The Most Complex Machine:

Lab Manual for Macintosh

(July 1995)

David J. Eck

Department of Mathematics and Computer Science
Hobart and William Smith Colleges
Geneva, New York 14456

Contents

Lab Number 0 for

The Most Complex Machine

by David J. Eck

Introduction to the Macintosh

About this lab: This is an introductory lab which is meant mostly to introduce you to the use of the Macintosh computer. The Macintosh is designed to be easy to use, and the programs you will use in this course are written to require minimal computer expertise. Even if you have never used a computer before, this lab will teach you most of the general facts you need to know in order to feel comfortable doing the rest of the labs. If you already know how to use a Macintosh computer, you probably won't learn much from this introductory lab; however, there are several good reasons for you to quickly work through the lab anyway. First, you might pick up a few useful techniques you didn't know before. Second, you need to know how to access the software you will use in the labs. And finally, xSimpleEdit—the program used in this lab—will familiarize you with some features that it shares with other programs that you will use in later labs.

You should have a floppy disk containing the software used in the labs, or, if the software is already stored on the computer system you will be using, you should know where to find it. It will be useful, though not essential, to have read Chapter 1 of *The Most Complex Machine* in order to obtain an overview of how computers really work.

This lab and the software used in it accompany *The Most Complex Machine: A Survey of Computers and Computing*, an introductory computer science textbook by David Eck (published 1995 by AK Peters, Ltd., 289 Linden Street, Wellesley, Massachusetts 02181; ISBN 1-56881-054-7). Support for the text, software and labs is available on the World Wide Web at http://math.hws.edu/TMCM.html. David Eck can be reached at the e-mail address eck@hws.edu.

You should *always* read through any lab worksheet in advance—not only will this let you do some planning, but it will often make the lab easier to do and more meaningful. (Also, you can make note of things you need to do at the computer; such things are usually written in slanted type, *like this.*) This is especially important for this lab, since in this case if you don't read the lab in advance you will spend most of your time at the computer just reading it.

Getting Started: Most of the remaining labs in this book begin by asking you to start up some specific program. It is assumed that you know where to find the programs and how to start them. How you do so will depend on the exact situation in which you are doing the labs. There are two major possibilities: Either you carry the programs around with you on your own floppy disk, or the programs are installed on the computer system you are using. I will describe the general procedure, but there are many possible variations, especially if you are using a computer in a computer-equipped classroom or laboratory. In that case, you might need to find out about local customs. (If you are using the book as part of a course, you will undoubtably be told what you need to know; if by some chance you are working on your own, ask anyone who looks like they know what they are doing.)

First of all, you need to know how to turn your computer on and off. Many models of Macintosh can be turned on by pressing a button in the upper right corner of the keyboard. (In some cases, it might be necessary to turn on the monitor with a separate switch.) These computers can be turned off simply by selecting the command **Shut Down** from the **Special** menu. If you don't know anything about menus, don't worry. You will by the end of this lab.

Other Macintosh models have a power switch located on the front or back of the machine. (There might be a separate power switch for the monitor.) The power switch can be used to turn the computer on; to turn it off, you should first use the **Shut Down** command from the **Special** menu and *only then* turn off the power. It is important that you *do not* simply turn off the power without using the **Shut Down** command first.

Many computers have a program called a ***screen saver*** installed that will either dim the screen or display some changing image after the computer has been left turned on but idle for some time. This is supposed to prevent damage to the screen that can result from displaying the same image for too long. If the screen saver has been activated, you can deactivate it by moving the mouse.

When you turn on the computer, the screen will display a work area called the ***desktop***, similar to the one shown in Figure 0.1. The desktop contains

