0112</td>
<td></td>
</tr>
<tr>
<td>76A3</td>
<td>6DCB</td>
<td></td>
</tr>
<tr>
<td>590A</td>
<td>3A1B</td>
<td></td>
</tr>
<tr>
<td>2222</td>
<td>3333</td>
<td></td>
</tr>
<tr>
<td>0000</td>
<td>0004</td>
<td></td>
</tr>
</tbody>
</table>

*Figure 9-7. Two-Byte Subtraction Exercises*

The problem involved with the last two exercises in Figure 9-7 is that you are trying to subtract a positive number from a smaller positive number.

\[
\begin{align*}
2222 & \hspace{1cm} \text{and} \hspace{1cm} 0000 \\
-3333 & \hspace{1cm} \text{and} \hspace{1cm} -0004 \\
\end{align*}
\]

Algebra students know that the result of each of these exercises is a negative number. You actually saw these results displayed on the screen.

EEEF \hspace{1cm} \text{and} \hspace{1cm} FFFC

This means that a method must exist to determine whether a number is positive or negative.

**NEGATIVE NUMBERS**

It is possible to look at the way data is represented in the computer in a different way. If signed numbers (those that are either positive, negative, or zero) are to be represented, the computer must have some way to tell them apart.

Consider an 8-bit block of data as being composed of one sign bit and seven data bits.
1. If the sign position holds a zero, the data is considered to be a positive number.

Examples:

- $01111111 = +127 (64+32+16+8+4+2+1)$
- $01111110 = +126 (64+32+16+8+4+2)$
- $01111101 = +125 (64+32+16+8+4+1)$
- $01111100 = +124 (64+32+16+8+4)$

2. If the sign position holds a one (1), the data is considered to be a negative number.

Examples:

- $10000000 = -128$
- $10000001 = -127$
- $10000010 = -126$
- $10000011 = -125$
- $10000100 = -124$

Zero is considered a positive number by Branch Instructions.
We have learned to interpret positive binary numbers as positive decimal numbers, but what about these negative critters? They don’t look familiar at all. However, it is plain to see that each 8-bit code could represent all the integers from -128 through +127.

<table>
<thead>
<tr>
<th>Binary</th>
<th>Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
<td>0</td>
</tr>
<tr>
<td>00000001</td>
<td>1</td>
</tr>
<tr>
<td>01111110</td>
<td>+126</td>
</tr>
<tr>
<td>01111111</td>
<td>+127</td>
</tr>
<tr>
<td>10000000</td>
<td>-128</td>
</tr>
<tr>
<td>10000001</td>
<td>-127</td>
</tr>
<tr>
<td>10000010</td>
<td>-126</td>
</tr>
<tr>
<td>11111110</td>
<td>-2</td>
</tr>
<tr>
<td>11111111</td>
<td>-1</td>
</tr>
</tbody>
</table>

Let's take a look at the negatives and see if there is any meaningful relationship to their positive counterparts.

Consider the positive number 126 = 01111110.

- Its one's complement = 1000001
- Its two's complement = 10000010

Compare the latter with -126 = 10000010

The binary representation of a negative number (-1 through -127) is equal to the two's complement of its positive counterpart.

- -127 = the two's complement of +127
- -126 = the two's complement of +126
- -125 = the two's complement of +125
- -2 = the two's complement of +2
- -1 = the two's complement of +1

Therefore, you can interpret the form of the results of the last two examples in Figure 9-7 by converting the displayed results as follows:
1. 2222
   -3333
   EEEF hex = 1110 1110 1110 1111 binary
   take the one’s complement 0001 0001 0001 0000
   add one for the two’s complement 0001 0001 0001 0001
   Therefore, EEEF hex = negative 1111 hex

2. 0000
   -0004
   FFFC hex = 1111 1111 1111 1100 binary
   one’s complement 0000 0000 0000 0011
   two’s complement 0000 0000 0000 0100
   Therefore, FFFC hex = negative 0004 hex

When thinking in terms of signed two-byte hex numbers, all numbers from 8000 through FFFF are considered negative. All two-bytes hex numbers from 0000 through 7FFF, are considered positive.

<table>
<thead>
<tr>
<th>Unsigned Value</th>
<th>Signed Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0001</td>
<td>+0001</td>
</tr>
<tr>
<td>0002</td>
<td>+0002</td>
</tr>
<tr>
<td>0003</td>
<td>+0003</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>7FFD</td>
<td>+7FFD</td>
</tr>
<tr>
<td>7FFE</td>
<td>+7FFE</td>
</tr>
<tr>
<td>7FFF</td>
<td>+7FFF</td>
</tr>
<tr>
<td>8000</td>
<td>-8000</td>
</tr>
<tr>
<td>8001</td>
<td>-7FFF</td>
</tr>
<tr>
<td>8002</td>
<td>-7FFE</td>
</tr>
<tr>
<td>8003</td>
<td>-7FFD</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>FFFD</td>
<td>-0003</td>
</tr>
<tr>
<td>FFFE</td>
<td>-0002</td>
</tr>
<tr>
<td>FFFF</td>
<td>-0001</td>
</tr>
</tbody>
</table>

*Figure 9-8. Two-Byte Hex Signed Numbers*
For two-byte numbers, the sign position is considered to be in the Most Significant Bit of the Most Significant Byte.

Thus numbers such as:

\[ \begin{align*}
0000000000000000 &= 0010 \text{ hex} \\
0010011111100110 &= 27E6 \text{ hex} \\
0110111110101001 &= 6FA9 \text{ hex}
\end{align*} \]

would be interpreted as positive numbers, but numbers such as:

\[ \begin{align*}
1000000000000000 &= 8000 \text{ hex} \\
1010011111100101 &= A7E5 \text{ hex} \\
1111011010111100 &= F6BC \text{ hex}
\end{align*} \]

would be interpreted as negative numbers.

A complete discussion of the arithmetic of signed numbers is beyond the scope of this book. A more thorough discussion of signed numbers and signed number arithmetic can be found in the MOS Technology Programming Manual available in some computer stores or from MOS Technology, Inc., 950 Rittenhouse Road, Norristown, PA 19401.

For our purposes, we must realize that certain branch instructions test the result of numbers to see whether they are negative or positive. This determination depends on whether or not the Negative flag of the Processor Status Register has been set to 1 or not. The Negative flag is set to 1 when the computer interprets the results of certain instructions as negative numbers (a 1 in bit 7 resulting from the operation performed by the instruction). In Chapter 5, a table shows the effect that each instruction has on the various flags in the Processor Status Register.

**MULTIPLE-BYTE ADDITION AND SUBTRACTION**

To add or subtract numbers that require more than two bytes, an extension of this two-byte procedure can be made. The operation is always performed from the Least Significant Byte forward (or from right to left).
DECIMAL ARITHMETIC

Are you tired of converting binary to hex to decimal? If so, the 6502 instruction set includes an instruction to help you out. If you are careful to express the values that you wish to add or subtract as binary-coded-decimal (BCD) numbers, the Atari can add or subtract those numbers and express the result as a decimal value. What are binary-coded decimal numbers? That’s just a fancy name for a binary number that has been separated into two 4-bit parts. These parts are then interpreted as decimal digits.

Examples:

<table>
<thead>
<tr>
<th>Binary</th>
<th>Binary-coded-decimal</th>
<th>Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>01011000</td>
<td>0101 1000</td>
<td>58</td>
</tr>
<tr>
<td>10010011</td>
<td>1001 0011</td>
<td>93</td>
</tr>
<tr>
<td>00010110</td>
<td>0001 0110</td>
<td>16</td>
</tr>
</tbody>
</table>
Since each four bits is interpreted as a decimal digit, the binary inputs must be chosen with care.

1100 1001
NOT a BCD value

1010 1011
NOT BCD values

Each 4-bit part must be one of these:

<table>
<thead>
<tr>
<th>BCD</th>
<th>DECIMAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
</tr>
<tr>
<td>0010</td>
<td>2</td>
</tr>
<tr>
<td>0011</td>
<td>3</td>
</tr>
<tr>
<td>0100</td>
<td>4</td>
</tr>
<tr>
<td>0101</td>
<td>5</td>
</tr>
<tr>
<td>0110</td>
<td>6</td>
</tr>
<tr>
<td>0111</td>
<td>7</td>
</tr>
<tr>
<td>1000</td>
<td>8</td>
</tr>
<tr>
<td>1001</td>
<td>9</td>
</tr>
</tbody>
</table>

The instruction needed to request the computer to perform decimal addition or subtraction is:

SED (SEt Decimal mode)
Op Code = F8
Implied Addressing mode
One byte long.
Status flags affected: D

This instruction sets the Decimal flag in the Processor Status Register to 1.
Once this instruction has been used, all of the Add and Subtract instructions will be carried out as decimal operations because of the status of the Decimal flag. The operation of any of the other instructions are not affected. If the SED instruction has been executed in a program, and a binary addition or subtraction is desired, the computer must execute the Clear Decimal mode instruction.

CLD (Clear Decimal mode)
Op Code = D8
Implied Addressing mode
One byte long
Status flags affected: D

This instruction resets the decimal flag in the Processor Status Register to zero.

Suppose that you want to add the decimal numbers 18 and 23. You might use this program.

ADD TWO DECIMAL NUMBERS
SED Set decimal mode
CLC Clear the carry bit
LDA$1000 Load accumulator from memory location $1000
ADC$1001 Add value from memory location $1001
STA $1100 Store the sum

Enter and assemble:

```
EDIT
NUM 100,10

100 *=1000 ← Enter it
110 SED
120 CLC
130 LDA $1100
140 ADC $1101
150 STA $1002
160 END
170 ASM ← Assemble it
0000 100 *=1000
```
| 1000 | F8 | 110 SED |
| 1001 | 18 | 120 CLC |
| 1002 | AD011 | 130 LDA $1000 |
| 1005 | 6D011 | 140 ADC $1001 |
| 1008 | 8D021 | 150 STA $1002 |

End

Go to the DEBUGGER, execute the program, and display the result. Be sure to load the data in 1100 and 1101 first.

```
EDIT
BUG
DEBUG G1000
100B A=41 X=00 Y=00 P=38 S=00
DEBUG D1102,1102
1102 41
DEBUG
```

The decimal result of 18+23 is 41. If we had been adding the hexadecimal values of 18 and 23, the result would have been 3B. Remember, 18 and 23 in hexadecimal notation are different values than 18 and 23 in decimal notation.

3B hex and 41 decimal are not equivalent values

Now let's execute the program starting from location 1001 (omitting the Set Decimal instruction) and observe the result.
Once the SED instruction has been executed, all additions in the program are performed in the decimal mode. The CLD (Clear Decimal mode) instruction must be executed to get back to binary addition.

The advantage of decimal addition is that it relieves you of converting numbers from one base to another for interpretation. You can enter decimal values (such as 23 and 18) and obtain decimal results (41).

Try the Add Two Decimal Numbers Program with other pairs of decimal values. Substitute them for the values that you originally put in memory locations 1100 and 1102. Use the Change Memory command to input the values in Figure 9-10.

The first change would be:

C1100<51,23

<table>
<thead>
<tr>
<th>First Number</th>
<th>Second Number</th>
<th>Sum Displayed</th>
</tr>
</thead>
<tbody>
<tr>
<td>51</td>
<td>23</td>
<td></td>
</tr>
<tr>
<td>68</td>
<td>14</td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>17</td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>99</td>
<td></td>
</tr>
<tr>
<td>72</td>
<td>45</td>
<td></td>
</tr>
</tbody>
</table>

Figure 9-10. Decimal Addition Exercises
What is wrong with the last two exercises in Figure 9-10? Make sure that the sum of the two numbers is less than 100 in this program. Any sum whose value is greater than 99 will not fit in the accumulator or any memory location. Remember, this is an 8-bit computer. Here is what will happen to you if you insist on experimenting (we secretly encourage you).

Last exercise example:

```
C1100<72,45         Load data
G1000                Execute
100B A=17 X=00 P=F9 S=00
DEBUG D1102,1102     Display result
1102 17              You know it means 117, but it just can't show all of it
DEBUG
```

Since the 6502 microprocessor handles blocks of data in 8-bit sizes, some provision must be made for the situation which results when a sum is larger than can be held in 8 bits. This technique was shown in earlier chapters with hexadecimal arithmetic. The same method will work with decimal addition and subtraction.

**SUMMARY**

In this chapter you have learned:

- To break large numbers into more than one byte in order to perform two-byte arithmetic,
- To retrieve, add (or subtract), and store separate bytes of two-byte numbers using the Absolute Addressing mode,
- How the carry flag is used for two-byte addition and subtraction,
- How to move blocks of data from one area of memory to another,
- How signed numbers may be interpreted: negative if Most Significant Bit is set to 1 positive if Most Significant Bit is reset to 0,
- The significance of the status flags on the Processor Status Register for branch instructions,
- That addition and subtraction can be performed on numbers larger than two bytes,
• To write numbers in binary-coded-decimal (BCD) form so that the computer can perform decimal addition and subtraction,
• How to use the Set Decimal mode and Clear Decimal mode to perform either decimal or binary arithmetic.

EXERCISES

1. If the two-byte hex numbers 9B and 66 are added, will the sum be too large to be held in the accumulator? _______________________

2. Tell how the computer will handle the sum in Exercise 1.
   (a) Number in accumulator _______________________
   (b) The Carry flag will be ________________
       (zero, one)

3. The following data is stored (or to be stored) in the specified memory locations.

<table>
<thead>
<tr>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1100</td>
<td>3F</td>
</tr>
<tr>
<td>1111</td>
<td>A3</td>
</tr>
<tr>
<td>1112</td>
<td></td>
</tr>
</tbody>
</table>

   ← Put sum here

Give the assembly language instructions to load the number from address 1110, add the number from address 1111 and store the result using the Absolute Addressing mode.
   (a) _______________________
   (b) _______________________
   (c) _______________________

4. In performing an addition or subtraction operation on the Least Significant Bytes, the Carry flag must be appropriately set or reset. The assembly instruction:
   (a) for addition is _______________________
   (b) for subtraction is _______________________

5. Show where the data would be placed by the following instruction.
   C1120<15,A4,32,CC

<table>
<thead>
<tr>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1120</td>
<td></td>
</tr>
<tr>
<td>1121</td>
<td></td>
</tr>
<tr>
<td>1122</td>
<td></td>
</tr>
<tr>
<td>1123</td>
<td></td>
</tr>
</tbody>
</table>
6. Write the DEBUGGER command that would move the data from where it is in memory in Exercise 5 to memory block 1105 through 1108.

7. Tell whether the following signed numbers would be interpreted as negative or positive.
   (a) 10100111 binary
   (b) 7F hex
   (c) 01011111 binary
   (d) A3 hex

8. Two-byte signed numbers are considered:
   (a) positive if in the range of ________ through ________
   (b) negative if in the range of ________ through ________

ANSWERS

Figure 9-6

<table>
<thead>
<tr>
<th>First Number</th>
<th>Second Number</th>
<th>Sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>000A</td>
<td>000B</td>
<td>0015</td>
</tr>
<tr>
<td>13C5</td>
<td>0F24</td>
<td>22E9</td>
</tr>
<tr>
<td>6666</td>
<td>333E</td>
<td>99A4</td>
</tr>
<tr>
<td>37AB</td>
<td>A09D</td>
<td>D848</td>
</tr>
<tr>
<td>E111</td>
<td>2000</td>
<td>0111</td>
</tr>
</tbody>
</table>

Most Significant Bit is lost

Figure 9-7

<table>
<thead>
<tr>
<th>First Number</th>
<th>Last Number</th>
<th>Difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>FFFF</td>
<td>0112</td>
<td>FEED</td>
</tr>
<tr>
<td>76A3</td>
<td>08D8</td>
<td></td>
</tr>
<tr>
<td>590A</td>
<td>1EEF</td>
<td></td>
</tr>
<tr>
<td>2222</td>
<td>EEEF</td>
<td></td>
</tr>
<tr>
<td>0000</td>
<td>FFFC</td>
<td></td>
</tr>
</tbody>
</table>

Last two are negative results
Figure 9-10

<table>
<thead>
<tr>
<th>First Number</th>
<th>Second Number</th>
<th>Sum Displayed</th>
</tr>
</thead>
<tbody>
<tr>
<td>51</td>
<td>23</td>
<td>74</td>
</tr>
<tr>
<td>68</td>
<td>14</td>
<td>82</td>
</tr>
<tr>
<td>29</td>
<td>17</td>
<td>46</td>
</tr>
<tr>
<td>33</td>
<td>99</td>
<td>32</td>
</tr>
<tr>
<td>72</td>
<td>45</td>
<td>17</td>
</tr>
</tbody>
</table>

Most Significant Bit is lost in last two answers

1. yes
2. (a) 01
   (b) one (true answer is 101)
3. (a) LDA $1110$
   (b) ADC $1111$
   (c) STA $1112$
4. (a) CLC
   (b) SEC
5. | Address | Data |
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1120</td>
<td>15</td>
</tr>
<tr>
<td>1121</td>
<td>A4</td>
</tr>
<tr>
<td>1122</td>
<td>32</td>
</tr>
<tr>
<td>1123</td>
<td>CC</td>
</tr>
</tbody>
</table>
6. M1105<1120,1123
7. (a) negative
   (b) positive
   (c) positive
   (d) negative
8. (a) 0000 through 7FFF
   (b) 8000 through FFFF
Chapter 10

Shift and Rotate

You learned to add and subtract in Chapter 9, but these two operations are only half of the four basic arithmetic operations. What about multiplication and division? The 6502 microprocessor has no instructions that will directly perform multiplication and division. However, think of the way that you perform these operations with pencil and paper.

