



Department of Computer Science

# An Implementation of Donald Knuth's MIX

Project 1998-1999

**Author**

**Andrew Doran**

`csubm@dcs.warwick.ac.uk`

**Supervisor**

**Dr Mike Joy**

`M.S.Joy@dcs.warwick.ac.uk`

**Abstract:** This project is intended to produce a working software implementation of Donald Knuth's MIX 1009 computer. Realisation of this hypothetical machine would enable users to view running algorithms in real-time, and alter their structure to observe gains in efficiency. Usefulness of the software is maximized by being available as a Java applet over the World Wide Web. It is intended as a learning tool for people interested in Computer Science and general programming, and to be used in conjunction with Knuth's book *The Art of Computer Programming*, from which the machine originates.

**Keyword List:** Algorithm, Applet, Assembly, Java, Knuth, Software, Education.

Software available on-line at:

<http://andrew.doran.com/project/>



# Contents

<b>1 Background</b>	<b>9</b>
<b>1.1 Donald Knuth</b>	<b>9</b>
<b>1.2 The Art of Computer Programming</b>	<b>9</b>
<b>1.3 The MIX 1009 - Machine Overview</b>	<b>10</b>
1.3.1 Description of MIX	10
1.3.1.1 Words	10
1.3.1.2 Fields	11
1.3.1.3 'Packing'	11
1.3.1.4 Memory	11
1.3.1.5 Registers	12
1.3.1.6 Instruction Set	12
1.3.1.7 Character Set	13
1.3.1.8 Peripherals	14
1.3.2 Description of MIXAL	14
1.3.2.1 Overview	14
1.3.2.2 Use of Symbols	15
1.3.2.3 The Location Counter	15
1.3.2.4 Constants	16
1.3.2.5 Character-Code Assembly	16
1.3.2.6 Comments	16
1.3.2.7 Program Input and Storage	17
1.3.2.8 Formal Language Definition	17
<b>1.4 Future Developments of MIX</b>	<b>18</b>
1.4.1 MMIX	18
 <b>2 A Java Implementation</b>	 <b>19</b>
<b>2.1 Java Applets</b>	<b>19</b>
2.1.1 Advantages	19
2.1.2 Disadvantages	19
2.1.3 The Choice of Java for the Implementation	20
2.1.4 The Choice of Java Version 1.1	20

<b>3 Design and Functionality</b>	<b>23</b>
<b>3.1 Introductory Applet Display</b>	<b>23</b>
<b>3.2 Main MIX Machine Window</b>	<b>25</b>
3.2.1 Components	25
3.2.2 Scrollable Memory	26
3.2.3 Menu Bar Options	26
<b>3.3 Program Window</b>	<b>29</b>
3.3.1 Greek Characters	29
3.3.2 The Clipboard	30
<b>3.4 Line Printer Window</b>	<b>31</b>
<b>3.5 Clock Window</b>	<b>32</b>
<b>3.6 Control Console</b>	<b>33</b>
<b>3.7 Program Loader (Temporary Window)</b>	<b>34</b>
<b>4 Implementation and Internal Structure</b>	<b>37</b>
<b>4.1 MIX Words</b>	<b>37</b>
4.1.1 Memory Cells	38
4.1.2 A-Register and X-Register	38
4.1.3 I-Registers	38
4.1.4 J-Register	39
<b>4.2 Overflow Indicator</b>	<b>39</b>
<b>4.3 Comparison Indicator</b>	<b>39</b>
<b>4.4 MIXAL Programs and Assembly</b>	<b>39</b>
4.4.1 Bootstrapping	40
4.4.2 Assembly	40
4.4.2.1 Symbols and the First Pass	41
4.4.2.2 The Second Pass	42
<b>4.5 Line Printer</b>	<b>44</b>
<b>4.6 Clock</b>	<b>44</b>
<b>4.7 Unimplemented Peripherals</b>	<b>45</b>
<b>4.8 Program Execution and Control</b>	<b>45</b>
4.8.1 Instructions	46
4.8.2 Timing	57
4.8.3 Plus and Minus Zero	58

<b>5 Testing</b>	<b>59</b>
<b>5.1 Java Classes</b>	<b>59</b>
<b>5.2 MIX Instructions</b>	<b>60</b>
<b>5.3 Program Execution</b>	<b>60</b>
<b>5.4 Complete Applet</b>	<b>61</b>
<b>6 Example Programs</b>	<b>63</b>
<b>6.1 Maximum - 'Program M'</b>	<b>63</b>
<b>6.2 Primes - 'Program P'</b>	<b>65</b>
<b>7 Other MIX Emulators</b>	<b>67</b>
<b>7.1 MIX/360</b>	<b>67</b>
<b>7.2 UT-MIX</b>	<b>67</b>
<b>7.3 Mixal</b>	<b>68</b>
<b>7.4 MIX Builder 98</b>	<b>69</b>
<b>8 Further Work</b>	<b>71</b>
<b>8.1 Unimplemented Features</b>	<b>71</b>
8.1.1 The Assembler	71
8.1.2 Peripherals	71
8.1.3 Floating-Point Operations	72
<b>8.2 The Graphical User Interface</b>	<b>72</b>
<b>8.3 MIXAL Extensions</b>	<b>73</b>
<b>8.3 Program Architecture</b>	<b>73</b>
<b>9 Conclusions</b>	<b>75</b>
<b>10 References</b>	<b>77</b>

## 11 Acknowledgments 79

## Appendix A - Authored Program Code 81

A.a	CharNotASign.java	81
A.b	ComparisonIndicator.java	82
A.c	IRegister.java	84
A.d	IndexOutOfRangeException.java	85
A.e	InfoWindow.java	86
A.f	InputConsole.java	87
A.g	JRegister.java	91
A.h	JRegisterMustBePositiveException.java	92
A.i	LinePrinter.java	93
A.j	MIX1009.java	96
A.k	MIXByte.java	98
A.l	MIXClock.java	99
A.m	MIXMachine.java	101
A.n	MIXSign.java	154
A.o	MIXWord.java	155
A.p	NotAMIXCharacterException.java	159
A.q	NotAValidStateException.java	160
A.r	OverFlowIndicator.java	161
A.s	ValueOutOfBoundsException.java	162

## Appendix B - Incorporated Program Code 163

B.a	ImageLabel.java	163
-----	-----------------	-----

## Appendix C - HTML Code 173

C.a	MIXApplet.html	173
-----	----------------	-----

## Appendix D - MIXAL Programs 175

D.a	Program P	175
D.b	Program M	177





# 1 Background

## 1.1 Donald Knuth

Donald E Knuth is widely recognised as one of the greatest Computer Scientists. His work on algorithms and programming techniques, 19 published books, over 160 papers, and invention of the T<sub>E</sub>X and METAFONT systems has earned him worldwide recognition (Knuth, 1997, inside back cover). He is the recipient of notable accolades such as the prestigious Association of Computing Machinery Turing Award (1974), the National Medal of Science (1978) and the Inamori Foundation Kyoto Prize (1996). Currently, Knuth is Professor Emeritus in the Art of Computer Programming at Stanford University (Knuth, 1998).

## 1.2 The Art of Computer Programming

The most prominent of Knuth's publications is the multi-volume work *The Art of Computer Programming* (TAOCP). Begun in 1962, TAOCP was originally intended to be a single book, with chapters devoted to important topics such as information structures, random numbers, sorting, searching and recursion. However, during the course of his writing, Knuth decided that it was “more important to treat the subjects in depth rather than to skim over them lightly” (Knuth, 1997, page vii). This decision has meant that the original single book has expanded to a planned five volumes, of which the first three are currently in print. Volumes four and five are estimated to be complete by the years 2004 and 2009 respectively (Knuth, 1999a). The first three volumes of TAOCP have, to date, sold over one million copies (Knuth, 1996), and their importance as a fundamental text in the field of Computer Science is unchallenged.

Algorithms and programs presented by Knuth in TAOCP are written in the language MIXAL and intended for execution on MIX, a hypothetical machine. MIX was formulated specifically for use with this work for a number of good reasons. The use of an existing language and machine would have given readers the impression that the books were just for users of those particular systems. Also, the rapid advancements in technology would mean that the system would very likely become obsolete over a relatively short period of time. Furthermore, idiosyncrasies of a chosen machine and language would have to be explained to the reader, confusing the issues at hand. All of these problems are avoided by the formulation of a near-‘ideal’ machine (Knuth, 1997, page ix).

The advantages gained from using a hypothetical machine are clear. However, since the machine does not exist, it is not possible for a user to ‘run’ the algorithm and view its performance in real-time. Knuth states himself that “[u]nder favorable [sic] circumstances the reader will have access to a MIX assembler and a MIX simulator, on which various exercises in this book can be worked out” (Knuth, 1997, page 153). MIX simulators are already available in various forms, the most recent being *MIX Builder 98* by Bill Menees, and vary in their degree of usability and performance (see section 7). All of them are

currently platform-dependent. It is the intent of this project to provide a working MIX machine upon which programs from the book can be viewed, run and analyzed by users, in a (relatively) platform-independent environment. This platform independence will give maximum access to potential readers of TAOCP (see section 2.1.1).

## **1.3 The MIX 1009 - Machine Overview**

### **1.3.1 Description of MIX**

MIX has been designed to resemble many computers widely available and used during the 1960s and 1970s. It possesses the basic features common to these systems. The number '1009' was given to MIX as an indicator of its origins - the figure is formed by taking the average of the numbers given to 16 popular computers of its era. Knuth believes that readers should not be afraid of learning the characteristics and programming language of a new machine; if they are interested in Computer Science, they will doubtless become familiar with a particular computer at some point (Knuth, 1997, page 124).

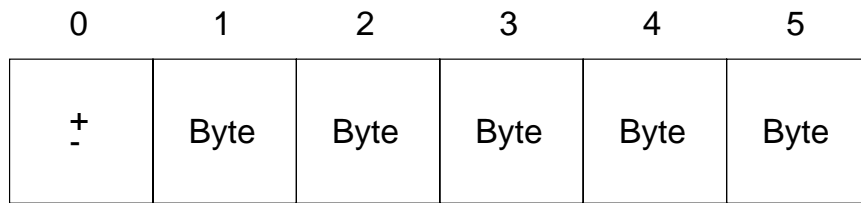
The following description of MIX originates from TAOCP Volume 1 (Knuth, 1997, pages 125-128). Information is summarised into particular subjects, and references point to particular locations in the book.

#### ***1.3.1.1 Words***

The basic unit of information used in MIX is called a *byte*. As the origins of MIX are pre-1975, this term is not used in the same sense that it is today. Each byte in the machine contains an unspecified amount of information, but must be capable of holding at least 64 distinct values. In other words, the numbers between 0 and 63 inclusive can be represented by a single byte. Also, a byte can contain at most 100 distinct values.

Knuth states that programs that depend on a particular byte-size value are not valid. For example, if an algorithm is dealing with the number 80, two adjacent bytes should be used even if one will suffice. He argues that byte-size dependent programs are against the spirit of the book and should be avoided. This restriction should not (and in practice, does not) cause major problems.

A MIX *word* consists of five adjacent bytes plus a sign, as shown in Figure 1.1, below. The five bytes allow the representation of  $64^5$  different states - the numbers between 0 and 1 073 741 823 inclusive. As shown here, the sign of a word can take the value of either '+' or '-', effectively doubling the number of possible states.



**Figure 1.1 : A MIX Word**

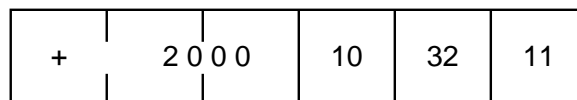
#### 1.3.1.2 Fields

Each section, or field, of a word is numbered. The sign is located in field 0, and the least significant byte in field 5. Programmers can specify contiguous subsections of a word using a 'field specification'. This takes the form of (L:R), where L is the number of the leftmost field and R the number of the rightmost field in the subsection. For example, (1:5) would indicate the five bytes without a sign, and (0:1) represents the most significant byte along with its sign. This field specification can be used internally by MIX in the form of the number  $8L+R$ , which fits easily into one byte.

#### 1.3.1.3 'Packing'

Representation of the contents of a MIX word is in the form of 'box' notation, illustrated in Figure 1.1 (above) and Figure 1.2 (below). As you can see from the latter, the  $\pm AA$  bytes are 'joined' together as no solid vertical dividers separate them. The sign and bytes 1 and 2 of the word are said to be 'packed'.

Packing is used graphically to give the user of the machine the impression that words are holding large values. The value 2000, for example, cannot be represented in a single byte. Two bytes (at least) are needed to accommodate this number, as shown in this example word:



This illustration is typical of what the user sees when using MIX. Contents of the first and second bytes are not individually known since the byte size varies from one machine to the next. Packing bytes in this way allows them to represent a greater range of values than they could do on their own.

There are no commands in the MIX language that specifically 'pack' two bytes together; assembled instructions take the format of that shown in Figure 1.2 and packing occurs throughout the life of a program as a side-effect of certain instructions. Knuth does not make this clear, but it is inferred from his description of the various commands.

#### 1.3.1.4 Memory

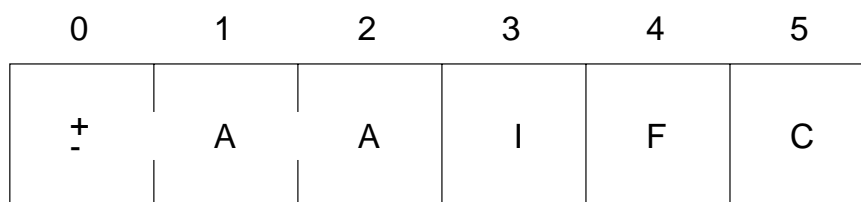
The machine is assumed to contain 4 000 memory cells, each with a location number between 0 and 3 999. These are used in conjunction with MIX's registers to store programs and their outputs.

### 1.3.1.5 Registers

MIX has nine registers. The A-register (Accumulator) and X-register (Extension) both consist of five bytes and a sign, a normal MIX word. These are typically used for arithmetic instructions. There are six I-registers (Index), numbered I1...I6; these consist of just two bytes together with a sign. The main use of these is the referencing of memory locations, but they are also commonly employed for counting. MIX's J-register (jump) also consists of just two bytes. It does have a sign field, but this is always assumed to be positive. This register is used when branching and jumping to subroutines within a program.

### 1.3.1.6 Instruction Set

When holding instructions, memory cells are interpreted in the form shown in Figure 1.2, below.



**Figure 1.2 : A MIX Instruction**

The rightmost byte, C, contains the *operation code*. This code is used to indicate which instruction is to be performed. MIX has a total of 149 defined instructions, a number that does not fit into a single byte. Thus, it is used in conjunction with the F-field - a *modification* of the operation code. Usually, the F-byte contains the field specification of the operation, in the form of 8L+R (see section 1.3.1.2), but it may also be an indicator that a particular variation of operation C should be performed. By using the F-byte in this manner for values of C that do not need field specifications, Knuth has managed to expand the command set of MIX significantly.

Fields 0, 1 and 2 of a MIX instruction are combined to give the *address*. This is used in conjunction with field 3, the *index specification*. The contents of I determine how  $\pm AA$  are to be used. If I is zero,  $\pm AA$  is used in the instruction without change. Otherwise, I should contain a number  $i$  between 1 and 6 inclusive; in this case, the contents of register  $i$  are added algebraically to  $\pm AA$  before execution of the instruction. This indexing process takes place on every instruction, and the value produced is referred to by Knuth simply as M.

Although M usually refers to some location in memory, and thus would have a value between 0 and 3 999, it is not always the case. The value of M has another significance for certain instructions, and they take account of the value, including the sign, for their operations.

Each instruction takes a certain amount of time to execute. This differs between instructions, but is guaranteed to be the same for every instance of a particular instruction. As MIX is hypothetical, Knuth has chosen to measure instruction execution time in  $\mu$ . The actual time that each instruction takes to execute may differ between implementations of MIX, but the same algorithm will take the same number

of  $u$  on each machine. This measurement of algorithm running time is intended for relative interpretation only, and has little meaning as a value by itself (see section 4.8.2).

Individual instructions are discussed in section 4.8.1.

#### *1.3.1.7 Character Set*

There are a total of 55 symbols in the MIX character set, each corresponding with a particular number as shown in Table 1, below. Note that character number 00 is a blank space.

no.	char	no.	char	no.	char	no.	char	no.	char	no.	char
00		10	$\Delta$	20	$\Sigma$	30	0	40	.	50	<
01	A	11	J	21	$\Pi$	31	1	41	,	51	>
02	B	12	K	22	S	32	2	42	(	52	@
03	C	13	L	23	T	33	3	43	)	53	;
04	D	14	M	24	U	34	4	44	+	54	:
05	E	15	N	25	V	35	5	45	-	55	`
06	F	16	O	26	W	36	6	46	*		
07	G	17	P	27	X	37	7	47	/		
08	H	18	Q	28	Y	38	8	48	=		
09	I	19	R	29	Z	39	9	49	\$		

**Table 1: The MIX Character Set**

### 1.3.1.8 Peripherals

MIX can be attached to various input-output peripheral equipment. However, every machine may not necessarily have every peripheral available to it. Each device is located on a specific 'port' on the machine, as outlined in Table 2, below.

Unit number	Peripheral device	Block size
$t$	Tape unit number $t$ ( $0 \leq t \leq 7$ )	100 words
$d$	Disk or drum unit number $d$ ( $8 \leq d \leq 15$ )	100 words
16	Card reader	16 words
17	Card punch	16 words
18	Line printer	24 words
19	Typewriter terminal	14 words
20	Paper tape	14 words

**Table 2: MIX Peripherals**

Each device has a specific 'block size' associated with it, as the table above shows. This is the standard number of MIX words that are transferred in any one input/output operation. Input and output with magnetic tape, disk or drum units is in the form of whole MIX words - five bytes and a sign. Units 16-20, however, use *character codes*, where each byte represents one alphanumeric character (see section 1.3.1.7). Character codes cannot be used to read in or write out all possible values that a byte may have as the numbers 56 onwards have no corresponding character. In addition, some peripherals may not be able to handle all the symbols in the character set - the greek symbols, for example, are not supported by the card reader. Character-code output does not involve the sign of each word; on input, the sign is assumed to be '+'. If the typewriter is being used for input, the 'carriage return' at the end of each line causes the remainder of the line to be filled with blanks (character code 00).

## 1.3.2 Description of MIXAL

### 1.3.2.1 Overview

Programming MIX has been made considerably easier by the invention of MIXAL, the MIX Assembly Language (Knuth, 1997, pages 144-153). This language is designed to enrich the basic MIX instruction set (see section 1.3.1.6), and reduces the possibility of programming errors. Its main features are the optional use of alphabetic names to stand for numbers, and a location field to associate names with memory locations.

Programs written in MIXAL are converted into machine instructions by an appropriate assembler (see section 4.4). Instructions are usually assembled into sequential memory locations, unless otherwise specified by the program.

Each line in a MIXAL program consists of three fields: LOC, OP and ADDRESS. The LOC field is used to define symbol values, for use throughout the program. Instructions, either MIX instructions or commands specifically for the assembler, are placed in the OP field. Finally, the ADDRESS field contains the instruction's operand. This last field also supports limited arithmetic, which occurs during instruction assembly.

#### *1.3.2.2 Use of Symbols*

Program instructions will often refer to another instruction's location in memory. If this is done without the use of symbols (i.e. using absolute references), altering the program by adding or removing lines will mean that the referring numbers may be inaccurate. Symbols defined in a LOC field, however, will always point to the instruction they precede no matter where in the program they are assembled. This provides a great deal of flexibility in programming. Symbols appearing in this field are normally given the value of the current location, but it is also possible to assign user-defined values to them.

The MIXAL operator 'EQU' is used to assign a value to a symbol. For example, the command:

```
X EQU 1000
```

sets the symbol X to be equivalent to the value 1000. The use of X in an instruction's ADDRESS field would effectively be substituted on assembly for the value 1000.

MIXAL also provides ten *local symbols*. These are symbols that can be (re)defined many times within a program. They take the form *nH* (*n* here), where *n* is a value between and including 0 and 9, and are placed in the LOC field as usual. However, reference to them takes the form of either *nF* (*n* forward) or *nB* (*n* backward); the former denotes the closest subsequent occurrence and the latter the closest preceding occurrence of *nH*. The reference symbols *nF* and *nB* cannot be used in a LOC field and *nH* cannot be used anywhere other than a LOC field. Provision in MIXAL of local symbols avoids the problem of programmers having to define a new symbol every time they wish to refer to a location.

#### *1.3.2.3 The Location Counter*

If a symbol precedes an instruction, it is assigned the same value as that currently held by the assembler's *location counter*. This counter is used by the assembler to correctly place instructions in memory; after each program line is assembled, the counter is incremented by 1. The programmer has a certain degree of control over this value, however. For example, the line:

```
ORIG 3000
```

would assign the value 3 000 to the location counter<sup>1</sup>. Assembly of the next instruction would occur in memory location 3 000; the above command does not assemble a MIX instruction itself.

Commands in MIXAL can refer to the current value of the location counter. It is denoted by the character \*, and can be used in the same way as any other symbol. Multiplication is also denoted by the same character, but this does not present a problem. The assembler interprets, for example, the instruction \*\*\* as '\* times \*'.

### 1.3.2.4 Constants

The MIXAL operator CON is used to assemble a constant value in memory. If a CON operator is encountered in the OP field of a program line, the value denoted by the ADDRESS field is assembled in a MIX word. For example, the instruction:

CON 2035

will assemble the word:

+				2035
---	--	--	--	------

into the current memory location denoted by the location counter.

A constant value enclosed in '=' characters is called a *literal constant* and is assembled in a different way. Literal constants can be used as operands for commands, and do not need a CON instruction. When the assembler encounters a literal constant, it is exchanged for an symbol (whose name is hidden from the programmer). At the end of the assembly procedure, the assembler creates CON instructions for these numbers and associates the internal symbols with their locations. This is effectively a programming 'short cut', and avoids the declaration of constants each time they are needed within a program.

### 1.3.2.5 Character-Code Assembly

MIXAL's ALF command is used to create a five-byte constant in MIX character code. The ADDRESS field of an ALF command is composed of five characters; their corresponding character codes are assembled in memory.

### 1.3.2.6 Comments

If the first character on a program line is an asterisk, the rest of the line is ignored. Programmers can use this facility to comment their programs.

---

1. Note that there is no symbol in the LOC field of this instruction; only the OP and ADDRESS fields are given.



#### *1.3.2.7 Program Input and Storage*

Programs written in MIXAL can be stored in the form of punched cards or typed-in via a terminal. If punched cards are used, they must take the following form:

Columns	1-10	LOC (location) field;
Columns	12-15	OP field;
Columns	17-80	ADDRESS field and optional remarks;
Columns	11, 16	blank.

The only exception to this is when the OP field is ALF. In this case, the remarks always start in column 22.

Input via a terminal is much more flexible. Here, the LOC field ends with the first blank space whilst the OP and ADDRESS fields (if present) begin with a nonblank character and continue to the next blank. The remainder of each line contains optional remarks.

#### *1.3.2.8 Formal Language Definition*

Knuth defines MIXAL in a formal manner in TAOCP (Knuth, 1997, pages 153-156). This is not reproduced here.

## 1.4 Future Developments of MIX

### 1.4.1 MMIX

The latest edition of *The Art of Computer Programming* contains information on a new machine, the MMIX 2009. Knuth admits that “MIX is now quite obsolete” (Knuth, 1997, page 124) and this new machine, based on RISC architecture, is more similar to computers typically used in the 1990s. However, the task of converting all the text in TAOCP will be long and laborious, and volunteers are being solicited to help with this process.

At the time of writing, Knuth has this week published documents outlining the basic structure of MMIX on his web site (Knuth, 1999b). The machine operates primarily on 64-bit words, has 256 general-purpose 64-bit registers and 256 operation codes. An operating system, NNIX, is being developed in parallel with the machine's specification. A preliminary basic MMIX assembler and interpreter has also been written by Knuth, and is available at the same location. Specifications have been designed with the aid of real-world microprocessor designers (such as ALPHA and MIPS), so the prospects for the machine are very exciting.

The MMIXMasters Homepage<sup>1</sup> has been organized by Vladimir Ivanovic to co-ordinate groups of students from around the world involved in converting TAOCP to MMIX, and Knuth envisages “friendly competition” (Knuth, 1999b) between them. His current work on volumes 4 and 5 of the book does not give him time to be involved in this process at present. However, he perceives that by 2011, the final versions of all the volumes will be complete, having each been updated to the new machine.

Unfortunately, work on the MMIX is beyond the scope of this project, but it does appear to be a very interesting project for the future.

---

1. At <http://www.mmixmasters.org/~mmixmasters/>

## 2 A Java Implementation

### 2.1 Java Applets

#### 2.1.1 Advantages

The introduction of Sun Microsystems' Java in 1995 has made the development of platform-independent programs a real possibility. Files resulting from Java program compilation are designed to be executed on any Java-supporting machine.

Platform-independence is taken one stage further in the form of applets - programs written in Java that can be embedded in the content of World-Wide-Web pages. Typically, applets are loaded by the web browser after the enclosing page is retrieved, and then executed. It is therefore possible to use these programs via any internet-connected, Java-compatible browser on any machine.

The ability to run programs on any system must come with restrictions, however. Java does not allow an Applet to access files on a user's machine<sup>1</sup>, so 'safe' operation is guaranteed. This is a useful restriction, and ensures that downloads of directly malicious programs is impossible.

Java makes the creation of graphical user interfaces extremely easy. Its sophisticated Abstract Window Toolkit (AWT) allows a programmer to create fully-functioning GUIs with only a few lines of code. This ease of use at the programming level is not found in similar languages such as C++.

#### 2.1.2 Disadvantages

The primary problem with Java-based programs is that of speed. It has been estimated that "interpreted Java runs in the range of 20 times slower than C" (Eckel, 1998, page 1039). Java applications *can* be fully compiled and executed in that form, but this eliminates the possibility of platform-independent code. To be able to run applets across the Internet, slow speed must be tolerated.

Applets cannot access the user's local disk - whether to read or write information. This restriction is in place to stop the unconscious use of malicious programs. However, it also means that it is impossible for applets to save and load data legitimately, with the user's consent.

Independent windows generated by an applet are often labelled with a prominent warning bar, containing text such as 'Warning: Applet Window' or 'Java Unsigned Applet Window'. This is to inform users that the windows do not belong to a native program and any details they enter may be transmitted back to the

---

1. Unless the applet is 'digitally signed', whereby the author attaches a personal digital signature to the program.

applet's host server. Although a handy security feature, this can leave applets looking unsightly (Eckel, 1998, page 625).

Applets must be downloaded from a server for use; they cannot be stored on a user's local disk by default. During high levels of internet activity, this download process may take a relatively long time and discourage users from running the software.

Intermittent Internet connections or browser crashes are potentially catastrophic for a running applet. There is no method of saving the current state and it is likely that information would be lost. Even a misplaced click of the mouse that leads the user to a different web page may be fatal to the applet's contents.

### **2.1.3 The Choice of Java for the Implementation**

The decision to use Java for the MIX implementation was based on the nature of *The Art of Computer Programming*. This work is very educational, and intended as such. MIX is a computer for learning about computers, programming and algorithms; Knuth states that “[u]nder favorable [sic] circumstances the reader will have access to a MIX assembler and a MIX simulator, on which various exercises in this book can be worked out” (Knuth, 1997, page 153). Java allows an assembler and simulator to be written for MIX that can be executed on any Java-supported platform. By coupling this with the accessibility of the World-Wide-Web in the form of an applet, any user who wishes to use a MIX machine can do so by visiting the appropriate web site.

The advantage of being able to run the MIX program on a variety of platforms vastly outweighs the disadvantages discussed in section 2.1.2. Programs written for MIX are typically very small; the inconvenience, therefore, of being unable to save them to the local disk is not a major one. If users wish to retain their programs for later use, they can simply be copied by hand into a suitable text editor.

### **2.1.4 The Choice of Java Version 1.1**

At the present time, three main releases of Java exist: 1.0, 1.1 and 1.2. Naturally, each release has improved on the previous version of the language. These improvements are particularly evident with the Java libraries used for building GUIs.

Java 1.0 was intended to give programmers the opportunities to build a GUI that looks equally good on all platforms. However, a “GUI that looks equally mediocre on all systems” (Eckel, 1998, page 587) is arguably the result. In addition, the GUI programming environment is untypically non-object-oriented, with very clumsy techniques having to be employed to obtain a functional program. These problems were addressed to a certain extent in Java 1.1, which introduced the second version of the Abstract Window Toolkit (AWT). The programming model of this later version is relatively simple and easy-to-use, although the GUI look-and-feel is still poor.

Release 1.2's Java Foundation Classes include a collection of GUI-based components called 'Swing'. This is a large collection of classes designed to facilitate the construction of all user interfaces, whether simple or complex. Swing's power comes from its simplicity and ease-of-use, coupled with great depth. It has the ability to present an applet or application to users in the form they expect, with the same look-and-feel as native applications. Thus, the same applet running concurrently on different platforms will look native to both desktops.

The choice of using Java 1.1 for this applet was made after careful consideration of a number of factors. At this time, the current releases of the most popular web-browsers<sup>1</sup> do not support the Java 1.2 extensions. Development of web-browser technology is rapid, and it is extremely likely that the next generation of these products will support Swing. However, this would still render a Java 1.2 applet unusable in the short term. This, coupled with the delay of user transition to the new browsers would mean that the number of potential users is small. Java 1.1 has been standard for a relatively long time, and is supported by most current browsers, so it is a natural choice for the applet. It is intended that the program's development should occur with constant feedback from users, and this can only occur if the software is usable.

At some point in the future, when Swing-supporting browsers are standard, it should be a simple task to convert the program to Java 1.2. Swing's good design means that "in many cases you can simply put a 'J' in front of the class names of each of your old AWT components" (Eckel, 1998, page 722) and the applet should function correctly. For now, Java 1.1 appears to be the best choice for the applet's development.

---

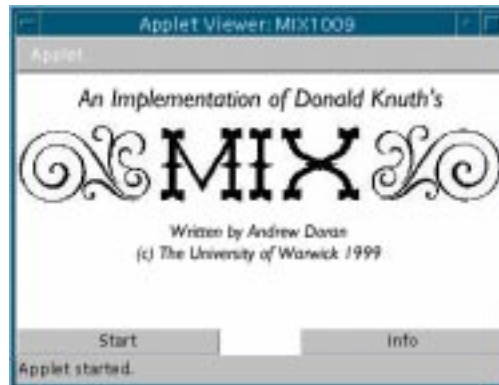
1. Both Netscape Navigator and Microsoft Internet Explorer, release 4.



## 3 Design and Functionality<sup>1</sup>

### 3.1 Introductory Applet Display

As the applet is designed to fit neatly into a web page, the initial execution of the program produces a display as shown in Figure 3.1, below.



**Figure 3.1 : Introductory Display**

This display is used to ‘present’ the applet to the user in a polished form. The applet is ‘embedded’ in an HTML file<sup>2</sup> and referenced by <APPLET> tags; this file is then viewed using an appropriate browser. AppletViewer is being used to view the applet in this case - usually, the display would be one of a number of components on a web page.

The display consists of three primary components: the introductory graphic, a ‘Start’ button and an ‘Info’ button<sup>3</sup>. Clicking the ‘Info’ button results in the appearance of the Applet Information window, shown in Figure 3.2 (below).

---

1. Images in this section are taken from ‘AppletViewer’ running on the *Common Desktop Environment* on the Sun Solaris platform.

2. See section C.a for the HTML code.

3. See section A.j for the Java code.



**Figure 3.2 : The Applet Information Window**

This window simply gives general information about the applet, such as a version number and copyright details<sup>1</sup>. At present, the graphic containing the information is completely static. A single button, 'Dismiss', closes the window.

Clicking 'Start' on the introductory display creates an instance of the MIX machine, which opens the following windows by default:

Main MIX Machine	(see section 3.2)
Control	(see section 3.6)
Program Loader (temporary window)	(see section 3.7)

After the 'Start' or 'Info' buttons are clicked, immediate further presses of them has no effect. Only one instance of the Information window and MIX machine are allowed to be active at any time. This is handled internally by two simple boolean flags that are set when each object is created and unset when they are dismissed.

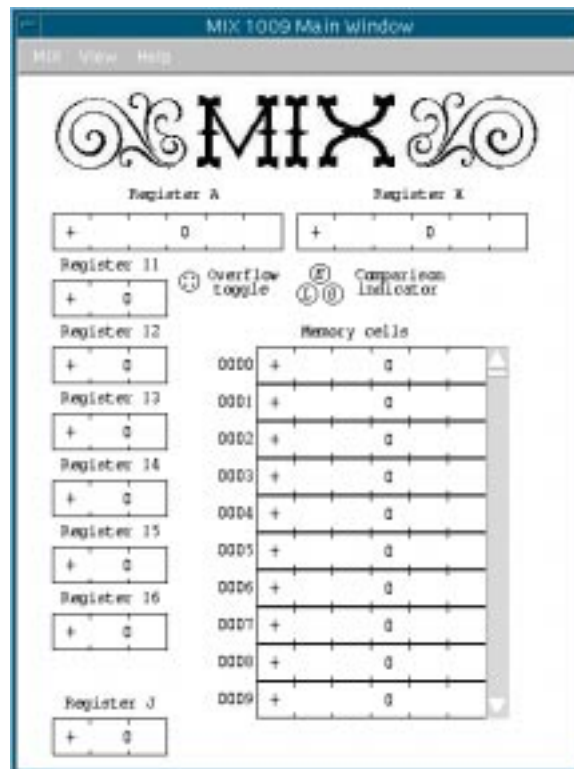
---

1. See section A.e for the Java code.



## 3.2 Main MIX Machine Window

Depression of the 'Start' button on the introductory display (see section 3.1) invokes the creation of the machine<sup>1</sup> and its associated components. The most prominent of these is the MIX machine window, shown here in Figure 3.3, below.



**Figure 3.3 : Main MIX Machine Window**

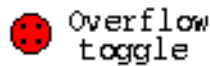
This window has been created to resemble Knuth's vision of MIX as illustrated in TAOCP (Knuth, 1997, page 126). It displays the A, X, I and J-registers along with the overflow toggle, comparison indicator and memory cells.

### 3.2.1 Components

The word-based components in the main MIX machine window display their current contents, together with the packed status of each byte (see section 1.3.1.3). The overflow toggle and comparison indicator are displayed in the form of 'lights' (see sections 4.2 and 4.3). In Figure 3.3 above, every light is off.

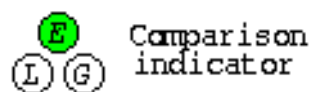
1. See section A.m for the Java code.

In the event of an overflow during the operation of MIX, the overflow indicator light is set to 'on'. This state is shown in Figure 3.4, below.



**Figure 3.4 : Overflow Toggle Set To 'On'**

Results of comparison operations are also viewed graphically via the comparison indicator. Figure 3.5 (below) shows the comparison indicator after two values have been deemed to be equal.



**Figure 3.5 : Comparison Indicator Set To 'Equal To'**

### 3.2.2 Scrollable Memory

The majority of the main MIX machine window is composed of a panel of memory cells. From Knuth's diagram and the dimensions occupied by the I-registers, it has been assumed that ten memory cells are visible at any one time. Initially, cells 0000 to 0009 are visible. By using the vertical scroll bar to the right of the set of memory cells, any of the 4 000 cells can be brought into view. Memory cells update during program execution in the same way as the A- and X-registers.

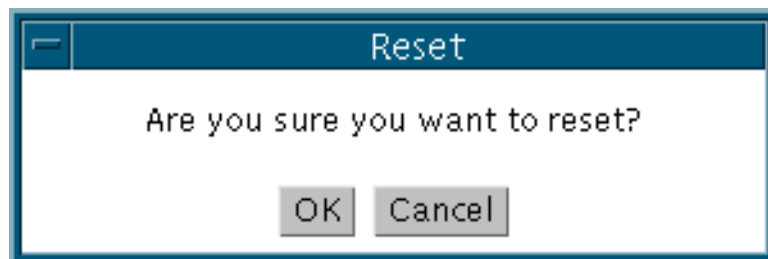
### 3.2.3 Menu Bar Options

Three menus occupy the 'menu bar' at the top of the main MIX machine window. These are titled, from left to right, 'MIX', 'View' and 'Help'.

The 'MIX' menu contains two options, 'Reset All' and 'Quit', as shown in Figure 3.6 below. Selecting 'Reset All' does not reset the machine straight away; the user is first asked whether they wish to proceed with this operation via a modal dialogue box. This is visible in Figure 3.7, also below. If the user wishes to proceed, 'OK' is clicked and the machine completely resets to the state it was in when first started. It should be noted, however, that the opened/closed state of peripheral windows and the position of the memory scroll bar are not affected by a reset.

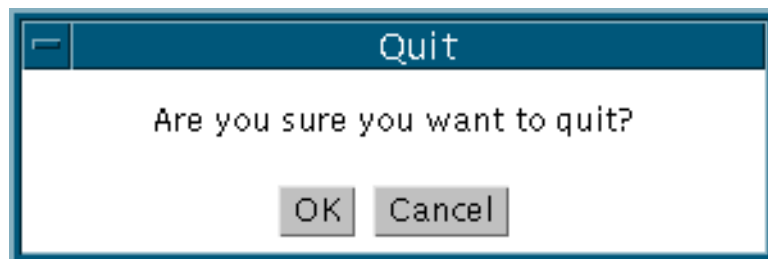


**Figure 3.6 : The MIX Menu**



**Figure 3.7 : The 'Reset All' Dialogue Box**

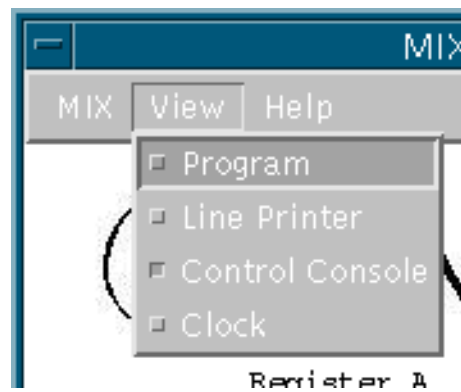
'Quit' also invokes a modal dialogue box, shown here in Figure 3.8. Here, the user is asked to confirm that they want to quit from the applet.



**Figure 3.8 : The 'Quit' Dialogue Box**

These dialogue boxes are intended as a safety feature of the machine. Resetting and quitting the applet are irreversible operations, and both have the effect of losing information currently contained in MIX. Inadvertent selection of these menu options can, by choosing 'Cancel' in the dialogue boxes, be remedied.

The 'View' menu contains four options, shown in Figure 3.9 (below), each of which has the effect of toggling the visibility of a particular peripheral window. Each window's current state is indicated by a small check box to the left of its name. In this Figure, only the control console's check box is depressed, indicating that it is currently visible. The three other peripheral windows are, in this case, hidden from the user. This is the default state in which the machine starts.