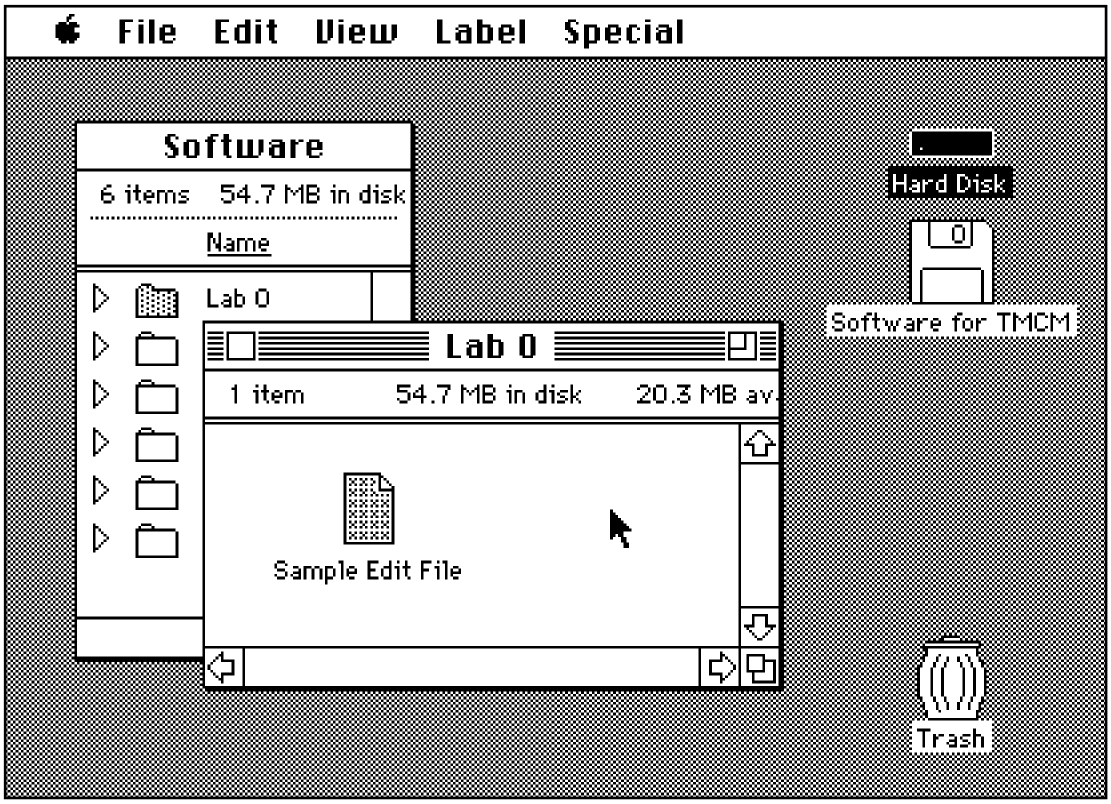


Figure 0.1. *A Macintosh desktop, showing several icons, two open windows and the menu bar. The arrow, which is called the* **cursor**, *is used to point to a location on the screen. Its position is controlled by moving the mouse.*

icons, which are pictorial symbols for information stored on the computer. An icon can represent a program, a document (also called a ***file*** or ***data file***), a ***folder***, or a ***disk***. There is also an icon called the ***trash*** which I will talk about later. Each icon has a name, which is displayed under the icon.

A folder is a container that can hold programs, documents and other folders. A disk can also hold programs, documents and folders. A disk is different from a folder because it is an actual physical object, while a folder is merely a collection of information stored somewhere on a disk. In addition to icons, the desktop can display ***windows***. Windows are used by programs to display information. There are also "directory windows" that are used to show what information is stored in a folder or disk. Such directory windows are the only ones you will see when there is no program running.

Finally, across the top of the screen is the ***menu bar***. The menu bar contains several individual menus, with names such as **File** and **Edit**. Each menu contains a selection of commands that can be accessed using the mouse. The apple on the left of the menu bar also represents a menu, called the **Apple** menu. There will probably be other menus represented by single symbols on the far right of the menu bar. When you run a program, it will have its own menu bar containing menus and commands appropriate for that program. The program's menu bar replaces the usual desktop menu bar when the program is running.

Mousing Around: Now, it is finally time to do something with the computer. To use a Macintosh, you need to learn how to use the mouse. The mouse is used to select items, to open them and to drag them around. It is also used to choose commands from a menu. There are a lot of different things to learn here, but most of them are very natural. If you have previous experience with Macintosh computers, you should already know everything in this section. If you have used the Windows system on IBM-type computers, you know most of it, but there are a few differences that you will have to get used to.