You perform a series of additions, subtractions, and shifts to get the partial answers into the correct place value.

Examples:

(a) Multiplication

\[
\begin{array}{c}
23 \\
\times 32 \\
46 & \quad 1) \text{ Two } 23\text{'s } = 23+23 = 46 \\
69 & \quad 2) \text{ Shift one place left } \\
736 & \quad 3) \text{ Three } 23\text{'s } = 23+23+23 = 69 \\
\end{array}
\]

(b) Division

\[
\begin{array}{c}
32 \\
\overline{23} 736 \\
69 & \quad 1) \text{ Three } 23\text{'s } = 23+23+23 = 69 \\
46 & \quad 2) \text{ Subtract } \\
66 & \quad 3) \text{ Shift right and bring down } \\
46 & \quad 4) \text{ Two } 23\text{'s } = 23+23 = 46 \\
0 & \quad 5) \text{ Subtract }
\end{array}
\]

You know that the computer can add and subtract, but you must learn how to manipulate numbers with instructions that will shift the numbers left or right before you tackle the multiplication and division operations.

Let's take another look at the structure of 8-bit binary numbers so that you can better understand the consequences of changing place values.
You can see from Figure 10-1 that each place value is double that of the place value immediately to its right. In other words, shifting a binary digit one place to the left doubles its value.

Study the following examples which demonstrate the results of a single shift left for each digit.

1. | Binary | Hex | Decimal |
   | 00000101 | 5   | 5       |
   | Shifted  | 00001010 | A   | 10      |
From the previous examples, you can see that a shift of one place to the left doubles the value of an 8-bit number. In the decimal paper-and-pencil multiplication and division examples, you made shifts which changed the place values by powers of ten.

\[
\begin{array}{c}
23 \\
\times 32 \\
\hline
46 \quad \text{shift left a power of ten because } 3 \times 23 \text{ is really } 30 \times 23 \\
69 \\
\hline
736
\end{array}
\]

In binary multiplication:

\[
\begin{array}{c}
101 \\
\times 11 \\
\hline
101 \quad \text{shift left a power of two because } 1 \times 101 \\
101 \\
\hline
1111 \quad 1 \times 101 \text{ is really } 10 \times 101
\end{array}
\]

You really do it in two parts

\[
\begin{array}{c}
101 \\
\times 1 \\
\hline
101 \\
\times 10 \\
\hline
101 \quad + \quad 1010 \quad = \quad 1111
\end{array}
\]

The shifting is just a short cut to perform separate operations as one unit.

In this chapter, we will describe the four shift and rotate instructions in the 6502 instruction set. We will also use subroutines, short mini-programs that can be branched to from several different locations.

First, we will demonstrate how the shift and rotate instructions work and then show some applications.
The four instructions are:
1. ASL (Arithmetic Shift Left),
2. LSR (Logical Shift Right),
3. ROL (ROtate Left), and
4. ROR (ROtate Right).

**ARITHMETIC SHIFT LEFT**

The first instruction that we will look at is ASL. This instruction shifts every bit in a particular byte of data one position to the left, as we have been discussing in previous examples.

One thing we didn't discuss earlier. If the 8 bits are shifted left, what happens at each end?

The $2^0$ bit goes to $2^1$
- $2^1$ bit goes to $2^2$
- $2^2$ bit goes to $2^3$
- $2^3$ bit goes to $2^4$
- $2^4$ bit goes to $2^5$
- $2^5$ bit goes to $2^6$
- $2^6$ bit goes to $2^7$

But where does the $2^7$ bit go, and where does the $2^0$ bit come from?

Each time that the ASL instruction is executed, the Carry flag assumes the value that was in the leftmost bit ($2^7$) before the shift occurred. All the rest of the bits are moved left one position, and a zero is moved into the rightmost position ($2^0$).

Here is an example that shows the contents of the accumulator before, during, and after the instruction is executed. The value, B1, is in the accumulator before execution.
Enter the following short little program, and you can see the computer shift the value in the accumulator when the ASL instruction is executed. The program places a one in the accumulator and shifts it left one place.

```
EDIT
NUM 100,10
100  *=\$1000
110  CLC
120  LDA #1    --- Load a 1 (00000001 binary)
130  ASL A     --- Shift left once
140  END
150  •
```
Assemble the program.

```
0000  100  *$ 1000
1000  18   110  CLC
1001  A901 120  LDA #$01
1003  0A   130  ASL A
1403 END
```

Now, enter the DEBUGGER and set the accumulator to zero by using the Charge Register command.

```
EDIT
DEBUG
CR<0
```

Trace the program by typing: T1000 and pressing the RETURN key.

```
T1000
1000  18   CLC  Accumulator
      A=00 X=00 Y=00 P=32 S=00  00000000
1001  A9 01  LDA #$01
      A=01 X=00 Y=00 P=30 S=00  00000001
See the
Accum.
Now shifted
1003  0A   ASL A
      A=02 X=00 Y=00 P=30 S=00  00000010
1004  00   BRK
      A=02 X=00 Y=00 P=30 S=00
```

Debug
Now, reset the accumulator to zero before going on to the next program.

```
DEBUG
CR<0
```

You only shifted left one time in the previous program. What happens if you shift left 8 times in succession? Would there be anything left in the accumulator? What would be the condition of the Carry flag? Try it.

Enter the program shown below and assemble it.

```
010  *=$1000
020  CLC
030  LDA #1
040  ASL A
050  ASL A
060  ASL A
070  ASL A
080  ASL A
090  ASL A
100  ASL A
110  ASL A
120  END
```

This program sets the carry bit to zero, puts a 1 in the accumulator, and shifts the accumulator left 8 times. When the program is done, the accumulator should be zero and the Carry flag should be set, because the bit which forms the 1 in the accumulator was shifted left through every bit and then out of the accumulator and into the Carry bit. To see this for yourself, type "BUG" and press RETURN. Then type "DR" and press RETURN again. You should see the registers as they are below.

```
A=00  X=00  Y=00  P=30  S=00
```

You typed these

```
BUG
DEBUG
DR
```

The accumulator is zero, and as you can see, the Carry bit is not set in the Processor Status Register (P).
Now, execute the program by typing: G1000, and press the RETURN key.

You should see the following:

```
DEBUG
G1000
100B  
A=00 X=00 Y=00 P=33 S=00
DEBUG
```

The Processor Status Register has changed, and the Carry flag (bit 0) is set.

```
0 0 1 1 0 0 1 1
```

Processor Status Register = 33 hex

Now try typing "T1000" to Trace the program. You will be able to see the registers as they are changing after each step in the program. Be sure to press RETURN after typing "T1000".

```
DEBUG
T1000
1000 1A  CLC
A=00 X=00 Y=00 P=32 S=00
1001 A9 01  LDA #$01
  A=01 X=00 Y=00 P=30 S=00
1002 0A  ASL A
  A=02 X=00 Y=00 P=30 S=00
1004 0A  ASL A
  A=04 X=00 Y=00 P=30 S=00
1005 0A  ASL A
  A=06 X=00 Y=00 P=30 S=00
1006 0A  ASL A
  A=10 X=00 Y=00 P=30 S=00
1007 0A  ASL A
  A=20 X=00 Y=00 P=30 S=00
1008 0A  ASL A
  A=40 X=00 Y=00 P=30 S=00
1009 0A  ASL A
  A=80 X=00 Y=00 P=80 S=00
100A 0A  ASL A
  A=00 X=00 Y=00 P=33 S=00
100B 00  BRK
  A=00 X=00 Y=00 P=33 S=00
```

*Figure 10-3. Trace of the Shift Program*
After the first instruction of the program (CLC) is executed, the Processor Status Register (P register) shows a value of 32. This means the Carry flag is zero. If a carry had occurred, the P register would read 33. Note that the Zero flag (bit 1 of the P register) is set.

After the next step (LOA #$01), which loads the 1 into the accumulator, the A register = 01 and P=30, indicating that when the 1 (a non-zero value) was loaded, the Zero bit was turned off.

- Note the pattern in the accumulator after each left shift. As the original 1 is shifted, the value in the accumulator is effectively doubled.
- Notice that when the value of 80 was in the accumulator, the P register changed to B0. When a value greater than 7F results from a shift, the Negative flag (bit 7 of the P register) is turned on. As you found out earlier in the book, the computer considers any number with its 7th bit on (all hex numbers between 80 and FF) as negative for comparing purposes (such as those used by branches).

After the last shift instruction is performed, the A register becomes zero and the bit that formed the value of 80 has been moved over to the Carry bit of the P register. The Negative flag is off (since the A register is zero). This leaves the P register with its Carry and Zero bits set to 1.

As we mentioned earlier, shifting a number left has the effect of doubling that number. This can be used to make a simple multiplication program. This program can only multiply a number by a multiple of two, though. If you want to multiply a number by 2, shift it once. You shift it twice to multiply by 4, three times for 8, etc. The program shown below is an example of multiplying 8 by 4. Enter the program and assemble.
ARITHMETIC SHIFT LEFT  197

Now enter DEBUG and run it.

BUG
DEBUG
G1000
1005
DEBUG

A=20  X=00  Y=00  P=30  S=00

The accumulator now holds the number 20 (hex), which is the same as 32 in decimal, $8 \times 4 = 32$, right?

You have to be careful when you multiply numbers this way, though, because if you have a number like 80 (hex), and you try to multiply it by 4 by shifting it left twice, you are going to get an answer of zero in the accumulator and the Carry bit.

Accumulator

\[
\begin{array}{cccccccc}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & & & & & & & \\
\end{array}
\]

Carry bit

\[
\begin{array}{cccccccc}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & & & & & & \\
\end{array}
\]

Carry bit

Figure 10-4(a). Original Value

Figure 10-4(b). After One Shift

Accumulator

\[
\begin{array}{cccccccc}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & & & & & & \\
\end{array}
\]

Carry bit

The early bird gets the bit

Figure 10-4(c). After Two Shifts
We have used the ASL instruction in the Accumulator Addressing mode. This instruction as well as the other shift and rotate instructions may be used in that mode or the Zero Page, Zero Page Indexed, the Absolute, or the Absolute indexed mode as indicated in the tables in Chapter 5.

The ASL instruction, as well as other shift and rotate instructions affect the Negative, Zero, and Carry flags in the Processor Status Register when they are executed.

LOGICAL SHIFT RIGHT

The LSR (Logical Shift Right) instruction, despite the word "Logical" rather than "Arithmetic," does the same thing as the ASL, except in the opposite direction. Each time it is executed, a zero is moved into the leftmost bit, the rest of the bits are shifted to the right, and the rightmost bit goes over into the Carry flag.

![Figure 10-5. Operation of LSR](image)

We will demonstrate this instruction by loading a value to be shifted into the accumulator from a memory location, and shift it right a given number of times. After each shift, the value of the accumulator is stored in another memory location. Every shift after the first stores the value in the next highest memory location.

<table>
<thead>
<tr>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1100</td>
<td>Original value</td>
</tr>
<tr>
<td>1101</td>
<td>Number of shifts</td>
</tr>
<tr>
<td>1102</td>
<td>Result of 1 shift</td>
</tr>
<tr>
<td>1103</td>
<td>Result of 2 shifts</td>
</tr>
<tr>
<td>1104</td>
<td>Result of 3 shifts</td>
</tr>
<tr>
<td>1105</td>
<td>etc.</td>
</tr>
</tbody>
</table>

*Figure 10-6. Memory Use for Shift Right*
The program is detailed below.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>*=1000</td>
<td>Start the program at location 1000</td>
</tr>
<tr>
<td>CLC</td>
<td>Clear the Carry bit</td>
</tr>
<tr>
<td>LDX #0</td>
<td>Set the memory pointer to zero</td>
</tr>
<tr>
<td>LDA $1100</td>
<td>Get the value to be shifted</td>
</tr>
<tr>
<td>LDY $1101</td>
<td>Load the number of times that the number is to be shifted into Y</td>
</tr>
<tr>
<td>LOOP LSR A</td>
<td>Shift it once</td>
</tr>
<tr>
<td>STA $1102,X</td>
<td>Put it in the next memory location</td>
</tr>
<tr>
<td>INX</td>
<td>Point to the next location</td>
</tr>
<tr>
<td>DEY</td>
<td>Find out if we have shifted enough</td>
</tr>
<tr>
<td>BNE LOOP</td>
<td>If not, go back and shift</td>
</tr>
</tbody>
</table>

This program will take the number that you put in location 1100 (hex) and shift it right just as many times as you have told it to in location 1101. The results of each shift will be stored starting in location 1102 and continuing for as many locations as there are shifts. Enter the program below and assemble it.

**RIGHT SHIFTER PROGRAM**

```
010  *=1000
020  CLC
030  LDX   #0
040  LDA   $1100
050  LDY   $1101
060  LOOP  LSR A
070  STA   $1102,X
080  INX
090  DEY
100  BNE   LOOP
110  END
```
Now enter DEBUG and put any value you want to see shifted into location 1100. Suppose the value is 80 (hex). You would enter:

\[
\text{BUG DEBUG C1100<80}
\]

Now enter the number of times that you want the number to be shifted into location 1101. Suppose it is 7. You type:

\[
\text{C1101<07}
\]

You now have:

<table>
<thead>
<tr>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1100</td>
<td>80</td>
</tr>
<tr>
<td>1101</td>
<td>07</td>
</tr>
</tbody>
</table>

Now you are ready to run the program. Type: G1000, and press the RETURN key. To see the results, type D1102,1108. This will let you look at the seven locations used to store the shifted values.

\[
\text{G1000 1011 DEBUG D1102,1108}
\]

\[
\begin{align*}
1102 & \quad 40 \quad 20 \quad 10 \quad 08 \quad 04 \quad 02 \\
1108 & \quad 01 \\
\text{DEBUG}
\end{align*}
\]

And there they are! Note the values in the registers when the program was finished:

- The accumulator contains the last shifted result.
- The X register contains the number of times that the accumulator was shifted.
- The Y register is zero, indicating that all 7 shifts have been performed.
- The P register shows that the Carry flag is not set. That is because there were not enough shifts made for the one to be bumped over into the Carry bit.

If you could see the data in memory locations 1102 through 1108 in binary form, the shifts would be more apparent.
**LOGICAL SHIFT RIGHT**

<table>
<thead>
<tr>
<th>Address</th>
<th>Binary Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1102</td>
<td>01000000</td>
</tr>
<tr>
<td>1103</td>
<td>00100000</td>
</tr>
<tr>
<td>1104</td>
<td>00010000</td>
</tr>
<tr>
<td>1105</td>
<td>00001000</td>
</tr>
<tr>
<td>1106</td>
<td>00000100</td>
</tr>
<tr>
<td>1107</td>
<td>00000010</td>
</tr>
<tr>
<td>1108</td>
<td>00000001</td>
</tr>
</tbody>
</table>

Now try shifting 8 times and see what happens. Change memory location 1101 and execute the program again. Display memory locations 1102 through 1109 this time.

```
C1101<8
G1000
1011 A=00 X=08 Y=00 P=33 S=00
D1102,1109
1102 40 20 10 08 04 02
1108 01 00
DEBUG
```

Notice the P register contains 33 at the end of the run. The Carry bit was turned on by the last shift right.

Show what values would be in the four registers if you executed the program with the following values in memory.