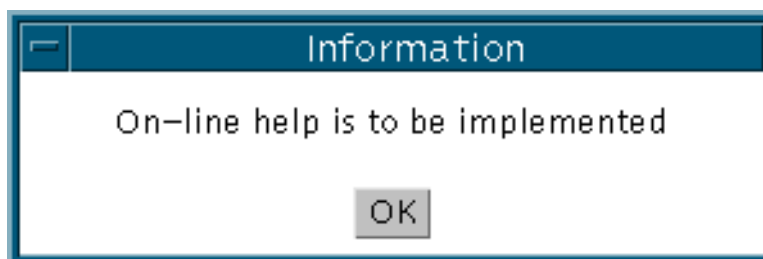
**Figure 3.9 : The View Menu**

Finally, the 'Help' menu, shown here in Figure 3.10, has one selectable menu item - 'Info'.



**Figure 3.10 : The Help Menu**

In its current state, the applet does not implement on-line help. Clicking 'Info' on this menu simply invokes the dialogue box shown in Figure 3.11.



**Figure 3.11 : The 'Info' Dialogue Box**

It is intended that on-line help should be provided with the applet. This is discussed in section 8.2.

## 3.3 Program Window

The 'Program' window is used to input MIXAL programs into the machine and acts as its 'typewriter terminal' peripheral<sup>1</sup> (see section 1.3.1.8). Choosing 'Program' from the 'View' menu on the main MIX machine (see section 3.2.3) displays the window. This is shown in Figure 3.12, below. For convenience, the window can be resized by the user; Java windows operate in the same way as those of native applications.



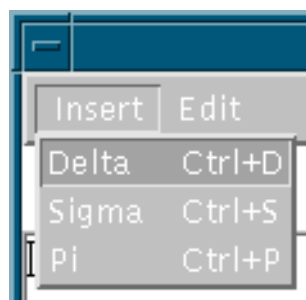
**Figure 3.12 : Program Window**

### 3.3.1 Greek Characters

The window consists mainly of a large text area. MIXAL programs are input here via the keyboard, ready for assembly. Standard QWERTY keyboards do not, however, consist of all the characters in the MIX character set (see section 1.3.1.7) - the three greek symbols of delta, sigma and pi are missing. Java complies with the Unicode character specification, so these characters can be made available for use. Users may wish to input these characters, and they can do so in one of three ways. Firstly, they are available as buttons in the top-left hand corner of the window. Clicking one of these will cause the corresponding character to be entered at the current cursor location in the text area. Secondly, the 'Insert' menu on the window's menu bar contains three selectable items, corresponding with each of the greek letters. This is shown in Figure 3.13, below. Viewing this menu reveals the last method of insertion - using keyboard shortcuts as indicated next to each menu item. Depressing the corresponding key combination for each letter, or choosing one of the 'Insert' menu options has the same effect as clicking the corresponding button.

---

1. See section A.f for the Java code.



**Figure 3.13 : The Insert Menu**

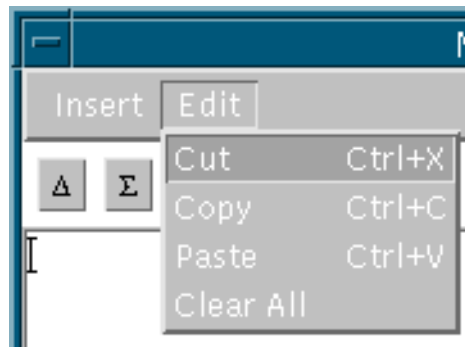
Giving the user a choice of three different character input methods allows them to use whichever technique they are accustomed to. Those who are not familiar with computer operations may find the button bar the most natural method, whereas experienced users will probably opt for the keyboard shortcuts. It is of great advantage to have these various methods present; users will spend less time attempting to discover how to input these characters.

### **3.3.2 The Clipboard**

Program editing is made significantly easier due to the presence of an internal 'clipboard'. This is an invisible area of applet memory that holds an arbitrary amount of text. Initially empty, the clipboard's contents can be set by performing a 'cut' or 'copy' operation.

The three clipboard buttons are located on the right of the button bar, shown in Figure 3.12 (above). The 'Copy' button places a copy of the currently selected text onto the clipboard. 'Cut' has a similar operation, but it also removes the selected text. 'Paste' has the effect of inserting the clipboard contents at the current cursor position.

The 'Edit' menu, shown below in Figure 3.14, also allows the user to perform these clipboard operations. In addition, corresponding keyboard shortcuts are available that have the same effect. Again, the variety of possible methods to perform a particular action gives the user a choice and is likely to assist them in achieving their goals.

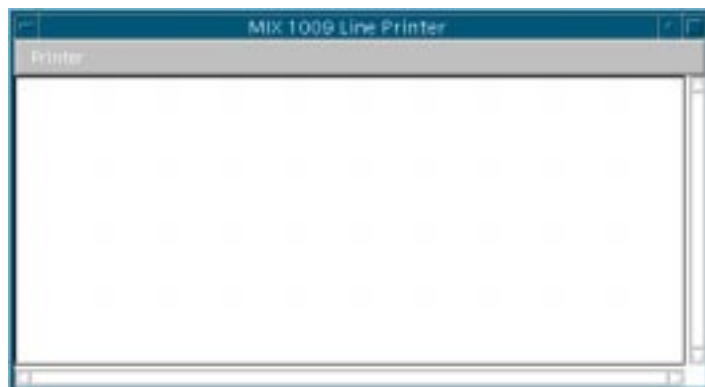


**Figure 3.14 : The Edit Menu**

Figure 3.14 shows one further menu item, 'Clear All'. At present, this item has no function, but it is intended that its selection, upon confirmation, will clear the entire contents of the text area.

## 3.4 Line Printer Window

This window acts as the 'Line Printer' peripheral (see section 1.3.1.8), and is shown in Figure 3.15, below<sup>1</sup>. It displays output from the machine. To view it, it must be selected from the main MIX machine window (see section 3.2.3). The window is very simple, consisting almost completely of a large read-only text area. If a line printer output command is executed during a MIX program, 24 words are sent to the line printer (see table 2 in section 1.3.1.8). As each word consists of five characters, a total of 120 characters are 'printed'. Thus, the total width of the text area is 120 characters; the length is unspecified.



**Figure 3.15 : Line Printer Window**

Text output to the line printer remains in its window until a 'clear all' operation is performed. There are two ways of doing this: the 'Clear All' menu item can be selected from the line printer's 'Printer' menu, or control-X can be pressed whilst the line printer has the input focus. This is shown in Figure 3.16, below. Once it selected, the contents of the text area is reset to its original empty state.

---

1. See section A.i for the Java code.

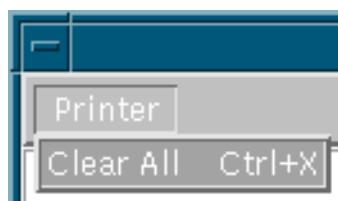


Figure 3.16 : The Printer Menu

## 3.5 Clock Window

The MIX clock<sup>1</sup> is a peripheral not originally invented by Knuth in his specification of the machine. It has been introduced as a useful tool for the experimentation with, and optimization of, algorithms. MIXAL instructions each take a particular amount of time to execute, measured in  $u$  (see section 1.3.1.6). Programs created in MIXAL, as long as they do not enter an infinite loop, thus have a definite running time; this is defined as the cumulative total of each instruction's execution time in the algorithm.

The clock can be viewed by selecting it from the main MIX machine window (see section 3.2.3). Figure 3.17, below, shows the clock in its default state.

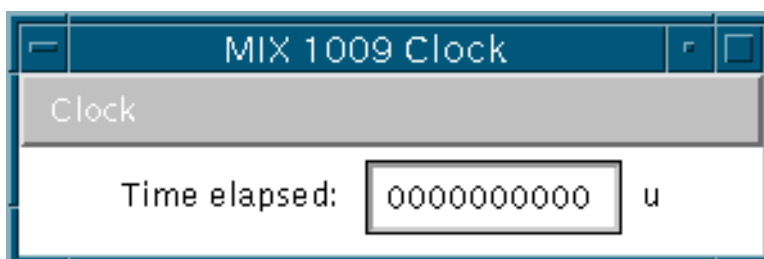


Figure 3.17 : Clock Window

During program execution, each instruction performed has the effect of incrementing the clock by a particular value. This incrementation has the same value for each instance of a particular instruction (see section 4.8.2 for individual timings). When an algorithm terminates, the clock will display the program's total running time in  $u$ .

The clock is intended to be used to compare program timings. Users can adjust their algorithms and then observe the effect on the total execution time. It is hoped that by using this peripheral in this way, efficient program composition and optimization can be explored.

---

1. See section A.1 for the Java code.



For convenience, the clock can be reset independently of the rest of the machine. The 'Reset' option from the 'Clock' menu or the key combination of control-X will perform this action, shown here in Figure 3.18. This has the simple effect of zeroing the clock value. It should be noted that the clock window must have the input focus for the key combination to work.



**Figure 3.18 : The Clock Menu**

## 3.6 Control Console

Control of the machine's operations is performed via the Control Console. The console is composed of five buttons. Of these, only 'Go' and 'Step' are implemented at present. It can be viewed by selecting it from the main MIX machine window (see section 3.2.3), and is shown in Figure 3.19, below.



**Figure 3.19 : Control Console**

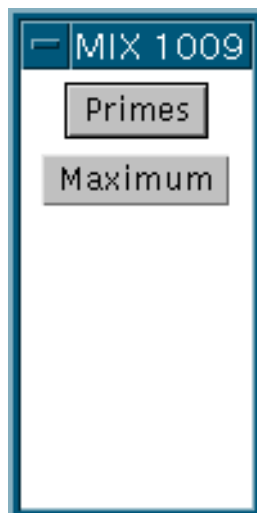
It is intended that the 'Go' button should have two functions, that of program assembly and program execution. After a MIXAL listing has been entered into the program window (see section 3.3), the first press of the 'Go' button would attempt an assembly of this program. If successful, an immediate further depression of the same button would run the program. Altering the program window's contents in any way after an assembly operation would cause the 'Go' button's action to again be that of assembly. It is not presently possible to assemble a MIXAL program; 'Go' currently just executes the program in memory (see section 4.4).

'Step' acts in much the same way as 'Go' in that it executes the current program in memory. However, as the button's name implies, only one instruction is executed. Multiple depressions allow the user to step through a program at their own pace and view the effect of each instruction. An assembled program, after being stepped through some way, can be run to completion by clicking 'Go' as long as the contents of the program window remain static.

Presently, running programs do not dynamically update the display on the main MIX machine window - stepping is the only way to see the effect of individual instructions. Upon program termination, however, each component in the system displays its final value. It is intended in a future version that the display will dynamically update, and the speed at which it will do so is to be altered by the 'Faster' and 'Slower' buttons. The default speed would be a median setting. Unfortunately, this, along with the 'Stop' operation to halt program execution, is not available in the current version.

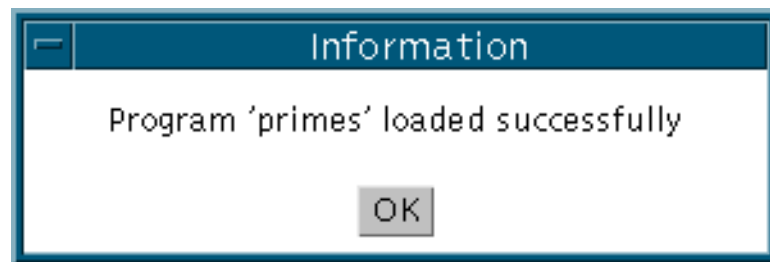
### **3.7 Program Loader (Temporary Window)**

Due to the absence of a program assembly feature in the current version, programs can be input into the machine via this temporary window. The Program Loader appears by default when the machine is started, and can be seen here in Figure 3.20.



**Figure 3.20 : Program Loader**

The two available programs, 'Primes' and 'Maximum' are specified internally in the applet code. Clicking on one of the program buttons loads its code into the machine. The user is then informed of the loading success in the form of a modal dialogue box. Figure 3.21 (below) shows the window produced when the 'Primes' button is clicked.



**Figure 3.21 : The 'Primes' Dialogue Box**

Once a program is loaded, it can be manipulated via the control console in the usual way (see section 3.6).



## 4 Implementation and Internal Structure

### 4.1 MIX Words

Word objects<sup>1</sup> are constructed by specifying an integer value denoting the number of bytes in the word. If no number is supplied, a default of five is used. Each component of a word is represented by a 30 pixels by 30 pixels image. Upon construction, the word calculates how large its graphic representation will be and resizes itself accordingly. For example, a normal-sized word consisting of five bytes and a sign will be represented by a 180 pixels by 30 pixels graphic (note that the sign here also consists of a 30 x 30 block of pixels).

Each word creates for itself an array of Byte objects to hold its values. In this implementation, bytes can hold any value between and including 0 and 63. An attempt to assign any other value to them will result in an exception. In addition, each byte has a boolean ‘packed flag’, indicating whether it is packed with the component to the immediate left (see section 1.3.1.3 for more information on packing). This packing only affects what the user sees, and not the values stored internally by the byte. The example below shows two bytes, holding the values 1 and 16 respectively, both with their packed bits set to ‘false’:

1	16
---	----

If the rightmost byte now has its packed flag set to ‘true’, it will be packed together with the byte to its immediate left. The resulting effect is the neighbouring byte is given a place value of 64 multiplied by the place value of the packed byte. In other words, the total value for the two bytes is  $(64 \times 1) + (64 \times 16) = 80$ , as shown below:

80
----

Using this method of place-value calculation, the word object correctly displays a graphical representation of its values.

After a change in state, such as the packed status or value of a byte or the value of the sign component, the entire graphic is redrawn to reflect the changes.

Words objects can be queried to return their current state as a String object or a long value. The sign, as well as individual byte values, can also be obtained.

---

1. See section A.o for the Java code.

### **4.1.1 Memory Cells**

Memory cells are standard-sized words (see section 1.3.1.1), consisting of five bytes and a sign. MIX has 4 000 of these cells, each viewable at any time by the user. The bottom-right hand corner of the main MIX machine window is dedicated to the memory display (see section 3.2.2). By dragging the scroll bar, positioned to the right of the memory cells, any contiguous block of ten cells can be brought into view.

Java does not allow the construction of a panel that holds 4 000 word objects, so the method of viewing the memory has to be dealt with internally. The solution is as follows. Ten additional 5-byte word objects are created, and each one placed on the main MIX machine window. To their left are ten text labels, each displaying a number corresponding to the cell being displayed. Movement of the scroll bar by the user triggers an internal event, which is captured by a display routine. The new value of the scroll bar is obtained, and the appropriate memory cells are copied across to the displayed cells from the memory array. In addition, the contents of the text labels are changed to indicate which cells are now being displayed.

The display technique outlined above works well. However, MIX instructions that update memory cell contents in some way would not naturally have the effects of their changes reflected in the state of the displayed word objects. To rectify this, each instruction that performs a memory-changing operation checks to see whether the cell being updated is currently displayed on screen. If so, two copies are made - one to the actual memory cell and one to its displayed counterpart.

### **4.1.2 A-Register and X-Register**

The A-register and X-register are each composed simply of a 5-byte word object, and placed directly onto the main MIX machine window. They display at all times the values currently held in the two registers.

### **4.1.3 I-Registers**

I-register objects<sup>1</sup> are a direct extension of the word object. Only one method is altered by the extension - the default constructor now explicitly uses the value 2 to denote only two bytes are needed per register. The objects are placed directly onto the main MIX machine window and dynamically update when their contents are altered.

---

1. See section A.c for the Java code.

#### 4.1.4 J-Register

The J-register object<sup>1</sup> is a direct extension of the I-register object. Like the I-register, it consists of only two bytes. However, it differs by always having a positive sign. If an attempt is made to set the J-register's sign to '-', an exception is generated.

### 4.2 Overflow Indicator

The overflow indicator is of very simple construction<sup>2</sup>, and consists primarily of a boolean value denoting its state. 'True' and 'false' indicate 'on' and 'off' respectively. The object is an extension of the Canvas object, and thus maintains its own display once placed on the main MIX machine window. If a change of state occurs, the overflow indicator redraws itself to show the new value.

### 4.3 Comparison Indicator

The comparison indicator<sup>3</sup> can be in any of four states - 'less than', 'equal to', 'greater than' or 'off'. It is appropriate, therefore, that integer values are used to set and examine the state. However, this implementation is 'hidden' from objects using the comparison indicator; the class' public static variables OFF, LESSTHAN, GREATER THAN and EQUALTO are used. Any attempt to set the state to an illegal value results in an exception. The current value of the indicator can be obtained by a simple method call.

Graphical representation of the comparison indicator's state is handled by the object itself. Comparison indicator is an extension of Canvas and can thus be placed directly onto the main MIX machine window. Changes in state are immediately reflected by the redrawing of the object.

### 4.4 MIXAL Programs and Assembly

In its pure form, a MIX computer would have to deal with the problem of loading programs and information from an external source. Whilst not really a problem in itself, information written in MIXAL would have to be converted to the language that the machine understands. Upon completion, the MIX applet will be able to perform this task of converting programs written in MIXAL directly to machine code ready for execution. They will be entered into the program window (see section 3.3) and then assembled directly. Knuth informs the reader of TAOCP that "[v]irtually all MIX programs in this book are written in MIXAL" (Knuth, 1997, page 145), and it is for this reason that an assembler should be present.

- 
1. See section A.g for the Java code.
  2. See section A.q for the Java code.
  3. See section A.b for the Java code.

### **4.4.1 Bootstrapping**

Knuth envisaged that MIX would have a 'bootstrap' routine to load programs into the machine's memory. His MIX would have a 'Go' button that, when pressed, starts the machine "from scratch" (Knuth, 1997, pages 143-144). Pushing this button results in the following actions:

- 1 A single card is read into locations 0000-0015; this is essentially equivalent to the instruction "IN 0(16)" (see section 4.8.1).
- 2 When the card has been completely read and the card reader is no longer busy, an unconditional jump to location 0000 occurs. The J-register is also set to zero.
- 3 The machine now begins to execute the program it has read from the card.

Although the above routine is written for use with a card-reader, it is possible to specify a similar algorithm for another input device. For the sake of convenience, Knuth assumes the existence of this peripheral. Using this model, the bootstrap routine would first load the MIXAL assembler into memory, before attempting to load and assemble a MIXAL program.

As our machine is implemented 'virtually', in software form, the need for this bootstrap routine no longer exists. The Java applet can take text input from the program window and create the machine code directly from it. Although bootstrapping is an important practicality of using a MIX-like machine, its use here would detract from programming and experimentation with algorithms; a hindrance rather than a help.

### **4.4.2 Assembly**

In the applet's current form, assembly of MIXAL programs is not possible. This section, however, describes the intended method to be used.

Knuth's formal definition of MIXAL, as well as his detailed description of its different features (see section 1.3.2), forms the basis for the construction of an assembler. There are many aspects to this, and the construction is not a trivial problem. Essentially, we require a full two-pass assembler for the language (Calingaert, 1979, pages 9-38).

A line of a MIXAL program is divided into three fields: LOC, OP and ADDRESS. Knuth specifies that when input is via a terminal, these three fields are delimited by blank spaces. For example, if the first character of the line is a blank space, the next nonblank character will be part of the OP field. This makes the assembly task considerably easier.

The assembly process consists mainly of two separate examinations of the program text, each known as a 'pass'. Input to the first pass is in the form of the raw text object, keyed into the program window by the user. This object is then broken up in two ways. Firstly it is tokenized, using a carriage return as a token.



Each line will thus be dealt with in turn. Secondly, the contents of the three fields are obtained by tokenizing the line, using a blank space as a token. One problem exists with this technique, however - if the first character of a line is a blank space, the first token produced by the tokenizer will be the contents of the OP field. This problem is solved by a simple check of the line's first character.

Consecutive program instructions are usually assembled in consecutive memory locations, unless explicitly specified otherwise in the program. In order to keep track of where each instruction will be placed, the assembler uses a *location counter*. Initially, the value of the location counter is zero. After each instruction, the counter's value is incremented by 1. However, assembler pseudo-commands such as ORIG and EQU do not cause a location counter increment as they do not result in assembled instructions<sup>1</sup>.

#### 4.4.2.1 Symbols and the First Pass

The main task undertaken by the first-pass is the construction of a *symbol table*, which relates MIXAL symbols to their corresponding values. Symbols in the LOC field of program instructions are placed in this table, and associated with the current value of the location counter. To ease processing, a Java class called Symbol will be created that has two components - a name (string) and an integer value. Symbol objects could then be placed into a HashSet (as provided by the Java Generic Library (JGL) (ObjectSpace, 1999)), using the same hash keys as those generated by their names. This HashSet is effectively our symbol table.

A second HashSet, this time containing just String objects, is created to hold those symbols that are used in the ADDRESS field but not yet defined, otherwise known as *forward references*. When symbols are encountered in the LOC field, this HashSet is checked for their presence. If they are in the HashSet, their entry is removed and a Symbol object is placed in the main HashSet as usual. Once all of the text has been parsed, the forward-reference HashSet is examined. If it contains any objects, they have been used in the program but not defined. Their value is deemed to be zero; Symbol objects are created for them and placed in the main HashSet but they are not removed.

The JGL HashSet object has a number of features that make it particularly suitable for use as a symbol table. Object equivalence in Java is user-definable, and Symbol objects can simply use their name field's (in other words, String name's) equivalence method. If an attempt is made to place an object into a HashSet that already contains an equivalent object, the object is returned by the method and the HashSet remains unchanged. In this way, the assembler can check the program naturally to identify any duplicated label names, which are illegal in MIXAL.

Symbols defined using the EQU operator are placed into the HashSet with their value as contained in the ADDRESS field, as opposed to the current location counter value. However, if the ADDRESS field contains an expression instead of an integer, an object is created that contains the name of the symbol, the current location counter value, the line number (in case of errors so that they can be reported to the user) and the contents of the ADDRESS field. This object is then placed in a list. The ADDRESS-field

---

1. ORIG actually specifies the memory location where the next instruction will be assembled, by changing the value of the location counter.

expressions may use other symbols, as long as they are already defined. In other words, no forward references may appear in expressions. Upon completion of the first pass, the following algorithm is used to resolve these final symbol definitions<sup>1</sup>:

- 1 Check how many items there are in the list.
- 2 Try and resolve the ADDRESS-field expression of the first object to an integer. If successful, create the corresponding Symbol object, place it in the symbol table and delete this list item.
- 3 If there are further objects in the list, move onto the next and repeat step 2.
- 4 We have reached the end of the list. If it still contains objects, check how many. Compare this number with that of step 1. If it is less, we have resolved one or more list objects and must iterate again; go to step 1. However, if the number is equivalent to that found in step 1, we have not managed to resolve any object on this pass.
- 5 If the list still contains objects, the expressions are using undefined symbols. Identify these symbols, create their Symbol objects with values of zero, and go back to step 1. It will now be possible to empty the list of objects and complete the symbol table.

Lines containing EQU commands, once resolved, are not placed in the intermediate program text, as they become redundant after entry to the symbol table. However, a note should be made by the assembler of the undefined symbols. New lines are effectively inserted before the 'END' line of a program, having the OP-field as 'CON' the ADDRESS as '0', and the name of the symbol in the LOC-field.

Local symbols are dealt with in the following manner. An array of ten integers (with indexes 0...9) is created. Initially, the value held in each location is -1. As 'here' symbols are encountered (see section 1.3.2.2), the corresponding integer value in the array is assigned the current value of the location counter. When the assembler then encounters a 'backward' reference, it can simply be substituted for the value in the array - the value of the location counter when the most recent corresponding 'here' reference was encountered.

'Backward' references can be easily checked for illegality. If one is used in the ADDRESS-field of an instruction before its corresponding 'here' symbol has been encountered, the value in the array will still be -1, as no redefinition of the value has occurred. Performing a check on this is a trivial operation.

'Forward' references present a different problem, however, and one that is well-known. In his book, Calingaert simply offers the following working example of the assembler used for the Univac I as a solution:

“That assembler, apparently the first to provide redefinable local labels, used magnetic tape for the input, intermediate and output texts. After the first pass, the intermediate

---

1. Although the majority of expression resolution takes place in the second-pass, a small amount has to occur here to complete the construction of the symbol table.

text had to be repositioned for...[p]ass 2 processing. Instead of rewinding, the assembler made an extra pass, reading the intermediate tape backward. During this backward pass, what were originally forward references could be processed in the same manner as were backward references on a forward pass.” (Calingaert, 1979, page 36)

It is not appropriate to implement this method exactly as it is specified here. MIXAL's ORIG command explicitly changes the value of the location counter; on a backwards pass, there is no simple way of maintaining thus (now-decrementing) variable. However, it is possible. Maintaining a table containing location-counter values for each program line, and then reading this table in reverse, would be one technique.

The CON operation is used to define ordinary constants; the resulting output is discussed in section 1.3.2.4. Literal constants are handled in a different way, however. When one is encountered, its address field is exchanged for an internally-generated symbol. The assembler will then insert a CON instruction just before the END line of the program, with the address field having the value of the constant. The generated symbol will appear in the LOC-field, effectively making the original instruction point to this new location. As the MIX character set is exclusively upper-case, internal symbols can be safely generated with names such as 'sym1' or 'sym2' as it is impossible for the user to define these elsewhere in the program.

As each program line is processed during the first pass, its adjusted or amended form is placed in an internal text object. It would be possible, and probably desirable, to maintain a line counter as each line is processed. If an problem arises when assembling, the program will be able to report not only the error but also the line number on which it occurs.

#### *4.4.2.2 The Second Pass*

At the end of the first pass the resulting text object is given, along with the symbol table, to the second pass as input. The second pass of the assembler is used mainly for the machine-code generation from intermediate program text. Transversal of the text object is the same as in pass one, and ORIG commands work as before. This time, however, values are known for all symbols in the text. Evaluation of ADDRESS-field expressions is thus possible for every program line, and machine-code instructions can be assembled accordingly.

The components of an ADDRESS-field do not, in certain cases, have to be specified explicitly by MIXAL commands. For example, if the programmer does not specify which subpart of a word should be loaded into the A-register in an LDA command, the entire word is implicitly transferred. The assembler will have access to a lookup table in which all of these 'normal' values will be specified for each command. Generated machine code can then use these values.

An important component of this second pass is an expression parser. MIXAL allows, and encourages, expressions to be placed in the ADDRESS-part of a program line, and these have to be resolved to

integer values. Knuth makes this task extremely easy, as “[o]perations within an expression are carried out from left to right” (Knuth, 1997, pages 153-154); there is no operator precedence.

The final command in a MIXAL program is an END instruction. The value contained in the ADDRESS-part of this line is used as the initial value of the *program counter*; execution will commence from this location.

It is arguable to a certain extent which of the tasks outlined above should occur in a particular pass. For example, a certain amount of expression parsing may have to occur to create the symbol table in pass one whereas the majority of it occurs in pass two. However, some tasks must occur at a certain stage, such as the processing of forward references in the second pass. It is believed that the above process separates the necessary tasks in a logical manner.

## 4.5 Line Printer

The line printer is an object of very simple operation<sup>1</sup>. It accepts as input single MIX words<sup>2</sup>. Each byte of the word is then examined in turn, and the corresponding character is appended to the end of the window's text area.

An integer variable, initially zero, is incremented each time a character is output. When this counter reaches 120, the maximum number of characters per line, a carriage return is added to the text area.

Contents of the line printer's window can be cleared either by an internal method call or by the user (see section 3.4). The text is simply replaced by an empty string.

## 4.6 Clock

MIX's clock holds an integer value denoting the current total running time of the program<sup>3</sup>. Initially zero, the value is incremented each time an instruction is executed by the machine. The amount of incrementation differs between instructions (see section 4.8.2); because of this, they each call the clock's 'update' method with their timing value as an argument when executed.

It can be seen in Figure 3.17 in section 3.5 that the clock always displays a 10-digit number. Each time the clock's value is incremented, the number of digits in the new value is calculated. This figure is then deducted from 10, and the resulting number of zeros added to the clock's value as a String. The previously displayed value is then replaced by this new object.

---

1. See section A.i for Java code.

2. A MIXAL print command always outputs 24 MIX words to the printer, but we assume that this is handled by the machine and not the peripheral.

3. See section A.l for Java code.

## 4.7 Unimplemented Peripherals

At the present time, magnetic tape units, disks and drums, the card reader, card punch and paper tape are not implemented. Each one of these differs mainly in block transfer size (see section 1.3.1.8), but also in the characters that they can recognise in the character set. The card reader and typewriter are the only two peripherals used for program input, the latter being the equivalent of our program window (see section 3.3), and it is important that they are implemented before the user actively experiments with programs from TAOCP. A suitable window should therefore be created that represents punched-cards which the user can alter.

Peripherals not used for input should be available to MIX internally. Data should be allowed to be written to and read from them. Although the user will not be able to alter their contents directly, windows should be constructed that allow the viewing and analysis of their contents during program execution.

These features are to be added in a future version of the applet (see section 8.1.2).

## 4.8 Program Execution and Control

Pressing either 'Go' or 'Step' on the control console causes the MIX machine's main method to be invoked, with a boolean flag argument denoting which of the two buttons has caused the invocation. The method is also called with the starting memory location as another argument. This value is assigned to the *program counter*, an internal integer variable that indicates the location of the next instruction to be executed. When the program finishes (by encountering a 'halt' instruction), or has executed a single instruction if being stepped, the method exits, with the current value of the program counter as a return value.

Figure 1.2 in section 1.3.1.6 shows the format of an instruction word. Each time the execution method is invoked, the following actions take place. Firstly, the *operation code* (C) is ascertained by reading byte 5 of the word pointed to by the program counter. The *modification* (F) of this code is then read from byte 4. As this code is used by the majority of instructions for denoting fields of words, the start and end bytes are obtained from this number. Integer division by 8 reveals the starting field, and the remainder denotes the end field. The *index specification* (I) is taken from byte 3, and finally the *address* is obtained by calculating:

$$(64 \times [\text{contents of byte 0}]) + [\text{contents of byte 1}]$$

If the sign of the word is negative, this value is multiplied by -1. This figure is then adjusted by adding the contents of I-register *i*, where *i* is the value of the index specification. Any illegal values are discovered at this point, and their existence reported to the user. These operations take place before every instruction is executed.

Each instruction changes the program counter to a new value, usually by incrementing it by 1. If the machine is not being stepped, the whole process takes place again for the next command until the machine is halted or an error occurs.

### **4.8.1 Instructions**

The operation code is used initially to ascertain which instruction is to be executed. Commands are listed below in order of this code, together with their MIX symbol equivalent and description of their action. It should be noted that many different commands may be accessible from a single code - in these cases, the F-value is specified for each instruction.

For sake of abbreviation, we define F and M as the values of the modification code and address of the instruction respectively. CONTENTS(M) is defined to be the contents of memory location M, and V the part of CONTENTS(M) specified by F.

#### **00 - NOP - No operation**

No operation occurs. Values of F and M are ignored.

#### **01 - ADD - Addition**

The value of V is added to the A-register. If the sum is too large to fit into five bytes, the result is as if a '1' had been carried to the left of the most significant byte and the overflow indicator is set to 'on'. In the case of this implementation this occurs if the sum exceeds 1 073 741 823, but varies between machines.

Knuth's examples of arithmetic are somewhat ambiguous, which he himself admits (Knuth, 1997, page 133), and this is none more evident than in the specification of packing. It is not clear from his examples how the packed status of rA's bytes should be left post-operation. The illustration provided for the SUB command (see operation 02, below) does give a small clue; the applet follows the convention that rA should end up being packed in the same way as M. Thus, the packed flags of rA's bytes are adjusted after the operation.

## 02 - SUB - Subtraction

This operation is equivalent to ADD (instruction 01, above), but with the sign of V reversed. Knuth's example of a subtraction offers a clue about the changes in the packed state of rA as a result of the operation (Knuth, 1997, page 132):

	-	1	2	3	4	0	0	9	A-register before
	-	1	0	0		1	5	0	Cell 1000
SUB 1000	+	1	3	3	4	1	4	9	A-register after

It can be argued from this that rA's packing takes on the same form as that of the operand, V, and this is the convention adopted by the applet.

The '?' in the above example is used to denote an unknown value, dependent on the byte-size of the particular machine. Knuth differentiates by means of example between arithmetic on packed and fully-unpacked words. The extra ambiguity that this introduces is clear, as illustrated above. Avoiding this problem is simple. The values of the two operands are calculated by:

$$\begin{aligned}
 &64^4 \times [\text{contents of byte 1}] + \\
 &64^3 \times [\text{contents of byte 2}] + \\
 &\dots \\
 &64^0 \times [\text{contents of byte 5}]
 \end{aligned}$$

...before performing the arithmetic operation. Once the resulting value has been assigned to the A-register, the packed status of the bytes are adjusted and a legal result obtained. Using this method does not appear to generate any ill-effects.

## 03 - MUL - Multiplication

The value V is multiplied by the contents of the A-register to create a 10-byte product. This is then stored in the A and X-registers, with the most significant byte being byte 1 of the A-register and the least significant byte being byte 5 of the X-register. Internally, the product is calculated and placed correctly into the registers by repeatedly dividing by 64 and storing the remainders in successive bytes. The signs of the A and X-registers are both set to the sign of the product.

Knuth's examples offer contradictory and ambiguous explanations of packing for multiplication operations. Therefore, the decision has been made to ignore the state of the packing and make no changes to it in this case.

### 04 - DIV - Division

The A and X-registers are treated as a 10-byte number with the sign of the A-register. This value is then divided by V. A division by zero causes the overflow indicator to be set to 'on', and the two registers to be "filled with undefined information" (Knuth, 1997, page 131). We leave them as they are. These also occurs if the quotient exceeds the maximum value of five bytes, in this case 1 073 741 823. Otherwise, the quotient is placed into the A-register and the remainder into the X-register. The sign of the X-register is set to the previous sign of the A-register, and the A-register's sign is set to the algebraic sign of the quotient.

The resulting values of the two registers do not have their packed states altered in any way. Knuth's examples are extremely unclear on this issue; by not altering the states the user should experience minimal confusion.

### 05 - NUM - Convert to numeric - *F must be 0*

This instruction is used to convert character codes into numeric code. The easiest method of explanation of its operation is by means of an example, shown below:

	A - register						X - register					
Initial contents	+	00	00	31	32	39	+	37	57	47	30	30
NUM 0	+			1 2 9 7 7 7 0 0			+	37	57	47	30	30

This diagram shows the states of the A and X-registers before a NUM operation. If we number the bytes of the A and X-registers from left to right as 0...9, the calculation is as follows:

$$\begin{aligned} & ( [ \text{contents of byte 9} ] \bmod 10 ) \times 10^0 + \\ & ( [ \text{contents of byte 8} ] \bmod 10 ) \times 10^1 + \\ & \dots \\ & ( [ \text{contents of byte 0} ] \bmod 10 ) \times 10^9 \end{aligned}$$

Thus, only the last digit of each byte is used to compose the numeric code. The A-register then takes on this value (ignoring the sign) and becomes completely unpacked. To achieve the opposite effect, the command CHAR is used (see below).



#### 05 - CHAR - Convert to characters - *F must be 1*

CHAR is used to convert numeric code into character codes, suitable for output to a peripheral such as the line printer. The numeric value stored in the A-register is converted by repeatedly dividing by 10, then storing the remainder plus 30. All signs are ignored. For example, if the A-register contained the value 12 977 700 (as in the example given in the above operation, NUM) a CHAR operation would produce the following result:

A - register						X - register					
+	30	30	31	32	39	+	37	37	37	30	30

It can be clearly seen that the second digit of each byte had composed the original value. Post-operation, the A-register is completely packed to reflect the change.

#### 05 - HLT - Halt - *F must be 2*

This operation stops the machine, and increments the program counter by 1. If 'Go' or 'Step' is then pressed, execution continues past this command. In thus case, the net effect is equivalent to a NOP (see instruction 00). At this point, the

#### 06 - SLA - Shift left A - *F must be 0*

SLA is used to shift the A-register's bytes to the left by M places. M must be nonnegative. The rightmost byte takes on the value zero once it is shifted to the left, and the leftmost byte when 'shifted off' the A-register is lost.

#### 06 - SRA - Shift right A - *F must be 1*

SRA is a 'mirror' of SLA, above. It shifts bytes in the A-register to the right by M places, where M is nonnegative.

#### 06 - SLAX - Shift left AX - *F must be 2*

Here, the A and X-registers are both shifted to the left M places, where M is nonnegative. Bytes shifted off to the left of the A-register are lost, and bytes of value 0 are shifted onto the far right of the X-register. The A and X-registers are taken to be a compound word; bytes shifted off to the left of the X-register are shifted onto the right of the A-register.

#### 06 - SRAX - Shift right AX - *F must be 3*

SRAX works in exactly the same way as SLAX, above, but with bytes moving to the right instead of left.

#### 06 - SLC - Shift left AX circularly - *F must be 4*

This operation is almost identical to that of SLAX, above. However, bytes shifted off to the left of the A-register reappear on the right of the X-register. The overall effect is a circular shift.

#### 06 - SRC - Shift right AX circularly - *F must be 5*

SRC works in exactly the same way as SLC, above, but with bytes moving to the right instead of left.

### **07 - MOVE - Move words**

In this operation, the value of F specifies the number of words to be moved (where F is zero or positive). M denotes the location in memory of the first byte to be moved, and the value of I-register 1 indicates the first location where words will be moved to. For example, if F = 50, M = 1 000 and the value of the first I-register = 2 000, words 1000 to 1049 would be moved to locations 2000 to 2049. The word 'move' is misleading; the words are actually copied to their new locations, leaving the originals unchanged. In addition to the movement of values, the state of the words' packing is also duplicated.

The movement takes place one word at a time. During this operation, an internal check is made to see whether the words being changed are displayed on screen. If they are, their on-screen equivalents are also updated (see section 4.1.1).

At the end of the operation, the I-register's value is increased by F.

### **08 - LDA - Load A**

In this operation, V replaces the previous contents of the A-register. If a partial field of V has been specified (in other words, the value of F is not 5 - see section 1.3.1.2), it is shifted over to the right of the A-register as it is loaded. All other bytes in the register are set to zero. For example, if F is the normal field specification of (0:5), the entire contents of location M are copied. If F is (1:5), everything bar the sign is transferred. A value of (4:4) would duplicate just the penultimate byte from the word.

In addition to transferring values, the packed state of each copied byte is also moved across. All other bytes are set to be unpacked.

### **09 - LD1 - Load I-register 1**

This is the same as LDA, except that the first I-register is loaded instead of the A-register. As I-registers only have two bytes instead of five, the first three are assumed to be zero. If an LD1 operation would result in setting bytes 1, 2 or 3 to anything but zero, the effect is undefined. In this implementation, an operation that attempts to transfer more than 2 bytes causes an error, and program execution halts. In practice, programs do not usually attempt this action.

### **10 to 14 - LD2 to LD6**

These commands are exactly the same as LD1 (above), but use I-registers 2 to 6 instead of 1.

### **15 - LDX - Load X**

This command is exactly the same as LDA, above, except that the X-register is loaded instead of the A-register.

### **16 - LDAN - Load A negative**

This is the same as LDA, above, except that the sign of V is reversed before the operation.

**17 - LD1N - Load I-register 1 negative**

This is the same as LD1, above, except that the sign of V is reversed before the operation.

**18 to 22 - LD2N to LD6N**

These are exactly the same as LD2 to LD6 (above), except that the sign of V is reversed before the operation.

**23 - LDXN - Load X negative**

This is the same as LDX, above, except that the sign of V is reversed before the operation.

**24 - STA - Store A**

With STA, the value of F does not specify a field of CONTENTS(M). Instead, it is used to denote a portion of the A-register. It also has the opposite significance to when it is used with a load operation (instructions 08 to 23 above); the number of bytes in the field is taken from the right-hand side of the register. These bytes are then placed into the correct part of memory location M denoted by F - any bytes not changed by this operation retain their original value. For example, if F was (2:3), the two rightmost bytes of the A-register would replace the second and third bytes of the word located at M. The sign of M is not altered unless it is part of the field, and the A-register is always left unchanged.

**25 - ST1 - Store I-register 1**

This has the same effect as STA (above), except that I-register 1 is used instead of the A-register. The first three bytes of an I-register are zero; if they are specified by F they are transferred as such.

**26 to 30 - ST2 to ST6**

These have the same effect as ST1, above, but use I-registers 2 to 6 instead of 1.

**31 - STX - Store X**

STX has the same effect as STA (above), except that the X-register is used instead of the A-register.

**32 - STJ - Store J**

This is the same as ST1 (above), except that the J-register is stored.

**33 - STZ - Store zero**

STZ works in the same way as STA (above), except that zero is stored instead of the A-register. Thus, the specified field of M is cleared to zero.

Knuth's description of STZ does not offer any information about the sign of M or the way it is packed (Knuth, 1997, page 130). In this implementation, if the sign is specified as part of the field it is set to the same value as that of the A-register. The bytes that are set to zero are packed in the same way if they had been set via STA.

### **34 - JBUS - Jump busy**

In this command, F refers to the numerical value of a peripheral (see Table 2 in section 1.3.1.8). If the unit specified is not ready for input or output, a jump occurs. This involves the J-register being set to the address of the next sequential memory location from which this JBUS command was read, and the program counter being set to M. Thus, the next instruction to be executed by the program will be that stored in location M.

If the unit is not busy, execution proceeds with the next instruction in memory - not the one stored in location M. The J-register remains unchanged.

In this implementation, we assume that the peripheral is never busy. As the computer and peripherals are all controlled by the same applet and thread, input and output will be possible at any point. MIX is not a multitasking machine. The implementation will simply stop executing the main program thread to transfer data between objects.

It is possible to create the machine and peripherals independent of each other. This command would then be given meaning; the machine could continue execution of its program whilst, say, a line of text is printed. Program running times would probably fall with this modification as the machine speed would not be dictated by peripherals (see section 8.4).

### **35 - IOC - Input-output control**

With an IOC command, F refers to the numerical value of a peripheral (see Table 2 in section 1.3.1.8). This instruction causes the machine to wait, if necessary, until the unit is not busy. A control operation is then performed. Different peripherals respond to different values of M and the MIX registers, each combination having a unique function for that device.

The majority of peripherals are not yet implemented here (see section 4.7), and it makes little sense to complete the code for this instruction beforehand. Knuth specifies an operation to skip the line printer to the top of the next page using this command, but as our version of the peripheral does not use 'real' paper the instruction is redundant.

As the MIX peripherals are implemented, the actions of this command will be constructed as specified in TAOCP (Knuth, 1997, page 137).

### **36 - IN - Input**

The IN instruction is used to transfer information from the peripheral specified by F (see Table 2 in section 1.3.1.8). M denotes the first memory location to be overwritten with this data, and the number of locations transferred is equal to the unit's 'block size'. This data is entered into consecutive memory locations.

If the specified unit is busy when this instruction is executed, the machine waits until the peripheral completes its current task. In our implementation, the unit is deemed never to be busy, for reasons discussed in instruction 34, above.

The only peripheral currently implemented is the typewriter - the program window in this implementation (see section 3.3). This is used to input MIXAL programs into the machine via the assembler, and hence the IN instruction is currently redundant. However, implementing the peripherals without this command would make no sense; both should be programmed concurrently for a future version.

### **37 - OUT - Output**

With the OUT command, F specifies the unit to which data is to be output (see Table 2 in section 1.3.1.8). A sequential block of memory starting at M and equal to the peripheral's block size is transferred. If the unit is not ready when the OUT instruction is executed, MIX waits until the peripheral finishes its current task.

Currently, for reasons specified in instruction 36 above, this command is only functional for the line printer peripheral. If F has the value 18, the number of this unit, 24 consecutive words starting at location M are output from memory to the line printer.

### **38 - JRED - Jump ready**

This command is the opposite of the JBUS command (instruction 34, above). A jump should occur if the peripheral specified by F is not busy. At present, the applet's architecture is such that both the machine and peripherals objects are controlled by a single thread. Therefore, as discussed in instruction 34, a peripheral is deemed to be always ready to receive input or give output.

In this implementation, if the peripheral has been implemented, it is always ready and a jump occurs. However, if we do not have the unit present in the system, no jump will occur. A unit that is not present will never be 'ready'.

### **39 - JMP - Jump - *F must be 0***

JMP causes an unconditional jump. The J-register is set to the current value of the program counter plus one, and the next instruction to be executed is taken from memory location M.

### **39 - JSJ - Jump, save J - *F must be 1***

A JSJ command is the same as a JMP instruction, above, except that the current contents of the J-register are not changed.

### **39 - JOV - Jump on overflow - *F must be 2***

If the overflow indicator is currently 'on', it is turned 'off' and a JMP occurs (see above). However, if it is 'off', program execution continues with the next sequential instruction; the machines registers and indicators are unchanged.

### **39 - JNOV - Jump on no overflow - *F must be 3***

If the overflow indicator is currently 'off', a JMP occurs (see above). However, if it is 'on', it is turned 'off' and the machine's registers are left unchanged; program execution continues with the next sequential instruction.

### **39 - JL - Jump on less - *F must be 4***

A JMP operation occurs if the comparison indicator is currently set to 'less than'. Otherwise, nothing happens and program execution continues with the next sequential instruction.

### **39 - JE - Jump on equal - *F must be 5***

This operation is the same as JL, above, except that a JMP instruction will occur if the comparison indicator is currently set to 'equal to'.

### **39 - JG - Jump on greater - *F must be 6***

This operation is the same as JL, above, except that a JMP instruction will occur if the comparison indicator is currently set to 'greater than'.

### **39 - JGE - Jump on greater-or-equal - *F must be 7***

This operation is the same as JL, above, except that a JMP instruction will occur if the comparison indicator is currently set to either 'greater than' or 'equal to'.

### **39 - JNE - Jump on unequal - *F must be 8***

This operation is the same as JL, above, except that a JMP instruction will occur if the comparison indicator is currently set to either 'less than' or 'greater than'. If the indicator is currently off, a jump does not occur.

### **39 - JLE - Jump on less-or-equal - *F must be 9***

This operation is the same as JL, above, except that a JMP instruction will occur if the comparison indicator is currently set to either 'less than' or 'equal to'.

### **40 - JAN - Jump A negative - *F must be 0***

If the A-register's sign is negative, a JMP operation occurs. Otherwise, nothing happens and the program continues with the next sequential instruction.

### **40 - JAZ - Jump A zero - *F must be 1***

If the A-register's value is zero, a JMP operation occurs. Otherwise, nothing happens and the program continues with the next sequential instruction.

### **40 - JAP - Jump A positive - *F must be 2***

If the value of the A-register is greater than zero, a JMP operation occurs. Otherwise, nothing happens and the program continues with the next sequential instruction.

**40 - JANN - Jump A nonnegative - *F must be 3***

If the value of the A-register is nonnegative - in other words, zero or greater - a JMP operation occurs. Otherwise, nothing happens and the program continues with the next sequential instruction.

**40 - JANZ - Jump A nonzero - *F must be 4***

If the value of the A-register is nonzero, a JMP operation occurs. Otherwise, nothing happens and the program continues with the next sequential instruction.

**40 - JANP - Jump A nonpositive - *F must be 5***

If the value of the A-register is nonpositive - in other words, zero or less - a JMP operation occurs. Otherwise, nothing happens and the program continues with the next sequential instruction.

**41 to 46**

These operations are equivalent to those specified in instruction 40, above, except that the I-registers are used instead of the A-register. The last digit of the command number dictates which I-register is to be used. For example, command 43 with a modification (F-value) of 5 would be the operation J1NP, causing a jump if I-register 3's value is nonpositive.

**47**

Operations contained in command 47 are equivalent to those in instruction 40, with the exception that the X-register is used instead of the A-register. For example, this command with a modification (F-value) of 2 would be the operation JXP, causing a jump if the X-register's value is positive.

**48 - INCA - Increase A - *F must be 0***

In this command, M does not refer to a memory location. Instead, the A-register's value is incremented by M. If the value is too large to fit into five bytes, an overflow occurs (see ADD, instruction 01, above).

**48 - DECA - Decrease A - *F must be 1***

DECA is equivalent to INCA, above, except that the sign of M is reversed before the operation.

**48 - ENTA - Enter A - *F must be 2***

This operation causes the A-register's value to be set to M. It is the quantity M that is used, *not* the memory location.

**48 - ENNA - Enter negative A - *F must be 3***

ENNA is the same as ENTA, above, except that the sign of M is reversed before the operation.

**49 to 54**

These operations are equivalent to those specified in instruction 49, above, except that the I-registers are used in place of the A-register. Instructions 49 to 54 are used for I-registers 1 to 6 respectively.

**55**

This command is identical to 48, above, except that the X-register is used instead of the A-register.

**56 - CMPA - Compare A**

CMPA compares the field specified by F in the A-register value V. The comparison indicator is then set to indicate the result of this operation. If the specified field does not include the sign of the words, the values are assumed to be nonnegative.

**57 to 62**

These commands are the same as CMPA, above, except that I-registers 1 to 6 are used instead of the A-register respectively.

**63 - CMPX - Compare X**

This instruction is the same as CMPA, above, except that the X-register is used instead of the A-register.



## 4.8.2 Timing

Each instruction, when executed, increments the machine's clock by a certain amount (see section 3.5). The cumulative result of these incrementations are used for comparisons between different algorithms. Table 3, below, shows 'how long' each instruction takes to execute:

Name	Time	Name	Time	Name	Time	Name	Time
ADD	2	INCX	1	JLE	1	MOVE	$x$
CHAR	10	IOC	1	JMP	1	MUL	10
CMPA	2	JAN	1	JNE	1	NOP	1
CMP $i$	2	JANN	1	JNOV	1	NUM	10
CMPX	2	JANP	1	JOV	1	SLA	2
DECA	1	JANZ	1	JRED	1	SLAX	2
DEC $i$	1	JAP	1	JSJ	1	SLC	2
DECX	1	JAZ	1	JXN	1	SRA	2
DIV	12	JBUS	1	JXNN	1	SRAX	2
ENNA	1	JE	1	JXNP	1	SRC	2
ENN $i$	1	JG	1	JXNZ	1	STA	2
ENNX	1	JGE	1	JXP	1	ST $i$	2
ENTA	1	J $i$ N	1	JXZ	1	STJ	2
ENT $i$	1	J $i$ NN	1	LDA	2	STX	2
ENTX	1	J $i$ NP	1	LDAN	2	STZ	2
HLT	1	J $i$ NZ	1	LD $i$	2	SUB	2
IN	1	J $i$ P	1	LD $i$ N	2		
INCA	1	J $i$ Z	1	LDX	2		
INC $i$	1	JL	1	LDXN	2		

Table 3: MIX Instruction Timings, in  $\mu$

In this table,  $i$  is any number between 1 and 6 inclusive and indicates a command that is used in conjunction with the I-registers. MOVE's time of  $x$  is interpreted as one unit plus two for each word of memory moved.

### **4.8.3 Plus and Minus Zero**

Two words that each have all of their bytes set to zero, but different signs, can be interpreted as different values. Knuth states that “minus zero is *equal to* a plus zero” (emphasis in original), and this implementation has been written to obey this convention. However, it is not completely clear how commands such as JAN should be handled when the value of the A-register is -0. In this version of the machine, the applet does not jump. If Knuth insists on the above statement, then I believe this action is correct as Zero is neither a positive nor a negative number (Weisstein, 1999).

## 5 Testing

### 5.1 Java Classes

Java's design makes it very easy to incorporate test code into programs. System components are naturally modelled in this applet by classes, and this makes testing relatively simple. Initially, each class was developed independently of its peers. This made sure that any internal errors could be located and eradicated very quickly, with no speculation as to which object was causing the problem.

Each class, whilst in development, contained a method with the signature shown below:

```
public static void main( String[] args )
{
    ...
}
```

After compilation, the classes could be individually invoked from the command-line; this would cause the `main()` method to be executed. Code placed in this method would typically create an instance of the class and call its methods, with the results displayed for easy viewing. For example, to test the Clock (see sections 3.5 and 4.6), the following code was used<sup>1</sup>:

```
public static void main( String args[] )
{
    MIXClock testClock = new MIXClock();
    testClock.setSize( 300, 85 );
    testClock.setVisible( true );
    for ( int i = 0; i < 10008; i++ )
        testClock.incrementClock( 1 );
    testClock.resetClock();
}
```

Here, an instance of the clock is created, resized and displayed. A simple for-loop then increments the clock 10 008 times. Finally, the clock is reset internally. These changes can be checked simply by observation.

Objects that do not adjust their display after an internal method invocation can still be tested. Using the `main()` method to create and view the object gives access to the graphical user interface (GUI) components. These can then be tested by pointing, clicking and typing. For example, the program window (see section 3.3) is completely GUI event-driven; its state only changes when the user selects menu options, performs button clicks, and types. By creating an instance of this component and using the GUI features provided, the object's functionality can be checked.

---

1. See section A.1 for the full Clock code.

In their final state, the applet components are created and used by other objects and do not exist completely independently. Apart from the objects' constructors, and the standard applet methods such as `update()` and `paint()`, no other methods are implicitly called. Thus, it is possible (and good practice) to leave the test code in the `main()` method of each class, as it will not be invoked inside the applet. Object testing is therefore self-contained and does not interfere with program operation.

## **5.2 MIX Instructions**

The majority of testing was performed on individual MIX instructions (see section 4.8.1). This took place for each instruction as it was coded in Java form. Initially, the HLT (halt) instruction was coded in order to stop the machine; its instruction code was then placed into memory location 0009. Locations 0000 to 0008 could then be used to test the other instructions as they were written, with the machine halting execution upon reaching location 0009.

Instructions, once coded, were placed into memory and exhaustively tested with a variety of values. Some commands are not given an exact definition by Knuth, such as SUB (subtraction) and DIV (division) (Knuth, 1997, pages 132-133); their exact operation had to be inferred to a certain extent. These design decisions are discussed for each instruction in section 4.8.1.

Each instruction causes a small change to the system state; programs are formed by using combinations of these atomic actions. This method of testing is therefore appropriate; if each atomic action is correct in itself, correct programs should work in the proper way.

## **5.3 Program Execution**

Instructions were tested individually immediately after coding, as discussed in section 5.2, with the idea that if each of their actions were correct, a complete program would be correct. This theory was tested by making the machine run a relatively complex program.

The first program to be executed on the machine was 'Program P'<sup>1</sup>, which results in a table of the first five hundred primes being output onto the line printer. This is taken directly from TAOCP and is described as being "deliberately...written in a slightly clumsy fashion in order to illustrate most of the features of MIXAL in a single program" (Knuth, 1997, page 148). From this statement, and examination of the program's contents, it was deemed to be of sufficient complexity to test the correct operation of the applet.

---

1. The operation of this program is discussed in section 6.2.

Program P is presented in MIXAL form in TAOCP<sup>1</sup>, and cannot be input directly into the machine. Instead, the code had to be converted by hand from its MIXAL source to MIX machine code; this conversion is shown in section D.a.

Once the machine code values had been ascertained and checked for correctness, the program was incorporated into the program loader (see section 3.7). Once loaded, the 'Go' button on the console was pressed. After a period of time, the machine halted itself and the contents of the line printer were examined. It complied completely with Knuth's intended output (Knuth, 1997, page 147), and the program had appeared to have executed correctly.

As a second test, 'Program M'<sup>2</sup> was loaded into the machine (Knuth, 1997, page 145). This program is used to find the maximum number in a set of integers<sup>3</sup>. To avoid infinite looping, the program's final instruction had to be changed from a JMP operation to a HLT command. Testing this algorithm on 1 000 random numbers proved successful; the program did indeed find the maximum of the values.

From these two tests, the machine's internal operation can be deemed to be correct. Later programs in TAOCP rely on peripherals that are not currently implemented (see section 4.7), and their code will thus not execute correctly at present. Once all the peripherals have been added to the applet, however, it is anticipated that they will run without problems.

## 5.4 Complete Applet

There are a number of tests that are not easily measured by standard scientific means, such as the effectiveness of the GUI design. However, this does not make them any less important. MIX has been devised to assist learning about computers and algorithms; a confusing implementation is likely to hamper the attainment of this goal.

There has not been sufficient time to extensively test and appraise how well the complete applet acts as a learning tool. The current absence of an assembler means that MIXAL programs cannot be devised and tested at present. However, the design is based very heavily on the MIX diagram found in TAOCP (Knuth, 1997, page 126) so as to produce minimal confusion to the user. Readers of the book will have a mental image of the computer, and it is hoped that a main window layout identical to the printed version will aid the user's transfer of knowledge.

It is intended that the Internet will facilitate changes and refinements to the current design. Giving universal access to the applet, by incorporating it into a web page, will allow users to provide feedback, suggestions and bug reports with ease. The applet is designed with end-users in mind, and it is of utmost importance that their comments on the software are heard and actioned. Over time, the software will

- 
1. See section D.a for the program's listing.
  2. The operation of this program is discussed in section 6.1.
  3. See section D.b for the program's listing.

evolve, become more usable, and hopefully provide many people with a useful accompaniment to Knuth's books.

## 6 Example Programs

The current version of the applet uses a temporary program loader to place programs into the machine's memory (see section 3.7). At present, two programs are available from this feature - 'Program M' (maximum) and 'Program P' (primes) - both of which are discussed below.

### 6.1 Maximum - 'Program M'

This program is the implementation of an algorithm to find the maximum of  $n$  elements. The program listing and assembled instructions can be found in section D.b, and are adapted from those found in TAOCP (Knuth, 1997, page 145).

Knuth has formulated Program M from the following simple algorithm, the programming of which serves as a useful introduction to MIX and MIXAL:

**Algorithm M** (*Find the maximum*). Given  $n$  elements  $X[1], X[2], \dots, X[n]$ , we will find  $m$  and  $j$  such that  $m = X[j] = \max_{1 \leq i \leq n} X[i]$ , where  $j$  is the largest index that satisfies this relation.

**M1.** [Initialize.] Set  $j \leftarrow n, k \leftarrow n-1, m \leftarrow X[n]$ . (During this algorithm we will have  $m = X[j] = \max_{k < i \leq n} X[i]$ .)

**M2.** [All tested?] If  $k = 0$ , the algorithm terminates.

**M3.** [Compare.] If  $X[k] \leq m$ , go to M5.

**M4.** [Change  $m$ .] Set  $j \leftarrow k, m \leftarrow X[k]$ . (This value of  $m$  is a new current maximum.)

**M5.** [Decrease  $k$ .] Decrease  $k$  by one and return to M2. ■

(Knuth, 1997, page 96).

Once the program is loaded into memory as specified in section D.b, a number of other settings of the machine must take place. I-register 1 is set to the value  $n$ , which has been chosen in this case to be 1 000. In addition, memory locations 1000 to 1999 are filled with random numbers, the values of which are between 1 and 1 000 000 000 inclusive. These correspond to the  $n$  elements that are going to be searched to find the maximum. At all times throughout the program's execution, the A-register, I-register 2 and I-register 3 hold the current values of  $m, j$  and  $k$  respectively.

After initializing all the necessary memory locations and I-register 1, the program can be executed by clicking 'Go' or 'Step' on the machine's control console (see section 3.6). Its operation is shown in Table 4, below:

Line Number	Operation
3000	Store the contents of the J-register into the (0:2) field of memory location 0009.
3001	Assign the value of $n$ to I-register 3.
3002	Jump unconditionally to memory location 3005.
3003	Compare the contents of the A-register with the contents of memory location ( $1000 + [ \text{contents of I-register 3} ]$ ). In other words, check to see if the value we are currently examining is greater than the largest one we have found so far.
3004	If the result of our comparison is 'greater than', and thus the A-register already held a larger value, jump to memory location 3007; we will do nothing more with this value. Otherwise...
3005	...we've found the largest value so far. Put its index ( $[ \text{memory location} ] - 1000$ ) into I-register 2 and...
3006	...put the value into the A-register.
3007	Decrement the I-register by 1. We are going to look at the next element.
3008	If I-register 3 is positive, we haven't examined every element in the list - go back to memory location 3003.
3009	Halt. We have completed the algorithm.

**Table 4: Operation of Program M**

Line 3009 has been changed here from an unconditional jump (JMP) instruction to a halt (HLT) instruction, and this makes line 3000 redundant. Knuth intended this code to be a subroutine for a larger program. Upon entry to the subroutine, this first line would set field (0:2) of location 0009 to be a jump command which would take the flow of control back to the point of exit from the main program. At present, the machine zeroes all of its memory locations and registers before a program is loaded; when line 3000 is executed, a value of zero is stored in location 3009. A jump instruction here would thus lead to an infinite loop - a situation avoided by the current change.



## 6.2 Primes - 'Program P'

'Program P' is far more complex than 'Program M', above. It is used to calculate, format and print a table of the first 500 prime numbers. The program listing and assembled instructions can be found in section D.a, and the former is also found in TAOCP (Knuth, 1997, pages 146-147).

Before revealing the MIXAL program to the reader, Knuth first explains the underlying algorithm, which is stated as follows:

**Algorithm P** (*Print table of 500 primes*). This algorithm has two distinct parts: Steps P1-P8 prepare an internal table of 500 primes, and steps P9-P11 print the answer in the form shown above. The latter part of the program uses two "buffers," in which line images are formed; while one buffer is being printed, the other is being filled.

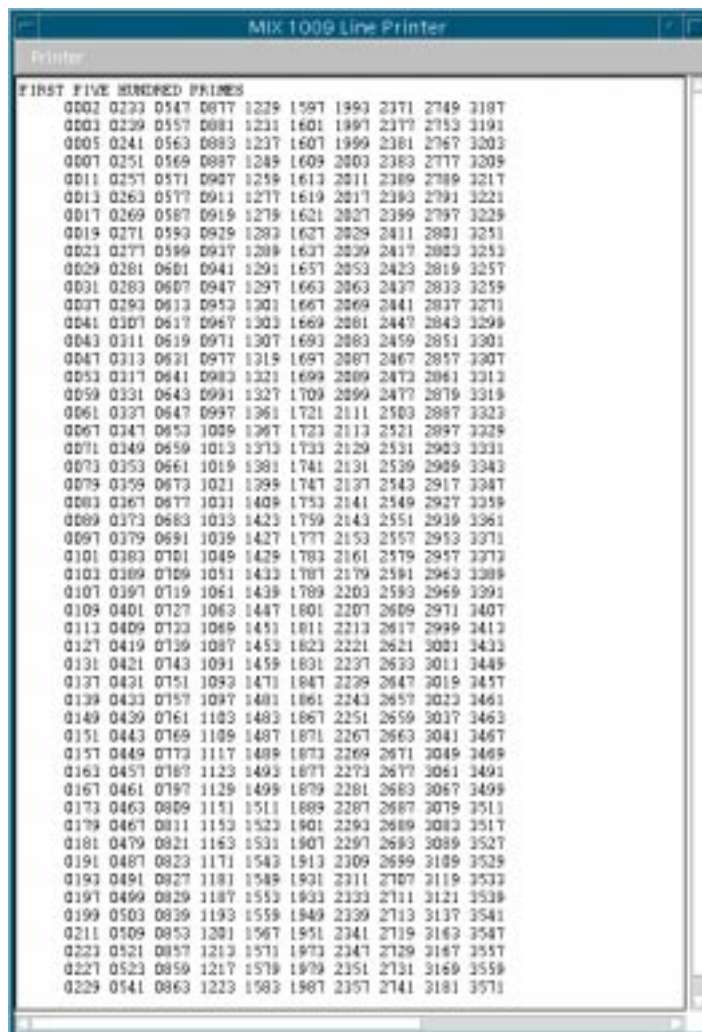
- P1.** [Start table.] Set  $\text{PRIME}[1] \leftarrow 2$ ,  $N \leftarrow 3$ ,  $J \leftarrow 1$ . (In this program,  $N$  runs through the odd numbers that are candidates for primes;  $J$  keeps track of how many primes have been found so far.)
- P2.** [ $N$  is prime.] Set  $J \leftarrow J + 1$ ,  $\text{PRIME}[J] \leftarrow N$ .
- P3.** [500 found?] If  $J = 500$ , go to step P9.
- P4.** [Advance  $N$ .] Set  $N \leftarrow N + 2$ .
- P5.** [ $K \leftarrow 2$ .] Set  $K \leftarrow 2$ . ( $\text{PRIME}[K]$  will run through the possible prime divisors of  $N$ .)
- P6.** [ $\text{PRIME}[K] \nmid N$ ?] Divide  $N$  by  $\text{PRIME}[K]$ ; let  $Q$  be the quotient and  $R$  the remainder. If  $R = 0$  (hence  $N$  is not prime), go to P4.
- P7.** [ $\text{PRIME}[K]$  large?] If  $Q \leq \text{PRIME}[K]$ , go to P2. (In such a case,  $N$  must be prime; the proof of this fact is interesting and a little unusual - see exercise 6.)
- P8.** [Advance  $K$ .] Increase  $K$  by 1, and go to P6.
- P9.** [Print title.] Now we are ready to print the table. Advance the printer to the next page. Set  $\text{BUFFER}[0]$  to the title line and print this line. Set  $B \leftarrow 1$ ,  $M \leftarrow 1$ .
- P10.** [Set up line.] Put  $\text{PRIME}[M]$ ,  $\text{PRIME}[50 + M]$ , ...,  $\text{PRIME}[450 + M]$  into  $\text{BUFFER}[B]$  in the proper format.
- P11.** [Print line.] Print  $\text{BUFFER}[B]$ ; set  $B \leftarrow 1 - B$  (thereby switching to the other buffer); and increase  $M$  by 1. If  $M \leq 50$ , return to P10; otherwise the algorithm terminates. ■

(Knuth, 1997, page 147).

Knuth's "interesting and a little unusual" fact at step P7 is a referral to the widely known fundamental theorem of arithmetic as proposed by Euclid. The theorem states that "[e]very positive integer greater than one can be expressed uniquely as a product of primes, apart from the rearrangement of terms" (Caldwell, 1999).

The program checks a number  $n$  for primality by dividing it by all the prime numbers less than  $n$ , which at this point will have been found. If none of these are factors, and hence the division (DIV) operations always leave a remainder value in the X-register, the number must be prime; the only factors that compose it are 1 and itself, from the above theorem.

Initially, the value of the first prime number, 2, is assembled as a constant in memory location 0000. As the program executes, locations 0001 onwards are filled with the values of consecutive prime numbers. Once 500 are found, steps P9 to P11 buffer and print the values in columnar format, with sequential primes listed in each column. Ten columns are used, each holding 50 primes. The technique used by Knuth to do this is to take the 1st, 51st, 101st, ..., 451st primes and copy them to consecutive memory locations. They are converted to character notation, and then printed in this format. The second line consists of the 2nd, 52nd, 102nd, ..., 452nd primes, and so on. Output of the program onto the line printer is shown here in Figure 6.1, below, and complies completely with Knuth's expected output (Knuth, 1997, page 147).



FIRST FIVE HUNDRED PRIMES									
0002	0223	0547	0877	1229	1597	1993	2371	2749	3187
0003	0229	0557	0881	1231	1601	1997	2377	2753	3191
0005	0241	0563	0883	1237	1607	1999	2381	2767	3203
0007	0251	0569	0887	1249	1609	2003	2383	2777	3209
0011	0257	0571	0907	1259	1613	2011	2389	2789	3217
0013	0263	0577	0911	1277	1619	2017	2393	2791	3221
0017	0269	0587	0919	1279	1621	2027	2399	2797	3229
0019	0271	0593	0929	1283	1627	2029	2411	2801	3251
0023	0277	0599	0937	1289	1637	2039	2417	2803	3253
0029	0281	0601	0941	1291	1657	2053	2423	2819	3257
0031	0283	0607	0947	1297	1663	2063	2437	2833	3259
0037	0293	0613	0953	1301	1667	2069	2441	2837	3271
0041	0307	0617	0967	1303	1669	2081	2447	2843	3299
0043	0311	0619	0971	1307	1693	2083	2459	2851	3301
0047	0313	0621	0977	1319	1697	2087	2467	2857	3307
0053	0317	0641	0983	1321	1699	2089	2473	2861	3313
0059	0331	0643	0991	1327	1709	2099	2477	2879	3319
0061	0337	0647	0997	1361	1721	2111	2503	2887	3323
0067	0347	0653	1029	1367	1723	2113	2521	2897	3329
0071	0349	0659	1033	1373	1733	2129	2531	2903	3331
0073	0353	0661	1039	1381	1741	2131	2539	2909	3343
0079	0359	0673	1039	1399	1747	2137	2543	2917	3347
0083	0367	0677	1031	1409	1753	2141	2549	2927	3359
0089	0373	0683	1033	1423	1759	2143	2551	2939	3361
0097	0379	0691	1039	1427	1777	2153	2557	2953	3371
0101	0383	0701	1049	1429	1783	2161	2579	2957	3373
0103	0389	0709	1051	1433	1787	2179	2581	2963	3389
0107	0397	0719	1061	1439	1789	2203	2593	2969	3391
0109	0401	0727	1063	1447	1801	2207	2609	2971	3407
0113	0409	0733	1069	1451	1811	2213	2617	2999	3413
0127	0419	0739	1087	1453	1823	2221	2621	3001	3433
0131	0421	0743	1091	1459	1831	2237	2633	3011	3449
0137	0431	0751	1093	1471	1847	2239	2647	3019	3457
0139	0433	0757	1097	1481	1861	2243	2657	3023	3461
0149	0439	0761	1103	1483	1867	2251	2659	3037	3463
0151	0443	0769	1109	1487	1871	2267	2663	3041	3467
0157	0449	0773	1117	1489	1873	2269	2671	3049	3469
0163	0457	0787	1123	1493	1877	2273	2677	3061	3491
0167	0461	0797	1129	1499	1879	2281	2683	3067	3499
0173	0463	0809	1131	1511	1889	2287	2687	3079	3511
0179	0467	0811	1153	1523	1901	2293	2689	3083	3517
0181	0479	0821	1163	1531	1907	2297	2693	3089	3527
0191	0487	0823	1171	1543	1913	2309	2699	3109	3529
0193	0491	0827	1181	1549	1931	2311	2707	3119	3533
0197	0499	0829	1187	1553	1933	2333	2711	3121	3539
0199	0503	0839	1193	1559	1949	2339	2713	3137	3541
0211	0509	0853	1201	1567	1951	2341	2719	3163	3547
0223	0521	0857	1213	1571	1973	2347	2729	3167	3557
0227	0523	0859	1217	1579	1979	2351	2731	3169	3559
0229	0541	0863	1223	1583	1987	2357	2741	3181	3571

Figure 6.1: Output From 'Program P'

## 7 Other MIX Emulators

This implementation of MIX is not the first; a number of others have been written since the 1970s. However, it is believed that this is the first time a MIX emulator has been written in Java. A Java implementation brings with it a number of advantages, such as being executable over the internet, and disadvantages - the most prominent of which is the lack of speed (see section 2). Other versions have their own particular benefits. This report would not be complete without an acknowledgment of the other known implementations of MIX, and this section is intended to give a brief overview of them.

### 7.1 MIX/360

There is little information available about this implementation; the only document to be found on the program is available from the Stanford University Computer Science Department Electronic Library<sup>1</sup>. It consists of a one-pass assembler and simulator for MIX (Knuth, 1971), and is assumed from its title to be created for use on the IBM System/360 computer.

The machine only simulates the card reader and the line printer, and offers a significant extension to the character set. Users are given significant 'traces' and 'dumps' after execution of their programs, typical of a shared card-punch based machine of this era.

Although the information supplied in Knuth's document is mainly aimed at users of MIX/360, he does include some implementation notes. The program is written completely in 360 assembly language and can execute 10 000 MIX instructions per second.

This speed is far in excess of that currently achievable by the applet. Java is known to be very slow; given the added complexities of updating the machine's components graphically, this speed is unachievable at present. Currently, the applet gives a speed of around 580 MIX instructions per second, running on an Sun Ultra-10. However, the benefits gained from being able to view the components graphically far outweighs the speed deficit. Section 7.4 contains more information on this issue.

### 7.2 UT-MIX

UT-MIX is an implementation devised for the CDC 6000-series of computers at the University of Texas (Peterson, 1977). As well as offering the basic machine, significant extensions have been added. Memory now consists of 4022 locations, with locations 4000 to 4021 being used specially by the simulator. Two magnetic tapes, two magnetic disks, a magnetic drum, one card reader and a line printer are provided as standard. The instructions JrE and JrO (where r specifies register A, X, or 1...6) have been added which jump if the register is even or odd respectively. SLB and SRB are provided for bit-

---

1. Found on the World Wide Web at <http://elib.stanford.edu/>

shifting (as opposed to bytes), XCH exchanges the values of the A and X-registers and SSP, SSN and CHS set the sign of the A-register to positive, negative or the opposite of its current value respectively. In addition, various logical operators are available for use with the A-register, missing in the original specification. MIXAL has also been significantly extended, with pseudo-instructions to access macros as well as conditional assembly of instructions. These extensions have great potential for increasing the usability and functionality of MIX, and are discussed in section 8.3 as possible additions to the applet.

Documentation for UT-MIX consists of a detailed user reference manual for the machine, with no third-party information available. The effectiveness of this implementation therefore cannot be commented upon.

### **7.3 Mixal**

Mixal is a very raw ‘load and go’ assembler and executor written in ANSI-C by Darius Bacon (no institutional affiliation). A program written in MIXAL can be executed by saving it as ASCII text and then specifying its filename as a command-line argument. The only devices currently ‘implemented’ are the “card punch” (Bacon, 1994) and line printer, which receive and send information to and from standard input and output respectively<sup>1</sup>.

The program is a fairly straightforward implementation of MIX as specified originally by Knuth, with only one minor change<sup>2</sup>. Running on a Sun Ultra-10, the software executes at an outstanding 716780 MIX instructions per second<sup>3</sup> - faster than any other implementation discovered so far. Other than output to the ‘line printer’, Mixal produces details of the register contents upon exiting a program and nothing more, which partially explains the extreme speed.

Trade-offs between speed and usability can be harnessed to a user’s advantage. The Java implementation is particularly noteworthy for its relative ease-of-use and graphical user interface, being able to help users in many ways that Mixal cannot. However, if the program is executing too slowly and its resulting output is judged to be the most important factor, Mixal will produce greater results.

---

1. The device specified for input is probably the card reader, and not the “card punch”.
2. ALF is replaced by CON “ ” to assemble character codes.
3. Tested using ‘Program P’ discussed in section 6.2.

## 7.4 MIX Builder 98

MIX Builder 98 is the most recent alternative implementation of MIX. Constructed using Delphi 4, the software is only available for use on 32-bit Windows platforms (Menees, 1998), and includes an assembler, simulator and debugger. Written by Bill Menees (no institutional affiliation), the program employs a very attractive, but non-standard, GUI<sup>1</sup> for user interaction which contains a number of special features, outlined below.

Menees has placed a new 'register' into the machine, the contents of which can be changed via the GUI but not accessed from within a program. Called LC, it contains the current value of the location counter. Assembly of literal constants now occurs after the END statement in a program and not before (see section 4.4); this apparently makes no difference as the END instruction doesn't assemble to a MIX-word anyway. The machine's character set has been altered, following the lead by Knuth with his MIX/360 implementation (see section 7.1), to make life easier for the programmer. Lastly, MIX Builder does not possess a 'Go' button - programs are "written into memory through a dedicated high-speed DMA transfer", leaving the user "free to assemble [their] programs into any memory location" (Menees, 1998).

MIX Builder's GUI contains a number of innovative features. The main window contains the program editor and debugger, the latter used for a variety of analysis purposes. In this window, each MIXAL program line is displayed adjacent to its assembled machine-code equivalent, along with a profiled execution-time of each instruction. This gives far more information about a program than our applet's simple Clock (see section 3.5); users are able to see how many times each instruction is executed and improve the total running time by adjusting the instructions. It is possible to go directly to a referenced program line from an instruction, or view the source line in the program editor. Each memory location and register is directly editable via the GUI, and easy access to a particular memory location is facilitated by entering its number into a text box.

Currently, only the card reader, card punch, line printer, typewriter and paper tape are implemented, but there are plans to implement the remaining ones in the near future.

Menees gives a good account of the speed of his emulator (Menees, 1998). He argues that the software has been coded for "clarity and safety over efficiency", and gives a performance of 2300 instructions per second on his Pentium Pro 200. This compares well with our Java implementation's performance of 580 instructions per second. Knuth's speed of 10000 instructions per second of his MIX/360 emulator (see section 7.1) is still far in excess of that achieved here, but in his defence Menees states that "Knuth is basically the God of Optimization, and he was working with Richard Sites (an architect for the Digital Alpha chip).

MIX Builder is a very well-presented, usable implementation with interesting features. However, this version still has the disadvantage that it cannot be used over the internet.

---

1. A screenshot of this can be found on-line at:  
<http://members.home.net/bmenees/Images/MBScreen.jpg>



## 8 Further Work

### 8.1 Unimplemented Features

At the present time, a number of MIX features are missing from this Java implementation. When looking to extend the project, it is very important that these are considered first. Giving the user a fully-functioning MIX machine will allow them to run every possible MIXAL program.

#### 8.1.1 The Assembler

There is currently no method of assembling MIXAL programs into MIX machine code, making the applet unprogrammable from an end-user perspective. The process of program assembly is outlined and discussed in section 4.4. By implementing this feature, users will be able to program and experiment with all algorithms that use the line printer for, or have no, output. This must be regarded as the most urgent of all future work.

#### 8.1.2 Peripherals

The line printer (see section 3.4) and the typewriter terminal (in the form of the program window - see section 3.3) are the only peripherals currently available for use. Implementation of the remaining units is a simple programming exercise. In their simplest form they can be represented by text areas in individual windows, very similar to the line printer window. The magnetic tapes, disks and drums, card reader and paper tape windows should all be user-editable before program execution, offering input to running program that requires it. The card punch window will work in the same way as the line printer, showing output to the user but disallowing its contents to be changed.

Java allows these text areas to exist on their own as objects, with no 'hidden' implementation. As their contents are written to or altered in any way the changes will be visible graphically to the user. Variations on input and output 'block size' (see section 1.3.1.8) is a little more complicated; magnetic disks, tapes and drums have to know exactly where their 'read/write' heads are located with respect to the media surface. The introduction of these peripherals brings with it the potential for many different program errors, and each should be dealt with accordingly.