<table>
<thead>
<tr>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1100</td>
<td>80</td>
</tr>
<tr>
<td>1101</td>
<td>0A</td>
</tr>
</tbody>
</table>

A= _____  X= _____  Y= _____  P= _____

Try entering the values in Figure 10-7 at locations 1100 and 1101. Fill in the values in memory at the end of the program's execution.
The two rotate instructions basically do the same thing as the shift instructions, with one important difference. Instead of throwing out the Carry bit and filling in a zero at one end of the byte, the Carry bit is rotated back into the original byte again.

**ROTATE LEFT**

The rotate left instruction (ROL) tells the computer to think of data in terms of blocks in a wheel instead of blocks in a straight line.
Or as is more conventional:

When a large number of rotations are made in succession, no bits are lost. They merely go round and round to different positions. It's something like the game of musical chairs except that no one takes any of the chairs away. There's always a place for each bit of data to go.

The next program sets the Carry bit and loads the accumulator with 1. Then the accumulator is rotated left eight times. On the last rotate, the Carry bit is set by the 1 that has been rotated through the accumulator. The accumulator ends up with a value of 80 due to the original carry that has rotated through the accumulator.

Enter the program and assemble it.

```
010  *=S1000
020  SEC
030  LDA  #1
040  ROL  A
050  ROL  A
060  ROL  A
070  ROL  A
080  ROL  A
090  ROL  A
100  ROL  A
110  ROL  A
```

Now run the program by entering the Debugger mode and typing: G1000. (Don’t forget the RETURN key.) You should see:

```
G1000
100B  A=80  X=00  Y=00  P=B1  S=00
DEBUG
```

```
1000000000
A Register
```

```
10110001
P Register
```

Negative flag

Carry flag
You can see from the P register that the Carry and Negative flags are set. The carry was set when the 1 that we had in the accumulator was shifted all the way over and out of the accumulator and into the Carry bit. The Negative flag was set when the value of the accumulator became C0 (and stayed set for 80) due to the shifting.

Examine the trace of the program that follows.

<table>
<thead>
<tr>
<th>Step</th>
<th>Instruction</th>
<th>A</th>
<th>X</th>
<th>Y</th>
<th>P</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>SEC</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>31</td>
<td>00</td>
</tr>
<tr>
<td>1001</td>
<td>LDA #$01</td>
<td>01</td>
<td>00</td>
<td>00</td>
<td>31</td>
<td>00</td>
</tr>
<tr>
<td>1003</td>
<td>ROL A</td>
<td>03</td>
<td>00</td>
<td>00</td>
<td>30</td>
<td>00</td>
</tr>
<tr>
<td>1004</td>
<td>ROL A</td>
<td>06</td>
<td>00</td>
<td>00</td>
<td>30</td>
<td>00</td>
</tr>
<tr>
<td>1005</td>
<td>ROL A</td>
<td>0C</td>
<td>00</td>
<td>00</td>
<td>30</td>
<td>00</td>
</tr>
<tr>
<td>1006</td>
<td>ROL A</td>
<td>18</td>
<td>00</td>
<td>00</td>
<td>30</td>
<td>00</td>
</tr>
<tr>
<td>1007</td>
<td>ROL A</td>
<td>30</td>
<td>00</td>
<td>00</td>
<td>30</td>
<td>00</td>
</tr>
<tr>
<td>1008</td>
<td>ROL A</td>
<td>60</td>
<td>00</td>
<td>00</td>
<td>30</td>
<td>00</td>
</tr>
<tr>
<td>1009</td>
<td>ROL A</td>
<td>C0</td>
<td>00</td>
<td>00</td>
<td>B0</td>
<td>00</td>
</tr>
<tr>
<td>100A</td>
<td>ROL A</td>
<td>80</td>
<td>00</td>
<td>00</td>
<td>B1</td>
<td>00</td>
</tr>
<tr>
<td>100B</td>
<td>BRK</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

In the first step, the Carry flag is set by the SEC instruction. Then a 1 is loaded into the accumulator. After the first ROL instruction, bit 7 of the accumulator (a zero) has been moved into the Carry bit, resetting it (as shown by P=30). The old value of the Carry bit (a one) is moved into bit 0 of the accumulator. All the rest of the bits in the accumulator have been shifted one position to the left, giving a value of 03 in the accumulator.

Figure 10-9 shows the binary results of each step in the program.

After the seventh rotate instruction has been executed, the accumulator has a value of C0. This sets the Negative flag, making P=B0. After the next ro-
### Rotate Right

The rotate-right instruction (ROL) works in the opposite direction as rotate left. As successive rotate-right instructions are executed, the bits in the accumulator and carry go around and around clockwise.

![Rotate-Right Wheel](image)

**Figure 10-10. Rotate-Right Wheel**

**Figure 10-9. Accumulator and Carry Rotates**

- Rotate, bit 7 of the accumulator (a one) gets moved into the Carry bit, setting it and making $P=B1$. The value, 80, is left in the accumulator.
A rotate-right instruction executed following a rotate-left instruction would put all bits back in their original place.

**Example:**

<table>
<thead>
<tr>
<th></th>
<th>Carry</th>
<th>Accumulator</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Originally</td>
<td>0</td>
<td>1 0 1 0 1 0 1 0</td>
</tr>
<tr>
<td>2. ROL A causes</td>
<td>1</td>
<td>0 1 0 1 0 1 0 0</td>
</tr>
<tr>
<td>3. ROR A causes Back to original</td>
<td>0</td>
<td>1 0 1 0 1 0 1 0</td>
</tr>
</tbody>
</table>

I don't know why anyone would want to do that, but you might think up a reason.

To demonstrate the ROR instruction, look back to the Right Shifter Program and see how it might be changed to make it a Right Rotater Program. Only this time provide the following initial values:

- Carry bit = 1
- Memory location 1100 = 0
- Memory location 1101 = 0
- Rotate right memory location 1100 8 times
- Store the result of each rotation in successive memory locations from 1111 through 1118

You might do it like this:

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0100</td>
<td>*= $1000</td>
</tr>
<tr>
<td>2</td>
<td>0110</td>
<td>SEC</td>
</tr>
<tr>
<td>3</td>
<td>0120</td>
<td>LDY $1101</td>
</tr>
<tr>
<td>4</td>
<td>0130</td>
<td>LOOP ROR $1100</td>
</tr>
<tr>
<td>5</td>
<td>0140</td>
<td>LDA $1100</td>
</tr>
<tr>
<td>6</td>
<td>0150</td>
<td>STA $1111,Y</td>
</tr>
<tr>
<td>7</td>
<td>0160</td>
<td>INY</td>
</tr>
<tr>
<td>8</td>
<td>0170</td>
<td>CPY #8</td>
</tr>
<tr>
<td>9</td>
<td>0180</td>
<td>BNE LOOP</td>
</tr>
<tr>
<td>10</td>
<td>0190</td>
<td>END</td>
</tr>
</tbody>
</table>

We changed the program quite a bit to show you that there's more than one way to achieve the same result. The Y register now serves as a loop counter and as an index to store the result of each rotate.
Enter the program and assemble it. Then go to the DEBUGGER to store the initial values in memory.

```
BUG
DEBUG
C1100<0,0 ← This will set both 1100 and 1101 to zero
```

Single step the program and watch the accumulator change as it is loaded from 1100 each time the instruction at 1007 is executed. Also watch the Y register change at 100D.

```
C1100<0,0
S1000
1000  38   SEC
    A=00 X=00 Y=00 P=33 S=00
DEBUG
S
1001  AC 01 11   LDY  $1101
    A=00 X=00 Y=00 P=33 S=00
DEBUG
S
1004  6E 00 11   ROR  $1100 ← You can't see it do it but . . . . .
    A=00 X=00 Y=000 P=B0 S=00
DEBUG
S
1007  AD 00 11   LDA  $1100 ← there it is
    A=80 X=00 Y=00 P=B0 S=00
DEBUG
S
100A  99 11 11   STA  $1111,Y
    A=80 X=00 Y=00 P=B0 S=00
DEBUG
S
100D  4C   INY
    A=80 X=00 Y=01 P=30 S=00 ← Y changed
```
Continue single stepping until the program ends. When finished, display memory locations 1111 through 1118.

You are now able to manipulate data in four ways. Proceed to the summary and then the exercises.
This chapter contains discussions of shift and rotate instructions to prepare you for multiplication and division operations which follow in the next chapter. You learned that:

- Multiplication and division operations are performed by a series of additions, subtractions, and place location shifts;
- Shifting a binary digit left one place doubles the place value of the digit;
- Shifting all bits of a binary number left one place doubles the value of the binary number;
- Shift and rotate instructions may be used in the following address modes:
  - Accumulator, Zero Page, Zero Page Indexed, Absolute, and Absolute Indexed (however, only the X register can be used for indexing);
- The following diagrams show the operation of the shift and rotate instructions:

![Operational diagrams of shift and rotate instructions]

**EXERCISES**

1. What would be the decimal equivalent of the result of shifting the following binary number two times left?

   0 0 0 1 0 1 1 0 (binary number before shifts)

   decimal equivalent after two shifts
Exercise 2 through 6 assume the following binary digits are in the accumulator and Processor Status Register. Show the contents of each after the specified instructions have been executed.

<table>
<thead>
<tr>
<th>Accumulator</th>
<th>01010011</th>
<th>at the beginning of each exercise</th>
</tr>
</thead>
<tbody>
<tr>
<td>P register</td>
<td>00110000</td>
<td>N V B D I Z C</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Exercise</th>
<th>Instructions</th>
<th>Accumulator</th>
<th>P register</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.</td>
<td>ASL A, ASL A</td>
<td></td>
<td>011000</td>
</tr>
<tr>
<td>3.</td>
<td>LSR A, LSR A</td>
<td></td>
<td>011000</td>
</tr>
<tr>
<td>4.</td>
<td>ROL A, ROL A</td>
<td></td>
<td>011000</td>
</tr>
<tr>
<td>5.</td>
<td>ROR A, ROR A</td>
<td></td>
<td>011000</td>
</tr>
<tr>
<td>6.</td>
<td>ROR A, LSR A, ROL A, ASL A</td>
<td></td>
<td>011000</td>
</tr>
</tbody>
</table>

7. Assume you are using the Right Shifter Program, and you use the following command sequence:

```
DEBUG
C1100< FF,08
G1000
```

(a) Show the bits contained in the accumulator at the end.

(b) the Carry bit is ________ (set, reset)
8. In exercise 7, show the contents of these memory locations in hexadecimal notation.

<table>
<thead>
<tr>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1102</td>
<td></td>
</tr>
<tr>
<td>1103</td>
<td></td>
</tr>
<tr>
<td>1104</td>
<td></td>
</tr>
<tr>
<td>1105</td>
<td></td>
</tr>
<tr>
<td>1106</td>
<td></td>
</tr>
<tr>
<td>1107</td>
<td></td>
</tr>
<tr>
<td>1108</td>
<td></td>
</tr>
<tr>
<td>1109</td>
<td></td>
</tr>
</tbody>
</table>

ANSWERS

From Figure 10-7

A=00 X=0A Y=00 P=32

<table>
<thead>
<tr>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1100</td>
<td>91</td>
</tr>
<tr>
<td>1101</td>
<td>03</td>
</tr>
<tr>
<td>1102</td>
<td>48</td>
</tr>
<tr>
<td>1103</td>
<td>24</td>
</tr>
<tr>
<td>1104</td>
<td>12</td>
</tr>
<tr>
<td>1105</td>
<td></td>
</tr>
</tbody>
</table>

1. 88 decimal (64+16+8)

2. Accumulator

<table>
<thead>
<tr>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

P register

<table>
<thead>
<tr>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

N=1, Z=0, C=0

3. Accumulator

<table>
<thead>
<tr>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

P register

<table>
<thead>
<tr>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

N=0, Z=0, C=1

4. Accumulator

<table>
<thead>
<tr>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

P register

<table>
<thead>
<tr>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

N=1, Z=0, C=0
5. Accumulator: 1 0 0 1 0 1 0 0
   P register: 1 0 1 1 0 0 0 1
   N=1, Z=0, C=1

6. Accumulator: 0 1 0 1 0 0 1 0
   P register: 0 0 1 1 0 0 0 0
   N=0, Z=0, C=0

7. (a) 0 0 0 0 0 0 0 0
   (b) set

8.

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1102</td>
<td>7F</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1103</td>
<td>3F</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1104</td>
<td>1F</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1105</td>
<td>0F</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1106</td>
<td>07</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1107</td>
<td>03</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1108</td>
<td>01</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1109</td>
<td>00</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Chapter 11

Multiplication, Division, and Subroutines

Since the 6502 instruction set does not contain instructions for multiplication and division, it is up to you to develop methods to perform these operations. Multiplication can be thought of as a series of additions.

Example:
\[3 \times 12 = 12 + 12 + 12\]
and
\[5 \times 16 = 16 + 16 + 16 + 16 + 16\]

You can very easily write a program using a loop to add one number a specified number of times to obtain the same answer that would result from multiplying the two numbers.

MULTIPLICATION BY ADDITION

```
CLC
LDX #5
LDA #0
LOOP ADC $1100
DEX
BNE LOOP
END
```

If a second number (10 for example) is loaded into memory location 1100, and the program is assembled and executed, the accumulator would contain the desired result at the end of the program.

Although it is very quick and easy to write a program for this method, you can see that it would be time consuming to execute the multiplication of large numbers. That would require many passes through the loop. It would work for 8-bit numbers but would have to be redesigned for larger numbers requiring more than 8 bits. Due to the inefficiency of the program, it is seldom used.

Division could be performed in a similar manner, subtracting the divisor each time through a loop. However, that method has the same drawbacks of the multiplication method.
Since you've learned all about shifts and rotates in Chapter 10, let's see if you can make use of them to solve multiplication and division problems.

EIGHT-BIT MULTIPLICATION

Let's start with 8-bit numbers and look at the multiplication process first. Remember, the computer does its arithmetic using binary numbers. First look at a pencil-and-paper example of binary multiplication. Multiply 18 (decimal) by 58 (decimal) using both decimal and binary multiplication.

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>18</td>
<td>0001 0010</td>
</tr>
<tr>
<td>x 58</td>
<td>x 10011010</td>
</tr>
<tr>
<td>144</td>
<td>0000 0000</td>
</tr>
<tr>
<td>90</td>
<td>0 0010 010</td>
</tr>
<tr>
<td>1044</td>
<td>00 0001 0010</td>
</tr>
<tr>
<td></td>
<td>0 0010 010</td>
</tr>
<tr>
<td></td>
<td>00 0000 00</td>
</tr>
<tr>
<td></td>
<td>000 0000 0</td>
</tr>
<tr>
<td></td>
<td>000 0100 0010 0100</td>
</tr>
</tbody>
</table>

Notice that your multiplication involves adding the multiplicand every time that a one appears in the multiplier. Of course, there is a shift to the left each time a bit of the multiplier is used just as there is in decimal multiplication. You also proceed from right to left as you "use up" the bits of the multiplier.

The program that we will use does much the same thing. The first part of the program will initialize the memories with the appropriate values. We will store these quantities in memory as follows:

<table>
<thead>
<tr>
<th>Memory Address</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>1100</td>
<td>Most Significant Byte of product</td>
</tr>
<tr>
<td>1101</td>
<td>Least Significant Byte of product</td>
</tr>
<tr>
<td>1102</td>
<td>Multiplicand (12)</td>
</tr>
<tr>
<td>1103</td>
<td>Multiplier (3A)</td>
</tr>
</tbody>
</table>

*Figure 11-1. Memory Use for Multiplication*
The accumulator will temporarily hold the Least Significant Byte of the product as the program is executed. You may notice that the multiplier is used from left to right (the opposite of the usual paper-and-pencil method) to simplify the process.

```
0 0 1 1 1 0 1 0
```

Used first

Used last

The multiplier is shifted left as it multiplies.

The flowchart in Figure 11-2 clarifies the action in the program.

*Figure 11-2. 8-bit Multiplication Flowchart*
The housekeeping chores at the beginning of the program are easy for you now. You have used each instruction in this section of the program before. They are all either load or store instructions in various addressing modes.

This section of the assembly language program would look like this:

```
EDIT
NUM 100,10
100  *=1000
110  LDX       #8
120  LDA       #$12
130  STA       $1102
140  LDA       #$3A
150  STA       $1103
160  LDA       #0
170  STA       $1100
180  STA       $1101
```

The loop section performs the multiplication. Two instructions appear that you learned about in the last chapter. One of them is ASL (Arithmetic Shift Left). It is used to shift the contents of the accumulator at statement number 190. It is also used in the Absolute Addressing mode at statement number 210. When used in this mode, the bits of the specified memory location (1103) are shifted left.