### 8.1.3 Floating-Point Operations

In Knuth's specification of MIX, he briefly mentions the machine's capability of performing floating point operations. However, information on this subject is just one or two lines long and he refers the reader to chapter 4 of TAOCP. At present, these operations have not been implemented, but it is not envisaged that they will present a problem. Knuth tells us that "few of the programs we will write for MIX will use this feature, since we will be primarily concerned with algorithms on integers." Other tasks, therefore, have been deemed more important than floating-point implementation for the duration of this project. They will, however, have to be implemented in the future.

## 8.2 The Graphical User Interface

Many extensions to the applet's GUI are both possible and desirable. The information window (see section 3.1) is an obvious candidate for elaboration. As the applet is going to be used in the vast majority of cases as part of a web page, it would be a good idea to use this window as a clickable image map. Users will be able to click on images that open browser windows displaying, for example, Donald Knuth's homepage, links to MIXAL program archives or on-line help. In the case of the latter, the help provided could be of a more in-depth nature than that contained within the applet, keeping download time low. Users that require help, or more information on a particular subject, can request it from the network at the point of need. However, a certain degree of instant help should be available from the applet itself, avoiding undue hindrance to the user.

A better use of icons could be made in the case of the program window. The 'cut', 'copy' and 'paste' buttons are currently labelled with the appropriate words. Most modern applications typically use a pair of scissors, a camera and a pasting brush respectively to denote these actions; their inclusion in this applet may increase its ease of use.

At present, each peripheral window is 'owned' by the main MIX machine window and can be displayed or hidden by using its menu bar. Each window has, by default, a 'close' icon which at present does nothing. It is a simple task to provide these icons with functionality, closing the windows that own them. However, it should be realised what it naturally means to close a particular window. In the case of a peripheral it can assume to have been hidden, whereas if the main MIX machine is dismissed, the machine is 'destroyed'. Appropriate dialogue boxes should be incorporated to give the user a chance to rectify dangerous actions.

A useful feature currently not present is an undo/redo buffer for the program window. This would allow users to correct their mistakes and revert to previous versions of their program with ease. In its simplest form, a doubly-linked list of string objects could record the state at each compilation attempt. The contents of the displayed text area could then be set to any of these recorded states at will.



MIX Builder 98 (see section 7.4) offers a number of innovative GUI features. Using this machine, the user is able to see at a glance what instruction is currently being executed, as the location counter is visible. This would be a simple addition to the Java applet, with the counter being placed as a component of the main MIX machine window or displayed separately.

The presence of a program profiler in MIX Builder's debugger window is extremely useful for program design. Profiling a program post-execution is a relatively easy task; each memory location would have a unique counter, incremented by unity each time its instruction is executed. This would go some way to giving users an insight to the most computation-heavy parts of their programs.

## 8.3 MIXAL Extensions

The implementation of MIX devised at the University of Texas (see section 7.2) contains a number of extensions to the MIX assembly language. These commands are mainly simple tools that allow the programmer to perform operations with one instruction, instead of a sequence of them. Instructions such as AND, OR and LNG (logical negate) should arguably have been included in the original MIX command set, but their exclusion simply creates more work for the programmer.

There is some value in extending the MIX command set, especially if sophisticated algorithms are being used. If memory is tight, commands such as those outlined above will save space and allow a greater number of programs to be executed. However, it is debatable whether the number of instructions should be complicated any further; the machine is built as a beginner's tool, and advanced programmers will probably implement their algorithms in a more 'mainstream' language. This is an aspect of MIX that should be developed once user demand is great enough.

## 8.4 Program Architecture

Currently, the flow of control in the MIX applet is contained in a single Java thread. Thus, whilst output is being passed to a peripheral, the machine cannot continue execution until the unit has finished. Allowing the peripherals to become separate threads of execution is likely to increase the speed, and give a more realistic model of a computer's operation. The machine and peripherals will have to be synchronized, with information passing between the two only when they are both ready for the transaction. This would involve changing the JBUS instruction, for example, to jump if a peripheral is busy (see section 4.8.1).

Changing the program architecture in this way would involve restructuring of the applet's code, whereas other extensions in this chapter consist of additions to it. The program is likely also to gain an increased degree of modularity and reusability; components are developed separately and only communicate with each other through synchronization.



## 9 Conclusions

The majority of the aims of this project as outlined in section 1.2 have been realised by the implementation. However, it has not been possible during the time allowed to write a MIXAL assembler, disallowing the user from experimenting with program code. As a future extension, this has utmost priority before modifications and additions to the design are considered. It is perfectly feasible to incorporate the assembler into MIXMachine.java (see section A.m), whilst adding a button to translate the contents of the program window into machine code.

The ability to download and use MIX from a web-page is a very attractive prospect. Two of the four alternative implementations outlined in section 7 are intended for (now) obsolete computers, and another is an exclusively windows-based application. An increasing number of people have access to the internet, and their interests in computer programming can be explored using Knuth's books in conjunction with this machine. The graphical nature of the applet allows users to become acquainted with its operation very quickly, using the mental model constructed from Knuth's books.

Choosing Java, as well as a graphical display, for the implementation slows the machine down considerably. However, the intention is that MIX should be used as a learning tool - no serious application is likely to be written in MIXAL - so this does not present a major problem. As machine speeds increase, the execution time of the applet will decrease; using a just-in-time (JIT) compiler as part of a browser interface is also likely to give a good performance increase.

As it currently stands, the applet achieves the majority of the aims of the project. The potential for future expansion is great, and is outlined in section 8.



## 10 References

Bacon, 1994 - Bacon, Darius; *READ.ME File for the Mixal Distribution*;

downloaded as part of <http://www.tuxedo.org/~esr/retro/mixal-1.07.tar.gz>;

file date unknown; accessed on 21 April 1999.

Caldwell, 1999 - Caldwell, Chris K; *Fundamental Theorem of Arithmetic from The Prime Pages' Glossary*;

<http://www.utm.edu/research/primes/glossary/FundamentalTheorem.html>;

file date unknown; accessed on 17 April 1999.

Calingaert, 1979 - Calingaert, Peter; *Assemblers, Compilers and Program Translation*; Pitman Publishing.

Eckel, 1998 - Eckel, Bruce; *Thinking In Java*; Prentice-Hall.

Knuth, 1971 - Knuth, Donald E and Sites, R L; *MIX/360 User's Guide*; Computer Science Department, School of Humanities and Sciences, Stanford University; March 1971

Knuth, 1996 - Knuth, Donald E; *One-Paragraph Autobiography*;

<http://www-cs-staff.stanford.edu/~uno/autobio.ps.gz>;

file dated 22 August 1996; accessed on 17 April 1999.

Knuth, 1997 - Knuth, Donald E; *The Art of Computer Programming*; 3rd edition; Addison-Wesley.

Knuth, 1998 - Knuth, Donald E; *Brief Biography*;

<http://www-cs-staff.stanford.edu/~uno/bio.ps.gz>;

file dated 25 September 1998; accessed on 17 April 1999.

Knuth, 1999a - Knuth, Donald E; *The Art of Computer Programming*;

<http://www-cs-staff.stanford.edu/~uno/taocp.html>;

file date unknown; accessed on 17 April 1999.

Knuth, 1999b - Knuth, Donald E; *MMIX 2009*;

<http://www-cs-staff.stanford.edu/~uno/mmix.html>;

file date unknown; accessed on 23 April 1999.

Menees, 1998 - Menees, Bill; *MIX Builder 98*;

downloaded as part of <http://members.home.net/bmenees/Files/MIXBuilder.zip>;

file dated 11 October 1998; accessed on 22 April 1999.

ObjectSpace, 1999 - ObjectSpace; *ObjectSpace JGL - The Generic Collection Library for Java*;  
<http://www.objectspace.com/products/jgl/>;  
file date unknown; accessed on 17 April 1999.

Peterson, 1977 - Peterson, James L; *UT-MIX Reference Manual*; Department of Computer Sciences, The University of Texas at Austin; January 1971;  
<http://lonestar.texas.net/~jpeterso/html/papers/utmix/report.htm>;  
file date unknown; accessed on 21 April 1999.

Weisstein, 1999 - Weisstein, Eric W; *CRC Concise Encyclopedia of Mathematics*; CRC Press;  
<http://www.astro.virginia.edu/~eww6n/math/Zero.html>;  
file date unknown; accessed on 17 April 1999.

## 11 Acknowledgments

I would like to express my thanks to my supervisor, Dr Mike Joy, who was always available for consultation and feedback when I needed it throughout the course of my project. I must also acknowledge the help of Tom Evans, a third year Computer Systems Engineering student, who suggested different ideas to me whilst I was implementing the applet's graphical-user interface. Without the assistance of these two people, my project would be far from complete.





## Appendix A - Authored Program Code

### A.a CharNotASignException.java

```
//: CharNotASignException.java
//  An Exception thrown by MIXSign

/** CharNotASignException is an Exception thrown by MIXSign
 *   @author Andrew Doran
 *   @author http://andrew.doran.com/
 *   @version 0.11 - 30 March 1999
 *   @see MIXSign
 */
class CharNotASignException extends Exception { }

//::~~
```

## A.b ComparisonIndicator.java

```
//: ComparisonIndicator.java
// A four-state indicator that is set upon a MIX comparison
import java.awt.*;

/** ComparisonIndicator maintains the state of the comparison indicator in the MIX1009.
 * @author Andrew Doran
 * @author http://andrew.doran.com/
 * @version 0.11 - 30 March 1999
 */
class ComparisonIndicator extends Canvas
{
    static final int OFF = 0;
    static final int LESSTHAN = 1;
    static final int GREATERTHAN = 2;
    static final int EQUALTO = 3;

    private int state;

    private static final Font CFONT = new Font( "Monospaced", Font.PLAIN, 12 );
    private static final Font CFONTI = new Font( "Monospaced", Font.ITALIC, 12 );
    private static final int CINDICATORWIDTH = 130;
    private static final int CINDICATORHEIGHT = 35;

    /** Loads the comparison indicator images and puts the machine
     * into an initial 'indicator off' state.
     * @return No return value.
     */
    ComparisonIndicator()
    {
        state = OFF;
        setSize( CINDICATORWIDTH, CINDICATORHEIGHT );
    }

    /** Sets the state of the comparison indicator
     * @param newState a valid integer state value.
     * @return No return value.
     * @exception NotAValidStateException thrown if the newState is not a valid state.
     */
    void setState( int newState ) throws NotAValidStateException
    {
        if ( ( newState < 0 ) || ( newState > 3 ) )
            throw new NotAValidStateException();
        else if ( state != newState )
        {
            state = newState;
            repaint();
        }
    }

    /** Returns the numeric state of the comparison indicator
     * @return int value representing state of the comparison indicator.
     */
    int getState()
    {
        return state;
    }

    /** (Re)draws the comparison indicator, with the current state
     * highlighted, or every indicator off if we have not been used
     * yet.
     * @param g Graphics object
     * @return No return value.
     */
    public void paint( Graphics g )
    {
        g.setColor( Color.white );
        g.fillRect( 0, 0, CINDICATORWIDTH, CINDICATORHEIGHT );
        g.setColor( Color.green );
        if ( state == LESSTHAN )
            g.fillArc( 10, 18, 15, 15, 0, 360 );
        if ( state == EQUALTO )
            g.fillArc( 20, 2, 15, 15, 0, 360 );
        if ( state == GREATERTHAN )
    }
}
```

```
        g.fillArc( 30, 18, 15, 15, 0, 360 );
g.setColor( Color.black );
g.setFont( CFONT );
g.drawString( "Comparison", 55, 15 );
g.drawString( "indicator", 57, 26 );
g.setFont( CFONTI );
g.drawString( "E", 23, 14 );
g.drawString( "L", 13, 30 );
g.drawString( "G", 33, 30 );
g.drawArc( 20, 2, 15, 15, 0, 360 );
g.drawArc( 10, 18, 15, 15, 0, 360 );
g.drawArc( 30, 18, 15, 15, 0, 360 );
    }

    // Eliminates flicker (from Eckel p.686)
    public void update( Graphics g )
    {   paint(g);
    }
}

///~
```

## A.c IRegister.java

```
//: IRegister.java
//  A two-byte 'word' used for the MIX1009 I-Registers.

/** IRegister is a two-byte word
 *  @author Andrew Doran
 *  @author http://andrew.doran.com/
 *  @version 0.11 - 30 January 1999
 */
class IRegister extends MIXWord
{
    /** Default constructor
     *  @return No return value.
     */
    IRegister()
    {
        super( 2 );
    }
}

//:~
```

## A.d IndexOutOfRangeException.java

```
//: IndexOutOfRangeException.java
// An Exception thrown by MIXWord

/** IndexOutOfRangeException is an Exception thrown by MIXWord
 * @author Andrew Doran
 * @author http://andrew.doran.com/
 * @version 0.11 - 30 March 1999
 * @see MIXSign
 */
class IndexOutOfRangeException extends Exception { }

//::~~
```

## A.e InfoWindow.java

```
//: InfoWindow.java
// Displays a 'dialogue box' information window
import java.awt.*;
import java.awt.event.*;
import ImageLabel;

/** @author Andrew Doran
 *  @author http://andrew.doran.com/
 *  @version 0.11 - 30 March 1999
 */
public class InfoWindow extends Frame
{
    private Panel bottomPanel = new Panel();           // This panel holds the
    private Button dismissButton = new Button( "Dismiss" ); // Button to close the InfoWindow

    /** InfoWindow constructor
     *  @param legend ImageLabel to be displayed
     *  @param windowTitle String containing requested window title
     */
    public InfoWindow( ImageLabel legend, String windowTitle )
    {
        setLayout( new BorderLayout() );
        setBackground( Color.white );
        setResizable( false );                        // No resizing as it is an Info box
        setTitle( windowTitle );                      // Assign the proper title
        setSize( legend.getWidth(), legend.getHeight() + 50 ); // Make the window a viewable size
        bottomPanel.setLayout( new FlowLayout( FlowLayout.CENTER ) );
        bottomPanel.setBackground( Color.white );
        dismissButton.addActionListener( new DismissActionL() );
        dismissButton.setBackground( Color.lightGray ); // Give the button a 'pushy' feel
        bottomPanel.add( dismissButton );
        add( "South", bottomPanel );                  // Add the Dismiss button panel
        add( "Center", legend );                      // Add the ImageLabel
    }

    /** Closes the window and thus invokes a componentHidden event
     */
    private class DismissActionL implements ActionListener
    {
        public void actionPerformed( ActionEvent e )
        {
            setVisible( false );
        }
    }
}

///:~
```

## A.f InputConsole.java

```
//: InputConsole.java
// Window for editing MIXAL programs
import java.awt.*;
import java.awt.event.*;
import java.awt.datatransfer.*;

/** InputConsole allows the user to edit and prepare MIXAL programs
 *  for the MIX1009.
 *  @author Andrew Doran
 *  @author http://andrew.doran.com/
 *  @version 0.11 - 30 March 1999
 */
class InputConsole extends Frame
{
    private static final String SIGMA = ( "" + ((char)0x03A3)),
                                DELTA = ( "" + ((char)0x0394)),
                                PI    = ( "" + ((char)0x03A0));

    private static final Font MONOSPACEDFONT = new Font( "Monospaced", Font.PLAIN, 12 );

    private Button sigma,                // These are for character input...
                  delta,
                  pi,
                  cut,                  // ...and these are for window actions.
                  copy,
                  paste;

    private TextArea textinput;          // Main window for editing programs

    private Panel greekPanel,           // Panel to hold greek character buttons
                  utilPanel,            // Panel to hold action buttons
                  buttonBar;           // Holds both of the above panels

    private MenuBar mb;                 // Our menu bar

    private Menu insertMenu,
                  editMenu;

    private MenuItem[] greekItems = { new MenuItem( "Delta", new MenuShortcut( KeyEvent.VK_D ) ),
                                       new MenuItem( "Sigma", new MenuShortcut( KeyEvent.VK_S ) ),
                                       new MenuItem( "Pi"   , new MenuShortcut( KeyEvent.VK_P ) ) },
                  editItems = { new MenuItem( "Cut"   , new MenuShortcut( KeyEvent.VK_X ) ),
                               new MenuItem( "Copy"  , new MenuShortcut( KeyEvent.VK_C ) ),
                               new MenuItem( "Paste" , new MenuShortcut( KeyEvent.VK_V ) ),
                               new MenuItem( "Clear All" ) };
                                     // An item to clear the entire window

    private Clipboard clipbd;           // For cut and paste operations

    private InsertDeltaL deltaL;         // 'ActionListener' objects, created
    private InsertSigmaL sigmaL;         // this way so that they can be attached
    private InsertPiL    piL;           // to both the menu items and the button
    private CutL         cutL;          // bar.
    private CopyL        copyL;
    private PasteL       pasteL;

    /** Sole constructor for the ImageConsole
     *  @return No return value.
     */
    InputConsole()
    {
        greekPanel = new Panel();
        utilPanel  = new Panel();
        buttonBar  = new Panel();

        delta = new Button( DELTA );
        sigma = new Button( SIGMA );
        pi    = new Button( PI );
        cut   = new Button( "Cut" );
        copy  = new Button( "Copy" );
        paste = new Button( "Paste" );
    }
}
```

```
delta.setBackground( Color.lightGray );           // The buttons should be grey to make
sigma.setBackground( Color.lightGray );           // them look 'pushable' on the white
pi.setBackground( Color.lightGray );              // background
cut.setBackground( Color.lightGray );
copy.setBackground( Color.lightGray );
paste.setBackground( Color.lightGray );

textinput = new TextArea(1, 70);                  // 70 Characters wide is enough
textinput.setFont( MONOSPACEDFONT );              // Keep all the buttons and the text
delta.setFont( MONOSPACEDFONT );                  // area in our monospaced font for
sigma.setFont( MONOSPACEDFONT );                  // consistency.
pi.setFont( MONOSPACEDFONT );

// This section sets up the display
setLayout( new BorderLayout() );
greekPanel.setLayout( new FlowLayout( FlowLayout.LEFT ) );
greekPanel.add( delta );
greekPanel.add( sigma );
greekPanel.add( pi );

utilPanel.setLayout( new FlowLayout( FlowLayout.LEFT ) );
utilPanel.add( cut );
utilPanel.add( copy );
utilPanel.add( paste );

buttonBar.setLayout( new GridLayout(1,2) );
buttonBar.add( greekPanel );
buttonBar.add( utilPanel );

buttonBar.setBackground( Color.white );

add( "North", buttonBar );
add( "Center", textinput );

mb = new MenuBar();
insertMenu = new Menu( "Insert" );
editMenu = new Menu( "Edit" );

for ( int i = 0; i < greekItems.length; i++ )
    insertMenu.add( greekItems[i] );
for ( int i = 0; i < editItems.length; i++ )
    editMenu.add( editItems[i] );

mb.add( insertMenu );
mb.add( editMenu );
setMenuBar( mb );
textinput.requestFocus();                          // Start with the input focus in the
setTitle( "MIX 1009 Program" );                     // text window
// End of display setup section

clipbd = new Clipboard( "InputConsoleClipboard" ); // We cannot access the system clipboard
deltaL = new InsertDeltaL();                         // from an applet
sigmaL = new InsertSigmaL();
piL = new InsertPiL();
cutL = new CutL();
copyL = new CopyL();
pasteL = new PasteL();

delta.addActionListener( deltaL );
greekItems[0].addActionListener( deltaL );
sigma.addActionListener( sigmaL );
greekItems[1].addActionListener( sigmaL );
pi.addActionListener( piL );
greekItems[2].addActionListener( piL );
cut.addActionListener( cutL );
editItems[0].addActionListener( cutL );
copy.addActionListener( copyL );
editItems[1].addActionListener( copyL );
paste.addActionListener( pasteL );
editItems[2].addActionListener( pasteL );

editItems[3].addActionListener( new ClearAllL() );
}
```



```
/** Lets the calling object obtain the program from the console
 * @return String returned by getText()
 */
String getProgram()
{ return textinput.getText();
}

private boolean weHaveSelectedText()
{ if ( textinput.getSelectionStart() == textinput.getSelectionEnd() )
    return false;
  else return true;
}

private class InsertDeltaL implements ActionListener
{ public void actionPerformed( ActionEvent e )
  { if ( weHaveSelectedText() )
    textinput.replaceRange(DELTA,textinput.getSelectionStart(),textinput.getSelectionEnd());
    else textinput.insert( DELTA, textinput.getCaretPosition() );
  }
}

private class InsertSigmaL implements ActionListener
{ public void actionPerformed( ActionEvent e )
  { if ( weHaveSelectedText() )
    textinput.replaceRange(SIGMA,textinput.getSelectionStart(),textinput.getSelectionEnd());
    else textinput.insert( SIGMA, textinput.getCaretPosition() );
  }
}

private class InsertPiL implements ActionListener
{ public void actionPerformed( ActionEvent e )
  { if ( weHaveSelectedText() )
    textinput.replaceRange(PI, textinput.getSelectionStart(), textinput.getSelectionEnd());
    else textinput.insert( PI, textinput.getCaretPosition() );
  }
}

/* Originally, with the following classes, I wanted to allow access
 * to the system clipboard so that programs could be cut and pasted into
 * the window after being downloaded from the web site. However, due to
 * security reasons, applets are not allowed access to the system clipboard
 * so this has been modified to used an 'internal' clipboard.
 */

private class CutL implements ActionListener // From Eckel pp. 702
{ public void actionPerformed( ActionEvent e )
  { String selection = textinput.getSelectedText();
    StringSelection clipString = new StringSelection( selection );
    clipbd.setContents( clipString, clipString );
    textinput.replaceRange( "", textinput.getSelectionStart(), textinput.getSelectionEnd() );
  }
}

private class CopyL implements ActionListener // From Eckel pp. 702
{ public void actionPerformed( ActionEvent e )
  { String selection = textinput.getSelectedText();
    StringSelection clipString = new StringSelection( selection );
    clipbd.setContents( clipString, clipString );
  }
}

private class PasteL implements ActionListener // From Eckel pp. 702
{ public void actionPerformed( ActionEvent e )
  { Transferable clipData = clipbd.getContents( InputConsole.this );
    try
    { String clipString = (String)clipData.getTransferData( DataFlavor.stringFlavor );
      if ( weHaveSelectedText() )
        textinput.replaceRange( clipString, textinput.getSelectionStart(),
                                textinput.getSelectionEnd() );
      else textinput.insert( clipString, textinput.getCaretPosition() );
    }
    catch( Exception ex )
    { System.out.println( "Clipboard doesn't contain pastable text." );
    }
  }
}
```

```
    }  
}  
  
private class ClearAllL implements ActionListener  
{ public void actionPerformed((ActionEvent e)  
  { System.out.println("Clear All selected...");  
  }  
}  
  
// Test code  
public static void main( String args[] )  
{ InputConsole win = new InputConsole();  
  win.setSize( 500, 300 );  
  win.show();  
}  
}
```

## A.g JRegister.java

```
//: JRegister.java
//  A positive-only version of an I-Register.

/** JRegister is a positive-only form of IRegister
 *  @author Andrew Doran
 *  @author http://andrew.doran.com/
 *  @version 0.11 - 30 March 1999
 */
class JRegister extends IRegister
{
    /** We disallow the setting of the J-Register's sign to negative
     *  @param sign char denoting desired sign.
     *  @return No return value.
     *  @exception JRegisterMustBePositiveException Thrown if an attempt is made
     *          to set the sign of the J-Register to anything but '+'
     */
    void setSign( char sign ) throws JRegisterMustBePositiveException
    {
        if ( sign != '+' )
            throw new JRegisterMustBePositiveException();
    }
}

//:~
```

## A.h JRegisterMustBePositiveException.java

```
//: JRegisterMustBePositiveException.java
//  An Exception thrown by JRegister

/** JRegisterMustBePositiveException is an Exception thrown by JRegister
 *  @author Andrew Doran
 *  @author http://andrew.doran.com/
 *  @version 0.11 - 30 March 1999
 *  @see JRegister
 */
class JRegisterMustBePositiveException extends CharNotASignException { }

//:~
```

## A.i LinePrinter.java

```
//: LinePrinter.java
// The visible output of the MIX 1009.
import java.awt.*;
import java.awt.event.*;

/** LinePrinter is the window used for simulated physical output of
 * the MIX1009 machine, basically consisting of a TextArea.
 * @author Andrew Doran
 * @author http://andrew.doran.com/
 * @version 0.11 - 30 March 1999
 */
class LinePrinter extends Frame
{ private static final String SIGMA = ( "" + ((char)0x03A3)),
    DELTA = ( "" + ((char)0x0394)),
    PI = ( "" + ((char)0x03A0));

    private static final Font MONOSPACEDFONT = new Font( "Monospaced", Font.PLAIN, 12 );

    private TextArea textOutput;

    private MenuBar mb;

    private Menu printerMenu;

    private MenuItem clearAll;

    private int charCounter,
        MAXLINELENGTH = 120;

    /** Sole constructor for the LinePrinter.
     * @return No return value.
     */
    LinePrinter()
    { charCounter = 0;
      textOutput = new TextArea(1, 70);
      textOutput.setFont( MONOSPACEDFONT );
      textOutput.setEditable( false );
      textOutput.setBackground( Color.white );
      textOutput.setForeground( Color.black );

      mb = new MenuBar();
      printerMenu = new Menu( "Printer" );
      mb.add( printerMenu );
      clearAll = new MenuItem( "Clear All", new MenuShortcut( KeyEvent.VK_X ) );
      clearAll.addActionListener( new ClearAllActionL() );
      printerMenu.add( clearAll );

      setLayout( new BorderLayout() );
      setTitle( "MIX 1009 Line Printer" );

      setMenuBar( mb );

      add( "Center", textOutput );
    }

    /** Outputs the character equivalents of a MIX word to the line printer
     * @param word The MIXWord to be printed
     * @return No return value.
     * @exception NotAMIXCharacterException Thrown if the character is not in the range 0-55
     */
    void print( MIXWord word ) throws NotAMIXCharacterException
    { int characterCode;
      char character = ' ';

      try
      { for (int i=0; i<5; i++)
        { characterCode = word.getValue( i );

          switch (characterCode)
          { case 0 : character = ' '; break;
```

```
case 1 : character = 'A'; break;
case 2 : character = 'B'; break;
case 3 : character = 'C'; break;
case 4 : character = 'D'; break;
case 5 : character = 'E'; break;
case 6 : character = 'F'; break;
case 7 : character = 'G'; break;
case 8 : character = 'H'; break;
case 9 : character = 'I'; break;
case 10 : character = ((char)0x0394); break; // Delta
case 11 : character = 'J'; break;
case 12 : character = 'K'; break;
case 13 : character = 'L'; break;
case 14 : character = 'M'; break;
case 15 : character = 'N'; break;
case 16 : character = 'O'; break;
case 17 : character = 'P'; break;
case 18 : character = 'Q'; break;
case 19 : character = 'R'; break;
case 20 : character = ((char)0x03A3); break; // Sigma
case 21 : character = ((char)0x03A0); break; // Pi
case 22 : character = 'S'; break;
case 23 : character = 'T'; break;
case 24 : character = 'U'; break;
case 25 : character = 'V'; break;
case 26 : character = 'W'; break;
case 27 : character = 'X'; break;
case 28 : character = 'Y'; break;
case 29 : character = 'Z'; break;
case 30 : character = '0'; break;
case 31 : character = '1'; break;
case 32 : character = '2'; break;
case 33 : character = '3'; break;
case 34 : character = '4'; break;
case 35 : character = '5'; break;
case 36 : character = '6'; break;
case 37 : character = '7'; break;
case 38 : character = '8'; break;
case 39 : character = '9'; break;
case 40 : character = '.'; break;
case 41 : character = ','; break;
case 42 : character = '('; break;
case 43 : character = ')'; break;
case 44 : character = '+'; break;
case 45 : character = '-'; break;
case 46 : character = '*'; break;
case 47 : character = '/'; break;
case 48 : character = '='; break;
case 49 : character = '$'; break;
case 50 : character = '<'; break;
case 51 : character = '>'; break;
case 52 : character = '@'; break;
case 53 : character = ';'; break;
case 54 : character = ':'; break;
case 55 : character = '\\'; break;
default : throw new NotAMIXCharacterException();
}

textOutput.append( ""+character );
charCounter++;
if (charCounter == MAXLINELENGTH)
{ textOutput.append( ""+'\\n' );
  charCounter = 0;
}
}
}
catch (Exception e)
{ if (e instanceof NotAMIXCharacterException)
  throw (NotAMIXCharacterException)e;
}
}

/** Clears the contents of the LinePrinter by blanking the text area
 * @return No return value
```

```
*/
void clear()
{ textOutput.setText("");
  charCounter = 0;
}

private class ClearAllActionL implements ActionListener
{ public void actionPerformed( ActionEvent e )
  { System.out.println( "Clear All selected..." );
    textOutput.setText("");
    charCounter = 0;
  }
}

// Test code
public static void main(String args[]) throws NotAMIXCharacterException,
                                              ValueOutOfBoundsException,
                                              IndexOutOfRangeException
{ LinePrinter lpr = new LinePrinter();
  MIXWord word = new MIXWord();
  word.setValue(0, 1);
  word.setValue(1, 15);
  word.setValue(2, 4);
  word.setValue(3, 28);

  lpr.setSize( 550, 300 );
  lpr.setVisible( true );

  while (true)
    lpr.print( word );
}

///

---


```

### A.j MIX1009.java

```
//: MIX1009.java
// An Implementation of Donald Knuth's MIX
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
import java.net.URL;
import ImageLabel;

/** An Implementation of Donald Knuth's MIX
 * Third Year Project 1998-99
 * Department of Computer Science, University of Warwick
 * @author Andrew Doran
 * @author http://andrew.doran.com/
 * @version 0.11 - 30 March 1999
 */
public class MIX1009 extends Applet
{ static final boolean OPEN      = true,           // Set up some English definitions
  CLOSED      = false;

  private static Button beginButton = new Button( "Start" ),
    infoButton  = new Button( "Info" );
  private static Panel  bottomPanel = new Panel();    // Holds the buttons
  private static ImageLabel MIXLogo,                // Main applet window logo
    infoImage;                                       // Info window contents

  private static boolean begun      = CLOSED,        // We start with no windows open
    infoWindow = CLOSED;

  /** Gets us into a valid working state on web page.
   * @return No return value.
   */
  public void init()
  { showStatus( "Loading images..." );           // Let the user know
                                           // what's happening

    MIXLogo = new ImageLabel( getDocumentBase(), "graphics/MIX.jpg" ); // Load the MIX logo
    MIXLogo.waitForImage( true );
    infoImage = new ImageLabel( getDocumentBase(), "graphics/MIXInfo.jpg" ); // Load the Info
                                           // window information
    infoImage.waitForImage( true );
    showStatus( "" );

    beginButton.addActionListener( new BeginActionL() ); // Tie the buttons to their respective
    infoButton.addActionListener( new InfoActionL() );   // actions
    beginButton.setBackground( Color.lightGray );        // Colour them grey so they look
    infoButton.setBackground( Color.lightGray );         // 'pushable'

    setLayout( new BorderLayout() );                  // Intro window layout
    setBackground( Color.white );                     // Make the background the same color as
                                           // the image
    bottomPanel.setLayout( new GridLayout(1,2,60,0) ); // Button bar layout
    bottomPanel.setBackground( Color.white );
    bottomPanel.add( beginButton );
    bottomPanel.add( infoButton );
    add( "South", bottomPanel );                      // Put the button bar on the bottom
    add( "Center", MIXLogo );                         // Put the image in the center
  }

  /** Lets us know whether the Info window is open or closed.
   * @return boolean representing either OPEN or CLOSED
   */
  private boolean getInfoWindowStatus()
  { return infoWindow;
  }

  /** Sets the Info window status to open or closed.
   * @param status representing either OPEN or CLOSED
   */
  private void setInfoWindowStatus( boolean status )
  { infoWindow = status;
  }
}
```



```
}

/** Creates and displays the Info window.
 * @return No return value.
 */
private void displayInfoWindow()
{ final InfoWindow iw = new InfoWindow( infoImage, "MIX : Info" );
  setInfoWindowStatus( OPEN );
  iw.addComponentListener( new ComponentAdapter()
  { public void componentHidden( ComponentEvent e )
    { iw.dispose();
      setInfoWindowStatus( CLOSED );
    }
  } );
  iw.setVisible( true );
}

/** Will invoke the Info window if it is not already displayed.
 */
private class InfoActionL implements ActionListener
{ public void actionPerformed( ActionEvent e )
  { if ( getInfoWindowStatus() == CLOSED )
    displayInfoWindow();
  }
}

/** Lets us know if the MIX1009 machine window has been opened.
 * @return boolean representing either OPEN or CLOSED
 */
private boolean getMIXMachineStatus()
{ return begun;
}

/** Sets the status of the MIX1009 machine window to open or closed.
 * @param status representing either OPEN or CLOSED
 * @return No return value.
 */
private void setMIXMachineStatus( boolean status )
{ begun = status;
}

/** Creates and displays the MIX1009 machine window.
 * @return No return value.
 */
private void startMIXMachine()
{ final MIXMachine mm = new MIXMachine();
  setMIXMachineStatus( OPEN );
  mm.addComponentListener( new ComponentAdapter()
  { public void componentHidden( ComponentEvent e )
    { mm.dispose();
      setMIXMachineStatus( CLOSED );
    }
  } );
  mm.setVisible( true );
}

/** Starts the MIX1009 machine if it hasn't already been opened.
 */
private class BeginActionL implements ActionListener
{ public void actionPerformed( ActionEvent e )
  { if ( getMIXMachineStatus() == CLOSED )
    startMIXMachine();
  }
}
}

//::~~
```

### A.k MIXByte.java

```
//: MIXByte.java
// A component of a MIX1009 'word'

/** MIXByte holds a value between 0 and 63 inclusive. It is used to construct
 * MIX 'words' in conjunction with MIXSign.
 * @author Andrew Doran
 * @author http://andrew.doran.com/
 * @version 0.11 - 30 March 1999
 */
class MIXByte
{ private int value;
  private boolean packed;

  /** Sets up the initial state of the MIXByte.
   * @return No return value.
   */
  MIXByte()
  { value = 0; // This is dependent on the notion that each MIX
    packed = true; // word starts with the value of + 0 (i.e. packed)
  }

  /** Sets the value of the MIXByte (must be between 0 and 63 inclusive).
   * @param newValue int value between 0 and 63.
   * @return No return value.
   * @exception ValueOutOfBoundsException Thrown if the value specified does not lie between
   * 0 and 63 inclusive.
   */
  void setValue(int newValue) throws ValueOutOfBoundsException
  { if (( newValue < 0 ) || ( newValue > 63 ))
    throw new ValueOutOfBoundsException();
    else value = newValue;
  }

  /** Retrieves the value of the MIXByte
   * @return int value
   */
  int getValue()
  { return value;
  }

  /** Sets the status of the 'packed' flag for the MIXByte.
   * @param p boolean where ( true == packed ) and ( false == unpacked ).
   * @return No return value.
   */
  void setPacked( boolean p )
  { packed = p;
  }

  /** Retrieves the status of the 'packed' flag for the MIXByte.
   * @return boolean packed status.
   */
  boolean isPacked()
  { return packed;
  }
}

//:~
```

## A.I MIXClock.java

```
//: MIXClock.java
// Gives program timing information to the user.
import java.awt.*;
import java.awt.event.*;

/** MIXClock gives the user a readout of how long a program has
 * taken to execute. Measured in u, the time is mainly used
 * for comparative purposes between algorithms and optimization.
 * @author Andrew Doran
 * @author http://andrew.doran.com/
 * @version 0.11 - 30 March 1999
 */
class MIXClock extends Frame
{ private int clock = 0; // Zero to start with...

  private TextField clockDisplay;

  private Label timeElapsed,
               u;

  private MenuBar mb;
  private Menu clockMenu;
  private MenuItem resetClock;
  private Panel clockPanel;

  /** Sole constructor for the MIXClock.
   * @return No return value.
   */
  MIXClock()
  { clockDisplay = new TextField( 10 );
    timeElapsed = new Label( "Time elapsed: ", Label.RIGHT );
    u = new Label( "u", Label.LEFT );

    clockDisplay.setEditable( false );
    clockDisplay.setText( "0000000000" );

    clockPanel = new Panel();
    setLayout( new BorderLayout() );
    setTitle( "MIX 1009 Clock" );

    mb = new MenuBar();
    clockMenu = new Menu( "Clock" );
    mb.add( clockMenu );
    resetClock = new MenuItem( "Reset", new MenuShortcut( KeyEvent.VK_X ) );
    resetClock.addActionListener( new ResetClockActionL() );
    clockMenu.add( resetClock );

    setMenuBar( mb );

    clockPanel.setLayout( new FlowLayout() );
    clockPanel.setBackground( Color.white );

    clockPanel.add( timeElapsed );
    clockPanel.add( clockDisplay );
    clockPanel.add( u );

    add( BorderLayout.CENTER, clockPanel );
  }

  /** Updates the clock display to hold the correct number of
   * digits accounting for changes in the length of the number.
   * @return No return value.
   */
  private void updateClock()
  { int digitCount = 1,
    divisions = clock;

    String clockText = "";

    do
```

```
{ divisions = ( divisions / 10 );
  digitCount++;
}
while( divisions != 0 );

for ( int i = 1; i < (12 - digitCount); i++ )
  clockText += "0";

clockText += clock;
clockDisplay.setText( clockText );
}

/** Increments the clock by the specified number and updates the
 * display accordingly.
 * @param int value to increment the clock by.
 * @return No return value.
 */
void incrementClock( int increment )
{ clock += increment;
  updateClock();
}

/** Resets the clock to zero.
 * @return No return value.
 */
void resetClock()
{ clock = 0;
  updateClock();
}

private class ResetClockActionL implements ActionListener
{ public void actionPerformed((ActionEvent e) )
  { System.out.println( "Reset Clock selected..." );
    resetClock();
  }
}

// Test code
public static void main( String args[] )
{ MIXClock testClock = new MIXClock();
  testClock.setSize( 300, 85 );
  testClock.setVisible( true );
  for ( int i = 0; i < 10008; i++ )
    testClock.incrementClock( 1 );
  testClock.resetClock();
}
}
```

## A.m MIXMachine.java

```
//: MIXMachine.java
// The main MIX frame object that controls the machine
import java.awt.*;
import java.awt.event.*;
import java.util.StringTokenizer;
import ImageLabel;

/** MIXMachine 'controls' the main objects in the Applet and shows the
 * user the internal states of the MIX1009.
 * @author Andrew Doran
 * @author http://andrew.doran.com/
 * @version 0.11 - 30 March 1999
 */
class MIXMachine extends Frame
{ // ---- Memory and memory indicators
  private MIXWord[]      memory;
  private MIXWord        aRegister,
                        xRegister;
  private IRegister[]    iRegisters;
  private JRegister      jRegister;
  private ComparisonIndicator cIndicator;
  private OverFlowIndicator oIndicator;

  // ---- Peripherals
  // private MagneticTapeUnit[] tapeUnit;      // see Knuth p. 137
  // private DiskOrDrumUnit[]   diskUnit;
  // private CardReader         cardReader;
  // private CardPunch          cardPunch;
  private LinePrinter        lpr;
  // private PaperTape          paperTape;
  private InputConsole       program;

  // ---- My peripherals
  private ControlConsole     control;
  private MIXClock           clock;
  private ProgramLoader      pLoader;

  // ---- GUI declarations
  private static final Font MIXFont = new Font( "Monospaced", Font.PLAIN, 12 );
  private MenuBar mb;
  private Menu mixMenu,
              viewMenu,
              helpMenu;
  private MenuItem reset,
                  quit,
                  info;
  private CheckboxMenuItem[] viewOptions;
  private ImageLabel logo;
  private Label regA,
              regX,
              regJ,
              memoryCells;
  private Label[] regIx,
                  memoryLabels;
  private MIXWord[] memoryVisible;
  private Panel memoryPanel,
              aXRegPanel,
              mainPanel;
  private Scrollbar memoryScroller;
  private GridBagLayout mGBL,
                      pGBL;
  private GridBagConstraints mGBC,
                      pGBC;

  private int globalPC = 0; // SEE CONTROL SECTION - START VALUE
                          // OF STEPPED PROGRAM

  private boolean choiceFlag = false;

  /** Sole entry point to machine. This constructor establishes a MIX1009
   * in its default state.
```

```
* @return No return value.
*/
MIXMachine( ImageLabel logo )
{ // ---- Instantiate memory and indicator components
  memory      = new MIXWord[ 4000 ];
  for ( int i=0; i < 4000; i++ )
    memory[ i ] = new MIXWord();
  aRegister   = new MIXWord();
  xRegister   = new MIXWord();
  iRegisters  = new IRegister[ 6 ];
  for ( int i=0; i < 6; i++ )
    iRegisters[ i ] = new IRegister();
  jRegister   = new JRegister();
  cIndicator  = new ComparisonIndicator();
  oIndicator  = new OverFlowIndicator();

  // ---- Instantiate peripherals
  // tapeUnit = new MagneticTapeUnit[ 8 ];
  // diskUnit = new DiskOrDrumUnit[ 8 ];
  // cardReader = new CardReader();
  // cardPunch = new CardPunch();
  lpr         = new LinePrinter();
  // paperTape = new PaperTape();
  program     = new InputConsole();

  // ---- Instantiate my peripherals
  control     = new ControlConsole();
  clock       = new MIXClock();
  pLoader     = new ProgramLoader();

  // ---- Setup GUI - menus
  mb = new MenuBar();
  mixMenu = new Menu( "MIX" );
  viewMenu = new Menu( "View" );
  helpMenu = new Menu( "Help" );
  reset    = new MenuItem( "Reset All" );
  quit     = new MenuItem( "Quit" );
  info     = new MenuItem( "Info" );
  viewOptions = new CheckboxMenuItem[ 4 ];
  viewOptions[ 0 ] = new CheckboxMenuItem( "Program" );
  viewOptions[ 1 ] = new CheckboxMenuItem( "Line Printer" );
  viewOptions[ 2 ] = new CheckboxMenuItem( "Control Console" );
  viewOptions[ 3 ] = new CheckboxMenuItem( "Clock" );

  reset.addActionListener( new ResetActionL() );
  quit.addActionListener( new QuitActionL() );
  info.addActionListener( new HelpActionL() );
  viewOptions[0].addItemListener( new ProgramItemActionL() );
  viewOptions[1].addItemListener( new PrinterItemActionL() );
  viewOptions[2].addItemListener( new ControlItemActionL() );
  viewOptions[3].addItemListener( new ClockItemActionL() );

  mixMenu.add( reset );
  mixMenu.add( quit );
  for ( int i = 0; i < viewOptions.length; i++ )
    viewMenu.add( viewOptions[i] );
  helpMenu.add( info );
  mb.add( mixMenu );
  mb.add( viewMenu );
  mb.add( helpMenu );
  setMenuBar( mb );

  // ---- Setup GUI - frame
  setTitle( "MIX 1009 Main Window" );
  setSize( 450, 600 );
  setResizable( false );
  regA = new Label( "Register A", Label.CENTER );
  regX = new Label( "Register X", Label.CENTER );
  regJ = new Label( "Register J", Label.CENTER );
  regA.setFont( MIXFont );
  regX.setFont( MIXFont );
  regJ.setFont( MIXFont );
  regIx = new Label[ 6 ];
  for ( int i = 0; i < 6; i++ )
```

```

{   regIx[ i ] = new Label( "Register I" + (i+1), Label.CENTER );
    regIx[ i ].setFont( MIXFont );
}

// ----- Create memory panel
memoryPanel = new Panel();
memoryCells = new Label( "Memory cells", Label.CENTER );
memoryCells.setFont( MIXFont );
memoryScroller = new Scrollbar( Scrollbar.VERTICAL, 0, 10, 0, 4000 );
memoryScroller.addAdjustmentListener( new MemoryAdjustL() );
memoryLabels = new Label[ 10 ];
memoryVisible = new MIXWord[ 10 ];
for ( int i = 0; i < 10; i++ )
{   memoryLabels[ i ] = new Label( "000" + i, Label.RIGHT );
    memoryLabels[ i ].setFont( MIXFont );
    memoryVisible[ i ] = new MIXWord();
    mMOVE( memory[ i ], memoryVisible[ i ] );
}
mGBL = new GridBagLayout();
memoryPanel.setLayout( mGBL );
mGBC = new GridBagConstraints();
memoryPanel.setBackground( Color.white );
mGBC.fill = GridBagConstraints.HORIZONTAL;
addComponent( memoryCells, 0, 0, 3, 1, mGBL, mGBC, memoryPanel );
mGBC.fill = GridBagConstraints.NONE;
for ( int i = 0; i < 10; i++ )
    addComponent( memoryVisible[ i ], 1, i+1, 1, 1, mGBL, mGBC, memoryPanel );
mGBC.fill = GridBagConstraints.VERTICAL;
addComponent( memoryScroller, 2, 1, 1, 10, mGBL, mGBC, memoryPanel );
mGBC.anchor = GridBagConstraints.EAST;
for ( int i = 0; i < 10; i++ )
    addComponent( memoryLabels[ i ], 0, i+1, 1, 1, mGBL, mGBC, memoryPanel );

// ----- Create A/XRegister panel
aXRegPanel = new Panel();
aXRegPanel.setLayout( new GridLayout( 2, 2 ) );
aXRegPanel.setBackground( Color.white );
aXRegPanel.add( regA );
aXRegPanel.add( regX );
aXRegPanel.add( aRegister );
aXRegPanel.add( xRegister );

// ----- Create main panel
mainPanel = new Panel();
pGBL = new GridBagLayout();
mainPanel.setLayout( pGBL );
pGBC = new GridBagConstraints();
pGBC.fill = GridBagConstraints.HORIZONTAL;
mainPanel.setBackground( Color.white );
addComponent( logo, 0, 0, 3, 2, pGBL, pGBC, mainPanel );
addComponent( aXRegPanel, 0, 2, 3, 2, pGBL, pGBC, mainPanel );
pGBC.anchor = GridBagConstraints.EAST;
addComponent( oIndicator, 1, 4, 1, 2, pGBL, pGBC, mainPanel );
addComponent( cIndicator, 2, 4, 1, 2, pGBL, pGBC, mainPanel );
pGBC.anchor = GridBagConstraints.CENTER;
for ( int i = 0; i < 6; i++ )
    addComponent( iRegisters[ i ], 0, 5+(i*2), 1, 1, pGBL, pGBC, mainPanel );
pGBC.anchor = GridBagConstraints.SOUTH;
addComponent( regJ, 0, 16, 1, 1, pGBL, pGBC, mainPanel );
addComponent( jRegister, 0, 17, 1, 1, pGBL, pGBC, mainPanel );
pGBC.anchor = GridBagConstraints.CENTER;
addComponent( memoryPanel, 1, 6, 2, 11, pGBL, pGBC, mainPanel );
pGBC.fill = GridBagConstraints.HORIZONTAL;
for ( int i = 0; i < 6; i++ )
    addComponent( regIx[ i ], 0, 4+(i*2), 1, 1, pGBL, pGBC, mainPanel );

setLayout( new BorderLayout() );
add( "Center", mainPanel );

// ---- Setup GUI - peripheral windows
program.setSize( 500, 300 );
//     program.setVisible( true );
//     viewOptions[0].setState( true );
lpr.setSize( 550, 300 );

```

```
//      lpr.setVisible( true );
//      viewOptions[1].setState( true );
      control.setSize( 300, 80 );
      control.setVisible( true );
      viewOptions[2].setState( true );
      clock.setSize( 300, 100 );
//      clock.setVisible( true );
//      viewOptions[3].setState( true );
      pLoader.setSize( 100, 200 );
      pLoader.setResizable( false );
      pLoader.setVisible( true );
    }

/** addComponent method from Deitel & Deitel p.634
 */
private void addComponent( Component c, int row, int column, int width, int height,
                          GridBagConstraints gbc, Panel p )
{
    gbc.gridx = row;
    gbc.gridy = column;
    gbc.gridwidth = width;
    gbc.gridheight = height;
    gbl.setConstraints( c, gbc );
    p.add( c );
}

private class ResetActionL implements ActionListener
{
    public void actionPerformed( ActionEvent e )
    {
        System.out.println( "Reset selected..." );
        MIXChoiceDialog message = new MIXChoiceDialog( "Reset", "Are you sure you want to reset?" );
        if ( choiceFlag == true )
        {
            clearMemoryContents();
            memoryScroller.setValue( memoryScroller.getValue() ); // Repaints the memory values.
            clock.resetClock();
            lpr.clear();
        }
    }
}

private class QuitActionL implements ActionListener
{
    public void actionPerformed( ActionEvent e )
    {
        System.out.println( "Quit selected..." );
        MIXChoiceDialog message = new MIXChoiceDialog( "Quit", "Are you sure you want to quit?" );
        if ( choiceFlag == true )
        {
            program.dispose();
            lpr.dispose();
            control.dispose();
            clock.dispose();
            pLoader.dispose();
            setVisible( false );
        }
    }
}

private class HelpActionL implements ActionListener
{
    public void actionPerformed( ActionEvent e )
    {
        System.out.println( "Help selected..." );
        MIXDialog message = new MIXDialog( "Information", "On-line help is to be implemented" );
    }
}

/** Shows/hides the program window
 */
private class ProgramItemActionL implements ItemListener
{
    public void itemStateChanged( ItemEvent e )
    {
        if ( ((CheckboxMenuItem)e.getSource()).getState() )
        {
            System.out.println( "Program selected..." );
            program.setVisible( true );
        }
        else
        {
            System.out.println( "Program deselected..." );
            program.setVisible( false );
        }
    }
}
}
```



```
/** Shows/hides the printer window
*/
private class PrinterItemActionL implements ItemListener
{ public void itemStateChanged( ItemEvent e )
  { if ( ((CheckboxMenuItem)e.getSource()).getState() )
    { System.out.println( "Printer selected..." );
      lpr.setVisible( true );
    }
    else
    { System.out.println( "Printer deselected..." );
      lpr.setVisible( false );
    }
  }
}

/** Shows/hides the control window
*/
private class ControlItemActionL implements ItemListener
{ public void itemStateChanged( ItemEvent e )
  { if ( ((CheckboxMenuItem)e.getSource()).getState() )
    { System.out.println( "Control selected..." );
      control.setVisible( true );
    }
    else
    { System.out.println( "Control deselected..." );
      control.setVisible( false );
    }
  }
}

/** Shows/hides the clock window
*/
private class ClockItemActionL implements ItemListener
{ public void itemStateChanged( ItemEvent e )
  { if ( ((CheckboxMenuItem)e.getSource()).getState() )
    { System.out.println( "Clock selected..." );
      clock.setVisible( true );
    }
    else
    { System.out.println( "Clock deselected..." );
      clock.setVisible( false );
    }
  }
}

/** Adjusts what section of main memory is displayed according
 * to the position of the memory scroll bar
*/
private class MemoryAdjustL implements AdjustmentListener
{ public void adjustmentValueChanged( AdjustmentEvent e )
  { String zeros;
    int sVal = e.getValue();
    for ( int i = sVal; i < (sVal + 10); i++ )
    { if (i < 10)
      { zeros = "000";
      }
      else if (i < 100)
      { zeros = "00";
      }
      else if (i < 1000)
      { zeros = "0";
      }
      else zeros = "";
      mMOVE( memory[ i ], memoryVisible[ i-sVal ] );
      memoryLabels[ i - sVal ].setText( zeros + i );
    }
    mainPanel.repaint();
  }
}

/** A modal dialogue box with one button to dismiss
*/
class MIXDialog extends Dialog
{ Label dTextL;
  Button dismissDB = new Button( "OK" );
  Panel dButtonPanel = new Panel();
```

```
/** Sole constructor
 * @param dTitle String containing title of dialogue box
 * @param dText String containing text to be displayed
 * @return No return value.
 */
MIXDialog( String dTitle, String dText )
{ super( MIXMachine.this, dTitle, true );
  setLayout( new BorderLayout() );
  setBackground( Color.white );
  dismissDB.setBackground( Color.lightGray );
  dButtonPanel.setLayout( new FlowLayout() );
  dButtonPanel.setBackground( Color.white );
  dButtonPanel.add( dismissDB );
  add( BorderLayout.SOUTH, dButtonPanel );
  dTextL = new Label( dText, Label.CENTER );
  add( BorderLayout.CENTER, dTextL );
  setSize( 300, 100 );
  setResizable( false );
  dismissDB.addActionListener( new ActionListener()
  { public void actionPerformed( ActionEvent e )
    { setVisible( false );
    }
  });
  addWindowListener( new WindowAdapter()
  { public void windowClosing( WindowEvent e )
    { dispose();
    }
  });
  setVisible( true );
}

/** A modal binary-choice dialogue box. Clicking 'OK' will set the variable
 * choiceFlag to true, clicking 'Cancel' will set it to false. This can then
 * be used by other sections of the object to see the outcome of a choice.
 */
class MIXChoiceDialog extends Dialog
{ Label dTextL;
  Button dBOK = new Button("OK"),
    dBCancel = new Button("Cancel");
  Panel dButtonPanel = new Panel();

  /** Sole constructor
   * @param dTitle String containing title of dialogue box
   * @param dText String containing text to be displayed
   * @return No return value.
   */
  MIXChoiceDialog( String dTitle, String dText )
  { super( MIXMachine.this, dTitle, true );
    setLayout( new BorderLayout() );
    setBackground( Color.white );
    dBOK.setBackground( Color.lightGray );
    dBCancel.setBackground( Color.lightGray );
    dButtonPanel.setLayout( new FlowLayout() );
    dButtonPanel.setBackground( Color.white );
    dButtonPanel.add( dBOK );
    dButtonPanel.add( dBCancel );
    add( BorderLayout.SOUTH, dButtonPanel );
    dTextL = new Label( dText, Label.CENTER );
    add( BorderLayout.CENTER, dTextL );
    setSize( 300, 100 );
    setResizable( false );
    choiceFlag = false;
    dBOK.addActionListener( new ActionListener()
    { public void actionPerformed( ActionEvent e )
      { choiceFlag = true;
        setVisible( false );
      }
    });
    dBCancel.addActionListener( new ActionListener()
    { public void actionPerformed( ActionEvent e )
      { setVisible( false );
      }
    });
  }
```

```

    });
    addWindowListener( new WindowAdapter()
    {
        public void windowClosing( WindowEvent e )
        {
            dispose();
        }
    });
    setVisible( true );
}
}

/** ControlConsole is used to start/stop the machine,
 * step through program instructions one by one and adjust
 * the speed.
 */
class ControlConsole extends Frame
{
    private Button[] buttons = { new Button( "Go" ),
                                new Button( "Stop" ),
                                new Button( "Step" ),
                                new Button( "Faster" ),
                                new Button( "Slower" ) };

    /** Sole constructor for the ControlConsole.
     * @return No return value.
     */
    ControlConsole()
    {
        setLayout( new FlowLayout() );
        setBackground( Color.white );
        setTitle( "MIX 1009 Control" );

        for ( int i = 0; i < buttons.length; i++ )
        {
            buttons[ i ].setBackground( Color.lightGray );
            add( buttons[ i ] );
        }

        buttons[ 0 ].addActionListener( new goButtonActionL() );
        buttons[ 2 ].addActionListener( new stepButtonActionL() );
        setResizable( false );
    }

    private class goButtonActionL implements ActionListener
    {
        public void actionPerformed( ActionEvent e )
        {
            System.out.println( "Go selected..." );
            String error = assembleCode(); // Attempt assembly
            if ( !error.equals( "" ) ) // If we have an error...
                System.out.println( error ); // ...report it!
            execute( globalPC, false ); // Then run the code.
        }
    }

    private class stepButtonActionL implements ActionListener
    {
        public void actionPerformed( ActionEvent e )
        {
            globalPC = execute( globalPC, true );
        }
    }
}

/** ProgramLoader is used here to put programs into memory for the
 * purpose of demonstrating MIX features. As it stands, the MIX
 * 1009 can only be 'programmed' by creating inner classes inside
 * this class and placing the instructions in memory directly.
 * In the final version, I do not intend this class to exist, as
 * programs will be loaded by assembling them from their MIXAL
 * source.
 */
class ProgramLoader extends Frame
{
    private Button[] buttons = { new Button( "Primes" ),
                                new Button( "Maximum" ) };

    /** Sole constructor
     * @return No return value
     */
    ProgramLoader()
    {
        setLayout( new FlowLayout() );
        setBackground( Color.white );
    }
}

```

```
setTitle( "MIX 1009 Program Loader" );

for ( int i=0; i<buttons.length; i++ )
{ buttons[ i ].setBackground( Color.lightGray );
  add( buttons[ i ] );
}

buttons[ 0 ].addActionListener( new primesButtonActionL() );
buttons[ 1 ].addActionListener( new programMButtonActionL() );
}

private class primesButtonActionL implements ActionListener
{ public void actionPerformed((ActionEvent e) )
  { clearMemoryContents();

    try
    { memory[ 3000 ].setValue( 4, 35 );
      memory[ 3000 ].setValue( 3, 18 );

      memory[ 3001 ].setValue( 4, 9 );
      memory[ 3001 ].setValue( 3, 5 );
      memory[ 3001 ].setValue( 1, 2050 % 64 );
      memory[ 3001 ].setValue( 0, 2050 / 64 );

      memory[ 3002 ].setValue( 4, 10 );
      memory[ 3002 ].setValue( 3, 5 );
      memory[ 3002 ].setValue( 1, 2051 % 64 );
      memory[ 3002 ].setValue( 0, 2051 / 64 );

      memory[ 3003 ].setValue( 4, 49 );
      memory[ 3003 ].setValue( 1, 1 );

      memory[ 3004 ].setValue( 4, 26 );
      memory[ 3004 ].setValue( 3, 5 );
      memory[ 3004 ].setValue( 2, 1 );
      memory[ 3004 ].setValue( 1, 499 % 64 );
      memory[ 3004 ].setValue( 0, 499 / 64 );

      memory[ 3005 ].setValue( 4, 41 );
      memory[ 3005 ].setValue( 3, 1 );
      memory[ 3005 ].setValue( 1, 3016 % 64 );
      memory[ 3005 ].setValue( 0, 3016 / 64 );

      memory[ 3006 ].setValue( 4, 50 );
      memory[ 3006 ].setValue( 1, 2 );

      memory[ 3007 ].setValue( 4, 51 );
      memory[ 3007 ].setValue( 3, 2 );
      memory[ 3007 ].setValue( 1, 2 );

      memory[ 3008 ].setValue( 4, 48 );
      memory[ 3008 ].setValue( 3, 2 );

      memory[ 3009 ].setValue( 4, 55 );
      memory[ 3009 ].setValue( 3, 2 );
      memory[ 3009 ].setValue( 2, 2 );

      memory[ 3010 ].setValue( 4, 4 );
      memory[ 3010 ].setValue( 3, 5 );
      memory[ 3010 ].setValue( 2, 3 );
      memory[ 3010 ].setValue( 1, 1 );
      memory[ 3010 ].setSign( '-' );

      memory[ 3011 ].setValue( 4, 47 );
      memory[ 3011 ].setValue( 3, 1 );
      memory[ 3011 ].setValue( 1, 3006 % 64 );
      memory[ 3011 ].setValue( 0, 3006 / 64 );

      memory[ 3012 ].setValue( 4, 56 );
      memory[ 3012 ].setValue( 3, 5 );
      memory[ 3012 ].setValue( 2, 3 );
      memory[ 3012 ].setValue( 1, 1 );
      memory[ 3012 ].setSign( '-' );
```

```
memory[ 3013 ].setValue( 4, 51 );
memory[ 3013 ].setValue( 1, 1 );

memory[ 3014 ].setValue( 4, 39 );
memory[ 3014 ].setValue( 3, 6 );
memory[ 3014 ].setValue( 1, 3008 % 64 );
memory[ 3014 ].setValue( 0, 3008 / 64 );

memory[ 3015 ].setValue( 4, 39 );
memory[ 3015 ].setValue( 1, 3003 % 64 );
memory[ 3015 ].setValue( 0, 3003 / 64 );

memory[ 3016 ].setValue( 4, 37 );
memory[ 3016 ].setValue( 3, 18 );
memory[ 3016 ].setValue( 1, 1995 % 64 );
memory[ 3016 ].setValue( 0, 1995 / 64 );

memory[ 3017 ].setValue( 4, 52 );
memory[ 3017 ].setValue( 3, 2 );
memory[ 3017 ].setValue( 1, 2035 % 64 );
memory[ 3017 ].setValue( 0, 2035 / 64 );

memory[ 3018 ].setValue( 4, 53 );
memory[ 3018 ].setValue( 3, 2 );
memory[ 3018 ].setValue( 1, 50 );
memory[ 3018 ].setSign( '-' );

memory[ 3019 ].setValue( 4, 53 );
memory[ 3019 ].setValue( 1, 501 % 64 );
memory[ 3019 ].setValue( 0, 501 / 64 );

memory[ 3020 ].setValue( 4, 8 );
memory[ 3020 ].setValue( 3, 5 );
memory[ 3020 ].setValue( 2, 5 );
memory[ 3020 ].setValue( 1, 1 );
memory[ 3020 ].setSign( '-' );

memory[ 3021 ].setValue( 4, 5 );
memory[ 3021 ].setValue( 3, 1 );

memory[ 3022 ].setValue( 4, 31 );
memory[ 3022 ].setValue( 3, 12 );
memory[ 3022 ].setValue( 2, 4 );

memory[ 3023 ].setValue( 4, 52 );
memory[ 3023 ].setValue( 3, 1 );
memory[ 3023 ].setValue( 1, 1 );

memory[ 3024 ].setValue( 4, 53 );
memory[ 3024 ].setValue( 3, 1 );
memory[ 3024 ].setValue( 1, 50 );

memory[ 3025 ].setValue( 4, 45 );
memory[ 3025 ].setValue( 3, 2 );
memory[ 3025 ].setValue( 1, 3020 % 64 );
memory[ 3025 ].setValue( 0, 3020 / 64 );

memory[ 3026 ].setValue( 4, 37 );
memory[ 3026 ].setValue( 3, 18 );
memory[ 3026 ].setValue( 2, 4 );

memory[ 3027 ].setValue( 4, 12 );
memory[ 3027 ].setValue( 3, 5 );
memory[ 3027 ].setValue( 2, 4 );
memory[ 3027 ].setValue( 1, 24 );

memory[ 3028 ].setValue( 4, 45 );
memory[ 3028 ].setValue( 1, 3019 % 64 );
memory[ 3028 ].setValue( 0, 3019 / 64 );

memory[ 3029 ].setValue( 4, 5 );
memory[ 3029 ].setValue( 3, 2 );

memory[ 0000 ].setValue( 4, 2 );
```

```
memory[ 1995 ].setValue( 4, 23 );
memory[ 1995 ].setValue( 3, 22 );
memory[ 1995 ].setValue( 2, 19 );
memory[ 1995 ].setValue( 1, 9 );
memory[ 1995 ].setValue( 0, 6 );

memory[ 1996 ].setValue( 4, 5 );
memory[ 1996 ].setValue( 3, 25 );
memory[ 1996 ].setValue( 2, 9 );
memory[ 1996 ].setValue( 1, 6 );
memory[ 1996 ].setValue( 0, 0 );

memory[ 1997 ].setValue( 4, 4 );
memory[ 1997 ].setValue( 3, 15 );
memory[ 1997 ].setValue( 2, 24 );
memory[ 1997 ].setValue( 1, 8 );

memory[ 1998 ].setValue( 4, 17 );
memory[ 1998 ].setValue( 2, 4 );
memory[ 1998 ].setValue( 1, 5 );
memory[ 1998 ].setValue( 0, 19 );

memory[ 1999 ].setValue( 4, 22 );
memory[ 1999 ].setValue( 3, 5 );
memory[ 1999 ].setValue( 2, 14 );
memory[ 1999 ].setValue( 1, 9 );
memory[ 1999 ].setValue( 0, 19 );

memory[ 2024 ].setValue( 4, 2035 % 64 );
memory[ 2024 ].setValue( 3, 2035 / 64 );

memory[ 2049 ].setValue( 4, 2010 % 64 );
memory[ 2049 ].setValue( 3, 2010 / 64 );

memory[ 2050 ].setValue( 4, 499 % 64 );
memory[ 2050 ].setValue( 3, 499 / 64 );
memory[ 2050 ].setSign( '-' );

memory[ 2051 ].setValue( 4, 3 );

for( int i=3000; i<3030; i++)
{
    memory[ i ].setPacked( 4, false );
    memory[ i ].setPacked( 3, false );
    memory[ i ].setPacked( 2, false );
}
for (int i=1995; i<2000; i++)
{
    memory[ i ].setPacked( 4, false );
    memory[ i ].setPacked( 3, false );
    memory[ i ].setPacked( 2, false );
    memory[ i ].setPacked( 1, false );
    memory[ i ].setPacked( 0, false );
}

} catch(Exception x) {System.out.println("Exception encountered when loading program!");
    System.out.println( x ); }

memoryScroller.setValue( memoryScroller.getValue() ); // Repaints the memory values.
globalPC = 3000; // Where we should start execution.

MIXDialog message = new MIXDialog("Information", "Program 'primes' loaded successfully");
}
}

private class programMButtonActionL implements ActionListener
{
    long randomLong;

    public void actionPerformed( ActionEvent e )
    {
        clearMemoryContents();

        try
        {
            // Generate 1000 random numbers between 0 and 10000000000

```

```
for (int i=1000; i<2000; i++ )
{
    randomLong = (long)((Math.random())*1000000000);
    memory[ i ].setValue( 4, (int)(randomLong % 64) );
    randomLong = randomLong / 64;
    memory[ i ].setValue( 3, (int)(randomLong % 64) );
    randomLong = randomLong / 64;
    memory[ i ].setValue( 2, (int)(randomLong % 64) );
    randomLong = randomLong / 64;
    memory[ i ].setValue( 1, (int)(randomLong % 64) );
    memory[ i ].setValue( 0, (int)(randomLong / 64) );
}

// Let r11 know there are 1000 elements
iRegisters[ 0 ].setValue( 1, 1000 % 64 );
iRegisters[ 0 ].setValue( 0, 1000 / 64 );

// Main program
memory[ 3000 ].setValue( 4, 32 );
memory[ 3000 ].setValue( 3, 2 );
memory[ 3000 ].setValue( 1, 3009 % 64 );
memory[ 3000 ].setValue( 0, 3009 / 64 );

memory[ 3001 ].setValue( 4, 51 );
memory[ 3001 ].setValue( 3, 2 );
memory[ 3001 ].setValue( 2, 1 );

memory[ 3002 ].setValue( 4, 39 );
memory[ 3002 ].setValue( 1, 3005 % 64 );
memory[ 3002 ].setValue( 0, 3005 / 64 );

memory[ 3003 ].setValue( 4, 56 );
memory[ 3003 ].setValue( 3, 5 );
memory[ 3003 ].setValue( 2, 3 );
memory[ 3003 ].setValue( 1, 1000 % 64 );
memory[ 3003 ].setValue( 0, 1000 / 64 );

memory[ 3004 ].setValue( 4, 39 );
memory[ 3004 ].setValue( 3, 7 );
memory[ 3004 ].setValue( 1, 3007 % 64 );
memory[ 3004 ].setValue( 0, 3007 / 64 );

memory[ 3005 ].setValue( 4, 50 );
memory[ 3005 ].setValue( 3, 2 );
memory[ 3005 ].setValue( 2, 3 );

memory[ 3006 ].setValue( 4, 8 );
memory[ 3006 ].setValue( 3, 5 );
memory[ 3006 ].setValue( 2, 3 );
memory[ 3006 ].setValue( 1, 1000 % 64 );
memory[ 3006 ].setValue( 0, 1000 / 64 );

memory[ 3007 ].setValue( 4, 51 );
memory[ 3007 ].setValue( 3, 1 );
memory[ 3007 ].setValue( 1, 1 );

memory[ 3008 ].setValue( 4, 43 );
memory[ 3008 ].setValue( 3, 2 );
memory[ 3008 ].setValue( 1, 3003 % 64 );
memory[ 3008 ].setValue( 0, 3003 / 64 );

// Code altered - Knuth's infinite loop is changed here to a HLT.
memory[ 3009 ].setValue( 4, 5 );
memory[ 3009 ].setValue( 3, 2 );

for( int i=3000; i<3010; i++)
{
    memory[ i ].setPacked( 4, false );
    memory[ i ].setPacked( 3, false );
    memory[ i ].setPacked( 2, false );
}

// Start execution at location 3000
globalPC = 3000;
} catch (Exception x) { System.out.println("Exception encountered when loading program!");
```

```
        System.out.println( x ); }

        memoryScroller.setValue( memoryScroller.getValue() ); // Repaints the memory values.
        MIXDialog message = new MIXDialog("Information", "Program 'Maximum' loaded successfully");

    }
}

/** Clears the memory of the MIXMachine - zeroes all values and
 * resets all indicators.
 * @return No return value.
 */
private void clearMemoryContents()
{
    try
    {
        for (int i=0; i<4000; i++)
        {
            for (int j=0; j<5; j++)
            {
                memory[ i ].setValue( j, 0 );
                memory[ i ].setPacked( j, true );
            }
            memory[ i ].setSign( '+' );
        }

        for (int i=0; i<5; i++)
        {
            aRegister.setValue( i, 0 );
            xRegister.setValue( i, 0 );
            aRegister.setPacked( i, true );
            xRegister.setPacked( i, true );
        }

        for (int i=0; i<6; i++)
        {
            for (int j=0; j<2; j++)
            {
                iRegisters[ i ].setValue( j, 0 );
                iRegisters[ i ].setPacked( j, true );
            }
            iRegisters[ i ].setSign( '+' );
        }

        jRegister.setValue( 0, 0 );
        jRegister.setValue( 1, 0 );
        jRegister.setPacked( 0, true );
        jRegister.setPacked( 1, true );
        jRegister.setSign( '+' );

        oIndicator.setState( false );
        cIndicator.setState( ComparisonIndicator.OFF );
    } catch (Exception e) { System.out.println("Problem occurred when clearing memory contents.");
        System.out.println( e ); }

    for (int i=0; i<10; i++)
        memoryVisible[ i ].repaint();
    repaint();

    globalPC = 0;
}

/** Takes the code entered into the Input Console and assembles
 * the resulting program in Memory. This takes the form of a
 * two-pass assembler in order to cleanly deal with symbols,
 * local declarations, and literal constants.
 * This method was created using the book "Assemblers, Compilers
 * and Program Translation" by Peter Calingaert, 1979.
 * INCOMPLETE AND NOT FUNCTIONAL AT PRESENT.
 * @return String denoting error encountered during assembly
 */
String assembleCode()
{
    System.out.println( "Assembling..." );
}

/*      String rawProg = program.getProgram(),
        pass1text,
        line,
        word,
        error = "";

    int locCounter = 0;                // Location counter
    int lineCounter = 0;               // Line counter (for debugging)
    int wordCounter = 0;               // Word counter (for assembly
```



```

// error detection) - NEEDED?
StringTokenizer st1 = new StringTokenizer( rawProg, "\n" ), // Take each line in turn
st2;

while ( st1.hasMoreTokens() )
{
    lineCounter++; // New line...increment line
    line = st1.nextToken(); // counter...get the line and...
    st2 = new StringTokenizer( line );
    wordCounter = 0; // ...reset word counter.
    if (line.indexOf(" ") > 0) // Something in the LOC field...
    { word = st2.nextToken(); // ...so we obtain it.
      if ( word.equals("**") ) // If it's a comment line,
        break; // ...move onto the next one.
      if ( word.equals

    st2 = new StringTokenizer( st1.nextToken(), " " ); // Take each word in turn
    while ( st2.hasMoreTokens() )
    { wordCounter++; // New word...increment word
      word = st2.nextToken(); // counter
      if (( wordCounter == 1 ) && ( word.equals("**") )) // If the line starts with a '*'
        break; // just skip it.
      System.out.println( word + " line:" + lineCounter + " word:" + wordCounter );
    }
}
return "";

/** Method to execute the program in memory.
 * @param startLoc int representing start location
 * @param stepped boolean value denoting whether this execution is stepped,
 * and therefore consist of one execution only.
 * @return int value denoting the new position of the program counter.
 */
private int execute( int startLoc, boolean stepped )
{
    System.out.println( "Executing code..." );
    int pc = startLoc, // Set the program counter to the
                        // start location.
        code = 0, // Command identifier code
        field = 0, // Field identifier = 8L + R
        fStart = 0, // L
        fEnd = 0, // R
        fOurStart = 0, // 'Our' start - avoids array problems
        index = 0, // Index code
        addr = 0, // Address code
        power = 0, // Used to calculate contents of packed
                  // bytes
        counter = 0, // Used in LDA and LDX etc.
        scrollValue = 0; // Used to update the display
    boolean halt = false; // Halt flag
    long value; // Contents of a packed (or unpacked)
                // byte
    String rtError = ""; // Holds the runtime error

    do
    {
        try
        {
            code = memory[ pc ].getValue( 4 );
            field = memory[ pc ].getValue( 3 );
            fStart = field / 8;
            fEnd = field % 8;
            if (fStart-1 == -1)
                fOurStart = 0;
            else
                fOurStart = fStart-1;
            index = memory[ pc ].getValue( 2 );
            addr = ( memory[ pc ].getValue( 0 ) * 64 ) + memory[ pc ].getValue( 1 );
            addr = ( memory[ pc ].getSign() == '+' ) ? addr : -addr;
            index--; // Adjust to access array.
            if ( index > 5 )
            {
                rtError = "Bad index value at pc="+pc;
                break;
            }
            if ( index >= 0 )
            {
                addr += (( iRegisters[ index ].getValue(0) * 64 ) + iRegisters[ index ].getValue(1) )