Example:

```
Memory 1103 before ASL $1103

To carry
bit of Processor Status Register

Memory 1103 after ASL $1103
```

We will make use of the fact that the Most Significant Bit from the shift is moved to the carry bit of the Processor Status Register.

At statement 200, you will make use of the ROL (Rotate Left) instruction in the Absolute Addressing mode. In Chapter 10, you used this instruction with the accumulator. This time it is used to rotate the bits in memory address 1100.
Example:

<table>
<thead>
<tr>
<th>Memory 1100 before ROL $1100</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 0 0 1 0</td>
</tr>
<tr>
<td>To Carry bit of</td>
</tr>
<tr>
<td>Processor Status Register</td>
</tr>
<tr>
<td>Old carry bit</td>
</tr>
<tr>
<td>X is either 0 or 1 depending on the Carry bit when the instruction is executed</td>
</tr>
<tr>
<td>Memory 1100 after ROL $1100</td>
</tr>
<tr>
<td>0 0 0 0 0 0 1 0 X</td>
</tr>
</tbody>
</table>

As each instruction is executed, the Most Significant Bit moves into the Carry bit of the Processor Status Register. In the ASL instruction, the Least Significant Bit is filled with a zero. In the ROL instruction, the Least Significant Bit is filled from the Carry bit of the status register (either a 1 or a 0). The instructions derive their names from the action that they cause.

Here is the section that performs the loop, which is the heart of the program.

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>190</td>
<td>LOOP</td>
<td>ASL</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>200</td>
<td>ROL</td>
<td>$1100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>210</td>
<td>ASL</td>
<td>$1103</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>220</td>
<td>BCC</td>
<td>SKIP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>230</td>
<td>CLC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>240</td>
<td>ADC</td>
<td>$1102</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>250</td>
<td>BCC</td>
<td>SKIP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>260</td>
<td>INC</td>
<td>$1100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>270</td>
<td>SKIP</td>
<td>DEX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>280</td>
<td>BNE</td>
<td>LOOP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

After the loop has been executed 8 times (once for each bit in the multiplier, the Least Significant Byte of the result will be in the accumulator. This is stored in the final section of the program in memory location 1101 for later display. The program then ends.

The Most Significant Byte of the result is now in memory location 1100, and the Least Significant Byte of the result is in memory location 1101.
 USING THE 8-BIT MULTIPLICATION PROGRAM

Enter the program in the Writer/Editor mode. Then assemble it. We used the Atari 820 Printer as we assembled the Multiplication Program so that we could see the complete program at one time. It is a long program, and it may not all show on the video screen. Here is what our printed assembled program looks like.

<table>
<thead>
<tr>
<th>Address</th>
<th>Opcode</th>
<th>Data</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0100</td>
<td>*</td>
<td>$1000</td>
</tr>
<tr>
<td>1000</td>
<td>A208</td>
<td>0110</td>
<td>LDX #8</td>
</tr>
<tr>
<td>1002</td>
<td>A912</td>
<td>0120</td>
<td>STA #$12</td>
</tr>
<tr>
<td>1004</td>
<td>8D0211</td>
<td>0130</td>
<td>LDA #$3A</td>
</tr>
<tr>
<td>1007</td>
<td>8D0311</td>
<td>0140</td>
<td>STA $1103</td>
</tr>
<tr>
<td>1009</td>
<td>A900</td>
<td>0160</td>
<td>LDA $0</td>
</tr>
<tr>
<td>100E</td>
<td>8D0011</td>
<td>0170</td>
<td>STA $1100</td>
</tr>
<tr>
<td>1011</td>
<td>8D0111</td>
<td>0180</td>
<td>STA $1101</td>
</tr>
<tr>
<td>1014</td>
<td>0A</td>
<td>0190</td>
<td>LOOP ASL</td>
</tr>
<tr>
<td>1015</td>
<td>2E0011</td>
<td>0200</td>
<td>ROL $1100</td>
</tr>
<tr>
<td>1018</td>
<td>0E0311</td>
<td>0210</td>
<td>ASL $1103</td>
</tr>
<tr>
<td>101B</td>
<td>9009</td>
<td>0220</td>
<td>BCC SKIP</td>
</tr>
<tr>
<td>101D</td>
<td>18</td>
<td>0230</td>
<td>CLC</td>
</tr>
<tr>
<td>101E</td>
<td>6D0211</td>
<td>0240</td>
<td>ADC $1102</td>
</tr>
<tr>
<td>1021</td>
<td>9003</td>
<td>0250</td>
<td>BCC SKIP</td>
</tr>
<tr>
<td>1023</td>
<td>EE0011</td>
<td>0260</td>
<td>INC $1100</td>
</tr>
<tr>
<td>1026</td>
<td>CA</td>
<td>0270</td>
<td>SKIP DEX</td>
</tr>
<tr>
<td>1027</td>
<td>D0EB</td>
<td>0280</td>
<td>BNE LOOP</td>
</tr>
<tr>
<td>1029</td>
<td>8D0111</td>
<td>0290</td>
<td>STA $1101</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0300</td>
<td>END</td>
</tr>
</tbody>
</table>

Figure 11-3. Printout of Assembled Multiplication Program

You're ready to execute the program. You don't have to load any data this time as the program is going to do it for you. Enter the DEBUGGER and execute the program.
The result of the multiplication can be found in memory locations 1100 (Most Significant Byte) and 1101 (Least Significant Byte). You can display them both at once by typing:

```
D1100,1101
```

Hmmm, let's see:

```
0414 hex =
0000010000010100 binary =
4+16+1024 = 1044 decimal.
```

It checks with our previously hand-calculated result.

While you are in the Debugger mode, make changes to the multiplier and multiplicand. They are in memory locations 1003 and 1008. Try using the following pairs of hexadecimal numbers and put your results in the appropriate places in Figure 11-4.

<table>
<thead>
<tr>
<th>Multiplier (1003)</th>
<th>Multiplicand (1008)</th>
<th>Result (1100 and 1010)</th>
</tr>
</thead>
<tbody>
<tr>
<td>03</td>
<td>09</td>
<td></td>
</tr>
<tr>
<td>1A</td>
<td>E4</td>
<td></td>
</tr>
<tr>
<td>3C</td>
<td>D8</td>
<td></td>
</tr>
<tr>
<td>A4</td>
<td>C2</td>
<td></td>
</tr>
<tr>
<td>FF</td>
<td>FF</td>
<td></td>
</tr>
</tbody>
</table>

*Figure 11-4. Multiplication Exercises*
Remember, there is a Change Memory command in the Debugger mode. Here are two ways to get ready for the first modification.

```
DEBUG C1003<03
DEBUG C1008<09
```

or

```
DEBUG C1003<03 , , , , 09
```

Most people working with machine language programming agree that an assembler removes much of the drudgery. If the previous multiplication program had been programmed in machine language directly, there would have been many details to take care of.

- Individual addresses would have had to be assigned to each instruction and its operand.
- Branch locations would have had to be calculated.
- Individual Op Codes would have had to be looked up in an instruction table.
- Conversion between decimal and hexadecimal notation would have been necessary.

The Atari Assembler programs make life much easier. The Writer/Editor Program cannot execute a program directly, but it takes care of assigning Op Codes and the operands that go into the machine language Object Program.

The Assembler relies on the Debugger Program for execution of the assembled program and for examination of memory for results. The combination of Editor, Assembler, and Debugger are hard to beat. The Assembler does all the detail work, and the Debugger provides the execution and debugging capabilities. Of course, the Editor allows you to communicate your wishes to the Assembler.

After that brief introduction to multiplication, let's move on and take a look at the division process.
EIGHT-BIT DIVISION

Once again, we’ll take a look at pencil-and-paper division before looking at the computer’s method. Since division is the inverse operation of multiplication, we’ll use the same numbers that we used in the multiplication example with one exception. That exception is made to the dividend so that the division example will not come out even. There will be a remainder.

Example:

\[
1046 \text{ (decimal)} \div 58 \text{ (decimal)}
\]

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>1046</td>
<td>0011 1010</td>
</tr>
<tr>
<td>58</td>
<td>0011 1010</td>
</tr>
<tr>
<td>466</td>
<td>111 011</td>
</tr>
<tr>
<td>464</td>
<td>111 010</td>
</tr>
</tbody>
</table>

2 remainder

It checks

\[
\begin{align*}
10010 &= 12 \text{ HEX} = 18 \text{ dec.} \\
10 &= 2 \text{ HEX} = 2 \text{ dec.}
\end{align*}
\]

Just as in multiplication, you can see shifts being made as the division takes place. Place value is very important in this process. Notice that a subtraction is made only if the divisor is smaller than part of the dividend that is being tested. A one then is placed in the quotient. If the divisor is larger, a zero is placed in the quotient.

Our assembly language program for division will look much the same as the multiplication program. The first part of the program places the appropriate values in their respective memory locations. Figure 11-5 shows the placement of the data.

<table>
<thead>
<tr>
<th>Memory Address</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>At the Start</td>
</tr>
<tr>
<td>Accumulator</td>
<td>Most Significant Byte of dividend</td>
</tr>
<tr>
<td>1100</td>
<td>Divisor</td>
</tr>
<tr>
<td>1101</td>
<td>Least Significant Byte of dividend</td>
</tr>
<tr>
<td>1102</td>
<td>—</td>
</tr>
</tbody>
</table>

*Figure 11-5. Memory Use for Division*
The flowchart for the program is given in Figure 11-6. The accumulator and memory location 1101 begin with the original dividend. Each time through the loop, the contents of memory location 1101 are shifted left, and the accumulator is rotated left. If a carry occurs from the shift of memory location 1101, it will appear as a 1 in the Least Significant Bit of the accumulator when it is rotated. If no carry occurs from the memory 1101 shift, a zero appears in the Least Significant Bit of the accumulator. Thus each time through the loop, the dividend is shifted one place from memory 1101 to the accumulator. This lets the computer compare the divisor with the most significant part of the dividend for a trial division.

Figure 11-6. Division Flowchart
The division is accomplished by comparing the divisor with the accumulator. Every time that the divisor is smaller than, or equal to, the accumulator, the divisor is subtracted from the accumulator and memory 1101 is incremented by one. This means a 1 is appearing in the quotient.

If the divisor is larger than the accumulator, no subtraction is made and memory 1101 is not incremented. This means a zero appears in the quotient.

As the bits in the accumulator and memory 1101 move to the left, the quotient appears in memory 1101 from the right (0 if divisor did not go into dividend, 1 if divisor did go into dividend).

After the loop is completed (8 passes through), the remainder is placed in memory location 1102. When the program is complete, you’ll find the 8-bit quotient in memory location 1101 and the remainder in memory location 1102.

The housekeeping chores are similar to those of the Multiplication Program.

```
10  *=$1000
20   LDX  #8
30   LDA  #$3A
40   STA  $1100
50   LDA  #$16
60   STA  $1101
70   LDA  #4

The main part of the program comes next. The rotate instruction appears in a different form in the loop in this program. We are using ROL A to rotate the accumulator. It works the same way as it did when we rotated a memory location. Only this time, the accumulator's contents are rotated. The loop is again executed 8 times using the X register as a counter.

80   LOOP  ASL  $1101
90   ROL  A
100  CMP  $1100
110  BCC  BRANCH
120  SBC  $1100
130  INC  $1101
140  BRANCH  DEX
150  BNE  LOOP

When the loop has been completed 8 times, the contents of the accumulator (the remainder) is stored in memory location 1102 for ease of display.

160   STA  $1102
170   END
```
Enter and assemble the program now. The assembled program should look like this:

```
0000 10 * = $1000
1000 A208 20 LDX #8
1002 A93A 30 LDA #$3A
1004 8D0011 40 STA $1100
1007 A916 50 LDA #$16
1009 8D0111 60 STA $1101
100C A904 70 LDA #4
100E 0E0111 80 LOOP ASL $1101
1011 2A 90 ROL A
1012 CD0011 0100 CMP $1100
1015 9006 0110 BCC BRANCH
1017 ED0011 0120 SBC $1100
101A EE0111 0130 INC $1101
101D CA 0140 BRANCH DEX
101E D0EE 0150 BNE LOOP
1020 8D0211 0160 STA $1102
1023 0170 END
```

*Figure 11-7. Printout of Assembled Division Program*

Enter the Debugger mode, execute the program, and display the results.

```
EDIT
BUG

DEBUG
G1000
1023 A=22 X=00 Y=00 P=32 S=00
DEBUG
D1101,1102

1101 12 02 Quotient
1102 00 02 Remainder
```

This checks with the hand calculated result.

Try the additional division problems given in Figure 11-8. The answers appear at the end of the chapter exercises.
SUBROUTINES

Dividend | Divisor | Result
---------|---------|--------
MSB (100D) | LSB (1008) | (1003) | Quotient (1101) | Remainder (1102)

| 01 | 00 | 10 |
| 0A | BC | DE |
| 0A | 05 | 9F |
| 05 | AA | 83 |
| 7F | FF | FF |

Figure 11-8. Division Exercises

If you try any other examples, make sure that the quotients do not exceed FF. The program will not work for quotients larger than one byte.

SUBROUTINES

Quite often, a set of instructions must be executed at several different places within a program. Rather than write the instructions at each place they are used in the program, it is more efficient to write the instructions once, as a subroutine to your program. Each time that you wish to execute the set of instructions, you insert the Jump to SubRoutine instruction.

Example:

Suppose that you have a subroutine labeled SUBEEP. The JSR instruction would be:

JSR SUBEEP

Once you have executed the subroutine, you must have a way to get back to the main program. And you must come back to the instruction that follows the JSR instruction that sent you to the subroutine. This is done by placing a RTS (ReTurn from Subroutine) instruction as the last instruction executed in the subroutine.
The subroutine could be “called” from the main program by the JSR SUBEEP instruction. The subroutine would be executed, and control would be returned to the main program as follows:

![Subroutine Flow Diagram]

How does the computer remember where it left the main program?

**Figure 11-9. Subroutine Flow**

Remember the stack that was discussed in Chapters 1 and 5? When a JSR instruction is executed, the computer automatically places the memory address contained in the program counter on the stack. Then, when the RTS instruction is executed at the end of the subroutine, the computer takes the address from the stack and places it back in the program counter. The execution of the main program then continues from that point.
When the JSR instruction is executed, the program counter is “pointing” to the next instruction.

The address 1021 is placed on top of the stack.

The address of the subroutine is placed in the program counter.

The program counter then forces the computer to execute the instructions beginning at location 1100 (the subroutine).

Returning from the Subroutine

The subroutine is executed

When the RTS instruction is executed, the computer goes to the stack . . . .

The address is removed from the stack . . . .

And placed back in the Program Counter.

This forces the computer to execute the next instruction starting at location 1021.
You should remember that the stack is made up of 8-bit memory locations. Addresses may be 16 bits long. Therefore, the process is a little more complicated than the previous description indicates. The address is actually placed on the stack as two separate bytes.

When the RTS instruction is executed, the return address is also picked off the stack in two separate bytes. In other words, the RTS instruction reverses the process performed by the JSR instruction. They work together just like the GO-SUB and RETURN instructions of BASIC.

Subroutines in assembly language may be nested, just as subroutines in BASIC.

**USING A SUBROUTINE**

Let's return to the 8-bit multiplication program used earlier in this chapter and add a sound subroutine that will play a different sound each time one of the bits of the multiplier is used. You will need to insert many new instructions in the original program. Several new instructions will be introduced which use the stack to save and retrieve information.