```

```
        * ( (iRegisters[ index ].getSign() == '-') ? -1 : 1 );
    }

    power = 0;
    value = 0;
    switch (code)
    { case 0: pc++; // NOP
      clock.incrementClock( 1 );
      break;

    case 1: for ( int i = (fEnd-1); i>=fOurStart; i-- ) // ADD
      { // WORK OUT VALUE
        value += (long)(Math.pow( 64, power ) * memory[ addr ].getValue( i ));
        power++;
      }

      // SET THE SIGN
      if (fStart == 0)
        value = (memory[ addr ].getSign() == '-') ? -value : value;

      // ADD THE VALUE TO rA
      mADD( aRegister, value );

      // 'PACK' rA THE SAME WAY AS THE WORD THE VALUE CAME FROM
      for ( int i = 0; i<5; i++ )
        aRegister.setPacked( i, memory[ addr ].isPacked( i ) );

      // INCREMENT CLOCK & PROGRAM COUNTER
      pc++;
      clock.incrementClock( 2 );
      break;

    case 2: for ( int i = (fEnd-1); i>=fOurStart; i-- ) // SUB
      { // WORK OUT VALUE
        value += (long)(Math.pow( 64, power ) * memory[ addr ].getValue( i ));
        power++;
      }

      // SET THE SIGN
      if (fStart == 0)
        value = (memory[ addr ].getSign() == '-') ? -value : value;

      // SUBTRACT THE VALUE FROM rA
      value = -value;
      mADD( aRegister, value );

      // 'PACK' rA THE SAME WAY AS THE WORD THE VALUE CAME FROM
      for ( int i = 0; i<5; i++ )
        aRegister.setPacked( i, memory[ addr ].isPacked( i ) );

      // INCREMENT CLOCK & PROGRAM COUNTER
      pc++;
      clock.incrementClock( 2 );
      break;

    case 3: for ( int i = (fEnd-1); i>=fOurStart; i-- ) // MUL
      { // WORK OUT VALUE
        value += (long)(Math.pow( 64, power ) * memory[ addr ].getValue( i ));
        power++;
      }

      // SET THE SIGN
      if (fStart == 0)
        value = (memory[ addr ].getSign() == '-') ? -value : value;

      // MULTIPLY BY rA AND PLACE IN rA AND rX
      mMUL( value );

      // INCREMENT CLOCK & PROGRAM COUNTER
      pc++;
      clock.incrementClock( 10 );
      break;

    case 4: for ( int i = (fEnd-1); i>=fOurStart; i-- ) // DIV
```

```
{ // WORK OUT VALUE
    value += (long)(Math.pow( 64, power ) * memory[ addr ].getValue( i ));
    power++;
}

// SET THE SIGN
if (fStart == 0)
    value = (memory[ addr ].getSign() == '-') ? -value : value;

// PERFORM THE DIVISION AND PLACE IN rA and rX
mDIV( value );

// INCREMENT CLOCK & PROGRAM COUNTER
pc++;
clock.incrementClock( 12 );
break;

case 5: switch (field)
{   case 0 : mNUM();                                // NUM

        // INCREMENT CLOCK & PROGRAM COUNTER
        pc++;
        clock.incrementClock( 10 );
        break;

        case 1 : mCHAR();                            // CHAR

        // INCREMENT CLOCK & PROGRAM COUNTER
        pc++;
        clock.incrementClock( 10 );
        break;

        case 2 : halt = true;                        // HLT

        // INCREMENT CLOCK & PROGRAM COUNTER
        pc++;
        clock.incrementClock( 1 );
        System.out.println( "HLT encountered. Stopping. PC now at "+pc );
        break;
    }
    break;

case 6: // CHECK SIGN                                // shift operators
    if (addr<0)
    {   rtError = "SLA operand cannot be negative at pc="+pc;
        break;
    }

    switch (field)
    {   case 0 : mSLA( (long)addr );                  // SLA
        break;

        case 1 : mSRA( (long)addr );                  // SRA
        break;

        case 2 : mSLAX( (long)addr, false );          // SLAX
        break;

        case 3 : mSRAX( (long)addr, false );          // SRAX
        break;

        case 4 : mSLAX( (long)addr, true );           // SLC
        break;

        case 5 : mSRAX( (long)addr, true );           // SRC
        break;
    }

    // INCREMENT CLOCK & PROGRAM COUNTER
    pc++;
    clock.incrementClock( 2 );
    break;

case 7: if (field == 0)                                // MOVE
```

```
        break;

int startMove = ((iRegisters[0].getValue(0))*64)+
    (iRegisters[0].getValue(1));
scrollValue = memoryScroller.getValue();

for (int i = 0; i<field; i++)
{ mMOVE( memory[ addr+i ], memory[ startMove+i ] );
  if (((startMove+i)>=(scrollValue)) && ((startMove+i)<=(scrollValue+9)))
    mMOVE( memory[ startMove+i ],
        memoryVisible[ (startMove+i) - (scrollValue) ] );
}

// INCREMENT I1 BY FIELD
startMove += field;
iRegisters[0].setValue( 0, startMove / 64 );
iRegisters[0].setValue( 1, startMove % 64 );

// INCREMENT CLOCK & PROGRAM COUNTER
pc++;
clock.incrementClock( 1+field );
break;

case 8: if (fStart == 0)                                     // LDA
        aRegister.setSign( memory[ addr ].getSign() );
    else
        aRegister.setSign( '+' );

    counter = 4;

    for (int i=(fEnd-1); i>=fOurStart; i--)
    { aRegister.setValue( counter, memory[ addr ].getValue( i ) );
      aRegister.setPacked( counter, memory[ addr ].isPacked( i ) );
      counter--;
    }
    for (int i=counter; i>=0; i--)
    { aRegister.setValue( i, 0 );
      aRegister.setPacked( i, false );
    }

    // INCREMENT CLOCK & PROGRAM COUNTER
    pc++;
    clock.incrementClock( 2 );
    break;

case 9:                                     // LD1
    if (((fEnd-fOurStart)+1)>2)
        for (int i=(fEnd-3); i>=(fOurStart); i--)
            if (memory[ addr ].getValue( i ) != 0 )
                { rtError = "I Register 1 can only be loaded with two bytes at pc="
                    +pc;
                }

    if (!(rtError.equals("")))
        break;

    mLDi( 0, memory[ addr ], fEnd-2, fEnd-1, (fStart==0) );

    // INCREMENT CLOCK & PROGRAM COUNTER
    pc++;
    clock.incrementClock( 2 );
    break;

case 10:                                     // LD2
    if (((fEnd-fOurStart)+1)>2)
        for (int i=(fEnd-3); i>=(fOurStart); i--)
            if (memory[ addr ].getValue( i ) != 0 )
                { rtError = "I Register 2 can only be loaded with two bytes at pc="
                    +pc;
                }

    if (!(rtError.equals("")))
        break;
```

```
mLDi( 1, memory[ addr ], fEnd-2, fEnd-1, (fStart==0) );

// INCREMENT CLOCK & PROGRAM COUNTER
pc++;
clock.incrementClock( 2 );
break;

case 11: // LD3
    if (((fEnd-fOurStart)+1)>2)
        for (int i=(fEnd-3); i>=(fOurStart); i--)
            if (memory[ addr ].getValue( i ) != 0 )
                { rtError = "I Register 3 can only be loaded with two bytes at pc="
                    +pc;
                }

    if (!(rtError.equals(""))))
        break;

    mLDi( 2, memory[ addr ], fEnd-2, fEnd-1, (fStart==0) );

    // INCREMENT CLOCK & PROGRAM COUNTER
    pc++;
    clock.incrementClock( 2 );
    break;

case 12: // LD4
    if (((fEnd-fOurStart)+1)>2)
        for (int i=(fEnd-3); i>=(fOurStart); i--)
            if (memory[ addr ].getValue( i ) != 0 )
                { rtError = "I Register 4 can only be loaded with two bytes at pc="
                    +pc;
                }

    if (!(rtError.equals(""))))
        break;

    mLDi( 3, memory[ addr ], fEnd-2, fEnd-1, (fStart==0) );

    // INCREMENT CLOCK & PROGRAM COUNTER
    pc++;
    clock.incrementClock( 2 );
    break;

case 13: // LD5
    if (((fEnd-fOurStart)+1)>2)
        for (int i=(fEnd-3); i>=(fOurStart); i--)
            if (memory[ addr ].getValue( i ) != 0 )
                { rtError = "I Register 5 can only be loaded with two bytes at pc="
                    +pc;
                }

    if (!(rtError.equals(""))))
        break;

    mLDi( 4, memory[ addr ], fEnd-2, fEnd-1, (fStart==0) );

    // INCREMENT CLOCK & PROGRAM COUNTER
    pc++;
    clock.incrementClock( 2 );
    break;

case 14: // LD6
    if (((fEnd-fOurStart)+1)>2)
        for (int i=(fEnd-3); i>=(fOurStart); i--)
            if (memory[ addr ].getValue( i ) != 0 )
                { rtError = "I Register 6 can only be loaded with two bytes at pc="
                    +pc;
                }

    if (!(rtError.equals(""))))
        break;

    mLDi( 5, memory[ addr ], fEnd-2, fEnd-1, (fStart==0) );
```

```
// INCREMENT CLOCK & PROGRAM COUNTER
pc++;
clock.incrementClock( 2 );
break;

case 15: if (fStart == 0) // LDX
        xRegister.setSign( memory[ addr ].getSign() );
    else
        xRegister.setSign( '+' );

    counter = 4;

    for (int i=(fEnd-1); i>=fOurStart; i--)
    { xRegister.setValue( counter, memory[ addr ].getValue( i ) );
      xRegister.setPacked( counter, memory[ addr ].isPacked( i ) );
      counter--;
    }
    for (int i=counter; i>=0; i--)
    { xRegister.setValue( i, 0 );
      xRegister.setPacked( i, false );
    }

    // INCREMENT CLOCK & PROGRAM COUNTER
    pc++;
    clock.incrementClock( 2 );
    break;

case 16: if (fStart == 0) // LDAN
        { if (memory[ addr ].getSign() == '-')
            aRegister.setSign( '-' );
          else
            aRegister.setSign( '+' );
        }
    else
        aRegister.setSign( '-' );

    counter = 4;

    for (int i=(fEnd-1); i>=fOurStart; i--)
    { aRegister.setValue( counter, memory[ addr ].getValue( i ) );
      aRegister.setPacked( counter, memory[ addr ].isPacked( i ) );
      counter--;
    }
    for (int i=counter; i>=0; i--)
    { aRegister.setValue( i, 0 );
      aRegister.setPacked( i, false );
    }

    // INCREMENT CLOCK & PROGRAM COUNTER
    pc++;
    clock.incrementClock( 2 );
    break;

case 17: // LD1N
    if ( ((fEnd>(fStart+1)) && (fStart > 0)) || ((fStart == 0) && (fEnd>2)) )
    { rtError = "I Register 1 can only be loaded with two bytes at pc="+pc;
      break;
    }

    mLDiN( 0, memory[ addr ], fOurStart, fEnd, (fStart==0) );

    // INCREMENT CLOCK & PROGRAM COUNTER
    pc++;
    clock.incrementClock( 2 );
    break;

case 18: // LD2N
    if ( ((fEnd>(fStart+1)) && (fStart > 0)) || ((fStart == 0) && (fEnd>2)) )
    { rtError = "I Register 2 can only be loaded with two bytes at pc="+pc;
      break;
    }

    mLDiN( 1, memory[ addr ], fOurStart, fEnd, (fStart==0) );
```

```
// INCREMENT CLOCK & PROGRAM COUNTER
pc++;
clock.incrementClock( 2 );
break;

case 19: // LD3N
if ( ((fEnd>(fStart+1)) && (fStart > 0)) || ((fStart == 0) && (fEnd>2)) )
{ rtError = "I Register 3 can only be loaded with two bytes at pc="+pc;
  break;
}

mLDiN( 2, memory[ addr ], fOurStart, fEnd, (fStart==0) );

// INCREMENT CLOCK & PROGRAM COUNTER
pc++;
clock.incrementClock( 2 );
break;

case 20: // LD4N
if ( ((fEnd>(fStart+1)) && (fStart > 0)) || ((fStart == 0) && (fEnd>2)) )
{ rtError = "I Register 4 can only be loaded with two bytes at pc="+pc;
  break;
}

mLDiN( 3, memory[ addr ], fOurStart, fEnd, (fStart==0) );

// INCREMENT CLOCK & PROGRAM COUNTER
pc++;
clock.incrementClock( 2 );
break;

case 21: // LD5N
if ( ((fEnd>(fStart+1)) && (fStart > 0)) || ((fStart == 0) && (fEnd>2)) )
{ rtError = "I Register 5 can only be loaded with two bytes at pc="+pc;
  break;
}

mLDiN( 4, memory[ addr ], fOurStart, fEnd, (fStart==0) );

// INCREMENT CLOCK & PROGRAM COUNTER
pc++;
clock.incrementClock( 2 );
break;

case 22: // LD6N
if ( ((fEnd>(fStart+1)) && (fStart > 0)) || ((fStart == 0) && (fEnd>2)) )
{ rtError = "I Register 6 can only be loaded with two bytes at pc="+pc;
  break;
}

mLDiN( 5, memory[ addr ], fOurStart, fEnd, (fStart==0) );

// INCREMENT CLOCK & PROGRAM COUNTER
pc++;
clock.incrementClock( 2 );
break;

case 23: if (fStart == 0) // LDXN
{ if (memory[ addr ].getSign() == '+')
  xRegister.setSign( '-' );
  else
  xRegister.setSign( '+' );
}
else
  xRegister.setSign( '-' );

counter = 4;

for (int i=(fEnd-1); i>=fOurStart; i--)
{ xRegister.setValue( counter, memory[ addr ].getValue( i ) );
  xRegister.setPacked( counter, memory[ addr ].isPacked( i ) );
  counter--;
}
for (int i=counter; i>=0; i--)
```

```
{ xRegister.setValue( i, 0 );
  xRegister.setPacked( i, false );
}

// INCREMENT CLOCK & PROGRAM COUNTER
pc++;
clock.incrementClock( 2 );
break;

case 24: if (fStart == 0) // STA
        memory[ addr ].setSign( aRegister.getSign() );

        counter = 4;
        scrollValue = memoryScroller.getValue();

        for (int i=(fEnd-1); i>=fOurStart; i--)
        { memory[ addr ].setValue( i, aRegister.getValue(counter) );
          memory[ addr ].setPacked( i, aRegister.isPacked(counter) );
          counter--;
        }
        if (((addr)>=(scrollValue)) && ((addr)<=(scrollValue+9)))
            mMOVE( memory[ addr ], memoryVisible[ addr - scrollValue ] );

        // INCREMENT CLOCK & PROGRAM COUNTER
        pc++;
        clock.incrementClock( 2 );
        break;

case 25: if (fStart == 0) // ST1
        memory[ addr ].setSign( iRegisters[0].getSign() );

        counter = 1;
        scrollValue = memoryScroller.getValue();

        for (int i=(fEnd-1); i>=fOurStart; i--)
        { memory[ addr ].setValue( i, (counter < 0) ? 0 :
                                     iRegisters[0].getValue( counter ) );
          memory[ addr ].setPacked( i, (counter < 0) ? false :
                                     iRegisters[0].isPacked( counter ) );
          counter--;
        }
        if (((addr)>=(scrollValue)) && ((addr)<=(scrollValue+9)))
            mMOVE( memory[ addr ], memoryVisible[ addr - scrollValue ] );

        // INCREMENT CLOCK & PROGRAM COUNTER
        pc++;
        clock.incrementClock( 2 );
        break;

case 26: if (fStart == 0) // ST2
        memory[ addr ].setSign( iRegisters[1].getSign() );

        counter = 1;
        scrollValue = memoryScroller.getValue();

        for (int i=(fEnd-1); i>=fOurStart; i--)
        { memory[ addr ].setValue( i, (counter < 0) ? 0 :
                                     iRegisters[1].getValue( counter ) );
          memory[ addr ].setPacked( i, (counter < 0) ? false :
                                     iRegisters[1].isPacked( counter ) );
          counter--;
        }
        if (((addr)>=(scrollValue)) && ((addr)<=(scrollValue+9)))
            mMOVE( memory[ addr ], memoryVisible[ addr - scrollValue ] );

        // INCREMENT CLOCK & PROGRAM COUNTER
        pc++;
        clock.incrementClock( 2 );
        break;

case 27: if (fStart == 0) // ST3
        memory[ addr ].setSign( iRegisters[2].getSign() );

        counter = 1;
```



```
scrollValue = memoryScroller.getValue();

for (int i=(fEnd-1); i>=fOurStart; i--)
{ memory[ addr ].setValue( i, (counter < 0) ? 0 :
                             iRegisters[2].getValue( counter ) );
  memory[ addr ].setPacked( i, (counter < 0) ? false :
                             iRegisters[2].isPacked( counter ) );
  counter--;
}
if (((addr)>=(scrollValue)) && ((addr)<=(scrollValue+9)))
  mMOVE( memory[ addr ], memoryVisible[ addr - scrollValue ] );

// INCREMENT CLOCK & PROGRAM COUNTER
pc++;
clock.incrementClock( 2 );
break;

case 28: if (fStart == 0) // ST4
        memory[ addr ].setSign( iRegisters[3].getSign() );

        counter = 1;
        scrollValue = memoryScroller.getValue();

        for (int i=(fEnd-1); i>=fOurStart; i--)
        { memory[ addr ].setValue( i, (counter < 0) ? 0 :
                                     iRegisters[3].getValue( counter ) );
          memory[ addr ].setPacked( i, (counter < 0) ? false :
                                     iRegisters[3].isPacked( counter ) );
          counter--;
        }
        if (((addr)>=(scrollValue)) && ((addr)<=(scrollValue+9)))
          mMOVE( memory[ addr ], memoryVisible[ addr - scrollValue ] );

        // INCREMENT CLOCK & PROGRAM COUNTER
        pc++;
        clock.incrementClock( 2 );
        break;

case 29: if (fStart == 0) // ST5
        memory[ addr ].setSign( iRegisters[4].getSign() );

        counter = 1;
        scrollValue = memoryScroller.getValue();

        for (int i=(fEnd-1); i>=fOurStart; i--)
        { memory[ addr ].setValue( i, (counter < 0) ? 0 :
                                     iRegisters[4].getValue( counter ) );
          memory[ addr ].setPacked( i, (counter < 0) ? false :
                                     iRegisters[4].isPacked( counter ) );
          counter--;
        }
        if (((addr)>=(scrollValue)) && ((addr)<=(scrollValue+9)))
          mMOVE( memory[ addr ], memoryVisible[ addr - scrollValue ] );

        // INCREMENT CLOCK & PROGRAM COUNTER
        pc++;
        clock.incrementClock( 2 );
        break;

case 30: if (fStart == 0) // ST6
        memory[ addr ].setSign( iRegisters[5].getSign() );

        counter = 1;
        scrollValue = memoryScroller.getValue();

        for (int i=(fEnd-1); i>=fOurStart; i--)
        { memory[ addr ].setValue( i, (counter < 0) ? 0 :
                                     iRegisters[5].getValue( counter ) );
          memory[ addr ].setPacked( i, (counter < 0) ? false :
                                     iRegisters[5].isPacked( counter ) );
          counter--;
        }
        if (((addr)>=(scrollValue)) && ((addr)<=(scrollValue+10)))
          mMOVE( memory[ addr ], memoryVisible[ addr - scrollValue ] );
```

---

```
// INCREMENT CLOCK & PROGRAM COUNTER
pc++;
clock.incrementClock( 2 );
break;

case 31: if (fStart == 0) // STX
        memory[ addr ].setSign( xRegister.getSign() );

        counter = 4;
        scrollValue = memoryScroller.getValue();

        for (int i=(fEnd-1); i>=fOurStart; i--)
        { memory[ addr ].setValue( i, xRegister.getValue(counter) );
          memory[ addr ].setPacked( i, xRegister.isPacked(counter) );
          counter--;
        }
        if ((addr)>=(scrollValue)) && ((addr)<=(scrollValue+9))
            mMOVE( memory[ addr ], memoryVisible[ addr - scrollValue ] );

        // INCREMENT CLOCK & PROGRAM COUNTER
        pc++;
        clock.incrementClock( 2 );
        break;

case 32: if (fStart == 0) // STJ
        memory[ addr ].setSign( '+' );

        counter = 1;
        scrollValue = memoryScroller.getValue();

        for (int i=(fEnd-1); i>=fOurStart; i--)
        { memory[ addr ].setValue( i, (counter < 0) ? 0 :
                                     jRegister.getValue( counter ) );
          memory[ addr ].setPacked( i, (counter < 0) ? false :
                                     jRegister.isPacked( counter ) );
          counter--;
        }
        if ((addr)>=(scrollValue)) && ((addr)<=(scrollValue+10))
            mMOVE( memory[ addr ], memoryVisible[ addr - scrollValue ] );

        // INCREMENT CLOCK & PROGRAM COUNTER
        pc++;
        clock.incrementClock( 2 );
        break;

case 33: if (fStart == 0) // STZ
        memory[ addr ].setSign( aRegister.getSign() );

        counter = 4;
        scrollValue = memoryScroller.getValue();

        for (int i=(fEnd-1); i>=fOurStart; i--)
        { memory[ addr ].setValue( i, 0 );
          memory[ addr ].setPacked( i, aRegister.isPacked(counter) );
          counter--;
        }
        if ((addr)>=(scrollValue)) && ((addr)<=(scrollValue+10))
            mMOVE( memory[ addr ], memoryVisible[ addr - scrollValue ] );

        // INCREMENT CLOCK & PROGRAM COUNTER
        pc++;
        clock.incrementClock( 2 );
        break;

case 34: if ((field < 15) || ((field > 17) && (field < 21))) // JBUS
        pc++; // If we have the peripheral, it's not going to be busy!
        else
        { jRegister.setValue( 0, (pc+1) / 64 );
          jRegister.setValue( 1, (pc+1) % 64 );
          pc = addr;
        }

        // INCREMENT CLOCK
```

```
        clock.incrementClock( 1 );
        break;

case 35: pc++;                                // IOC
        break;
        // THIS NEEDS IMPLEMENTING WHEN WE DO THE PERIPHERALS.

case 36: pc++;                                // IN
        break;
        // THIS NEEDS IMPLEMENTING WHEN WE DO THE PERIPHERALS.

case 37: if (field == 18)                      // OUT
        { if (!(addr==0))                      // ** Doesn't clear page **
          { for (int i=0; i<24; i++)
            { lpr.print( memory[ addr + i ] );
              clock.incrementClock( 1 );
            }
          }
        pc++;
        break;
        // THIS NEEDS IMPLEMENTING WHEN WE DO THE PERIPHERALS.

case 38: if ((field < 15) || ((field > 17) && (field < 21))) // JRED
        { jRegister.setValue( 0, (pc+1) / 64 );
          jRegister.setValue( 1, (pc+1) % 64 );
          pc = addr;
        }
        else
            pc++; // If we have the peripheral, it's ready!

        // INCREMENT CLOCK
        clock.incrementClock( 1 );
        break;

case 39: switch(field)
        { case 0 : jRegister.setValue( 0, (pc+1) / 64 ); // JMP
              jRegister.setValue( 1, (pc+1) % 64 );
              pc = addr;

              // INCREMENT CLOCK
              clock.incrementClock( 1 );
              break;

          case 1 : pc = addr; // JSJ

              // INCREMENT CLOCK
              clock.incrementClock( 1 );
              break;

          case 2 : if (oIndicator.getState()) // JOV
                    { jRegister.setValue( 0, (pc+1) / 64 );
                      jRegister.setValue( 1, (pc+1) % 64 );
                      pc = addr;
                      oIndicator.setState( false );
                    }
                    else
                        pc++;

                    // INCREMENT CLOCK
                    clock.incrementClock( 1 );
                    break;

          case 3 : if (oIndicator.getState()) // JNOV
                    { pc++;
                      oIndicator.setState( false );
                    }
                    else
                    { jRegister.setValue( 0, (pc+1) / 64 );
                      jRegister.setValue( 0, (pc+1) % 64 );
                      pc = addr;
                    }

                    // INCREMENT CLOCK
                    clock.incrementClock( 1 );
```

```
        break;

case 4 :                                     // JL
    if (cIndicator.getState() == ComparisonIndicator.LESSTHAN)
    { jRegister.setValue( 0, (pc+1) / 64 );
      jRegister.setValue( 1, (pc+1) % 64 );
      pc = addr;
    }
    else
        pc++;

    // INCREMENT CLOCK
    clock.incrementClock( 1 );
    break;

case 5 :                                     // JE
    if (cIndicator.getState() == ComparisonIndicator.EQUALTO)
    { jRegister.setValue( 0, (pc+1) / 64 );
      jRegister.setValue( 1, (pc+1) % 64 );
      pc = addr;
    }
    else
        pc++;

    // INCREMENT CLOCK
    clock.incrementClock( 1 );
    break;

case 6 :                                     // JG
    if (cIndicator.getState() == ComparisonIndicator.GREATERTHAN)
    { jRegister.setValue( 0, (pc+1) / 64 );
      jRegister.setValue( 1, (pc+1) % 64 );
      pc = addr;
    }
    else
        pc++;

    // INCREMENT CLOCK
    clock.incrementClock( 1 );
    break;

case 7 :                                     // JGE
    if ((cIndicator.getState() == ComparisonIndicator.GREATERTHAN)
        || (cIndicator.getState() == ComparisonIndicator.EQUALTO))
    { jRegister.setValue( 0, (pc+1) / 64 );
      jRegister.setValue( 1, (pc+1) % 64 );
      pc = addr;
    }
    else
        pc++;

    // INCREMENT CLOCK
    clock.incrementClock( 1 );
    break;

case 8 :                                     // JNE
    if ((cIndicator.getState() == ComparisonIndicator.LESSTHAN)
        || (cIndicator.getState() ==
            ComparisonIndicator.GREATERTHAN))
    { jRegister.setValue( 0, (pc+1) / 64 );
      jRegister.setValue( 1, (pc+1) % 64 );
      pc = addr;
    }
    else
        pc++;

    // INCREMENT CLOCK
    clock.incrementClock( 1 );
    break;

case 9 :                                     // JLE
    if ((cIndicator.getState() == ComparisonIndicator.LESSTHAN)
        || (cIndicator.getState() == ComparisonIndicator.EQUALTO))
    { jRegister.setValue( 0, (pc+1) / 64 );
```

```

        jRegister.setValue( 1, (pc+1) / 64 );
        pc = addr;
    }
    else
        pc++;

    // INCREMENT CLOCK
    clock.incrementClock( 1 );
    break;
}
break;

case 40: switch(field)
{ case 0 : for (int i=4; i>=0; i--)                                // JAN
    { // WORK OUT VALUE
        value += (long)(Math.pow( 64, power ) *
            aRegister.getValue( i ));
        power++;
    }

    if ((aRegister.getSign() == '-') && (value > 0 ))
    { jRegister.setValue( 0, (pc+1) / 64 );
      jRegister.setValue( 1, (pc+1) % 64 );
      pc = addr;
    }
    else
        pc++;

    // INCREMENT CLOCK
    clock.incrementClock( 1 );
    break;

    case 1 : for (int i=4; i>=0; i--)                                // JAZ
    { // WORK OUT VALUE
        value += (long)(Math.pow( 64, power ) *
            aRegister.getValue( i ));
        power++;
    }

    if (value == 0)
    { jRegister.setValue( 0, (pc+1) / 64 );
      jRegister.setValue( 1, (pc+1) % 64 );
      pc=addr;
    }
    else
        pc++;

    // INCREMENT CLOCK
    clock.incrementClock( 1 );
    break;

    case 2 : for (int i=4; i>=0; i--)                                // JAP
    { // WORK OUT VALUE
        value += (long)(Math.pow( 64, power ) *
            aRegister.getValue( i ));
        power++;
    }

    if ((aRegister.getSign() == '+') && (value > 0))
    { jRegister.setValue( 0, (pc+1) / 64 );
      jRegister.setValue( 1, (pc+1) % 64 );
      pc=addr;
    }
    else
        pc++;

    // INCREMENT CLOCK
    clock.incrementClock( 1 );
    break;

    case 3 : for (int i=4; i>=0; i--)                                // JANN
    { // WORK OUT VALUE
        value += (long)(Math.pow( 64, power ) *
            aRegister.getValue( i ));

```

```
        power++;
    }

    if ((aRegister.getSign() != '-' ) || (value == 0))
    {
        jRegister.setValue( 0, (pc+1) / 64 );
        jRegister.setValue( 1, (pc+1) % 64 );
        pc=addr;
    }
    else
        pc++;

    // INCREMENT CLOCK
    clock.incrementClock( 1 );
    break;

case 4 : for (int i=4; i>=0; i--)                                // JANZ
    {
        // WORK OUT VALUE
        value += (long)(Math.pow( 64, power ) *
            aRegister.getValue( i ));
        power++;
    }

    if (value != 0 )
    {
        jRegister.setValue( 0, (pc+1) / 64 );
        jRegister.setValue( 1, (pc+1) % 64 );
        pc=addr;
    }
    else
        pc++;

    // INCREMENT CLOCK
    clock.incrementClock( 1 );
    break;

case 5 : for (int i=4; i>=0; i--)                                // JANP
    {
        // WORK OUT VALUE
        value += (long)(Math.pow( 64, power ) *
            aRegister.getValue( i ));
        power++;
    }

    if ((aRegister.getSign() != '+' ) || (value == 0))
    {
        jRegister.setValue( 0, (pc+1) / 64 );
        jRegister.setValue( 1, (pc+1) % 64 );
        pc=addr;
    }
    else
        pc++;

    // INCREMENT CLOCK
    clock.incrementClock( 1 );
    break;
}
break;

case 41: switch(field)
    {
        case 0 : for (int i=1; i>=0; i--)                        // J1N
            {
                // WORK OUT VALUE
                value += (long)(Math.pow( 64, power ) *
                    iRegisters[0].getValue( i ));
                power++;
            }

            if ((iRegisters[0].getSign() == '-' ) && (value > 0))
            {
                jRegister.setValue( 0, (pc+1) / 64 );
                jRegister.setValue( 1, (pc+1) % 64 );
                pc=addr;
            }
            else
                pc++;

            // INCREMENT CLOCK
            clock.incrementClock( 1 );
            break;
    }
}
```

```
case 1 : for (int i=1; i>=0; i--)                                // J1Z
{ // WORK OUT VALUE
  value += (long)(Math.pow( 64, power ) *
    iRegisters[0].getValue( i ));
  power++;
}

if (value == 0)
{ jRegister.setValue( 0, (pc+1) / 64 );
  jRegister.setValue( 1, (pc+1) % 64 );
  pc=addr;
}
else
  pc++;

// INCREMENT CLOCK
clock.incrementClock( 1 );
break;

case 2 : for (int i=1; i>=0; i--)                                // J1P
{ // WORK OUT VALUE
  value += (long)(Math.pow( 64, power ) *
    iRegisters[0].getValue( i ));
  power++;
}

if ((iRegisters[0].getSign() == '+') && (value > 0))
{ jRegister.setValue( 0, (pc+1) / 64 );
  jRegister.setValue( 1, (pc+1) % 64 );
  pc=addr;
}
else
  pc++;

// INCREMENT CLOCK
clock.incrementClock( 1 );
break;

case 3 : for (int i=1; i>=0; i--)                                // J1NN
{ // WORK OUT VALUE
  value += (long)(Math.pow( 64, power ) *
    iRegisters[0].getValue( i ));
  power++;
}

if ((iRegisters[0].getSign() != '-' ) || (value == 0))
{ jRegister.setValue( 0, (pc+1) / 64 );
  jRegister.setValue( 1, (pc+1) % 64 );
  pc=addr;
}
else
  pc++;

// INCREMENT CLOCK
clock.incrementClock( 1 );
break;

case 4 : for (int i=1; i>=0; i--)                                // J1NZ
{ // WORK OUT VALUE
  value += (long)(Math.pow( 64, power ) *
    iRegisters[0].getValue( i ));
  power++;
}

if (value != 0 )
{ jRegister.setValue( 0, (pc+1) / 64 );
  jRegister.setValue( 1, (pc+1) % 64 );
  pc=addr;
}
else
  pc++;

// INCREMENT CLOCK
```

---

```
        clock.incrementClock( 1 );
        break;

    case 5 : for (int i=1; i>=0; i--) // J1NP
    { // WORK OUT VALUE
        value += (long)(Math.pow( 64, power ) *
            iRegisters[0].getValue( i ));
        power++;
    }

    if ((iRegisters[0].getSign() != '+' ) || (value == 0))
    { jRegister.setValue( 0, (pc+1) / 64 );
      jRegister.setValue( 1, (pc+1) % 64 );
      pc=addr;
    }
    else
        pc++;

    // INCREMENT CLOCK
    clock.incrementClock( 1 );
    break;
}
break;

case 42: switch(field)
{ case 0 : for (int i=1; i>=0; i--) // J2N
    { // WORK OUT VALUE
        value += (long)(Math.pow( 64, power ) *
            iRegisters[1].getValue( i ));
        power++;
    }

    if ((iRegisters[1].getSign() == '-' ) && (value > 0))
    { jRegister.setValue( 0, (pc+1) / 64 );
      jRegister.setValue( 1, (pc+1) % 64 );
      pc=addr;
    }
    else
        pc++;

    // INCREMENT CLOCK
    clock.incrementClock( 1 );
    break;

    case 1 : for (int i=1; i>=0; i--) // J2Z
    { // WORK OUT VALUE
        value += (long)(Math.pow( 64, power ) *
            iRegisters[1].getValue( i ));
        power++;
    }

    if (value == 0)
    { jRegister.setValue( 0, (pc+1) / 64 );
      jRegister.setValue( 1, (pc+1) % 64 );
      pc=addr;
    }
    else
        pc++;

    // INCREMENT CLOCK
    clock.incrementClock( 1 );
    break;

    case 2 : for (int i=1; i>=0; i--) // J2P
    { // WORK OUT VALUE
        value += (long)(Math.pow( 64, power ) *
            iRegisters[1].getValue( i ));
        power++;
    }

    if ((iRegisters[1].getSign() == '+' ) && (value > 0))
    { jRegister.setValue( 0, (pc+1) / 64 );
      jRegister.setValue( 1, (pc+1) % 64 );
      pc=addr;
    }
}
```



```
    }
    else
        pc++;

    // INCREMENT CLOCK
    clock.incrementClock( 1 );
    break;

case 3 : for (int i=1; i>=0; i--) // J2NN
    { // WORK OUT VALUE
        value += (long)(Math.pow( 64, power ) *
            iRegisters[1].getValue( i ));
        power++;
    }

    if ((iRegisters[1].getSign() != '-' ) || (value == 0))
    { jRegister.setValue( 0, (pc+1) / 64 );
      jRegister.setValue( 1, (pc+1) % 64 );
      pc=addr;
    }
    else
        pc++;

    // INCREMENT CLOCK
    clock.incrementClock( 1 );
    break;

case 4 : for (int i=1; i>=0; i--) // J2NZ
    { // WORK OUT VALUE
        value += (long)(Math.pow( 64, power ) *
            iRegisters[1].getValue( i ));
        power++;
    }

    if (value != 0 )
    { jRegister.setValue( 0, (pc+1) / 64 );
      jRegister.setValue( 1, (pc+1) % 64 );
      pc=addr;
    }
    else
        pc++;

    // INCREMENT CLOCK
    clock.incrementClock( 1 );
    break;

case 5 : for (int i=1; i>=0; i--) // J2NP
    { // WORK OUT VALUE
        value += (long)(Math.pow( 64, power ) *
            iRegisters[1].getValue( i ));
        power++;
    }

    if ((iRegisters[1].getSign() != '+' ) || (value == 0))
    { jRegister.setValue( 0, (pc+1) / 64 );
      jRegister.setValue( 1, (pc+1) % 64 );
      pc=addr;
    }
    else
        pc++;

    // INCREMENT CLOCK
    clock.incrementClock( 1 );
    break;
}
break;

case 43: switch(field)
    { case 0 : for (int i=1; i>=0; i--) // J3N
        { // WORK OUT VALUE
            value += (long)(Math.pow( 64, power ) *
                iRegisters[2].getValue( i ));
            power++;
        }
    }
```

```
        if ((iRegisters[2].getSign() == '-') && (value > 0))
        {
            jRegister.setValue( 0, (pc+1) / 64 );
            jRegister.setValue( 1, (pc+1) % 64 );
            pc=addr;
        }
        else
            pc++;

        // INCREMENT CLOCK
        clock.incrementClock( 1 );
        break;

case 1 : for (int i=1; i>=0; i--)                                // J3Z
        {
            // WORK OUT VALUE
            value += (long)(Math.pow( 64, power ) *
                            iRegisters[2].getValue( i ));
            power++;
        }

        if (value == 0)
        {
            jRegister.setValue( 0, (pc+1) / 64 );
            jRegister.setValue( 1, (pc+1) % 64 );
            pc=addr;
        }
        else
            pc++;

        // INCREMENT CLOCK
        clock.incrementClock( 1 );
        break;

case 2 : for (int i=1; i>=0; i--)                                // J3P
        {
            // WORK OUT VALUE
            value += (long)(Math.pow( 64, power ) *
                            iRegisters[2].getValue( i ));
        }

        if ((iRegisters[2].getSign() == '+') && (value > 0))
        {
            jRegister.setValue( 0, (pc+1) / 64 );
            jRegister.setValue( 1, (pc+1) % 64 );
            pc=addr;
        }
        else
            pc++;

        // INCREMENT CLOCK
        clock.incrementClock( 1 );
        break;

case 3 : for (int i=1; i>=0; i--)                                // J3NN
        {
            // WORK OUT VALUE
            value += (long)(Math.pow( 64, power ) *
                            iRegisters[2].getValue( i ));
            power++;
        }

        if ((iRegisters[2].getSign() != '-' ) || (value == 0))
        {
            jRegister.setValue( 0, (pc+1) / 64 );
            jRegister.setValue( 1, (pc+1) % 64 );
            pc=addr;
        }
        else
            pc++;

        // INCREMENT CLOCK
        clock.incrementClock( 1 );
        break;

case 4 : for (int i=1; i>=0; i--)                                // J3NZ
        {
            // WORK OUT VALUE
            value += (long)(Math.pow( 64, power ) *
                            iRegisters[2].getValue( i ));
            power++;
        }
```

```
    }

    if (value != 0)
    { jRegister.setValue( 0, (pc+1) / 64 );
      jRegister.setValue( 1, (pc+1) % 64 );
      pc=addr;
    }
    else
      pc++;

    // INCREMENT CLOCK
    clock.incrementClock( 1 );
    break;

case 5 : for (int i=1; i>=0; i--) // J3NP
{ // WORK OUT VALUE
  value += (long)(Math.pow( 64, power ) *
    iRegisters[2].getValue( i ));
  power++;
}

if ((iRegisters[2].getSign() != '+' ) || (value == 0))
{ jRegister.setValue( 0, (pc+1) / 64 );
  jRegister.setValue( 1, (pc+1) % 64 );
  pc=addr;
}
else
  pc++;

// INCREMENT CLOCK
clock.incrementClock( 1 );
break;
}
break;

case 44: switch(field)
{ case 0 : for (int i=1; i>=0; i--) // J4N
{ // WORK OUT VALUE
  value += (long)(Math.pow( 64, power ) *
    iRegisters[3].getValue( i ));
  power++;
}

if ((iRegisters[3].getSign() == '-' ) && (value > 0))
{ jRegister.setValue( 0, (pc+1) / 64 );
  jRegister.setValue( 1, (pc+1) % 64 );
  pc=addr;
}
else
  pc++;

// INCREMENT CLOCK
clock.incrementClock( 1 );
break;

case 1 : for (int i=1; i>=0; i--) // J4Z
{ // WORK OUT VALUE
  value += (long)(Math.pow( 64, power ) *
    iRegisters[3].getValue( i ));
  power++;
}

if (value == 0)
{ jRegister.setValue( 0, (pc+1) / 64 );
  jRegister.setValue( 1, (pc+1) % 64 );
  pc=addr;
}
else
  pc++;

// INCREMENT CLOCK
clock.incrementClock( 1 );
break;
```

```
case 2 : for (int i=1; i>=0; i--) // J4P
{ // WORK OUT VALUE
    value += (long)(Math.pow( 64, power ) *
        iRegisters[3].getValue( i ));
    power++;
}

if ((iRegisters[3].getSign() == '+') && (value > 0))
{ jRegister.setValue( 0, (pc+1) / 64 );
  jRegister.setValue( 1, (pc+1) % 64 );
  pc=addr;
}
else
    pc++;

// INCREMENT CLOCK
clock.incrementClock( 1 );
break;

case 3 : for (int i=1; i>=0; i--) // J4NN
{ // WORK OUT VALUE
    value += (long)(Math.pow( 64, power ) *
        iRegisters[3].getValue( i ));
    power++;
}

if ((iRegisters[3].getSign() != '-' ) || (value == 0))
{ jRegister.setValue( 0, (pc+1) / 64 );
  jRegister.setValue( 1, (pc+1) % 64 );
  pc=addr;
}
else
    pc++;

// INCREMENT CLOCK
clock.incrementClock( 1 );
break;

case 4 : for (int i=1; i>=0; i--) // J4NZ
{ // WORK OUT VALUE
    value += (long)(Math.pow( 64, power ) *
        iRegisters[3].getValue( i ));
    power++;
}

if (value != 0 )
{ jRegister.setValue( 0, (pc+1) / 64 );
  jRegister.setValue( 1, (pc+1) % 64 );
  pc=addr;
}
else
    pc++;

// INCREMENT CLOCK
clock.incrementClock( 1 );
break;

case 5 : for (int i=1; i>=0; i--) // J4NP
{ // WORK OUT VALUE
    value += (long)(Math.pow( 64, power ) *
        iRegisters[3].getValue( i ));
    power++;
}

if ((iRegisters[3].getSign() != '+' ) || (value == 0))
{ jRegister.setValue( 0, (pc+1) / 64 );
  jRegister.setValue( 1, (pc+1) % 64 );
  pc=addr;
}
else
    pc++;

// INCREMENT CLOCK
clock.incrementClock( 1 );
```

```
        break;
    }
    break;

case 45: switch(field)
{ case 0 : for (int i=1; i>=0; i--) // J5N
    { // WORK OUT VALUE
        value += (long)(Math.pow( 64, power ) *
            iRegisters[4].getValue( i ));
        power++;
    }

    if ((iRegisters[4].getSign() == '-') && (value > 0))
    { jRegister.setValue( 0, (pc+1) / 64 );
      jRegister.setValue( 1, (pc+1) % 64 );
      pc=addr;
    }
    else
        pc++;

    // INCREMENT CLOCK
    clock.incrementClock( 1 );
    break;

case 1 : for (int i=1; i>=0; i--) // J5Z
    { // WORK OUT VALUE
        value += (long)(Math.pow( 64, power ) *
            iRegisters[4].getValue( i ));
        power++;
    }

    if (value == 0)
    { jRegister.setValue( 0, (pc+1) / 64 );
      jRegister.setValue( 1, (pc+1) % 64 );
      pc=addr;
    }
    else
        pc++;

    // INCREMENT CLOCK
    clock.incrementClock( 1 );
    break;

case 2 : for (int i=1; i>=0; i--) // J5P
    { // WORK OUT VALUE
        value += (long)(Math.pow( 64, power ) *
            iRegisters[4].getValue( i ));
        power++;
    }

    if ((iRegisters[4].getSign() == '+') && (value > 0))
    { jRegister.setValue( 0, (pc+1) / 64 );
      jRegister.setValue( 1, (pc+1) % 64 );
      pc=addr;
    }
    else
        pc++;

    // INCREMENT CLOCK
    clock.incrementClock( 1 );
    break;

case 3 : for (int i=1; i>=0; i--) // J5NN
    { // WORK OUT VALUE
        value += (long)(Math.pow( 64, power ) *
            iRegisters[4].getValue( i ));
        power++;
    }

    if ((iRegisters[4].getSign() != '-' ) || (value == 0))
    { jRegister.setValue( 0, (pc+1) / 64 );
      jRegister.setValue( 1, (pc+1) % 64 );
      pc=addr;
    }
}
```