**THE MAIN PROGRAM (Unchanged Portion)**

<table>
<thead>
<tr>
<th>Line</th>
<th>Instruction</th>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>=8100</td>
<td></td>
<td>Data Initialization</td>
</tr>
<tr>
<td>110</td>
<td>LDX #8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>120</td>
<td>LDA #$12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>130</td>
<td>STA $1102</td>
<td></td>
<td></td>
</tr>
<tr>
<td>140</td>
<td>LDA #$3A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>150</td>
<td>STA $1103</td>
<td></td>
<td></td>
</tr>
<tr>
<td>160</td>
<td>LDA #0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>170</td>
<td>STA $1100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>180</td>
<td>STA $1101</td>
<td></td>
<td></td>
</tr>
<tr>
<td>190</td>
<td>LOOP ASL A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>200</td>
<td>ROL $1100</td>
<td></td>
<td>Multiplication Loop</td>
</tr>
<tr>
<td>210</td>
<td>ASL $1103</td>
<td></td>
<td></td>
</tr>
<tr>
<td>220</td>
<td>BCC SKIP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>230</td>
<td>CLC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>240</td>
<td>ADC $1102</td>
<td></td>
<td></td>
</tr>
<tr>
<td>250</td>
<td>BCC SKIP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>260</td>
<td>INC $1100</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
USING A SUBROUTINE

MAIN PROGRAM (Changed Portion)

270 SKIP PHA
280 PHP
290 TXA
300 PHA
310 JSR SUBEEP —— Here's where you GOSUB
320 PLA —— You come back here
330 TAX
340 PLP
350 PLA
360 DEX
370 BNE LOOP
380 STA $1101
390 BRK —— Program ends here

Subroutine SUBEEP

1024
1025 400 SUBEEP LDA #$C8
1026 410 STA $CE
1027 420 LDA #A0
1028 430 STA $D201
1029 440 LDA $1103,X
1030 450 STA $D200

SUBEEP

1031 460 LDA #$AF
1032 470 ALOOP STA $D201
1033 480 LDA $CE
1034 490 JSR DELAY —— Jump to nested sub

1051 500 SEC —— Return here
1052 510 SBC #$01
1053 520 CMP #$9F
1054 530 BNE ALOOP
1055 540 RTS

1059 550 DELAY LDY #$13 —— Start of nested sub
1060 560 DELAY2 DEY

Nested Subroutine

570 BNE DELAY2
580 DEX
590 BNE DELAY

1061 600 RTS —— End of Nested sub
1062 610 END

The program is executed in the order shown in Figure 11-10. The subroutine SUBEEP is called eight times from the main program. The subroutine labeled DELAY is nested within subroutine SUBEEP.
The frequency of the sound is controlled by the value stored in D200 from the data table at 1103+X. The audio control register at D201 controls the volume of the sound.

Since the 6502 microprocessor has only two index registers (X and Y), a new programming technique is demonstrated at 270–300 and 320–350. The main program and the subroutines use the X register and accumulator for different purposes. Therefore, before the subroutines are called, all the values in registers X, A, and P are saved on the stack so that the registers may be used by the subroutines in their own way.

PHA  Push the Accumulator on the stack
PHP  Push the P register on the stack
TXA and PHA  Transfer X to Z, then Push A on the stack

When JSR is executed, the information goes on the stack in this way.

When SUBEEP's RTS instruction is executed, the return address is pulled off the stack leaving the following data.
The computer then returns to the main program and executes in order:

- **PLA** → Pull off top of stack and
- **TAX** → transfer to X register
- **PLP** → Pull off stack and place in P register
- **PLA** → Pull off stack and place in A register

Notice that values are pulled off the stack in the reverse order of the way they were pushed on. This restores everything to the proper place.

Enter the program as given and assemble it. Then go to DEBUG and enter the data in the data table at 1104 through 110B.

<table>
<thead>
<tr>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1104</td>
<td>48</td>
</tr>
<tr>
<td>1105</td>
<td>60</td>
</tr>
<tr>
<td>1106</td>
<td>78</td>
</tr>
<tr>
<td>1107</td>
<td>90</td>
</tr>
<tr>
<td>1108</td>
<td>A8</td>
</tr>
<tr>
<td>1109</td>
<td>C0</td>
</tr>
<tr>
<td>110A</td>
<td>D8</td>
</tr>
<tr>
<td>110B</td>
<td>F0</td>
</tr>
</tbody>
</table>

*Figure 11-11. Data for Sound Program*

Execute the program and see how you like the notes that are produced by the sound subroutine SUBEEP. If you want to change the notes, change the data shown in Figure 11-11. The frequencies are picked off in reverse order from 110B through 1104.
SUMMARY

You have now seen how two operations for which the computer has no instructions can be performed by putting together groups of other instructions. Although the 6502 microprocessor has no multiplication or division instructions, the operations can be performed as a series of additions, subtractions, and shifts or rotations. You have learned in this chapter to:

- Apply the shift and rotate instructions in programs to multiply and divide numbers;
- Use a demonstration program that multiplies two 8-bit numbers to produce a 16-bit result;
- Use a demonstration program that divides an 8-bit divisor into a 16-bit dividend producing an 8-bit quotient and an 8-bit remainder;
- Use a subroutine called by a JSR (Jump to SubRoutine) instruction and return to the main program with a RTS (ReTurn from Subroutine) instruction;
- Use the stack to save and retrieve information when registers or counters must be used in more than one way;
- Use the following other new instructions:
  - PHA Push the value in the accumulator onto the stack
  - PHP Push the P register's contents onto the stack
  - PLA Pull the top value off the stack and put it into the accumulator
  - PLP Pull the top value off the stack and put it into the P register
- Use a subroutine producing musical tones

EXERCISES

1. Multiply the decimal numbers 28 and 37.
   (a) Using decimal notation
   (b) Using binary notation

   \[
   \begin{array}{c}
   28 \\
   \times 37 \\
   \end{array}
   \]

   (c) Check the binary result

   + __________________
   + __________________
2. According to the flowchart in Figure 11-2, which bit of the multiplier is used first? (most significant, least significant)

Exercises 3–7 refer to the following sketches of the Carry bit and the accumulator before and after one of the shift or rotate instructions are given.

(a) Carry  Accumulator  before  

<table>
<thead>
<tr>
<th>1</th>
<th>01010111</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>10101110</td>
</tr>
</tbody>
</table>

after  

<table>
<thead>
<tr>
<th>1</th>
<th>01010111</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>10101111</td>
</tr>
</tbody>
</table>

(b) Carry  Accumulator  before  

<table>
<thead>
<tr>
<th>1</th>
<th>01010111</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>10101111</td>
</tr>
</tbody>
</table>

after  

<table>
<thead>
<tr>
<th>1</th>
<th>01010111</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10101011</td>
</tr>
</tbody>
</table>

(c) Carry  Accumulator  before  

<table>
<thead>
<tr>
<th>1</th>
<th>01010111</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>00101011</td>
</tr>
</tbody>
</table>

after  

<table>
<thead>
<tr>
<th>1</th>
<th>01010111</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10101011</td>
</tr>
</tbody>
</table>

3. Name the shift or rotate instruction performed in a. 

4. Name the shift or rotate instruction performed in b. 

5. Name the shift or rotate instruction performed in c. 

6. Name the shift or rotate instruction performed in d. 

7. Fill in the blanks which would result from successive operations for your answers from Exercises 3, 4, 5, and 6.

Originally:  Carry  Accumulator  

| 1 | 01010111 |

The instructions for your answers to: 3, then 4, then 5, then 6 are executed.

Final conditions:  Carry  Accumulator  

| 0 | 01010111 |

8. The 8-bit multiplication program of Figure 11-3 is to be used to multiply 120 (decimal) by 73 (decimal).

(a) What hexadecimal values must be loaded into which two memory locations?

<table>
<thead>
<tr>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
</table>
(b) Show how to use the DEBUGGER to load the values by one command.

```
DEBUG
```

9. The assembly language instruction used to call a subroutine labeled SUBBY would be:

10. The last instruction executed by a subroutine must be: ____________

**ANSWERS**

Answers to Figure 11-4

<table>
<thead>
<tr>
<th>Multiplier</th>
<th>Multiplicand</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>03</td>
<td>09</td>
<td>00 1B</td>
</tr>
<tr>
<td>1A</td>
<td>E4</td>
<td>17 28</td>
</tr>
<tr>
<td>3C</td>
<td>D8</td>
<td>32 A0</td>
</tr>
<tr>
<td>A4</td>
<td>C2</td>
<td>7C 48</td>
</tr>
<tr>
<td>FF</td>
<td>FF</td>
<td>FE 01</td>
</tr>
</tbody>
</table>

Answers to Figure 11-8

<table>
<thead>
<tr>
<th>Dividend</th>
<th>Divisor</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>01 00</td>
<td>10</td>
<td>10 00</td>
</tr>
<tr>
<td>0A BC</td>
<td>DE</td>
<td>0C 54</td>
</tr>
<tr>
<td>0A 05</td>
<td>9F</td>
<td>10 15</td>
</tr>
<tr>
<td>05 AA</td>
<td>83</td>
<td>0B 09</td>
</tr>
<tr>
<td>7F FF</td>
<td>FF</td>
<td>80 7F</td>
</tr>
</tbody>
</table>

1. (a) Using decimal notation

```
28
x 37
196
84
1036
```
(b) Using binary notation

<table>
<thead>
<tr>
<th>18</th>
<th>00011100</th>
</tr>
</thead>
<tbody>
<tr>
<td>x37</td>
<td>00100101</td>
</tr>
<tr>
<td></td>
<td>00011100</td>
</tr>
<tr>
<td></td>
<td>00000000</td>
</tr>
<tr>
<td></td>
<td>00011100</td>
</tr>
<tr>
<td></td>
<td>00000000</td>
</tr>
<tr>
<td></td>
<td>00000000</td>
</tr>
<tr>
<td></td>
<td>00111100</td>
</tr>
<tr>
<td></td>
<td>01000001100</td>
</tr>
</tbody>
</table>

(c) 

\[
\begin{array}{c}
4 \\
+8 \\
\hline
+1024 \\
1036 \\
\end{array}
\]

2. Most Significant Bit
3. ASL A
4. ROL A
5. LSR A
6. ROR A
7. Final conditions:

<table>
<thead>
<tr>
<th>Carry</th>
<th>Accumulator</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00010111</td>
</tr>
</tbody>
</table>

8. (a) 

<table>
<thead>
<tr>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1003</td>
<td>49</td>
</tr>
<tr>
<td>1008</td>
<td>78</td>
</tr>
</tbody>
</table>

(b) 

DEBUG
C1003<49 , , , , 78

9. JSR SUBBY
10. RTS
Chapter 12

Programming Practice

Much can be learned about programming by using, analyzing, and modifying programs written by other people. By this time, you are familiar with the use of the Atari Assembler Cartridge and should feel at ease using it. Therefore, we have devoted this last chapter to programs for you to use and modify to fit your own needs. Suggestions are given for some modifications, and you will no doubt think of others.

We have not covered all of the features of the Assembler Cartridge but have tried to present those that we think you will use most.

Programming requires much practice. Therefore, you should try other programs to investigate all aspects of the 6502 instruction set, as well as Atari Assembler Cartridge commands and statements. Seek out those instructions that you are not familiar with, study the descriptions, and write some short simple programs that make use of them.

The assembler can be used to develop machine language subroutines that can speed up and make your BASIC programs more powerful. Many applications using sound effects, high speed graphics, animation, and certain functions not available in BASIC can be performed by machine language subroutines.

As we cautioned earlier, your machine language programs require their own area of memory. If you are using assembled subroutines accessed from BASIC, you must be careful that your machine language programs are assembled in an area of memory that is not used by the BASIC program. There is an area of memory from 0600 (1536 decimal) through 06FF (1791 decimal) that can be used for this purpose. These 256 bytes that are available will be adequate for most of your machine language subroutines. All programs in this chapter use this area. There are also 7 bytes of zero page memory that have been reserved for your use by the BASIC cartridge (locations CB through D1 or 203 decimal, through 209, decimal). In addition, locations D4 and D5 of zero page are used to return parameters (numerical values) from machine language subroutines to BASIC through the USR function.

The Assembler User's Manual, supplied with your Assembler Cartridge, contains several entertaining programs. The manual also shows a method to prepare an assembled subroutine and a BASIC program which will access the sub-
routine. We will concentrate in this chapter on similar programs that use only the Assembler Cartridge. You may rewrite them to be accessed from BASIC if you wish. Our first program will demonstrate both methods of using the assembled program.

BASIC language can perform logic functions by means of the statements NOT, AND and OR. There is another logic function in the 6502 instruction set, the Exclusive OR. Our first program demonstrates its use.

**USING A LOGIC FUNCTION**

The 6502 microprocessor has instructions that extend the logic functions NOT, AND and OR used in BASIC. One of these instructions performs the logic function Exclusive OR (Assembly mnemonic EOR). This instruction compares two binary numbers, one bit position at a time. If the two bits are alike, a zero is placed in the corresponding bit position of the result. If one of the compared bits is a 1 (one) and the other is a 0 (zero), a 1 (one) is placed in the corresponding bit position of the result.

Example:

\[
\begin{array}{ccccccc}
1 & 0 & 1 & 1 & 0 & 1 & 1 \\
0 & 0 & 0 & 1 & 1 & 1 & 0 \\
1 & 0 & 1 & 0 & 1 & 0 & 1 \\
\end{array}
\]

You see that a 1 results in a given bit position if there is a one in the corresponding bit position of either one number or the other number, but not if there is a one in the corresponding bit position of both numbers.

The above example shows that the result of performing an Exclusive OR on the hex numbers B7 and 1E is A9.

Let's now write an assembly language program to perform the same Exclusive OR as the example.

**EOR SOURCE PROGRAM**

```
100 *=$0600
110 LDA #$B7
120 EOR #$1E
130 STA $1000
140 END
```
Enter and assemble the program.

**ASSEMBLED DISPLAY**

<table>
<thead>
<tr>
<th>Line</th>
<th>Instruction</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>130</td>
<td>STA $1000</td>
<td></td>
</tr>
<tr>
<td>140</td>
<td>END</td>
<td></td>
</tr>
</tbody>
</table>

**ASM**

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>100</td>
<td>*= $0600</td>
</tr>
<tr>
<td>0600</td>
<td>A9B7</td>
<td>110</td>
</tr>
<tr>
<td>0602</td>
<td>491E</td>
<td>120</td>
</tr>
<tr>
<td>0604</td>
<td>8D0010</td>
<td>130</td>
</tr>
</tbody>
</table>

**END**

Enter the DEBUGGER and execute the program.

**DEBUG**

```
G0600
0607   A=A9 X=00 Y=00 P=B0 S=00
```

Display memory
1000 to see EOR result

```
D1000,1000
```

1000 A9

```
```
There it is, very short and quick. Now let’s see how we might access a similar subroutine from BASIC. The values to be Exclusive ORed will be passed from the BASIC subroutine to the subroutine. The subroutine will pull the values off the stack, EOR them and pass the result back to BASIC. The flow of the program and subroutine is shown in Figure 12-1.

![Logic Program Flowchart](image)

The BASIC program is very short. Do not enter it in the computer yet. Remember the procedure mentioned earlier.

```
100   REM ** EOR SUBROUTINE PROGRAM **
110   GR.0
120   A = USR(1536, 183, 30)
130   PRINT "183 EXCLUSIVE OR WITH 30 IS":A
140   END
```

Notice the USR function at line 120. The data in parentheses following the address 1536 is placed on the stack for use by the subroutine in this order:

<table>
<thead>
<tr>
<th>STACK</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Top</td>
<td></td>
</tr>
<tr>
<td>aa</td>
<td>number of bytes passed</td>
</tr>
<tr>
<td>bb</td>
<td>high byte of 1st number</td>
</tr>
<tr>
<td>cc</td>
<td>low byte of 1st number</td>
</tr>
<tr>
<td>dd</td>
<td>high byte of 2nd number</td>
</tr>
<tr>
<td>ee</td>
<td>low byte of 2nd number</td>
</tr>
</tbody>
</table>

where aa, bb, cc, dd, and ee are hexadecimal representations
The machine language subroutine will pull the top byte off the stack. It is not used. The high byte of the 1st number is then pulled off and saved (in memory $CC$). Then the low byte of the 1st number is pulled off and saved (in memory $CD$). You now have this condition:

<table>
<thead>
<tr>
<th>STACK</th>
<th>MEMORY</th>
</tr>
</thead>
<tbody>
<tr>
<td>Top</td>
<td></td>
</tr>
<tr>
<td>dd</td>
<td>00CC</td>
</tr>
<tr>
<td>ee</td>
<td>00CD</td>
</tr>
</tbody>
</table>

The subroutine then pulls off the high byte of the second number (from the stack) and Exclusive ORs it with the high byte of the first number (from memory $CC$). The result is stored in memory $D5$. You now have:

<table>
<thead>
<tr>
<th>STACK</th>
<th>MEMORY</th>
</tr>
</thead>
<tbody>
<tr>
<td>Top</td>
<td></td>
</tr>
<tr>
<td>ee</td>
<td>00CC</td>
</tr>
<tr>
<td></td>
<td>00CD</td>
</tr>
<tr>
<td></td>
<td>00D5</td>
</tr>
</tbody>
</table>

Next, the subroutine pulls off the low byte of the second number (from the stack) and Exclusive ORs it with the low byte of the first number (from memory $CD$). The result is stored in memory $D4$. You now have:

<table>
<thead>
<tr>
<th>STACK</th>
<th>MEMORY</th>
</tr>
</thead>
<tbody>
<tr>
<td>Top</td>
<td></td>
</tr>
<tr>
<td>??</td>
<td>00CC</td>
</tr>
<tr>
<td></td>
<td>00CD</td>
</tr>
<tr>
<td></td>
<td>00D4</td>
</tr>
<tr>
<td></td>
<td>00D5</td>
</tr>
</tbody>
</table>

Last of all, the RTS (Return from subroutine) instruction is executed. The values stored in $D5$ and $D4$ are passed back to the BASIC program as the decimal value for the variable A.

This is what the machine language subroutine looks like:
ENTERING THE SUBROUTINE

Now, go through the steps outlined in the Assembler User's Manual.

1. Place the Assembler Cartridge in the computer and enter the assembly language subroutine.

2. Assemble the program to make sure that you have no errors.

<table>
<thead>
<tr>
<th>EDIT</th>
<th>ASM</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0100</td>
</tr>
<tr>
<td>0600</td>
<td>68</td>
</tr>
<tr>
<td>0601</td>
<td>68</td>
</tr>
<tr>
<td>0602</td>
<td>85CC</td>
</tr>
<tr>
<td>0603</td>
<td>68</td>
</tr>
<tr>
<td>0604</td>
<td>85CD</td>
</tr>
<tr>
<td>0605</td>
<td>68</td>
</tr>
<tr>
<td>0606</td>
<td>45CC</td>
</tr>
<tr>
<td>060A</td>
<td>85D5</td>
</tr>
<tr>
<td>060C</td>
<td>68</td>
</tr>
<tr>
<td>060D</td>
<td>45CD</td>
</tr>
<tr>
<td>060F</td>
<td>85D4</td>
</tr>
<tr>
<td>0611</td>
<td>60</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
3. Use 3(a) for cassette storage or 3(b) for disk storage and assemble again.
   (a) For cassette:

   [Diagram: Cassette]
   EDIT
   ASM,,#C:
   Press RETURN
   After the "BEEP" press RETURN AGAIN

   (b) For disk:

   [Diagram: Disk]
   ASM,,#D:EXOR.OBJ
   Press RETURN

4. Take out the Assembler Cartridge and replace it with the BASIC cartridge.
5. Wait for the BASIC READY prompt.

6. Use 6(a) for cassette or 6(b) for disk.
   (a) Follow the normal procedure for loading from tape with:

   [Diagram: Tape]
   CLOAD
   and press RETURN

   (b) Type in DOS to raise the Disk Operating System.

   DISK OPERATING SYSTEM
   COPYRIGHT 1979 ATARI

   A. DISK DIRECTORY I. FORMAT DISK
   B. RUN CARTRIDGE J. DUPLICATE DISK
   C. COPY FILE K. BINARY SAVE
   D. DELETE FILE(S) L. BINARY LOAD
   E. RENAME FILE M. RUN AT ADDRESS
   F. LOCK FILE N. DEFINE DEVICE
   G. UNLOCK FILE O. DUPLICATE FILE
   H. WRITE DOS FILE

   SELECT ITEM
Then type L to load a binary file.

SELECT ITEM
L
LOAD FROM WHAT FILE?

Now, type: EXOR.OBJ and press RETURN

SELECT ITEM
L
LOAD FROM WHAT FILE?
EXOR.OBJ

After the file is loaded, the computer will return to the SELECT ITEM PROMPT.

SELECT ITEM

Type B and press return to return to BASIC.

SELECT ITEM
B
READY

7. You should now be back in the BASIC READY mode.
8. Enter the BASIC program.

```
100 REM ** EOR SUBROUTINE PROGRAM **
110 CLS
120 A = USR(1536,183,30)
130 PRINT "183 EXCLUSIVE OR WITH 30 IS";A
140 END
```

You are now ready to execute your program—ALMOST. First check to make sure the machine language subroutine is in memory. This can be done by checking the first location of the subroutine. Type:

```
PRINT PEEK(1536) and press RETURN
```

The computer should respond with the value 104 (the decimal equivalent of the first machine language instruction, PLA).

If the computer responds correctly, execute the program with the RUN command. If it doesn't correspond correctly, go back to the Assembler Cartridge and try it all again. Verrrrry carefully.

![RUN 169](image)

The decimal equivalent of the EOR result.

128+32+8+1 or

10101001 in binary

You may use the Exclusive OR with other values by changing line 120 of the BASIC program.

```
120 A = USR(1536,183,30)
```

These values may change.

_Do not_ change the value 1536 as that is the memory location where the subroutine begins. The other two values (183 and 30) may be changed to any positive integer in the range of 0 through 65535.

The remaining programs in this chapter will show only the versions of the programs that may be run directly in the Debugger mode of the assembler. Feel free to change them to subroutines to be accessed from BASIC as in the EOR program.
PROGRAM TO SOUND OFF

This program uses three sound registers (or channels) for playing music chords. The data is placed in the appropriate registers, and the sound produced is sent to your TV set. Be sure that you have the volume of your TV set turned to a reasonable level before running the program. The program controls the volume produced, but the TV volume control must be set high enough to make an audible sound.

The last chord produced will keep playing until a new note is placed in the sound registers. Therefore, the last part of the program sets the volume down to zero. There are other ways to accomplish the same result, but why not let the computer do it for you?

We have used comments in the source program that label each part of the program and the data values used. It is good programming practice to do so. If you save the source program on tape or disk, the comments will help you decipher the program when you enter it again. The flow of the program is shown in Figure 12-2.

Figure 12-2. Sound Off Program Flow

SOUND OFF SOURCE PROGRAM

100 *=$0600
110 ;FIRST CHORD SECTION
120 LDA #$79            FIRST NOTE
130 STA $D200
140 LDA #$88            SECOND NOTE
150 STA $D202
160 LDA #$99            THIRD NOTE
170 STA $D204
180 LDA #$AF            TONE AND VOLUME
190 STA $D201
Enter the program, assemble it, and execute it in the usual way. See if you like the sounds produced. If not, take a look at the table of music notes in Figure 12-3. You may alter frequency values in the program as long as you stay in the range of 0 (highest note) to FF hexadecimal (lowest note).

To understand the tone and volume controls, you must look at the volume in binary form.

Example:

195 decimal = AF hexadecimal

= 1010 1111

<table>
<thead>
<tr>
<th>A</th>
<th>F</th>
</tr>
</thead>
</table>

selects pure tone (no noise)
<table>
<thead>
<tr>
<th>Natural Scale Note</th>
<th>Frequency Value (hex)</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>1D</td>
</tr>
<tr>
<td>B</td>
<td>1F</td>
</tr>
<tr>
<td>A#</td>
<td>21</td>
</tr>
<tr>
<td>A</td>
<td>23</td>
</tr>
<tr>
<td>G#</td>
<td>25</td>
</tr>
<tr>
<td>G</td>
<td>28</td>
</tr>
<tr>
<td>F#</td>
<td>2A</td>
</tr>
<tr>
<td>F</td>
<td>2D</td>
</tr>
<tr>
<td>E</td>
<td>2F</td>
</tr>
<tr>
<td>D#</td>
<td>32</td>
</tr>
<tr>
<td>D</td>
<td>35</td>
</tr>
<tr>
<td>C#</td>
<td>39</td>
</tr>
<tr>
<td>C</td>
<td>3C</td>
</tr>
<tr>
<td>B</td>
<td>40</td>
</tr>
<tr>
<td>A#</td>
<td>44</td>
</tr>
<tr>
<td>A</td>
<td>48</td>
</tr>
<tr>
<td>G#</td>
<td>4C</td>
</tr>
<tr>
<td>G</td>
<td>51</td>
</tr>
<tr>
<td>F#</td>
<td>55</td>
</tr>
<tr>
<td>F</td>
<td>5B</td>
</tr>
<tr>
<td>E</td>
<td>60</td>
</tr>
<tr>
<td>D#</td>
<td>66</td>
</tr>
<tr>
<td>D</td>
<td>6C</td>
</tr>
<tr>
<td>C#</td>
<td>72</td>
</tr>
<tr>
<td>C</td>
<td>79</td>
</tr>
<tr>
<td>B</td>
<td>80</td>
</tr>
<tr>
<td>A#</td>
<td>88</td>
</tr>
<tr>
<td>A</td>
<td>90</td>
</tr>
<tr>
<td>G#</td>
<td>99</td>
</tr>
<tr>
<td>G</td>
<td>A2</td>
</tr>
<tr>
<td>F#</td>
<td>AC</td>
</tr>
<tr>
<td>F</td>
<td>B6</td>
</tr>
<tr>
<td>E</td>
<td>C1</td>
</tr>
<tr>
<td>D#</td>
<td>CC</td>
</tr>
<tr>
<td>D</td>
<td>D9</td>
</tr>
<tr>
<td>C#</td>
<td>E6</td>
</tr>
<tr>
<td>C</td>
<td>F3</td>
</tr>
</tbody>
</table>

*Figure 12-3. Approximate Values for Three-Octave Scale*
The first hex digit (A in example) controls the noise content of the tone. We chose a pure tone. The second hex digit controls the volume. Its range may be from 0 (lowest volume = off) to F (highest volume).

You should try different values to obtain different sound effects. All you need to do is change the values for frequency, tone and volume. Check the comments in the source program to find the values to be changed.

1. Changes may be made in the source program.
   (a) In the Edit mode change lines:
       120, 140, 160, 180 for the first chord
       240, 260, 280, and 300 for second chord
   (b) Assemble the program again.
   (c) Execute it from the Debugger mode.

2. Changes may be made in the object program.
   Use the Debugger mode to change memories:
   0601, 0606, 060B and 0610 for first chord
   061E, 0623, 0628 and 062D for second chord

PLAY NOTES PROGRAM

This program uses a single sound channel to play a series of notes. A time delay is used between the notes so that they can be heard for a reasonable length of time. Once again, make sure that the volume on the TV set is turned up. By looking at the frequency table of Figure 12-3, you can see that the program plays one octave of the scale.

ASSEMBLER PROGRAM

```
100  *=0600
110  ;PLAY NOTES
120  LDA #$AF
130  STA $D201
140  LDA #$F3
150  JSR RPT
160  LDA #$D9
170  JSR RPT
180  LDA #$C1
190  JSR RPT
200  LDA #$B6
210  JSR RPT
220  LDA #$A2
230  JSR RPT
```
Enter the program and assemble and execute it. Does it sound reasonable to you? If not, changes can be made where the frequencies are loaded into the accumulator. The duration of the notes can be changed by the LDY and LDX instructions in the RPT subroutine.

This program could be changed so that it would access the frequencies from a data table. The frequencies could then be accessed by an indexed load instruction. This might be a little difficult since the X and Y registers are already being used in the time delay subroutine. You would have to save their values on the stack, and retrieve them after the time delay had been completed. We have covered the use of the stack instructions before. Therefore, you know how to make these changes if you wish.

Notice in line 300 and 310 that the value AO was stored in memory location D201. This sets the volume to zero. If those two lines were omitted, the last note would keep playing even though the program had ended.

PROGRAM TO SHAPE SOUND

This program uses one sound channel to shape the sound that is produced. If you look at a graph of the volume of the sound produced over a period of time, it would look something like this:
You can see that the sound goes through three distinct periods. You have used the frequency, tone and volume variables before. Now, you have a chance actually to vary the shape of the sound produced.

We have chosen one of the three time constants—the decay time. It will be changed as the program is executed. It is varied from 50 to 200 (decimal) within a loop in steps of 25 (50, 75, 100, 125, 150, 175 and 200). The sound is thus shaped as follows:

```
SOUND SHAPER PROGRAM
100 *=0600
110 ;INITIALIZE TIMES
120  LDA #$3C    FREQUENCY
130  STA $D200
140  LDA #$32    PEAK
150  STA #$CD
160  LDA #$0A    ATTACK
170  STA #$CC
180  LOOP LDA $1000    COUNTER
190  TAX
200  LDA $1000,X
210  STA #$CE    DECAY
220  DEC $1000
230  LDA #$A0    VOLUME
240  ATTK STA $D201
250  LDX #$CC
260  JSR DELAY
270  CLC
```
280 ADC #$01
290 CMP #$80
300 BNE ATTK
310 LDA #$0E
320 PEAK LDX $CD
330 JSR DELAY
340 SEC
350 SBC #$01
360 BNE PEAK
370 LDA #$AF TONE,VOLUME
380 DCA Y STA $D201
390 LDX $CE
400 JSR DELAY
410 SEC
420 SBC #$01
430 CMP #$9F
440 BNE DCA Y
450 LDA $1000
460 BNE LOOP
470 BRK
480 DELAY LDY #$13
490 ROUND DEY
500 BNE ROUND
510 DEX
520 BNE DELAY
530 RTS
540 END

This may be your longest program yet. Enter and assemble it. Check the highest memory location used. Remember, we must stay between 0600 and 06FF or we will run into some memory that we shouldn’t be using.

---

EDIT

BUG

DEBUG

Last location looks OK

So go on to DEBUG mode
Before you execute the program, remember that you may not have entered the data to be used as the counter and the decay times. Refer to the table below, and enter the data in the Debugger mode.

<table>
<thead>
<tr>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>07</td>
</tr>
<tr>
<td>1001</td>
<td>C8</td>
</tr>
<tr>
<td>1002</td>
<td>AF</td>
</tr>
<tr>
<td>1003</td>
<td>96</td>
</tr>
<tr>
<td>1004</td>
<td>7D</td>
</tr>
<tr>
<td>1005</td>
<td>64</td>
</tr>
<tr>
<td>1006</td>
<td>4B</td>
</tr>
<tr>
<td>1007</td>
<td>32</td>
</tr>
</tbody>
</table>

7 notes to be played

the decay times

Figure 12-4. Data for Sound Shaper Program

Now execute the program. You should be able to detect the differences in decay time.

You might want to experiment with the other parameters (attack time, peak time, frequency or volume). Feel free to modify our program to suit your own purposes.

PROGRAM TO PRINT ON THE SCREEN

Hidden away in your Atari’s memory are lots of nice little subroutines that might save you much time and effort. We found one hidden away at memory location $F6A4. It will display on the screen the character whose ATASCII code is in the accumulator.

Example:

```
LDA #$41
ATASCII code for the letter A
JSR $F6A4
location of display subroutine

jump to subroutine
```

The instructions above would display the letter A, and then the cursor would move one place to the right, ready to display whatever comes next.

You must be careful when you use this subroutine if you are using the X and Y register in your program. Your values (in X and Y) will be destroyed by the subroutine as it also uses these registers. Therefore, you must make provision to save them (the stack is the ideal place for this).

The program that we will use is short, but you will have to load in lots of
data in a data table. The X register and the Absolute Indexed Addressing mode will be used to load the accumulator with the ATASCII codes from the data table. The ATASCII Character Set is given in Appendix F.

PRINT ON THE SCREEN

100  *=0600
110  ;PRINT ON THE SCREEN
120  LDX #$2B NUMBER OF CHARACTERS
130  LOOP TXA
140  PHA SAVE NUMBER
150  LDA $1100,X LOAD CODE
160  JSR $F6A4 DISPLAY CHARACTER
170  PLA GET NUMBER BACK
180  TAX
190  DEX COUNT DOWN
200  BNE LOOP GO BACK IF NOT DONE
210  LOOP1 JMP LOOP1 CIRCLE HERE
220  END

Enter and assemble the program. Now comes the chore of loading the data. Go into the Debugger mode and enter the data as shown.

Press RETURN at end of each line

What is all that?
We're not going to spoil the purpose by telling you what all those numbers represent yet. If you can't wait, Figure 12-5 shows what character is represented by each code.

Before you run the program, clear the screen by holding down the \textbf{SHIFT} key and pressing the \textbf{CLEAR} key.

Screen is cleared

Now, execute the program by typing:\n
\texttt{G0600} \hspace{1cm} \text{and press RETURN}

This is what you see:

\texttt{G0600}
You have reached the END

The last lines of the program keep the display on the screen while the computer goes around and around doing nothing. Press the \texttt{BREAK} key to stop the program.