---

```
        else
            pc++;

        // INCREMENT CLOCK
        clock.incrementClock( 1 );
        break;

    case 4 : for (int i=1; i>=0; i--) // J5NZ
    { // WORK OUT VALUE
        value += (long)(Math.pow( 64, power ) *
            iRegisters[4].getValue( i ));
        power++;
    }

    if (value != 0 )
    { jRegister.setValue( 0, (pc+1) / 64 );
      jRegister.setValue( 1, (pc+1) % 64 );
      pc=addr;
    }
    else
        pc++;

    // INCREMENT CLOCK
    clock.incrementClock( 1 );
    break;

    case 5 : for (int i=1; i>=0; i--) // J5NP
    { // WORK OUT VALUE
        value += (long)(Math.pow( 64, power ) *
            iRegisters[4].getValue( i ));
        power++;
    }

    if ((iRegisters[4].getSign() != '+' ) || (value == 0))
    { jRegister.setValue( 0, (pc+1) / 64 );
      jRegister.setValue( 1, (pc+1) % 64 );
      pc=addr;
    }
    else
        pc++;

    // INCREMENT CLOCK
    clock.incrementClock( 1 );
    break;
}
break;

case 46: switch(field)
{ case 0 : for (int i=1; i>=0; i--) // J6N
    { // WORK OUT VALUE
        value += (long)(Math.pow( 64, power ) *
            iRegisters[5].getValue( i ));
        power++;
    }

    if ((iRegisters[5].getSign() == '-' ) && (value > 0))
    { jRegister.setValue( 0, (pc+1) / 64 );
      jRegister.setValue( 1, (pc+1) % 64 );
      pc=addr;
    }
    else
        pc++;

    // INCREMENT CLOCK
    clock.incrementClock( 1 );
    break;

    case 1 : for (int i=1; i>=0; i--) // J6Z
    { // WORK OUT VALUE
        value += (long)(Math.pow( 64, power ) *
            iRegisters[5].getValue( i ));
        power++;
    }
}
```

```
        if (value == 0)
        {
            jRegister.setValue( 0, (pc+1) / 64 );
            jRegister.setValue( 1, (pc+1) % 64 );
            pc=addr;
        }
        else
            pc++;

        // INCREMENT CLOCK
        clock.incrementClock( 1 );
        break;

case 2 : for (int i=1; i>=0; i--)                                // J6P
{
    // WORK OUT VALUE
    value += (long)(Math.pow( 64, power ) *
                    iRegisters[5].getValue( i ));
    power++;
}

if ((iRegisters[5].getSign() == '+') && (value > 0))
{
    jRegister.setValue( 0, (pc+1) / 64 );
    jRegister.setValue( 1, (pc+1) % 64 );
    pc=addr;
}
else
    pc++;

// INCREMENT CLOCK
clock.incrementClock( 1 );
break;

case 3 : for (int i=1; i>=0; i--)                                // J6NN
{
    // WORK OUT VALUE
    value += (long)(Math.pow( 64, power ) *
                    iRegisters[5].getValue( i ));
    power++;
}

if ((iRegisters[5].getSign() != '-' ) || (value == 0))
{
    jRegister.setValue( 0, (pc+1) / 64 );
    jRegister.setValue( 1, (pc+1) % 64 );
    pc=addr;
}
else
    pc++;

// INCREMENT CLOCK
clock.incrementClock( 1 );
break;

case 4 : for (int i=1; i>=0; i--)                                // J6NZ
{
    // WORK OUT VALUE
    value += (long)(Math.pow( 64, power ) *
                    iRegisters[5].getValue( i ));
    power++;
}

if (value != 0 )
{
    jRegister.setValue( 0, (pc+1) / 64 );
    jRegister.setValue( 1, (pc+1) % 64 );
    pc=addr;
}
else
    pc++;

// INCREMENT CLOCK
clock.incrementClock( 1 );
break;

case 5 : for (int i=1; i>=0; i--)                                // J6NP
{
    // WORK OUT VALUE
    value += (long)(Math.pow( 64, power ) *
                    iRegisters[5].getValue( i ));
    power++;
}
```

```
    }

    if ((iRegisters[5].getSign() != '+' ) || (value == 0))
    {
        jRegister.setValue( 0, (pc+1) / 64 );
        jRegister.setValue( 1, (pc+1) % 64 );
        pc=addr;
    }
    else
        pc++;

    // INCREMENT CLOCK
    clock.incrementClock( 1 );
    break;
}
break;

case 47: switch(field)
{
    case 0 : for (int i=4; i>=0; i--) // JXN
    {
        // WORK OUT VALUE
        value += (long)(Math.pow( 64, power ) *
            xRegister.getValue( i ));
        power++;
    }

    if ((xRegister.getSign() == '-') && (value > 0 ))
    {
        jRegister.setValue( 0, (pc+1) / 64 );
        jRegister.setValue( 1, (pc+1) % 64 );
        pc = addr;
    }
    else
        pc++;

    // INCREMENT CLOCK
    clock.incrementClock( 1 );
    break;

    case 1 : for (int i=4; i>=0; i--) // JXZ
    {
        // WORK OUT VALUE
        value += (long)(Math.pow( 64, power ) *
            xRegister.getValue( i ));
        power++;
    }

    if (value == 0)
    {
        jRegister.setValue( 0, (pc+1) / 64 );
        jRegister.setValue( 1, (pc+1) % 64 );
        pc=addr;
    }
    else
        pc++;

    // INCREMENT CLOCK
    clock.incrementClock( 1 );
    break;

    case 2 : for (int i=4; i>=0; i--) // JXP
    {
        // WORK OUT VALUE
        value += (long)(Math.pow( 64, power ) *
            xRegister.getValue( i ));
        power++;
    }

    if ((xRegister.getSign() == '+') && (value > 0))
    {
        jRegister.setValue( 0, (pc+1) / 64 );
        jRegister.setValue( 1, (pc+1) % 64 );
        pc=addr;
    }
    else
        pc++;

    // INCREMENT CLOCK
    clock.incrementClock( 1 );
    break;
}
```



```

case 3 : for (int i=4; i>=0; i--)                                // JXNN
{ // WORK OUT VALUE
    value += (long)(Math.pow( 64, power ) *
                    xRegister.getValue( i ));
    power++;
}

if ((xRegister.getSign() != '-' ) || (value == 0))
{ jRegister.setValue( 0, (pc+1) / 64 );
  jRegister.setValue( 1, (pc+1) % 64 );
  pc=addr;
}
else
    pc++;

// INCREMENT CLOCK
clock.incrementClock( 1 );
break;

case 4 : for (int i=4; i>=0; i--)                                // JXNZ
{ // WORK OUT VALUE
    value += (long)(Math.pow( 64, power ) *
                    xRegister.getValue( i ));
    power++;
}

if (value != 0 )
{ jRegister.setValue( 0, (pc+1) / 64 );
  jRegister.setValue( 1, (pc+1) % 64 );
  pc=addr;
}
else
    pc++;

// INCREMENT CLOCK
clock.incrementClock( 1 );
break;

case 5 : for (int i=4; i>=0; i--)                                // JXNP
{ // WORK OUT VALUE
    value += (long)(Math.pow( 64, power ) *
                    xRegister.getValue( i ));
    power++;
}

if ((xRegister.getSign() != '+' ) || (value == 0))
{ jRegister.setValue( 0, (pc+1) / 64 );
  jRegister.setValue( 1, (pc+1) % 64 );
  pc=addr;
}
else
    pc++;

// INCREMENT CLOCK
clock.incrementClock( 1 );
break;
}
break;

case 48: switch (field)
{ case 0 : // ADD THE VALUE TO rA                                // INCA
    mADD( aRegister, addr );

    // INCREMENT CLOCK & PROGRAM COUNTER
    pc++;
    clock.incrementClock( 1 );
    break;

case 1 : addr = -addr;                                           // DECA

    // ADD THE VALUE TO rA
    mADD( aRegister, addr );

    // INCREMENT CLOCK & PROGRAM COUNTER

```

```
        pc++;
        clock.incrementClock( 1 );
        break;

    case 2 : aRegister.setSign( (addr>=0) ? '+' : '-' );           // ENTA
        if (addr<0)
            addr = addr * (-1);
        aRegister.setValue( 4, addr % 64 );
        aRegister.setPacked( 4, memory[pc].isPacked( 1 ) );
        aRegister.setValue( 3, addr / 64 );
        aRegister.setPacked( 3, false );
        for (int i=2; i>=0; i--)
        { aRegister.setValue( i, 0 );
          aRegister.setPacked( i, false );
        }

        // INCREMENT CLOCK & PROGRAM COUNTER
        pc++;
        clock.incrementClock( 1 );
        break;

    case 3 : aRegister.setSign( (addr>=0) ? '-' : '+' );           // ENNA
        if (addr<0)
            addr = addr * (-1);
        aRegister.setValue( 4, addr % 64 );
        aRegister.setPacked( 4, memory[pc].isPacked( 1 ) );
        aRegister.setValue( 3, addr / 64 );
        aRegister.setPacked( 3, false );
        for (int i=2; i>=0; i--)
        { aRegister.setValue( i, 0 );
          aRegister.setPacked( i, false );
        }

        // INCREMENT CLOCK & PROGRAM COUNTER
        pc++;
        clock.incrementClock( 1 );
        break;
    }
    break;

case 49: switch (field)
{   case 0 : // ADD THE VALUE TO I1                               // INC1
        addr = addr + (((iRegisters[0].getValue(0) * 64) +
                        iRegisters[0].getValue(1))
                      * ((iRegisters[0].getSign() == '+') ? 1 : (-1)));

        if ( (addr<(-4096)) || (addr>4096) )
            oIndicator.setState( true );

        if (addr<0)
        { iRegisters[0].setSign( '-' );
          addr = addr * (-1);
        }
        else
            iRegisters[0].setSign( '+' );

        iRegisters[0].setValue( 1, addr % 64 );
        iRegisters[0].setValue( 0, (addr/64) % 64 );
        iRegisters[0].setPacked( 1, memory[pc].isPacked( 1 ) );
        iRegisters[0].setPacked( 0, memory[pc].isPacked( 0 ) );

        // INCREMENT CLOCK & PROGRAM COUNTER
        pc++;
        clock.incrementClock( 1 );
        break;

    case 1 : addr = -addr;                                           // DEC1

        // ADD THE VALUE TO I1
        addr = addr + (((iRegisters[0].getValue(0) * 64) +
                        iRegisters[0].getValue(1))
                      * ((iRegisters[0].getSign() == '+') ? 1 : (-1)));

        if ( (addr<(-4096)) || (addr>4096) )
```

```

        oIndicator.setState( true );

    if (addr<0)
    { iRegisters[0].setSign( '-' );
      addr = addr * (-1);
    }
    else
        iRegisters[0].setSign( '+' );

    iRegisters[0].setValue( 1, addr % 64 );
    iRegisters[0].setValue( 0, (addr/64) % 64 );
    iRegisters[0].setPacked( 1, memory[pc].isPacked( 1 ) );
    iRegisters[0].setPacked( 0, memory[pc].isPacked( 0 ) );

    // INCREMENT CLOCK & PROGRAM COUNTER
    pc++;
    clock.incrementClock( 1 );
    break;

case 2 : iRegisters[0].setSign( (addr>=0) ? '+' : '-' );          // ENT1
    if (addr<0)
        addr = addr * (-1);

    if (addr>4096)
        oIndicator.setState( true );

    iRegisters[0].setValue( 1, addr % 64 );
    iRegisters[0].setPacked( 1, memory[pc].isPacked( 1 ) );
    iRegisters[0].setValue( 0, (addr/64) % 64 );
    iRegisters[0].setPacked( 0, memory[pc].isPacked( 0 ) );

    // INCREMENT CLOCK & PROGRAM COUNTER
    pc++;
    clock.incrementClock( 1 );
    break;

case 3 : iRegisters[0].setSign( (addr>=0) ? '-' : '+' );          // ENN1
    if (addr<0)
        addr = addr * (-1);

    if (addr>4096)
        oIndicator.setState( true );

    iRegisters[0].setValue( 1, addr % 64 );
    iRegisters[0].setPacked( 1, memory[pc].isPacked( 1 ) );
    iRegisters[0].setValue( 0, (addr/64) % 64 );
    iRegisters[0].setPacked( 0, memory[pc].isPacked( 0 ) );

    // INCREMENT CLOCK & PROGRAM COUNTER
    pc++;
    clock.incrementClock( 1 );
    break;
}
break;

case 50: switch (field)
{ case 0 : // ADD THE VALUE TO I2                                // INC2
    addr = addr + ((iRegisters[1].getValue(0) * 64) +
                  iRegisters[1].getValue(1))
              * ((iRegisters[1].getSign() == '+') ? 1 : (-1));

    if ( (addr<(-4096)) || (addr>4096) )
        oIndicator.setState( true );

    if (addr<0)
    { iRegisters[1].setSign( '-' );
      addr = addr * (-1);
    }
    else
        iRegisters[1].setSign( '+' );

    iRegisters[1].setValue( 1, addr % 64 );
    iRegisters[1].setValue( 0, (addr/64) % 64 );
    iRegisters[1].setPacked( 1, memory[pc].isPacked( 1 ) );

```

```
        iRegisters[1].setPacked( 0, memory[pc].isPacked( 0 ) );

        // INCREMENT CLOCK & PROGRAM COUNTER
        pc++;
        clock.incrementClock( 1 );
        break;

    case 1 : addr = -addr;                                     // DEC2

        // ADD THE VALUE TO I2
        addr = addr + (((iRegisters[1].getValue(0) * 64) +
            iRegisters[1].getValue(1))
            * ((iRegisters[1].getSign() == '+' ) ? 1 : (-1)));

        if ( (addr < (-4096)) || (addr > 4096) )
            oIndicator.setState( true );

        if (addr < 0)
        { iRegisters[1].setSign( '-' );
          addr = addr * (-1);
        }
        else
            iRegisters[1].setSign( '+' );

        iRegisters[1].setValue( 1, addr % 64 );
        iRegisters[1].setValue( 0, (addr/64) % 64 );
        iRegisters[1].setPacked( 1, memory[pc].isPacked( 1 ) );
        iRegisters[1].setPacked( 0, memory[pc].isPacked( 0 ) );

        // INCREMENT CLOCK & PROGRAM COUNTER
        pc++;
        clock.incrementClock( 1 );
        break;

    case 2 : iRegisters[1].setSign( (addr >= 0) ? '+' : '-' );    // ENT2
        if (addr < 0)
            addr = addr * (-1);

        if (addr > 4096)
            oIndicator.setState( true );

        iRegisters[1].setValue( 1, addr % 64 );
        iRegisters[1].setPacked( 1, memory[pc].isPacked( 1 ) );
        iRegisters[1].setValue( 0, (addr/64) % 64 );
        iRegisters[1].setPacked( 0, memory[pc].isPacked( 0 ) );

        // INCREMENT CLOCK & PROGRAM COUNTER
        pc++;
        clock.incrementClock( 1 );
        break;

    case 3 : iRegisters[1].setSign( (addr >= 0) ? '-' : '+' );    // ENN2
        if (addr < 0)
            addr = addr * (-1);

        if (addr > 4096)
            oIndicator.setState( true );

        iRegisters[1].setValue( 1, addr % 64 );
        iRegisters[1].setPacked( 1, memory[pc].isPacked( 1 ) );
        iRegisters[1].setValue( 0, (addr/64) % 64 );
        iRegisters[1].setPacked( 0, memory[pc].isPacked( 0 ) );

        // INCREMENT CLOCK & PROGRAM COUNTER
        pc++;
        clock.incrementClock( 1 );
        break;
    }
    break;

case 51: switch (field)
{   case 0 : // ADD THE VALUE TO I3                             // INC3
        addr = addr + (((iRegisters[2].getValue(0) * 64) +
            iRegisters[2].getValue(1))
```

```

        * ((iRegisters[2].getSign() == '+' ) ? 1 : (-1)));

if ( (addr<(-4096)) || (addr>4096) )
    oIndicator.setState( true );

if (addr<0)
{
    iRegisters[2].setSign( '-' );
    addr = addr * (-1);
}
else
    iRegisters[2].setSign( '+' );

iRegisters[2].setValue( 1, addr % 64 );
iRegisters[2].setValue( 0, (addr/64) % 64 );
iRegisters[2].setPacked( 1, memory[pc].isPacked( 1 ) );
iRegisters[2].setPacked( 0, memory[pc].isPacked( 0 ) );

// INCREMENT CLOCK & PROGRAM COUNTER
pc++;
clock.incrementClock( 1 );
break;

case 1 : addr = -addr;                                     // DEC3

// ADD THE VALUE TO I3
addr = addr + ((iRegisters[2].getValue(0) * 64) +
               iRegisters[2].getValue(1))
        * ((iRegisters[2].getSign() == '+' ) ? 1 : (-1)));

if ( (addr<(-4096)) || (addr>4096) )
    oIndicator.setState( true );

if (addr<0)
{
    iRegisters[2].setSign( '-' );
    addr = addr * (-1);
}
else
    iRegisters[2].setSign( '+' );

iRegisters[2].setValue( 1, addr % 64 );
iRegisters[2].setValue( 0, (addr/64) % 64 );
iRegisters[2].setPacked( 1, memory[pc].isPacked( 1 ) );
iRegisters[2].setPacked( 0, memory[pc].isPacked( 0 ) );

// INCREMENT CLOCK & PROGRAM COUNTER
pc++;
clock.incrementClock( 1 );
break;

case 2 : iRegisters[2].setSign( (addr>=0) ? '+' : '-' );    // ENT3
if (addr<0)
    addr = addr * (-1);

if (addr>4096)
    oIndicator.setState( true );

iRegisters[2].setValue( 1, addr % 64 );
iRegisters[2].setPacked( 1, memory[pc].isPacked( 1 ) );
iRegisters[2].setValue( 0, (addr/64) % 64 );
iRegisters[2].setPacked( 0, memory[pc].isPacked( 0 ) );

// INCREMENT CLOCK & PROGRAM COUNTER
pc++;
clock.incrementClock( 1 );
break;

case 3 : iRegisters[2].setSign( (addr>=0) ? '-' : '+' );    // ENN3
if (addr<0)
    addr = addr * (-1);

if (addr>4096)
    oIndicator.setState( true );

iRegisters[2].setValue( 1, addr % 64 );

```

```
iRegisters[2].setPacked( 1, memory[pc].isPacked( 1 ) );
iRegisters[2].setValue( 0, (addr/64) % 64 );
iRegisters[2].setPacked( 0, memory[pc].isPacked( 0 ) );

// INCREMENT CLOCK & PROGRAM COUNTER
pc++;
clock.incrementClock( 1 );
break;
}
break;

case 52: switch (field)
{ case 0 : // ADD THE VALUE TO I4 // INC4
    addr = addr + (((iRegisters[3].getValue(0) * 64) +
        iRegisters[3].getValue(1))
        * ((iRegisters[3].getSign() == '+') ? 1 : (-1)));

    if ( (addr < (-4096)) || (addr > 4096) )
        oIndicator.setState( true );

    if (addr < 0)
    { iRegisters[3].setSign( '-' );
      addr = addr * (-1);
    }
    else
        iRegisters[3].setSign( '+' );

    iRegisters[3].setValue( 1, addr % 64 );
    iRegisters[3].setValue( 0, (addr/64) % 64 );
    iRegisters[3].setPacked( 1, memory[pc].isPacked( 1 ) );
    iRegisters[3].setPacked( 0, memory[pc].isPacked( 0 ) );

    // INCREMENT CLOCK & PROGRAM COUNTER
    pc++;
    clock.incrementClock( 1 );
    break;

case 1 : addr = -addr; // DEC4

    // ADD THE VALUE TO I4
    addr = addr + (((iRegisters[3].getValue(0) * 64) +
        iRegisters[3].getValue(1))
        * ((iRegisters[3].getSign() == '+') ? 1 : (-1)));

    if ( (addr < (-4096)) || (addr > 4096) )
        oIndicator.setState( true );

    if (addr < 0)
    { iRegisters[3].setSign( '-' );
      addr = addr * (-1);
    }
    else
        iRegisters[3].setSign( '+' );

    iRegisters[3].setValue( 1, addr % 64 );
    iRegisters[3].setValue( 0, (addr/64) % 64 );
    iRegisters[3].setPacked( 1, memory[pc].isPacked( 1 ) );
    iRegisters[3].setPacked( 0, memory[pc].isPacked( 0 ) );

    // INCREMENT CLOCK & PROGRAM COUNTER
    pc++;
    clock.incrementClock( 1 );
    break;

case 2 : iRegisters[3].setSign( (addr >= 0) ? '+' : '-' ); // ENT4
    if (addr < 0)
        addr = addr * (-1);

    if (addr > 4096)
        oIndicator.setState( true );

    iRegisters[3].setValue( 1, addr % 64 );
    iRegisters[3].setPacked( 1, memory[pc].isPacked( 1 ) );
    iRegisters[3].setValue( 0, (addr/64) % 64 );
```

```
        iRegisters[3].setPacked( 0, memory[pc].isPacked( 0 ) );

        // INCREMENT CLOCK & PROGRAM COUNTER
        pc++;
        clock.incrementClock( 1 );
        break;

    case 3 : iRegisters[3].setSign( (addr>=0) ? '-' : '+' );          // ENN4
        if (addr<0)
            addr = addr * (-1);

        if (addr>4096)
            oIndicator.setState( true );

        iRegisters[3].setValue( 1, addr % 64 );
        iRegisters[3].setPacked( 1, memory[pc].isPacked( 1 ) );
        iRegisters[3].setValue( 0, (addr/64) % 64 );
        iRegisters[3].setPacked( 0, memory[pc].isPacked( 0 ) );

        // INCREMENT CLOCK & PROGRAM COUNTER
        pc++;
        clock.incrementClock( 1 );
        break;
    }
    break;

case 53: switch (field)
    { case 0 : // ADD THE VALUE TO I5                                // INC5
        addr = addr + (((iRegisters[4].getValue(0) * 64) +
            iRegisters[4].getValue(1))
            * ((iRegisters[4].getSign() == '+') ? 1 : (-1)));

        if ( (addr<(-4096)) || (addr>4096) )
            oIndicator.setState( true );

        if (addr<0)
        { iRegisters[4].setSign( '-' );
          addr = addr * (-1);
        }
        else
            iRegisters[4].setSign( '+' );

        iRegisters[4].setValue( 1, addr % 64 );
        iRegisters[4].setValue( 0, (addr/64) % 64 );
        iRegisters[4].setPacked( 1, memory[pc].isPacked( 1 ) );
        iRegisters[4].setPacked( 0, memory[pc].isPacked( 0 ) );

        // INCREMENT CLOCK & PROGRAM COUNTER
        pc++;
        clock.incrementClock( 1 );
        break;

    case 1 : addr = -addr;                                           // DEC5

        // ADD THE VALUE TO I5
        addr = addr + (((iRegisters[4].getValue(0) * 64) +
            iRegisters[4].getValue(1))
            * ((iRegisters[4].getSign() == '+') ? 1 : (-1)));

        if ( (addr<(-4096)) || (addr>4096) )
            oIndicator.setState( true );

        if (addr<0)
        { iRegisters[4].setSign( '-' );
          addr = addr * (-1);
        }
        else
            iRegisters[4].setSign( '+' );

        iRegisters[4].setValue( 1, addr % 64 );
        iRegisters[4].setValue( 0, (addr/64) % 64 );
        iRegisters[4].setPacked( 1, memory[pc].isPacked( 1 ) );
        iRegisters[4].setPacked( 0, memory[pc].isPacked( 0 ) );
```

```
// INCREMENT CLOCK & PROGRAM COUNTER
pc++;
clock.incrementClock( 1 );
break;

case 2 : iRegisters[4].setSign( (addr>=0) ? '+' : '-' );      // ENT5
if (addr<0)
    addr = addr * (-1);

if (addr>4096)
    oIndicator.setState( true );

iRegisters[4].setValue( 1, addr % 64 );
iRegisters[4].setPacked( 1, memory[pc].isPacked( 1 ) );
iRegisters[4].setValue( 0, (addr/64) % 64 );
iRegisters[4].setPacked( 0, memory[pc].isPacked( 0 ) );

// INCREMENT CLOCK & PROGRAM COUNTER
pc++;
clock.incrementClock( 1 );
break;

case 3 : iRegisters[4].setSign( (addr>=0) ? '-' : '+' );      // ENN5
if (addr<0)
    addr = addr * (-1);

if (addr>4096)
    oIndicator.setState( true );

iRegisters[4].setValue( 1, addr % 64 );
iRegisters[4].setPacked( 1, memory[pc].isPacked( 1 ) );
iRegisters[4].setValue( 0, (addr/64) % 64 );
iRegisters[4].setPacked( 0, memory[pc].isPacked( 0 ) );

// INCREMENT CLOCK & PROGRAM COUNTER
pc++;
clock.incrementClock( 1 );
break;
}
break;

case 54: switch (field)
{ case 0 : // ADD THE VALUE TO I6                                // INC6
    addr = addr + (((iRegisters[5].getValue(0) * 64) +
                    iRegisters[5].getValue(1))
                  * ((iRegisters[5].getSign() == '+') ? 1 : (-1)));

    if ( (addr<(-4096)) || (addr>4096) )
        oIndicator.setState( true );

    if (addr<0)
    { iRegisters[5].setSign( '-' );
      addr = addr * (-1);
    }
    else
        iRegisters[5].setSign( '+' );

    iRegisters[5].setValue( 1, addr % 64 );
    iRegisters[5].setValue( 0, (addr/64) % 64 );
    iRegisters[5].setPacked( 1, memory[pc].isPacked( 1 ) );
    iRegisters[5].setPacked( 0, memory[pc].isPacked( 0 ) );

    // INCREMENT CLOCK & PROGRAM COUNTER
    pc++;
    clock.incrementClock( 1 );
    break;

case 1 : addr = -addr;                                           // DEC6

    // ADD THE VALUE TO I6
    addr = addr + (((iRegisters[4].getValue(0) * 64) +
                    iRegisters[4].getValue(1))
                  * ((iRegisters[4].getSign() == '+') ? 1 : (-1)));
```



```
        if ( (addr<(-4096)) || (addr>4096) )
            oIndicator.setState( true );

        if (addr<0)
        { iRegisters[5].setSign( '-' );
          addr = addr * (-1);
        }
        else
            iRegisters[5].setSign( '+' );

        iRegisters[5].setValue( 1, addr % 64 );
        iRegisters[5].setValue( 0, (addr/64) % 64 );
        iRegisters[5].setPacked( 1, memory[pc].isPacked( 1 ) );
        iRegisters[5].setPacked( 0, memory[pc].isPacked( 0 ) );

        // INCREMENT CLOCK & PROGRAM COUNTER
        pc++;
        clock.incrementClock( 1 );
        break;

case 2 : iRegisters[5].setSign( (addr>=0) ? '+' : '-' );           // ENT6
        if (addr<0)
            addr = addr * (-1);

        if (addr>4096)
            oIndicator.setState( true );

        iRegisters[5].setValue( 1, addr % 64 );
        iRegisters[5].setPacked( 1, memory[pc].isPacked( 1 ) );
        iRegisters[5].setValue( 0, (addr/64) % 64 );
        iRegisters[5].setPacked( 0, memory[pc].isPacked( 0 ) );

        // INCREMENT CLOCK & PROGRAM COUNTER
        pc++;
        clock.incrementClock( 1 );
        break;

case 3 : iRegisters[5].setSign( (addr>=0) ? '-' : '+' );           // ENN6
        if (addr<0)
            addr = addr * (-1);

        if (addr>4096)
            oIndicator.setState( true );

        iRegisters[5].setValue( 1, addr % 64 );
        iRegisters[5].setPacked( 1, memory[pc].isPacked( 1 ) );
        iRegisters[5].setValue( 0, (addr/64) % 64 );
        iRegisters[5].setPacked( 0, memory[pc].isPacked( 0 ) );

        // INCREMENT CLOCK & PROGRAM COUNTER
        pc++;
        clock.incrementClock( 1 );
        break;
    }
    break;

case 55: switch (field)
    { case 0 : // ADD THE VALUE TO rX                               // INCX
        mADD( xRegister, addr );

        // INCREMENT CLOCK & PROGRAM COUNTER
        pc++;
        clock.incrementClock( 1 );
        break;

        case 1 : addr = -addr;                                       // DECX

        // ADD THE VALUE TO rX
        mADD( xRegister, addr );

        // INCREMENT CLOCK & PROGRAM COUNTER
        pc++;
        clock.incrementClock( 1 );
        break;
```

```
case 2 : xRegister.setSign( (addr>=0) ? '+' : '-' );           // ENTX
        if (addr<0)
            addr = addr * (-1);
        xRegister.setValue( 4, addr % 64 );
        xRegister.setPacked( 4, memory[pc].isPacked( 1 ) );
        xRegister.setValue( 3, addr / 64 );
        xRegister.setPacked( 3, false );
        for (int i=2; i>=0; i--)
        { xRegister.setValue( i, 0 );
          xRegister.setPacked( i, false );
        }

        // INCREMENT CLOCK & PROGRAM COUNTER
        pc++;
        clock.incrementClock( 1 );
        break;

case 3 : xRegister.setSign( (addr>=0) ? '-' : '+' );           // ENNX
        if (addr<0)
            addr = addr * (-1);
        xRegister.setValue( 4, addr % 64 );
        xRegister.setPacked( 4, memory[pc].isPacked( 1 ) );
        xRegister.setValue( 3, addr / 64 );
        xRegister.setPacked( 3, false );
        for (int i=2; i>=0; i--)
        { xRegister.setValue( i, 0 );
          xRegister.setPacked( i, false );
        }

        // INCREMENT CLOCK & PROGRAM COUNTER
        pc++;
        clock.incrementClock( 1 );
        break;
    }
    break;

case 56: for (int i=(fEnd-1); i>=fOurStart; i--)                // CMPA
    { // WORK OUT VALUE
      value += (long)(Math.pow( 64, power ) * memory[ addr ].getValue( i ));
      power++;
    }

    // SET THE SIGN
    if (fStart == 0)
        value = (memory[ addr ].getSign() == '-') ? -value : value;

    // COMPARE THE VALUE TO rA
    mCMP( aRegister, value, fOurStart, fEnd-1, (fStart == 0) );

    // INCREMENT CLOCK & PROGRAM COUNTER
    pc++;
    clock.incrementClock( 2 );
    break;

case 57: for (int i=(fEnd-1); i>=fOurStart; i--)                // CMP1
    { // WORK OUT VALUE
      value += (long)(Math.pow( 64, power ) * memory[ addr ].getValue( i ));
      power++;
    }

    // SET THE SIGN
    if (fStart == 0)
        value = (memory[ addr ].getSign() == '-') ? -value : value;

    // COMPARE THE VALUE TO I1
    mCMPi( iRegisters[0], value, fOurStart, fEnd-1, (fStart == 0) );

    // INCREMENT CLOCK & PROGRAM COUNTER
    pc++;
    clock.incrementClock( 2 );
    break;

case 58: for (int i=(fEnd-1); i>=fOurStart; i--)                // CMP2
```

```
{ // WORK OUT VALUE
  value += (long)(Math.pow( 64, power ) * memory[ addr ].getValue( i ));
  power++;
}

// SET THE SIGN
if (fStart == 0)
  value = (memory[ addr ].getSign() == '-') ? -value : value;

// COMPARE THE VALUE TO I1
mCMPi( iRegisters[1], value, fOurStart, fEnd-1, (fStart == 0) );

// INCREMENT CLOCK & PROGRAM COUNTER
pc++;
clock.incrementClock( 2 );
break;

case 59: for (int i=(fEnd-1); i>=fOurStart; i--) // CMP3
{ // WORK OUT VALUE
  value += (long)(Math.pow( 64, power ) * memory[ addr ].getValue( i ));
  power++;
}

// SET THE SIGN
if (fStart == 0)
  value = (memory[ addr ].getSign() == '-') ? -value : value;

// COMPARE THE VALUE TO I1
mCMPi( iRegisters[2], value, fOurStart, fEnd-1, (fStart == 0) );

// INCREMENT CLOCK & PROGRAM COUNTER
pc++;
clock.incrementClock( 2 );
break;

case 60: for (int i=(fEnd-1); i>=fOurStart; i--) // CMP4
{ // WORK OUT VALUE
  value += (long)(Math.pow( 64, power ) * memory[ addr ].getValue( i ));
  power++;
}

// SET THE SIGN
if (fStart == 0)
  value = (memory[ addr ].getSign() == '-') ? -value : value;

// COMPARE THE VALUE TO I1
mCMPi( iRegisters[3], value, fOurStart, fEnd-1, (fStart == 0) );

// INCREMENT CLOCK & PROGRAM COUNTER
pc++;
clock.incrementClock( 2 );
break;

case 61: for (int i=(fEnd-1); i>=fOurStart; i--) // CMP5
{ // WORK OUT VALUE
  value += (long)(Math.pow( 64, power ) * memory[ addr ].getValue( i ));
  power++;
}

// SET THE SIGN
if (fStart == 0)
  value = (memory[ addr ].getSign() == '-') ? -value : value;

// COMPARE THE VALUE TO I1
mCMPi( iRegisters[4], value, fOurStart, fEnd-1, (fStart == 0) );

// INCREMENT CLOCK & PROGRAM COUNTER
pc++;
clock.incrementClock( 2 );
break;

case 62: for (int i=(fEnd-1); i>=fOurStart; i--) // CMP6
{ // WORK OUT VALUE
  value += (long)(Math.pow( 64, power ) * memory[ addr ].getValue( i ));
```

```
        power++;
    }

    // SET THE SIGN
    if (fStart == 0)
        value = (memory[ addr ].getSign() == '-') ? -value : value;

    // COMPARE THE VALUE TO I1
    mCMPi( iRegisters[5], value, fOurStart, fEnd-1, (fStart == 0) );

    // INCREMENT CLOCK & PROGRAM COUNTER
    pc++;
    clock.incrementClock( 2 );
    break;

case 63: for (int i=(fEnd-1); i>=fOurStart; i--) // CMPX
    { // WORK OUT VALUE
        value += (long)(Math.pow( 64, power ) * memory[ addr ].getValue( i ));
        power++;
    }

    // SET THE SIGN
    if (fStart == 0)
        value = (memory[ addr ].getSign() == '-') ? -value : value;

    // COMPARE THE VALUE TO rA
    mCMP( xRegister, value, fOurStart, fEnd-1, (fStart == 0) );

    // INCREMENT CLOCK & PROGRAM COUNTER
    pc++;
    clock.incrementClock( 2 );
    break;
}
} catch( Exception e )
{ if (e instanceof NotAMIXCharacterException)
    rtError = "Not a MIX character value at pc="+pc;
  else
  { System.out.println( e );
    break;
  }
}

} while ( (halt == false) && (rtError.equals( "" ) && (stepped == false) );

if (!(rtError.equals("")))
    System.out.println( rtError );

return pc;
}

// NOTE...See Knuth's "baffling" description for reasons why this implementation has been chosen...

/** Adds a value to an existing word/register in memory.
 * @param addTo the MIXWord that is going to be added to
 * @param div the BigInteger value to add
 * @return No return value
 */
private void mADD( MIXWord addTo, long div )
{ long mod;
  long valueOfWord = addTo.toLong();

  valueOfWord += div;
  try
  { if ( valueOfWord < 0 )
    { addTo.setSign( '-' );
      valueOfWord = -valueOfWord;
    }
    else
      addTo.setSign( '+' );

    for ( int i = 4; i >=0; i-- )
    { mod = valueOfWord % 64;
      addTo.setValue( i, (int)mod );
      valueOfWord = valueOfWord / 64;
    }
  }
}
```

```
    }

    } catch (Exception e) { System.out.println(
        "An exception has occurred when it shouldn't have in mADD()" ); }

    if ( (!(valueOfWord == 0)) && (addTo == aRegister) ) // If we overflow totally and it's the
        // A-Register
        oIndicator.setState( true );                    // Set the overflow indicator 'on'
}

private void mMUL( long mult )
{ long mod;
  long result = aRegister.toLong();

  result = result * mult;
  try
  { if ( result < 0 )
    { aRegister.setSign( '-' );
      xRegister.setSign( '-' );
      result = -result;                    // makes calculations easier
    }
    else
    { aRegister.setSign( '+' );
      xRegister.setSign( '+' );
    }

    for ( int i = 4; i>=0; i-- )
    { mod = result % 64;
      xRegister.setValue( i, (int)mod );
      result = result / 64;
    }

    if ( !(result == 0) )
    { for ( int i = 4; i>=0; i-- )
      { mod = result % 64;
        aRegister.setValue( i, (int)mod );
        result = result / 64;
      }
    }
    else
    { for ( int i = 4; i>=0; i-- )
      aRegister.setValue( i, 0 );
    }

    // AMBIGUOUS 'PACKING' INFORMATION SUPPLIED BY KNUTH ON p132 SO WE IGNORE PACK BITS

  } catch (Exception e) { System.out.println(
      "An exception has occurred when it shouldn't have in mMUL()" ); }
}

private void mDIV( long denominator )
{ long numerator = 0,
  quotient,
  remainder,
  aMod,
  xMod,
  biggestNum = 1073741823;

  int aPower = 5,
  xPower = 0;

  if ( denominator == 0 ) // see p.131
  { oIndicator.setState( true );
    System.out.println( "Division by zero" );
    return;
  }

  try
  { for ( int i = 4; i>=0; i-- ) // get numerator
    { numerator += (long)(Math.pow(64, aPower) * aRegister.getValue( i ) );
      numerator += (long)(Math.pow(64, xPower) * xRegister.getValue( i ) );
      aPower++;
      xPower++;
    }
  }
```

```
        numerator = (aRegister.getSign() == '-') ? -numerator : numerator; // set the numerator's
                                                                    // sign

quotient = numerator / denominator;
remainder = numerator % denominator;

if ((Math.abs(quotient)) > biggestNum )
{
    oIndicator.setState( true );
    System.out.println( "Returning...quotient too large." );
    return;
}

xRegister.setSign( aRegister.getSign() );
aRegister.setSign( (quotient < 0) ? '-' : '+' );

for ( int i = 4; i>=0; i-- )
{
    aMod = quotient % 64;
    aRegister.setValue( i, (int)aMod );
    quotient = quotient / 64;
    xMod = remainder % 64;
    xRegister.setValue( i, (int)xMod );
    remainder = remainder / 64;
}

} catch (Exception e) { System.out.println(
    "An exception has occurred when it shouldn't have in mDIV()" ); }

}

private void mNUM()
{
    long numVal = 0,
        aMult = 100000,
        xMult = 1;
    int mod;

    try
    {
        for (int i=4; i>=0; i--)
        {
            numVal += (((aRegister.getValue( i )) % 10)*aMult);
            numVal += (((xRegister.getValue( i )) % 10)*xMult);
            aMult = aMult*10;
            xMult = xMult*10;
        }

        for (int i=4; i>=0; i--)
        {
            mod = (int)(numVal % 64);
            numVal = numVal / 64;
            aRegister.setValue( i, mod );
            aRegister.setPacked( i, true );
        }
        aRegister.setPacked( 0, false );

        if (numVal > 0)
            oIndicator.setState( true );

    } catch (Exception e) { System.out.println(
        "An exception occurred when it shouldn't have in mNUM()" );
        System.out.println( e ); }

}

private void mCHAR()
{
    long numVal = aRegister.toLong();
    int mod;

    numVal = Math.abs(numVal);
    try
    {
        for (int i=4; i>=0; i--)
        {
            mod = (int)((numVal % 10) + 30);
            xRegister.setValue( i, mod );
            xRegister.setPacked( i, false );
            numVal = numVal / 10;
        }
        for (int i=4; i>=0; i--)
        {
            mod = (int)((numVal % 10) + 30);
```

```

        aRegister.setValue( i, mod );
        aRegister.setPacked( i, false );
        numVal = numVal / 10;
    }
    if (numVal > 0)
        oIndicator.setState( true );
} catch (Exception e) { System.out.println(
    "An exception occurred when it shouldn't have in mCHAR()" ); }
}

private void mSLA( long shift )
{ try
    { for (long i=0; i<shift; i++)
        { aRegister.setValue( 0, aRegister.getValue( 1 ) );
          aRegister.setValue( 1, aRegister.getValue( 2 ) );
          aRegister.setValue( 2, aRegister.getValue( 3 ) );
          aRegister.setValue( 3, aRegister.getValue( 4 ) );
          aRegister.setValue( 4, 0 );
        }
    } catch (Exception e) { System.out.println(
        "An exception occurred when it shouldn't have in mSLA()" ); }
}

private void mSRA( long shift )
{ try
    { for (long i=0; i<shift; i++)
        { aRegister.setValue( 4, aRegister.getValue( 3 ) );
          aRegister.setValue( 3, aRegister.getValue( 2 ) );
          aRegister.setValue( 2, aRegister.getValue( 1 ) );
          aRegister.setValue( 1, aRegister.getValue( 0 ) );
          aRegister.setValue( 0, 0 );
        }
    } catch (Exception e) { System.out.println(
        "An exception occurred when it shouldn't have in mSRA()" ); }
}

private void mSLAX( long shift, boolean circular )
{ int value;
  try
  {
    for (long i=0; i<shift; i++)
    { value = aRegister.getValue( 0 );
      aRegister.setValue( 0, aRegister.getValue( 1 ) );
      aRegister.setValue( 1, aRegister.getValue( 2 ) );
      aRegister.setValue( 2, aRegister.getValue( 3 ) );
      aRegister.setValue( 3, aRegister.getValue( 4 ) );
      aRegister.setValue( 4, xRegister.getValue( 0 ) );
      xRegister.setValue( 0, xRegister.getValue( 1 ) );
      xRegister.setValue( 1, xRegister.getValue( 2 ) );
      xRegister.setValue( 2, xRegister.getValue( 3 ) );
      xRegister.setValue( 3, xRegister.getValue( 4 ) );
      if (circular)
          xRegister.setValue( 4, value );
      else
          xRegister.setValue( 4, 0 );
    }
  } catch (Exception e) { System.out.println(
      "An exception occurred when it shouldn't have in mSLAX()" ); }
}

private void mSRAX( long shift, boolean circular )
{ int value;
  try
  {
    for (long i=0; i<shift; i++)
    { value = xRegister.getValue( 4 );
      xRegister.setValue( 4, xRegister.getValue( 3 ) );
      xRegister.setValue( 3, xRegister.getValue( 2 ) );
      xRegister.setValue( 2, xRegister.getValue( 1 ) );
      xRegister.setValue( 1, xRegister.getValue( 0 ) );
      xRegister.setValue( 0, aRegister.getValue( 4 ) );
      aRegister.setValue( 4, aRegister.getValue( 3 ) );
      aRegister.setValue( 3, aRegister.getValue( 2 ) );
      aRegister.setValue( 2, aRegister.getValue( 1 ) );
    }
  }
}