Here are the ATASCII codes used in the program.

<table>
<thead>
<tr>
<th>Memory Address</th>
<th>Hex ATASCII Code</th>
<th>Character Displayed</th>
</tr>
</thead>
<tbody>
<tr>
<td>112B</td>
<td>59</td>
<td>Y</td>
</tr>
<tr>
<td>112A</td>
<td>6F</td>
<td>o</td>
</tr>
<tr>
<td>1129</td>
<td>75</td>
<td>u</td>
</tr>
<tr>
<td>1128</td>
<td>20</td>
<td>SPACE</td>
</tr>
<tr>
<td>1127</td>
<td>68</td>
<td>h</td>
</tr>
<tr>
<td>1126</td>
<td>61</td>
<td>a</td>
</tr>
<tr>
<td>1125</td>
<td>76</td>
<td>v</td>
</tr>
<tr>
<td>Hex Code</td>
<td>ASCII Code</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>------------</td>
<td>-------------</td>
</tr>
<tr>
<td>1124</td>
<td>65</td>
<td>e</td>
</tr>
<tr>
<td>1123</td>
<td>9B</td>
<td>CARRIAGE RETURN</td>
</tr>
<tr>
<td>1122</td>
<td>72</td>
<td>r</td>
</tr>
<tr>
<td>1121</td>
<td>65</td>
<td>e</td>
</tr>
<tr>
<td>1120</td>
<td>61</td>
<td>a</td>
</tr>
<tr>
<td>111F</td>
<td>63</td>
<td>c</td>
</tr>
<tr>
<td>111E</td>
<td>68</td>
<td>h</td>
</tr>
<tr>
<td>111D</td>
<td>65</td>
<td>e</td>
</tr>
<tr>
<td>111C</td>
<td>64</td>
<td>d</td>
</tr>
<tr>
<td>111B</td>
<td>9B</td>
<td>CARRIAGE RETURN</td>
</tr>
<tr>
<td>111A</td>
<td>74</td>
<td>t</td>
</tr>
<tr>
<td>1119</td>
<td>68</td>
<td>h</td>
</tr>
<tr>
<td>1118</td>
<td>65</td>
<td>e</td>
</tr>
<tr>
<td>1117</td>
<td>20</td>
<td>SPACE</td>
</tr>
<tr>
<td>1116</td>
<td>45</td>
<td>E</td>
</tr>
<tr>
<td>1115</td>
<td>4E</td>
<td>N</td>
</tr>
<tr>
<td>1114</td>
<td>44</td>
<td>D</td>
</tr>
<tr>
<td>1113</td>
<td>9B</td>
<td>CARRIAGE RETURN</td>
</tr>
<tr>
<td>1112</td>
<td>00</td>
<td>☺</td>
</tr>
<tr>
<td>1111</td>
<td>00</td>
<td>☻</td>
</tr>
<tr>
<td>1110</td>
<td>00</td>
<td>☻</td>
</tr>
<tr>
<td>110F</td>
<td>9B</td>
<td>CARRIAGE RETURN</td>
</tr>
<tr>
<td>110E</td>
<td>06</td>
<td>☻</td>
</tr>
<tr>
<td>110D</td>
<td>0D</td>
<td>☻</td>
</tr>
<tr>
<td>110C</td>
<td>07</td>
<td>☻</td>
</tr>
<tr>
<td>110B</td>
<td>9B</td>
<td>CARRIAGE RETURN</td>
</tr>
<tr>
<td>110A</td>
<td>16</td>
<td>☻</td>
</tr>
<tr>
<td>1109</td>
<td>00</td>
<td>☻</td>
</tr>
<tr>
<td>1108</td>
<td>02</td>
<td>☻</td>
</tr>
<tr>
<td>1107</td>
<td>9B</td>
<td>CARRIAGE RETURN</td>
</tr>
<tr>
<td>1106</td>
<td>07</td>
<td>☻</td>
</tr>
<tr>
<td>1105</td>
<td>0E</td>
<td>☻</td>
</tr>
<tr>
<td>1104</td>
<td>06</td>
<td>☻</td>
</tr>
<tr>
<td>1103</td>
<td>9B</td>
<td>CARRIAGE RETURN</td>
</tr>
<tr>
<td>1102</td>
<td>9B</td>
<td>CARRIAGE RETURN</td>
</tr>
<tr>
<td>1101</td>
<td>9B</td>
<td>CARRIAGE RETURN</td>
</tr>
</tbody>
</table>

*Figure 12-5. ATASCII Codes for Print on the Screen*
Now, you can create your own graphics and messages by replacing our codes with any that you choose from Appendix F. You can lengthen or shorten the data table, but be sure you put the corresponding number in line 120 of the source program. Fill the screen and have fun!!

YOU'RE ON YOUR OWN

We are going to leave you at this point. You now have a fundamental knowledge of how the Assembler Cartridge works. You also have a background in the 6502 instruction set. With the understanding you now have, you can explore other assembly language features. These can be found in your Atari Assembler User's Manual. You should also explore the operation of all instructions in the 6502 set. The more that you work with assembly language programming, the more you will probably like it.

If you intend to do a sizable amount of assembly language programming, we recommend the use of the Atari 810 or 815 Disk Drive along with the Atari 820 or 822 or 825 Printer. We have used the 810 Disk Drive along with the Atari 820 printer and find them both very convenient for assembly language programming.

Good Programming!
Appendix A

6502 Instructions—
Flags Affected

X in the box means the flag is affected. The result will depend upon the condition, or status, resulting from a previous operation. A one (1) indicates that the flag is set, a zero (0) that a flag is reset.

<table>
<thead>
<tr>
<th>Mnemonic Code</th>
<th>Operation Performed</th>
<th>Status Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADC</td>
<td>Add memory to accumulator with carry</td>
<td>X X X X</td>
</tr>
<tr>
<td>AND</td>
<td>AND memory with accumulator</td>
<td>X     X</td>
</tr>
<tr>
<td>ASL</td>
<td>Shift left one bit (memory or accum.)</td>
<td>X     X X</td>
</tr>
<tr>
<td>BCC</td>
<td>Branch on carry clear (If C=0)</td>
<td></td>
</tr>
<tr>
<td>BCS</td>
<td>Branch on carry set (If C=1)</td>
<td></td>
</tr>
<tr>
<td>BEQ</td>
<td>Branch on result zero (If Z=1)</td>
<td></td>
</tr>
<tr>
<td>BIT</td>
<td>Test bits in accumulator with memory</td>
<td>X     X X</td>
</tr>
<tr>
<td>BMI</td>
<td>Branch on result minus (If N=1)</td>
<td></td>
</tr>
<tr>
<td>BNE</td>
<td>Branch on result not zero (If Z=0)</td>
<td></td>
</tr>
<tr>
<td>BPL</td>
<td>Branch on result plus (If N=0)</td>
<td></td>
</tr>
<tr>
<td>BRK</td>
<td>Force break</td>
<td>1</td>
</tr>
<tr>
<td>BVC</td>
<td>Branch on overflow clear (If V=0)</td>
<td></td>
</tr>
<tr>
<td>BVS</td>
<td>Branch on overflow set (If V=1)</td>
<td></td>
</tr>
<tr>
<td>CLC</td>
<td>Clear carry flag</td>
<td></td>
</tr>
<tr>
<td>CLD</td>
<td>Clear decimal mode</td>
<td>0     0</td>
</tr>
<tr>
<td>CLI</td>
<td>Clear interrupt disable flag</td>
<td></td>
</tr>
<tr>
<td>CLV</td>
<td>Clear overflow flag</td>
<td>0     0</td>
</tr>
<tr>
<td>CMP</td>
<td>Compare memory and accumulator</td>
<td>X     X X</td>
</tr>
<tr>
<td>CPX</td>
<td>Compare memory and index X</td>
<td>X     X X</td>
</tr>
<tr>
<td>CPY</td>
<td>Compare memory and index Y</td>
<td>X     X X</td>
</tr>
<tr>
<td>DEC</td>
<td>Decrement memory by one</td>
<td>X     X</td>
</tr>
<tr>
<td>DEX</td>
<td>Decrement index X by one</td>
<td>X     X X</td>
</tr>
<tr>
<td>DEY</td>
<td>Decrement index Y by one</td>
<td>X     X X</td>
</tr>
<tr>
<td>EOR</td>
<td>Exclusive OR memory with accumulator</td>
<td>X     X X</td>
</tr>
<tr>
<td>Mnemonic Code</td>
<td>Operation Performed</td>
<td>Status Flags</td>
</tr>
<tr>
<td>--------------</td>
<td>----------------------------------------------------------</td>
<td>--------------</td>
</tr>
<tr>
<td>INC</td>
<td>Increment memory by one</td>
<td>X X</td>
</tr>
<tr>
<td>INX</td>
<td>Increment index X by one</td>
<td>X X</td>
</tr>
<tr>
<td>INY</td>
<td>Increment index Y by one</td>
<td>X X</td>
</tr>
<tr>
<td>JMP</td>
<td>Jump to new location</td>
<td></td>
</tr>
<tr>
<td>JSR</td>
<td>Jump to new location save rtn. add.</td>
<td></td>
</tr>
<tr>
<td>LDA</td>
<td>Load accumulator from memory</td>
<td>X X</td>
</tr>
<tr>
<td>LDX</td>
<td>Load index X from memory</td>
<td>X X</td>
</tr>
<tr>
<td>LDY</td>
<td>Load index Y from memory</td>
<td>X X</td>
</tr>
<tr>
<td>LSR</td>
<td>Shift right one bit (memory or accum.)</td>
<td>0</td>
</tr>
<tr>
<td>NOP</td>
<td>No operation</td>
<td></td>
</tr>
<tr>
<td>ORA</td>
<td>OR memory with accumulator</td>
<td>X X</td>
</tr>
<tr>
<td>PHA</td>
<td>Push accumulator on stack</td>
<td></td>
</tr>
<tr>
<td>PHP</td>
<td>Push processor status on stack</td>
<td></td>
</tr>
<tr>
<td>PLA</td>
<td>Pull accumulator from stack</td>
<td>X X</td>
</tr>
<tr>
<td>PLP</td>
<td>Pull processor status from stack</td>
<td>X X X X X X X</td>
</tr>
<tr>
<td>ROL</td>
<td>Rotate one bit left (mem. or accum.)</td>
<td>X X</td>
</tr>
<tr>
<td>ROR</td>
<td>Rotate one bit right (mem. or accum.)</td>
<td>X X</td>
</tr>
<tr>
<td>RTI</td>
<td>Return from interrupt</td>
<td>X X X X X X X</td>
</tr>
<tr>
<td>RTS</td>
<td>Return from subroutine</td>
<td></td>
</tr>
<tr>
<td>SBC</td>
<td>Subtract memory and borrow from accum.</td>
<td>X X X</td>
</tr>
<tr>
<td>SEC</td>
<td>Set carry flag</td>
<td>1</td>
</tr>
<tr>
<td>SED</td>
<td>Set decimal mode</td>
<td>1</td>
</tr>
<tr>
<td>SEI</td>
<td>Set interrupt disable flag</td>
<td></td>
</tr>
<tr>
<td>STA</td>
<td>Store accumulator in memory</td>
<td></td>
</tr>
<tr>
<td>STX</td>
<td>Store index X in memory</td>
<td>X X</td>
</tr>
<tr>
<td>STY</td>
<td>Store index Y in memory</td>
<td>X X</td>
</tr>
<tr>
<td>TAX</td>
<td>Transfer accumulator to index X</td>
<td>X X</td>
</tr>
<tr>
<td>TAY</td>
<td>Transfer accumulator to index Y</td>
<td>X X</td>
</tr>
<tr>
<td>TSX</td>
<td>Transfer stack pointer to index X</td>
<td>X X</td>
</tr>
<tr>
<td>TXA</td>
<td>Transfer index X to accumulator</td>
<td>X</td>
</tr>
<tr>
<td>TXS</td>
<td>Transfer index X to stack pointer</td>
<td>X</td>
</tr>
<tr>
<td>TYA</td>
<td>Transfer index Y to accumulator</td>
<td>X</td>
</tr>
</tbody>
</table>

Flag abbreviations are:
- N Negative result flag
- V Overflow flag
- Expansion flag (not labeled)
- B Break command flag
- D Decimal mode flag
- I Interrupt disable flag
- Z Zero result flag
- C Carry flag
# 6502 Instructions—Addressing Modes

<table>
<thead>
<tr>
<th>Mnemonic Code</th>
<th>Op Codes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Accumulator</td>
</tr>
<tr>
<td>ADC</td>
<td>-</td>
</tr>
<tr>
<td>AND</td>
<td>-</td>
</tr>
<tr>
<td>ASL</td>
<td>0A</td>
</tr>
<tr>
<td>BCC</td>
<td>-</td>
</tr>
<tr>
<td>BCS</td>
<td>-</td>
</tr>
<tr>
<td>BEQ</td>
<td>-</td>
</tr>
<tr>
<td>BIT</td>
<td>-</td>
</tr>
<tr>
<td>BMI</td>
<td>-</td>
</tr>
<tr>
<td>BNE</td>
<td>-</td>
</tr>
<tr>
<td>BPL</td>
<td>-</td>
</tr>
<tr>
<td>BRK</td>
<td>-</td>
</tr>
<tr>
<td>BVC</td>
<td>-</td>
</tr>
<tr>
<td>BVS</td>
<td>-</td>
</tr>
<tr>
<td>CLC</td>
<td>-</td>
</tr>
<tr>
<td>CLD</td>
<td>-</td>
</tr>
<tr>
<td>CLI</td>
<td>-</td>
</tr>
<tr>
<td>CLV</td>
<td>-</td>
</tr>
<tr>
<td>CMP</td>
<td>-</td>
</tr>
<tr>
<td>CPX</td>
<td>-</td>
</tr>
<tr>
<td>CPY</td>
<td>-</td>
</tr>
<tr>
<td>DEC</td>
<td>-</td>
</tr>
<tr>
<td>DEX</td>
<td>-</td>
</tr>
<tr>
<td>DEY</td>
<td>-</td>
</tr>
<tr>
<td>EOR</td>
<td>-</td>
</tr>
<tr>
<td>Mnemonic Code</td>
<td>Op Codes</td>
</tr>
<tr>
<td>---------------</td>
<td>----------</td>
</tr>
<tr>
<td></td>
<td>Accumulator</td>
</tr>
<tr>
<td>INC</td>
<td>E6</td>
</tr>
<tr>
<td>INX</td>
<td></td>
</tr>
<tr>
<td>INY</td>
<td></td>
</tr>
<tr>
<td>JMP</td>
<td></td>
</tr>
<tr>
<td>JSR</td>
<td></td>
</tr>
<tr>
<td>LDA</td>
<td>A9</td>
</tr>
<tr>
<td>LDX</td>
<td>A2</td>
</tr>
<tr>
<td>LDY</td>
<td>A0</td>
</tr>
<tr>
<td>LSR</td>
<td>4A</td>
</tr>
<tr>
<td>NOP</td>
<td></td>
</tr>
<tr>
<td>ORA</td>
<td>09</td>
</tr>
<tr>
<td>PHA</td>
<td></td>
</tr>
<tr>
<td>PHP</td>
<td></td>
</tr>
<tr>
<td>PLA</td>
<td></td>
</tr>
<tr>
<td>PLP</td>
<td></td>
</tr>
<tr>
<td>ROL</td>
<td>2A</td>
</tr>
<tr>
<td>ROR</td>
<td>6A</td>
</tr>
<tr>
<td>RTI</td>
<td></td>
</tr>
<tr>
<td>RTS</td>
<td></td>
</tr>
<tr>
<td>SBC</td>
<td>E9</td>
</tr>
<tr>
<td>SEC</td>
<td></td>
</tr>
<tr>
<td>SED</td>
<td></td>
</tr>
<tr>
<td>SEI</td>
<td></td>
</tr>
<tr>
<td>STA</td>
<td>85</td>
</tr>
<tr>
<td>STX</td>
<td>86</td>
</tr>
<tr>
<td>STY</td>
<td>84</td>
</tr>
<tr>
<td>TAX</td>
<td></td>
</tr>
<tr>
<td>TAY</td>
<td></td>
</tr>
<tr>
<td>TSX</td>
<td></td>
</tr>
<tr>
<td>TXA</td>
<td></td>
</tr>
<tr>
<td>TXS</td>
<td></td>
</tr>
<tr>
<td>TYA</td>
<td></td>
</tr>
</tbody>
</table>
### Frequency Values for Three-Octave Scale

<table>
<thead>
<tr>
<th>Natural Scale Note</th>
<th>Frequency Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>1D</td>
</tr>
<tr>
<td>B</td>
<td>1F</td>
</tr>
<tr>
<td>A#</td>
<td>21</td>
</tr>
<tr>
<td>A</td>
<td>23</td>
</tr>
<tr>
<td>G#</td>
<td>25</td>
</tr>
<tr>
<td>G</td>
<td>28</td>
</tr>
<tr>
<td>F#</td>
<td>2A</td>
</tr>
<tr>
<td>F</td>
<td>2D</td>
</tr>
<tr>
<td>E</td>
<td>2F</td>
</tr>
<tr>
<td>D#</td>
<td>32</td>
</tr>
<tr>
<td>D</td>
<td>35</td>
</tr>
<tr>
<td>C#</td>
<td>39</td>
</tr>
<tr>
<td>C</td>
<td>3C</td>
</tr>
<tr>
<td>B</td>
<td>40</td>
</tr>
<tr>
<td>A#</td>
<td>44</td>
</tr>
<tr>
<td>A</td>
<td>48</td>
</tr>
<tr>
<td>G#</td>
<td>4C</td>
</tr>
<tr>
<td>G</td>
<td>51</td>
</tr>
<tr>
<td>F#</td>
<td>55</td>
</tr>
<tr>
<td>F</td>
<td>5B</td>
</tr>
<tr>
<td>E</td>
<td>60</td>
</tr>
<tr>
<td>D#</td>
<td>66</td>
</tr>
<tr>
<td>D</td>
<td>6C</td>
</tr>
<tr>
<td>C#</td>
<td>72</td>
</tr>
<tr>
<td>C</td>
<td>79</td>
</tr>
<tr>
<td>B</td>
<td>80</td>
</tr>
<tr>
<td>A#</td>
<td>88</td>
</tr>
<tr>
<td>A</td>
<td>90</td>
</tr>
<tr>
<td>G#</td>
<td>99</td>
</tr>
</tbody>
</table>

High

Middle
<table>
<thead>
<tr>
<th>Natural Scale Note</th>
<th>Frequency Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>G</td>
<td>A2</td>
</tr>
<tr>
<td>F#</td>
<td>AC</td>
</tr>
<tr>
<td>F</td>
<td>B6</td>
</tr>
<tr>
<td>E</td>
<td>C1</td>
</tr>
<tr>
<td>D#</td>
<td>CC</td>
</tr>
<tr>
<td>D</td>
<td>D9</td>
</tr>
<tr>
<td>C#</td>
<td>E6</td>
</tr>
<tr>
<td>C</td>
<td>F3, Low</td>
</tr>
</tbody>
</table>
When an error occurs, a short “beep” is heard and the error number is displayed.

<table>
<thead>
<tr>
<th>Error Number</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Insufficient memory for the program to be assembled.</td>
</tr>
<tr>
<td>2</td>
<td>The number xx cannot be found for the “DEL xx,yy” command.</td>
</tr>
<tr>
<td>3</td>
<td>An error in specifying an address (mini-assembler).</td>
</tr>
<tr>
<td>4</td>
<td>File named cannot be loaded.</td>
</tr>
<tr>
<td>5</td>
<td>Undefined reference label.</td>
</tr>
<tr>
<td>6</td>
<td>Syntax error in a statement.</td>
</tr>
<tr>
<td>7</td>
<td>Label defined more than once.</td>
</tr>
<tr>
<td>8</td>
<td>Buffer overflow.</td>
</tr>
<tr>
<td>9</td>
<td>No label given before “=” .</td>
</tr>
<tr>
<td>10</td>
<td>The value of an expression is greater than 255 where one byte was required.</td>
</tr>
<tr>
<td>11</td>
<td>Null string used where invalid.</td>
</tr>
<tr>
<td>12</td>
<td>Address or address type specified is incorrect.</td>
</tr>
<tr>
<td>13</td>
<td>Phase error-an inconsistent result found from Pass 1 to Pass 2.</td>
</tr>
<tr>
<td>14</td>
<td>Undefined forward reference.</td>
</tr>
<tr>
<td>15</td>
<td>Line is too large.</td>
</tr>
<tr>
<td>16</td>
<td>Source statement not recognized by the assembler.</td>
</tr>
<tr>
<td>17</td>
<td>Line number too large.</td>
</tr>
<tr>
<td>18</td>
<td>LOMEM command was attempted after other command(s) or instruction(s). LOMEM must be first command if used.</td>
</tr>
<tr>
<td>19</td>
<td>No starting address given.</td>
</tr>
</tbody>
</table>
Appendix E

Atari Operating System Errors

Errors numbered above 100 refer to Operating System and Disk Operating System. Refer to DOS manual for complete list of DOS errors.

<table>
<thead>
<tr>
<th>Error Number</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>The BREAK key was hit during an I/O operation.</td>
</tr>
<tr>
<td>130</td>
<td>A nonexistent device was specified.</td>
</tr>
<tr>
<td>132</td>
<td>Command is invalid for the device selected.</td>
</tr>
<tr>
<td>136</td>
<td>END OF FILE READ HAS BEEN REACHED. This error may occur when reading from cassette.</td>
</tr>
<tr>
<td>137</td>
<td>Record was longer than 256 characters.</td>
</tr>
<tr>
<td>138</td>
<td>Device specified in the command does not respond. Make sure it is connected and powered.</td>
</tr>
<tr>
<td>139</td>
<td>Device specified in command does not return an acknowledge signal.</td>
</tr>
<tr>
<td>140</td>
<td>Serial framing error.</td>
</tr>
<tr>
<td>142</td>
<td>Serial framing error.</td>
</tr>
<tr>
<td>143</td>
<td>Serial data checksum error.</td>
</tr>
<tr>
<td>144</td>
<td>Device done error.</td>
</tr>
<tr>
<td>145</td>
<td>Diskette error- read-after-write comparison failed.</td>
</tr>
<tr>
<td>146</td>
<td>Disk full.</td>
</tr>
<tr>
<td>165</td>
<td>File name error.</td>
</tr>
</tbody>
</table>
ATASCII is an abbreviation of ATARI ASCII. Letters and numbers have the same values as those in ASCII, but some of the special characters have been assigned differently. Characters 80-FF (hex) are just reverse colors of 1-7F except for the characters shown (9B-9F and FD, FE, and FF).

<table>
<thead>
<tr>
<th>HEX</th>
<th>CHARACTER</th>
<th>HEX</th>
<th>CHARACTER</th>
<th>HEX</th>
<th>CHARACTER</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>A</td>
<td></td>
<td>14</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>B</td>
<td></td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>C</td>
<td></td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>D</td>
<td></td>
<td>17</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>E</td>
<td></td>
<td>18</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>F</td>
<td></td>
<td>19</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>10</td>
<td></td>
<td>1A</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>11</td>
<td></td>
<td>1B</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>12</td>
<td></td>
<td>1C</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>13</td>
<td></td>
<td>1D</td>
<td></td>
</tr>
<tr>
<td>HEX</td>
<td>CHARACTER</td>
<td>HEX</td>
<td>CHARACTER</td>
<td>HEX</td>
<td>CHARACTER</td>
</tr>
<tr>
<td>-----</td>
<td>-----------</td>
<td>-----</td>
<td>-----------</td>
<td>-----</td>
<td>-----------</td>
</tr>
<tr>
<td>1E</td>
<td>←</td>
<td>32</td>
<td>2</td>
<td>46</td>
<td>F</td>
</tr>
<tr>
<td>1F</td>
<td>→</td>
<td>33</td>
<td>3</td>
<td>47</td>
<td>G</td>
</tr>
<tr>
<td>20</td>
<td>Space</td>
<td>34</td>
<td>4</td>
<td>48</td>
<td>H</td>
</tr>
<tr>
<td>21</td>
<td>!</td>
<td>35</td>
<td>5</td>
<td>49</td>
<td>I</td>
</tr>
<tr>
<td>22</td>
<td>&quot;</td>
<td>36</td>
<td>6</td>
<td>4A</td>
<td>J</td>
</tr>
<tr>
<td>23</td>
<td>#</td>
<td>37</td>
<td>7</td>
<td>4B</td>
<td>K</td>
</tr>
<tr>
<td>24</td>
<td>$</td>
<td>38</td>
<td>8</td>
<td>4C</td>
<td>L</td>
</tr>
<tr>
<td>25</td>
<td>%</td>
<td>39</td>
<td>9</td>
<td>4D</td>
<td>M</td>
</tr>
<tr>
<td>26</td>
<td>&amp;</td>
<td>3A</td>
<td>:</td>
<td>4E</td>
<td>N</td>
</tr>
<tr>
<td>27</td>
<td>'</td>
<td>3B</td>
<td>;</td>
<td>4F</td>
<td>O</td>
</tr>
<tr>
<td>28</td>
<td>(</td>
<td>3C</td>
<td>&lt;</td>
<td>50</td>
<td>P</td>
</tr>
<tr>
<td>29</td>
<td>)</td>
<td>3D</td>
<td>=</td>
<td>51</td>
<td>Q</td>
</tr>
<tr>
<td>2A</td>
<td>*</td>
<td>3E</td>
<td>&gt;</td>
<td>52</td>
<td>R</td>
</tr>
<tr>
<td>2B</td>
<td>+</td>
<td>3F</td>
<td>?</td>
<td>53</td>
<td>S</td>
</tr>
<tr>
<td>2C</td>
<td>,</td>
<td>40</td>
<td>@</td>
<td>54</td>
<td>T</td>
</tr>
<tr>
<td>2D</td>
<td>-</td>
<td>41</td>
<td>A</td>
<td>55</td>
<td>U</td>
</tr>
<tr>
<td>2E</td>
<td>.</td>
<td>42</td>
<td>B</td>
<td>56</td>
<td>V</td>
</tr>
<tr>
<td>2F</td>
<td>/</td>
<td>43</td>
<td>C</td>
<td>57</td>
<td>W</td>
</tr>
<tr>
<td>30</td>
<td>0</td>
<td>44</td>
<td>D</td>
<td>58</td>
<td>X</td>
</tr>
<tr>
<td>31</td>
<td>1</td>
<td>45</td>
<td>E</td>
<td>59</td>
<td>Y</td>
</tr>
<tr>
<td>HEX</td>
<td>CHARACTER</td>
<td>HEX</td>
<td>CHARACTER</td>
<td>HEX</td>
<td>CHARACTER</td>
</tr>
<tr>
<td>-----</td>
<td>-----------</td>
<td>-----</td>
<td>-----------</td>
<td>-----</td>
<td>-----------</td>
</tr>
<tr>
<td>5A</td>
<td>Z</td>
<td>6A</td>
<td>j</td>
<td>7A</td>
<td>z</td>
</tr>
<tr>
<td>5B</td>
<td>[</td>
<td>6B</td>
<td>k</td>
<td>7B</td>
<td></td>
</tr>
<tr>
<td>5C</td>
<td>\</td>
<td>6C</td>
<td>l</td>
<td>7C</td>
<td>↑</td>
</tr>
<tr>
<td>5D</td>
<td>}</td>
<td>6D</td>
<td>m</td>
<td>7D</td>
<td>←</td>
</tr>
<tr>
<td>5E</td>
<td>^</td>
<td>6E</td>
<td>n</td>
<td>7E</td>
<td>→</td>
</tr>
<tr>
<td>5F</td>
<td>－</td>
<td>6F</td>
<td>o</td>
<td>7F</td>
<td>↓</td>
</tr>
<tr>
<td>60</td>
<td></td>
<td>70</td>
<td>p</td>
<td>9B</td>
<td>Carriage Return</td>
</tr>
<tr>
<td>61</td>
<td>a</td>
<td>71</td>
<td>q</td>
<td>9C</td>
<td></td>
</tr>
<tr>
<td>62</td>
<td>b</td>
<td>72</td>
<td>r</td>
<td>9D</td>
<td></td>
</tr>
<tr>
<td>63</td>
<td>c</td>
<td>73</td>
<td>s</td>
<td>9E</td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>d</td>
<td>74</td>
<td>t</td>
<td>9F</td>
<td></td>
</tr>
<tr>
<td>65</td>
<td>e</td>
<td>75</td>
<td>u</td>
<td>FD</td>
<td>Buzzer!</td>
</tr>
<tr>
<td>66</td>
<td>f</td>
<td>76</td>
<td>v</td>
<td>FE</td>
<td>Delete character</td>
</tr>
<tr>
<td>67</td>
<td>g</td>
<td>77</td>
<td>w</td>
<td>FF</td>
<td>Insert character</td>
</tr>
<tr>
<td>68</td>
<td>h</td>
<td>78</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>69</td>
<td>i</td>
<td>79</td>
<td>y</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Index

#### A

- Absolute addressing mode, 90
- Absolute indexed addressing mode, 146
- Accumulator, 5, 74
- Addition, 164
- Addressing modes, 88
- Architecture, 4
- Arithmetic Shift Left, 191
- Assembler cartridge, 1, 53
- Assembler program (of the cartridge), 53, 57, 120
- Assembly language, advantages, 3
- ATASCII character codes, 252, 265

#### B

- BASIC cartridge, 1
- BASIC, review, 7
- Binary-coded-decimal (BCD) numbers, 179
- Binary-hexadecimal relationship, 14
- Binary numbers, 12
- Bits, 13
- Branch backward, 97, 101
- Branch forward, 96, 100
- Branch instructions, 102
- Bytes, 13

#### C

- Carry flag, 102
- Cartridge (assembler), 1
- Cartridge (BASIC), 1
- Central Processing Unit (CPU), 4

#### D

- Data bus, 5
- DEBUGGER program (of the cartridge), 53, 62, 131
- Decimal arithmetic, 179
- Disassemble, 135
- Division, 221

#### E

- EDIT mode (same as WRITER/EDITOR), 124
- Edit text buffer, 125
- Error codes, 121, 263, 264
- Exclusive OR, 237

#### F

- Fields used in assembler statements, 120
Flags (status), 7, 82
Format (assembly language), 55, 120

H
Hexadecimal-decimal relationship, 16
Hexadecimal notation, 14

I
Immediate addressing mode, 89
Implied addressing mode, 88
Instruction set (6502), 91, 257, 259

L
Label (assembler statement), 120
Least Significant Byte, 37
Logical Shift Right, 198

M
Machine Language, disadvantages, 2
Memory map, 28
Mnemonic codes, 60, 121
Most Significant Byte, 37
Multiplication, 214

N
Negative flag, 111
Negative numbers, 174

O
Object program, 56, 59
Operand, 121
Overflow flag, 116

P
Passing data: BASIC to machine language, 33
POKE machine language from BASIC, 19
Processor Status Register, 7, 82
Program counter, 6, 96, 227
Program design, 144

R
Registers, 5, 73
Relative addressing mode, 90
Rotate Left, 202
Rotate Right, 205

S
Signed numbers, 113
6502 instruction set, 91, 257, 259
Sound programs, 245, 248, 249
Source program, 55, 59, 120
Stack, 6, 22, 87, 226
Stack Pointer, 6, 87
Status flags affected by instructions, 7, 85, 257
Subroutines, 225
Subroutine to display on the screen, 252
Subtraction, 172

T
Trace (of a program), 45, 63

U
USR function, 31
<table>
<thead>
<tr>
<th>W</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>WRITER/EDITOR program (of cartridge), 52, 53, 124</td>
<td>Y Register, 74</td>
</tr>
<tr>
<td>X</td>
<td>Z</td>
</tr>
<tr>
<td>X Register, 74</td>
<td>Zero flag, 108</td>
</tr>
<tr>
<td></td>
<td>Zero page addressing mode, 90</td>
</tr>
</tbody>
</table>
Now you can master assembly language programming and learn how to use the Atari Assembler Cartridge with your Atari 400 or 800 model computer at the same time! This excellent guidebook is designed for readers with some BASIC language programming knowledge, but assumes no assembly language background.

Your BASIC background leads you gradually into assembly language programming. THE ATARI ASSEMBLER takes an easy pace through each program. Sketches of the video screen show you each stage of entering and executing the sample programs and step-by-step instructions help you make the transition from BASIC programming to the Atari’s own language.

The authors know just how important it is to learn by doing. As you explore the Atari Assembler Cartridge and discover how to program in the assembly language, you are urged to try out instructions and programs on the computer, checking your results on the video screen. By the time you complete this book, you will be designing your own creative assembly language programs.