```

```
        aRegister.setValue( 1, aRegister.getValue( 0 ) );
        if (circular)
            aRegister.setValue( 0, value );
        else
            aRegister.setValue( 0, 0 );
    }
} catch (Exception e) { System.out.println(
    "An exception occurred when it shouldn't have in mSRAX()" ); }
}

private void mMOVE( MIXWord from, MIXWord to )
{
    try
    {
        for (int i=4; i>=0; i--)
        {
            to.setValue( i, from.getValue( i ) );
            to.setPacked( i, from.isPacked( i ) );
        }
        to.setSign( from.getSign() );
    } catch (Exception e) { System.out.println(
        "An exception occurred when it shouldn't have in mMOVE()" ); }
}

private void mLDi( int regID, MIXWord word, int start, int end, boolean sign )
{
    int counter = 1;

    try
    {
        if (sign)
            iRegisters[ regID ].setSign( word.getSign() );
        else
            iRegisters[ regID ].setSign( '+' );

        for (int i=end; i>=start; i--)
        {
            iRegisters[ regID ].setValue( counter, word.getValue( i ) );
            iRegisters[ regID ].setPacked( counter, word.isPacked( i ) );
            counter--;
        }

        if (counter==0)
        {
            iRegisters[ regID ].setValue( 0, 0 );
            iRegisters[ regID ].setPacked( 0, false );
        }
    } catch (Exception e) { System.out.println(
        "An exception occurred when it shouldn't have in mLDi()" ); }
}

private void mLDiN( int regID, MIXWord word, int start, int end, boolean sign )
{
    int counter = 1;

    try
    {
        if (sign)
        {
            if (word.getSign() == '+')
                iRegisters[ regID ].setSign( '-' );
            else
                iRegisters[ regID ].setSign( '+' );
        }
        else
            iRegisters[ regID ].setSign( '-' );

        for (int i=(end-1); i>=start; i--)
        {
            iRegisters[ regID ].setValue( counter, word.getValue( i ) );
            iRegisters[ regID ].setPacked( counter, word.isPacked( i ) );
            counter--;
        }

        if (counter==0)
        {
            iRegisters[ regID ].setValue( 0, 0 );
            iRegisters[ regID ].setPacked( 0, false );
        }
    } catch (Exception e) { System.out.println(
        "An exception occurred when it shouldn't have in mLDi()" ); }
}

private void mCMP( MIXWord register, long value, int start, int end, boolean sign)
```



```
{ long rValue = 0;
  int power = 0;

  try
  { for (int i=end; i>=start; i--)
    { rValue += ((long)(Math.pow( 64, power ))) * register.getValue( i );
      power++;
    }

    if (sign == true)
      rValue = (register.getSign() == '+') ? rValue : -rValue;

    if (rValue < value)
      cIndicator.setState( ComparisonIndicator.LESSTHAN );
    else
      if (rValue > value)
        cIndicator.setState( ComparisonIndicator.GREATERTHAN );
      else
        cIndicator.setState( ComparisonIndicator.EQUALTO );

    } catch (Exception e) { System.out.println(
      "An exception occurred when it shouldn't have in mCMP()" ); }
  }

private void mCMPi( MIXWord register, long value, int start, int end, boolean sign )
{ long rValue = 0;
  int power = 0;

  try
  { for (int i=end; i>=start; i--)
    { if (i>2)
      { rValue += ((long)(Math.pow( 64, power ))) * register.getValue( i-3 );
        power++;
      }

      if (sign == true)
        rValue = (register.getSign() == '+') ? rValue : -rValue;

      if (rValue < value)
        cIndicator.setState( ComparisonIndicator.LESSTHAN );
      else
        if (rValue > value)
          cIndicator.setState( ComparisonIndicator.GREATERTHAN );
        else
          cIndicator.setState( ComparisonIndicator.EQUALTO );

    } catch (Exception e) { System.out.println(
      "An exception occurred when it shouldn't have in mCMPi()" ); }
  }
}
```

## A.n MIXSign.java

```
//: MIXSign.java
// A component of a MIX1009 'word'

/** MIXSign simply holds either '+' or '-', denoting the sign of
 * a MIX1009 word
 * @author Andrew Doran
 * @author http://andrew.doran.com/
 * @version 0.11 - 30 March 1999
 */
class MIXSign
{
    private char sign;

    /** Sets up the initial state of the MIXSign.
     * @return no return value
     */
    MIXSign()
    {
        sign = '+'; // We always start with a positive sign
    }

    /** Sets the Sign to either '+' or '-'.
     * @param newSign char value of '+' or '-'.
     * @return No return value
     * @exception CharNotASignException Thrown if the character supplied is not a valid sign.
     */
    void setSign( char newSign ) throws CharNotASignException
    {
        if ( ( newSign != '+' ) && ( newSign != '-' ) )
            throw new CharNotASignException();
        else sign = newSign;
    }

    /** Used to retrieve the value of the sign.
     * @return char denoting current sign.
     */
    char getSign()
    {
        return sign;
    }
}

//::~~
```

## A.o MIXWord.java

```
//: MIXWord.java
// A MIX1009 'word' constructed from arbitrary MIXBytes and a MIXSign.
import java.awt.*;
import java.awt.image.*;

/** MIXWord is a basic unit of memory for the MIX1009.
 * @author Andrew Doran
 * @author http://andrew.doran.com/
 * @version 0.1 - 24 January 1999
 */
class MIXWord extends Canvas
{ private MIXSign    wordSign;
  private MIXByte[] wordBytes;
  private int        displayWidth;
  private static final Font SMLMONOSPACEDFONT = new Font( "Monospaced", Font.PLAIN, 12 );
  private static final Font BIGMONOSPACEDFONT = new Font( "Monospaced", Font.PLAIN, 16 );

  /** Standard word in the MIX1009 consists of five MIXBytes and a MIXSign.
   * @return No return value.
   */
  MIXWord()
  { this( 5 );
  }

  /** Can have a MIX1009 'word' constructed with an arbitrary number of MIXBytes and a MIXSign.
   * @param wordLength int value denoting number of bytes in the word.
   * @return No return value.
   */
  MIXWord( int wordLength )
  { wordSign = new MIXSign();
    wordBytes = new MIXByte[ wordLength ];
    for( int i = wordLength - 1; i >= 0; i-- )
      wordBytes[i] = new MIXByte();
    displayWidth = (( wordLength + 1 ) * 30) - 1;
    setSize( displayWidth, 29 );
  }

  /** Indicates whether the byte array index specified is out of range.
   * @param index int value of index, starting at zero for LSB.
   * @return boolean true if value specified is out of range.
   */
  boolean indexOutOfRange( int index )
  { if (( index < 0 ) || ( index > ( wordBytes.length -1 )))
    return true;
    return false;
  }

  /** Discovers if a certain byte in the word is packed.
   * @param index int value of index, starting at zero for LSB.
   * @return boolean value denoting status of 'packed' for byte.
   * @exception IndexOutOfRangeException Thrown if the byte referenced does not exist in the word.
   */
  boolean isPacked( int index ) throws IndexOutOfRangeException
  { if ( indexOutOfRange( index ) )
    throw new IndexOutOfRangeException();
    else return wordBytes[ index ].isPacked();
  }

  /** Sets the 'packed' status for an individual byte in the word.
   * @param index int value of index, starting at zero for LSB.
   * @param p boolean value denoting 'packed' status to be set.
   * @return No return value.
   * @exception IndexOutOfRangeException Thrown if the byte referenced does not exist in the word.
   */
  void setPacked( int index, boolean p ) throws IndexOutOfRangeException
  { if ( indexOutOfRange( index ) )
    throw new IndexOutOfRangeException();
    else wordBytes[ index ].setPacked( p );
  }
}
```

```
/** Retrieves the value of a specified byte in the word.
 * @param index int vale of index, starting at zero for LSB.
 * @return int value of specified byte.
 * @exception IndexOutOfRangeException Thrown if the byte referenced does not exist in the word.
 */
int getValue( int index ) throws IndexOutOfRangeException
{ if ( indexOutOfRange( index ) )
    throw new IndexOutOfRangeException();
  else return wordBytes[ index ].getValue();
}

/** Sets the value of a specified byte in the word.
 * @param index int value of index, starting at zero for LSB.
 * @param newValue int value between 0 and 63 inclusive.
 * @return No return value.
 * @exception IndexOutOfRangeException Thrown if the byte referenced does not exist in the word.
 * @exception ValueOutOfBoundsException Thrown if the value specified does not lie between
 *         0 and 63 inclusive.
 */
void setValue( int index, int newValue ) throws IndexOutOfRangeException,
                                                ValueOutOfBoundsException
{ if ( indexOutOfRange( index ) )
    throw new IndexOutOfRangeException();
  else wordBytes[ index ].setValue( newValue );
  repaint();
}

/** Sets the sign of the word.
 * @param newSign char value of desired sign.
 * @return No return value.
 * @exception CharNotASignException Thrown if the character supplied is not a valid sign.
 */
void setSign( char newSign ) throws CharNotASignException
{ wordSign.setSign( newSign );
  repaint();
}

/** Retrieves the sign of the word.
 * @return char denoting word sign.
 */
char getSign()
{ return wordSign.getSign();
}

public String toString()
{ long value = 0;
  int power = 0;

  for ( int i = 4; i>=0; i-- )
  { value += (long)(Math.pow( 64, power ) * wordBytes[ i ].getValue() );
    power++;
  }

  if (wordSign.getSign() == '-')
    value = -value;

  return ( "" + value );
}

public long toLong()
{ long value = 0;
  long power = 0;

  for ( int i = 4; i>=0; i-- )
  { value += (long)(Math.pow( 64, power ) * wordBytes[ i ].getValue() );
    power++;
  }

  if (wordSign.getSign() == '-')
    value = -value;

  return ( value );
}
```

```

public void paint( Graphics g )
{
    int currentByte = displayWidth,
        xValue      = 0,
        placeCounter = 0;
    long valueToShow = 0;

    boolean packedFlag = false;

    g.setColor( Color.white );
    g.fillRect( 0, 0, displayWidth, 29 );
    g.setColor( Color.black );
    g.drawRect( 0, 0, displayWidth, 28 );
    g.setFont( SMLMONOSPACEDFONT );

    for( int i = displayWidth - 30; i >= 0; i -= 30 )
    {
        currentByte = ((i+1)/30) - 1;
        if ( packedFlag == false )
            if ( !wordBytes[ currentByte ].isPacked() )
                // If the last Byte wasn't
                // packed and this one isn't
                // either...

                {
                    valueToShow = wordBytes[ currentByte ].getValue();
                    if ( ( "" + valueToShow ).length() == 2 )
                        // ...then draw it!
                        xValue = i+8;
                    else
                        xValue = i+12;
                    g.drawString( "" + valueToShow, xValue, 19 );
                }
            else
                // But if this one is...
                // ...we're joined to our
                // neighbour...
                {
                    placeCounter = 0;
                    // ...have a place value
                    // of 64^0...
                    valueToShow = wordBytes[ currentByte ].getValue();
                    // ...and must start to
                    // establish our value.
                    if ( currentByte == 0 )
                        // However, if this is the
                        // last digit we *do*
                        {
                            if ( ( "" + valueToShow ).length() == 2 )
                                // need to display our
                                // contents even if we are
                                // packed with the sign of
                                // the word.
                                xValue = i+8;
                            else
                                xValue = i+12;
                            g.drawString( "" + valueToShow, xValue, 19 );
                        }
                }
        }

    // UNLESS WE ARE ON BYTE ZERO...OTHERWISE NOTHING GETS DISPLAYED!
    else
        // If the last Byte was
        // packed...
        {
            if ( ( wordBytes[ currentByte ].isPacked() ) && (currentByte > 0) )
                // ...and this one
                // is too...
                {
                    placeCounter++;
                    // ...recognise this position
                    // is 'worth' more...
                    valueToShow += (long)(Math.pow( 64, placeCounter ) *
                        wordBytes[ currentByte ].getValue());
                    // ...and add it to our
                    // 'running total'.
                }
            else
                // But if this one isn't...
                {
                    placeCounter++;
                    // ...recognise this position
                    // is 'worth' more...
                    valueToShow += (long)(Math.pow( 64, placeCounter ) *
                        wordBytes[ currentByte ].getValue());
                    // ...add it to our 'running
                    // total'...
                    xValue = (((30 * (placeCounter+1)) - ( "" + valueToShow ).length() * 7)/2) + i;
                    g.drawString( "" + valueToShow, xValue, 19 );
                    // ...draw it...
                    packedFlag = false;
                    // ...and let the next Byte
                    // know we weren't packed.
                }
        }

    if ( wordBytes[ currentByte ].isPacked() )
    {
        g.drawLine( i, 0, i, 4 );
        g.drawLine( i, 25, i, 29 );
    }
    else

```

```
        g.drawLine( i, 0, i, 29 );

    }
    g.setFont( BIGMONOSPACEDFONT );
    g.drawString( "" + wordSign.getSign(), 10, 20 );
}

// Eliminates flicker (from Eckel p.686)
public void update( Graphics g )
{   paint(g);
}

}

///  
:~
```

## A.p NotAMIXCharacterException.java

```
//: NotAMIXCharacterException.java
//  An Exception thrown by LinePrinter

/** NotAMIXCharacterException is an Exception thrown by LinePrinter
 *   @author Andrew Doran
 *   @author http://andrew.doran.com/
 *   @version 0.11 - 30 March 1999
 *   @see LinePrinter
 */
class NotAMIXCharacterException extends Exception { }

//::~~
```

## A.q NotAValidStateException.java

```
//: NotAValidStateException.java
// An Exception thrown by ComparisonIndicator

/** NotAValidStateException is an Exception thrown by ComparisonIndicator
 * @author Andrew Doran
 * @author http://andrew.doran.com/
 * @version 0.11 - 30 March 1999
 * @see ComparisonIndicator
 */
class NotAValidStateException extends Exception { }

//:~
```



## A.r OverFlowIndicator.java

```
//: OverFlowIndicator.java
// Maintains the state of the 'overflow' flag in the MIX1009
import java.awt.*;

/** OverFlowIndicator maintains the state of the 'overflow' flag in the MIX1009.
 * @author Andrew Doran
 * @author http://andrew.doran.com/
 * @version 0.1 - 26 January 1999
 */
class OverFlowIndicator extends Canvas
{
    private boolean overFlow;
    private int width,
        height;
    private static final Font OFONT = new Font( "Monospaced", Font.PLAIN, 12 );

    /** Puts the indicator into an initial 'off' state.
     * @return No return value
     */
    OverFlowIndicator()
    {
        overFlow = false;
        width = 90;
        height = 35;
        setSize( width, height );
    }

    /** Sets the state of the overflow indicator.
     * @param newState where true == on, false == off.
     * @return No return value.
     */
    void setState( boolean newState )
    {
        if ( overFlow != newState )
        {
            overFlow = newState;
            repaint();
        }
    }

    /** Retrieves the state of the overflow indicator.
     * @return boolean where true == on, false == off.
     */
    boolean getState()
    {
        return overFlow;
    }

    public void paint( Graphics g )
    {
        g.setColor( Color.white );
        g.fillRect( 0, 0, width, height );
        g.setColor( Color.black );
        g.setFont( OFONT );
        g.drawString( "Overflow", 30, 15 );
        g.drawString( "toggle", 35, 26 );
        if ( overFlow )
        {
            g.setColor( Color.red );
            g.fillArc( 8, 8, 15, 15, 0, 360 );
            g.setColor( Color.black );
        }
        g.drawArc( 8, 8, 15, 15, 0, 360 );
        g.drawArc( 12, 12, 1, 1, 0, 360 );
        g.drawArc( 18, 12, 1, 1, 0, 360 );
        g.drawArc( 18, 18, 1, 1, 0, 360 );
        g.drawArc( 12, 18, 1, 1, 0, 360 );
    }

    // Eliminates flicker (from Eckel p.686)
    public void update( Graphics g )
    {
        paint(g);
    }
}

//::~~
```

## A.s ValueOutOfRangeException.java

```
//: ValueOutOfRangeException.java
//  An Exception thrown by MIXByte

/** ValueOutOfRangeException is an Exception thrown by MIXByte
 *   @author Andrew Doran
 *   @author http://andrew.doran.com/
 *   @version 0.11 - 30 March 1999
 *   @see MIXByte
 */
class ValueOutOfRangeException extends Exception { }

//:~
```

## Appendix B - Incorporated Program Code

This appendix contains code used by the Java applet's components given in Appendix A, above, and is included here for completeness.

### B.a ImageLabel.java

```
import java.awt.*;
import java.net.*;

// This appears in Core Web Programming from
// Prentice Hall Publishers, and may be freely used
// or adapted. 1997 Marty Hall, hall@apl.jhu.edu.

//=====
/**
 * A class for displaying images. It places the Image
 * into a canvas so that it can moved around by layout
 * managers, will get repainted automatically, etc.
 * No mouseXXX or action events are defined, so it is
 * most similar to the Label Component.
 * <P>
 * By default, with FlowLayout the ImageLabel takes
 * its minimum size (just enclosing the image). The
 * default with BorderLayout is to expand to fill
 * the region in width (North/South), height
 * (East/West) or both (Center). This is the same
 * behavior as with the builtin Label class. If you
 * give an explicit resize or
 * reshape call <B>before</B> adding the
 * ImageLabel to the Container, this size will
 * override the defaults.
 * <P>
 * Here is an example of its use:
 * <P>
 * <PRE>
 * public class ShowImages extends Applet {
 *     private ImageLabel imagel, image2;
 *
 *     public void init() {
 *         imagel = new ImageLabel(getCodeBase(),
 *                                 "some-image.gif");
 *         image2 = new ImageLabel(getCodeBase(),
 *                                 "other-image.jpg");
 *         add(imagel);
 *         add(image2);
 *     }
 * }
 * </PRE>
 *
 * @author Marty Hall (hall@apl.jhu.edu)
 * @see Icon
 * @see ImageButton
 * @version 1.0 (1997)
 */

public class ImageLabel extends Canvas {
    //-----
    // Instance variables.

    // The actual Image drawn on the canvas.
    private Image image;

    // A String corresponding to the URL of the image
    // you will get if you call the constructor with
    // no arguments.
```

```
private static String defaultImageString
    = "http://java.sun.com/lib/images/" +
      "logo.java.color-transp.55x60.gif";

// The URL of the image. But sometimes we will use
// an existing image object (e.g. made by
// createImage) for which this info will not be
// available, so a default string is used here.
private String imageString = "<Existing Image>";

// Turn this on to get verbose debugging messages.
private boolean debug = false;

/** Amount of extra space around the image. */
private int border = 0;

/** If there is a non-zero border, what color should
 *  it be? Default is to use the background color
 *  of the Container.
 */
private Color borderColor = null;

// Width and height of the Canvas. This is the
// width/height of the image plus twice the border.
private int width, height;

/** Determines if it will be sized automatically.
 *  * If the user issues a resize() or reshape()
 *  * call before adding the label to the Container,
 *  * or if the LayoutManager resizes before
 *  * drawing (as with BorderLayout), then those sizes
 *  * override the default, which is to make the label
 *  * the same size as the image it holds (after
 *  * reserving space for the border, if any).
 *  * This flag notes this, so subclasses that
 *  * override ImageLabel need to check this flag, and
 *  * if it is true, and they draw modified image,
 *  * then they need to draw them based on the width
 *  * height variables, not just blindly drawing them
 *  * full size.
 */
private boolean explicitSize = false;
private int explicitWidth=0, explicitHeight=0;

// The MediaTracker that can tell if image has been
// loaded before trying to paint it or resize
// based on its size.
private MediaTracker tracker;

// Used by MediaTracker to be sure image is loaded
// before paint & resize, since you can't find out
// the size until it is done loading.
private static int lastTrackerID=0;
private int currentTrackerID;
private boolean doneLoading = false;

private Container parentContainer;

//-----
/** Create an ImageLabel with the default image.
 *
 *  * @see #getDefaultImageString
 *  * @see #setDefaultImageString
 */
// Remember that the funny "this()" syntax calls
// constructor of same class
public ImageLabel() {
    this(defaultImageString);
}

/** Create an ImageLabel using the image at URL
 *  * specified by the string.
 *
 *  * @param imageURLString A String specifying the
```

```
*   URL of the image.
*/
public ImageLabel(String imageURLString) {
    this(makeURL(imageURLString));
}

/** Create an ImageLabel using the image at URL
 * specified.
 *
 * @param imageURL The URL of the image.
 */
public ImageLabel(URL imageURL) {
    this(loadImage(imageURL));
    imageString = imageURL.toExternalForm();
}

/** Create an ImageLabel using the image in the file
 * in the specified directory.
 *
 * @param imageDirectory Directory containing image
 * @param file Filename of image
 */
public ImageLabel(URL imageDirectory, String file) {
    this(makeURL(imageDirectory, file));
    imageString = file;
}

/** Create an ImageLabel using the image specified.
 * The other constructors eventually call this one,
 * but you may want to call it directly if you
 * already have an image (e.g. created via
 * createImage).
 *
 * @param image The image
 */
public ImageLabel(Image image) {
    this.image = image;
    tracker = new MediaTracker(this);
    currentTrackerID = lastTrackerID++;
    tracker.addImage(image, currentTrackerID);
}

//-----
/** Makes sure that the Image associated with the
 * Canvas is done loading before returning, since
 * loadImage spins off a separate thread to do the
 * loading. Once you get around to drawing the
 * image, this will make sure it is loaded,
 * waiting if not. The user does not need to call
 * this at all, but if several ImageLabels are used
 * in the same Container, this can cause
 * several repeated layouts, so users might want to
 * explicitly call this themselves before adding
 * the ImageLabel to the Container. Another
 * alternative is to start asynchronous loading by
 * calling prepareImage on the ImageLabel's
 * image (see getImage).
 *
 * @param doLayout Determines if the Container
 * should be re-laid out after you are finished
 * waiting. <B>This should be true when called
 * from user functions</B>, but is set to false
 * when called from preferredSize to avoid an
 * infinite loop. This is needed when
 * using BorderLayout, which calls preferredSize
 * <B>before</B> calling paint.
 */
public void waitForImage(boolean doLayout) {
    if (!doneLoading) {
        debug("[waitForImage] - Resizing and waiting for "
            + imageString);
        try { tracker.waitForID(currentTrackerID); }
        catch (InterruptedException ie) {}
        catch (Exception e) {

```

```
        System.out.println("Error loading "
                           + imageString + ": "
                           + e.getMessage());
        e.printStackTrace();
    }
    if (tracker.isErrorID(0))
        new Throwable("Error loading image "
                      + imageString).printStackTrace();
    doneLoading = true;
    if (explicitWidth != 0)
        width = explicitWidth;
    else
        width = image.getWidth(this) + 2*border;
    if (explicitHeight != 0)
        height = explicitHeight;
    else
        height = image.getHeight(this) + 2*border;
    resize(width, height);
    debug("[waitForImage] - " + imageString + " is "
          + width + "x" + height + ".");

    // If no parent, you are OK, since it will have
    // been resized before being added. But if
    // parent exists, you have already been added,
    // and the change in size requires re-layout.
    if (((parentContainer = getParent()) != null)
        && doLayout) {
        setBackground(parentContainer.getBackground());
        parentContainer.layout();
    }
}

//-----
/** Moves the image so that it is <I>centered</I> at
 * the specified location, as opposed to the move
 * method of Component which places the top left
 * corner at the specified location.
 * <P>
 * <B>Note:</B> The effects of this could be undone
 * by the LayoutManager of the parent Container, if
 * it is using one. So this is normally only used
 * in conjunction with a null LayoutManager.
 *
 * @param x The X coord of center of the image
 *          (in parent's coordinate system)
 * @param y The Y coord of center of the image
 *          (in parent's coordinate system)
 * @see java.awt.Component#move
 */

public void centerAt(int x, int y) {
    debug("Placing center of " + imageString + " at ("
          + x + ", " + y + ")");
    move(x - width/2, y - height/2);
}

//-----
/** Determines if the x and y <B>(in the ImageLabel's
 * own coordinate system)</B> is inside the
 * ImageLabel. Put here because Netscape 2.02 has
 * a bug in which it doesn't process inside() and
 * locate() tests correctly.
 */

public synchronized boolean inside(int x, int y) {
    return((x >= 0) && (x <= width)
           && (y >= 0) && (y <= height));
}

//-----
/** Draws the image. If you override this in a
 * subclass, be sure to call super.paint.
 */
public void paint(Graphics g) {
```

```
    if (!doneLoading)
        waitForImage(true);
    else {
        if (explicitSize)
            g.drawImage(image, border, border,
                        width-2*border, height-2*border,
                        this);
        else
            g.drawImage(image, border, border, this);
        drawRect(g, 0, 0, width-1, height-1,
                border, borderColor);
    }
}

//-----
/** Used by layout managers to calculate the usual
 * size allocated for the Component. Since some
 * layout managers (e.g. BorderLayout) may
 * call this before paint is called, you need to
 * make sure that the image is done loading, which
 * will force a resize, which determines the values
 * returned.
 */
public Dimension preferredSize() {
    if (!doneLoading)
        waitForImage(false);
    return(super.preferredSize());
}

//-----
/** Used by layout managers to calculate the smallest
 * size allocated for the Component. Since some
 * layout managers (e.g. BorderLayout) may
 * call this before paint is called, you need to
 * make sure that the image is done loading, which
 * will force a resize, which determines the values
 * returned.
 */
public Dimension minimumSize() {
    if (!doneLoading)
        waitForImage(false);
    return(super.minimumSize());
}

//-----
// LayoutManagers (such as BorderLayout) might call
// resize or reshape with only 1 dimension of
// width/height non-zero. In such a case, you still
// want the other dimension to come from the image
// itself.

/** Resizes the ImageLabel. If you don't resize the
 * label explicitly, then what happens depends on
 * the layout manager. With FlowLayout, as with
 * FlowLayout for Labels, the ImageLabel takes its
 * minimum size, just enclosing the image. With
 * BorderLayout, as with BorderLayout for Labels,
 * the ImageLabel is expanded to fill the
 * section. Stretching GIF/JPG files does not always
 * result in clear looking images. <B>So just as
 * with builtin Labels and Buttons, don't
 * use FlowLayout if you don't want the Buttons to
 * get resized.</B> If you don't use any
 * LayoutManager, then the ImageLabel will also
 * just fit the image.
 * <P>
 * Note that if you resize explicitly, you must do
 * it <B>before</B> the ImageLabel is added to the
 * Container. In such a case, the explicit size
 * overrides the image dimensions.
 *
 * @see #reshape
 */
public void resize(int width, int height) {
```

```
    if (!doneLoading) {
        explicitSize=true;
        if (width > 0)
            explicitWidth=width;
        if (height > 0)
            explicitHeight=height;
    }
    super.resize(width, height);
}

/** Resizes the ImageLabel. If you don't resize the
 * label explicitly, then what happens depends on
 * the layout manager. With FlowLayout, as with
 * FlowLayout for Labels, the ImageLabel takes its
 * minimum size, just enclosing the image. With
 * BorderLayout, as with BorderLayout for Labels,
 * the ImageLabel is expanded to fill the
 * section. Stretching GIF/JPG files does not always
 * result in clear looking images. <B>So just as
 * with builtin Labels and Buttons, don't
 * use FlowLayout if you don't want the Buttons to
 * get resized.</B> If you don't use any
 * LayoutManager, then the ImageLabel will also
 * just fit the image.
 * <P>
 * Note that if you resize explicitly, you must do
 * it <B>before</B> the ImageLabel is added to the
 * Container. In such a case, the explicit size
 * overrides the image dimensions.
 *
 * @see #resize
 */
public void reshape(int x, int y,
                    int width, int height) {
    if (!doneLoading) {
        explicitSize=true;
        if (width > 0)
            explicitWidth=width;
        if (height > 0)
            explicitHeight=height;
    }
    super.reshape(x, y, width, height);
}

//-----
// You can't just set the background color to
// the borderColor and skip drawing the border,
// since it messes up transparent gifs. You
// need the background color to be the same as
// the container.

/** Draws a rectangle with the specified OUTSIDE
 * left, top, width, and height.
 * Used to draw the border.
 */
protected void drawRect(Graphics g,
                        int left, int top,
                        int width, int height,
                        int lineThickness,
                        Color rectangleColor) {
    g.setColor(rectangleColor);
    for(int i=0; i<lineThickness; i++) {
        g.drawRect(left, top, width, height);
        if (i < lineThickness-1) { // Skip last iteration
            left = left + 1;
            top = top + 1;
            width = width - 2;
            height = height - 2;
        }
    }
}

//-----
/** Calls System.out.println if the debug variable
```



```
* is true; does nothing otherwise.
*
* @param message The String to be printed.
*/
protected void debug(String message) {
    if (debug)
        System.out.println(message);
}

//-----
// Creates the URL with some error checking.

private static URL makeURL(String s) {
    URL u = null;
    try { u = new URL(s); }
    catch (MalformedURLException mue) {
        System.out.println("Bad URL " + s + ": " + mue);
        mue.printStackTrace();
    }
    return(u);
}

private static URL makeURL(URL directory,
                           String file) {
    URL u = null;
    try { u = new URL(directory, file); }
    catch (MalformedURLException mue) {
        System.out.println("Bad URL " +
                           directory.toExternalForm() +
                           ", " + file + ": " + mue);
        mue.printStackTrace();
    }
    return(u);
}

//-----
// Loads the image. Needs to be static since it is
// called by the constructor.

private static Image loadImage(URL url) {
    return(Toolkit.getDefaultToolkit().getImage(url));
}

//-----
/** The Image associated with the ImageLabel. */

public Image getImage() {
    return(image);
}

//-----
/** Gets the border width. */

public int getBorder() {
    return(border);
}

/** Sets the border thickness. */

public void setBorder(int border) {
    this.border = border;
}

//-----
/** Gets the border color. */

public Color getBorderColor() {
    return(borderColor);
}

/** Sets the border color. */

public void setBorderColor(Color borderColor) {
    this.borderColor = borderColor;
}
```

---

```
}

//-----
// You could just call size().width and size().height,
// but since we've overridden resize to record
// this, we might as well use it.

/** Gets the width (image width plus twice border). */

public int getWidth() {
    return(width);
}

/** Gets the height (image height plus 2x border). */

public int getHeight() {
    return(height);
}

//-----
/** Has the ImageLabel been given an explicit size?
 * This is used to decide if the image should be
 * stretched or not. This will be true if you
 * call resize or reshape on the ImageLabel before
 * adding it to a Container. It will be false
 * otherwise.
 */
protected boolean hasExplicitSize() {
    return(explicitSize);
}

//-----
/** Returns the string representing the URL that
 * will be used if none is supplied in the
 * constructor.
 */
public static String getDefaultImageString() {
    return(defaultImageString);
}

/** Sets the string representing the URL that
 * will be used if none is supplied in the
 * constructor. Note that this is static,
 * so is shared by all ImageLabels. Using this
 * might be convenient in testing, but "real"
 * applications should avoid it.
 */
public static void setDefaultImageString(String file) {
    defaultImageString = file;
}

//-----
/** Returns the string representing the URL
 * of image.
 */
protected String getImageString() {
    return(imageString);
}

//-----
/** Is the debugging flag set? */

public boolean isDebugging() {
    return(debug);
}

/** Set the debugging flag. Verbose messages
 * will be printed to System.out if this is true.
 */
public void setIsDebugging(boolean debug) {
    this.debug = debug;
}

//-----
```

}



## Appendix C - HTML code

### C.a MIXApplet.html

```
<HTML>
<HEAD></HEAD>
<BODY>

<applet code="MIX1009" width=380 height=220>
</applet>

</BODY>
</HTML>
```



# Appendix D - MIXAL Programs

## D.a Program P

Mem	Assembled Instruction					LOC	OP	ADDRESS
* EXAMPLE PROGRAM ... TABLE OF PRIMES								
*								
						L	EQU	500
						PRINTER	EQU	18
						PRIME	EQU	-1
						BUF0	EQU	2000
						BUF1	EQU	BUF0+25
							ORIG	3000
3000 :	+	0	0	18	35	START	IOC	0 (PRINTER)
3001 :	+	2050	0	5	9		LD1	=1-L=
3002 :	+	2051	0	5	10		LD2	=3=
3003 :	+	1	0	0	49	2H	INC1	1
3004 :	+	499	1	5	26		ST2	PRIME+L,1
3005 :	+	3016	0	1	41		J1Z	2F
3006 :	+	2	0	0	50	4H	INC2	2
3007 :	+	2	0	2	51		ENT3	2
3008 :	+	0	0	2	48	6H	ENTA	0
3009 :	+	0	2	2	55		ENTX	0,2
3010 :	-	1	3	5	4		DIV	PRIME,3
3011 :	+	3006	0	1	47		JXZ	4B
3012 :	-	1	3	5	56		CMPA	PRIME,3
3013 :	+	1	0	0	51		INC3	1
3014 :	+	3008	0	6	39		JG	6B
3015 :	+	3003	0	0	39		JMP	2B
3016 :	+	1995	0	18	37	2H	OUT	TITLE (PRINTER)
3017 :	+	2035	0	2	52		ENT4	BUF1+10
3018 :	-	50	0	2	53		ENT5	-50
3019 :	+	501	0	0	53	2H	INC5	L+1
3020 :	-	1	5	5	8	4H	LDA	PRIME,5

## An Implementation of Donald Knuth's MIX

Mem	Assembled Instruction					LOC	OP	ADDRESS
3021 :	+	0	0	1	5		CHAR	
3022 :	+	0	4	12	31		STX	0,4(1:4)
3023 :	+	1	0	1	52		DEC4	1
3024 :	+	50	0	1	53		DEC5	50
3025 :	+	3020	0	2	45		J5P	4B
3026 :	+	0	4	18	37		OUT	0,4(PRINTER)
3027 :	+	24	4	5	12		LD4	24,4
3028 :	+	3016	0	0	45		J5N	2B
3029 :	+	0	0	2	5		HLT	

\* INITIAL CONTENTS OF TABLES AND BUFFERS

0000 :	+	2					ORIG	PRIME+1
							CON	2
							ORIG	BUF0-5
1995 :	+	6	9	19	22	23	TITLE	ALF FIRST
1996 :	+	0	6	9	25	5		ALF FIVE
1997 :	+	0	8	24	15	4		ALF HUND
1998 :	+	19	5	4	0	17		ALF RED P
1999 :	+	19	9	14	5	22		ALF RIMES
							ORIG	BUF0+24
2024 :	+	2035					CON	BUF1+10
							ORIG	BUF1+24
2049 :	+	2010					CON	BUF0+10
2050 :	+	499						
2051 :	+	3						
							END	START



## D.b Program M

Mem	Assembled Instruction					LOC	OP	ADDRESS
						X	EQU	1000
							ORIG	3000
3000 :	+	3009	0	2	32	MAXIMUM	STJ	EXIT
3001 :	+	0	1	2	51	INIT	ENT3	0,1
3002 :	+	3005	0	0	39		JMP	CHANGEM
3003 :	+	1000	3	5	56	LOOP	CMPA	X,3
3004 :	+	3007	0	7	39		JGE	*+3
3005 :	+	0	3	2	50	CHANGEM	ENT2	0,3
3006 :	+	1000	3	5	8		LDA	X,3
3007 :	+	1	0	1	51		DEC3	1
3008 :	+	3003	0	2	43		J3P	LOOP
3009 :	+	0	0	2	5		HLT	