To ***select*** a disk or folder icon or a particular window, just point at it with the mouse and click the mouse button. The item you select will be ***hilited*** to indicate that it is selected. Clicking is also used for other purposes. For example, if you click on the small square box found in the upper left corner of a window, that window will close.

You can move an object around by ***dragging*** it. That is, point the mouse at it, press the mouse button and, *while continuing to hold the button down*, move the mouse; release the button when the mouse is at the position you want. If it's a window you want to drag, you must grab it by the ***title bar*** that runs along the top of the window. *Try selecting and dragging the disk icon at the upper right corner of your computer screen.*

You can move an item from one folder or disk to another by dragging the item and dropping it on the folder or disk to which you want to move it. If you drag something that's stored on one disk to another disk or to a folder that is stored on another disk, it will be copied instead of moved. If you drop a folder, document or program on the trash icon, it will be thrown away. (If you do this accidentally, you can open the trash by double-clicking it and drag the item back out of the trash; however, once the trash has been emptied, you can no longer get it back.) If you drop a floppy disk icon in the trash, the disk will merely be ejected from the computer and removed from the desktop; it will *not* be erased. In fact, this—and not the **Eject Disk** command in the **Special** menu—is the proper way to remove a disk.

You can ***open*** something by ***double-clicking*** on it. That is, point the mouse at it and quickly click the mouse button two times. If you double-click a disk or folder icon, a window will open showing you the contents of the disk or folder. The window can be closed as described above. *Try opening and closing some disks and folders. Try dragging things from one folder to another, then dragging them back. Try throwing something in the trash, then dragging it out again.*

To choose a menu command, point the mouse at the menu name in the menu bar. Press the mouse button and *hold it down.* A menu of commands will appear below the menu bar. While continuing to hold down the mouse button, move the mouse to the command you want. The command that the mouse is pointing at will be hilited (unless it is not legal at the moment). When you release the mouse button, the hilited command, if any, will be executed. *Try choosing the* **About This Macintosh** *command from the* **Apple** *menu.* (Recall that the **Apple** menu is indicated by the small apple symbol at the left end of the menu bar.) This will open a small window containing some information about the computer you are using.

If you double-click on a program icon, that program will be started up. If you double-click on a document icon, the program that created that file will open and, usually, a window will be opened showing the document. When a program opens, its own menu bar will appear on the top of the screen. In almost all cases, there will be a **File** menu containing a **Quit** command that can be used to terminate the program. Knowing this much, you should feel free to try starting up some programs to see what happens. Usually, there will be things for you to click on and commands for you to try. Generally, the first command in the **Apple** menu will be called "**About** something or other." This command might give some useful information about the program. Also, look for **Help** commands in the **Apple** menu or in other menus.

Disks and Software: To do the labs in this manual, you need access to certain programs and data files. You might have these programs and data files on a floppy disk, or they might be installed on the computer you will be using. I will assume that you know where to find this software. (If you have the software on a floppy disk, insert that disk into the slot on the front of the computer. An icon for the disk will appear on the desktop. Double-click on the icon to open it; it might actually open automatically when you insert the desk. If the software is installed on your computer, you might have to open one or more folders on the disk to find it.)

If you have the software only on a floppy disk, you should make a copy and keep the original as a backup. If you have your own computer, you can copy the software onto your computer's hard disk, if it has one. Otherwise, you should copy it to another floppy disk. If you don't know how to do this, ask someone or look it up in a Macintosh computer manual.

If you are using software installed on a computer that you don't own, you will need a floppy disk on which you can keep any files that you create during the labs. If you need to buy a disk, buy a "3-1/2 inch double-sided, double-density" disk. (Don't buy a "high-density" disk unless you know the computer you will be using can handle it—all newer Macintoshes can, but a Mac SE or a Mac Plus generally cannot.)

A new disk must be ***formatted*** before it can be used. When you insert an unformatted disk into the computer, a box will appear telling you that it needs to be initialized. What you need to do will be obvious, except that in the case of a double-density disk, there will be two buttons labeled **Single-sided** and **Double-sided**. You should click on the button labeled **Double-sided**. *Warning*: Do not initialize a disk that you have previously used on an IBM-type computer unless you want to throw away all the data on that disk; a disk can contain IBM data or Macintosh data but not both at the same time.

A Simple Program: For this lab, you need a program named *xSimpleEdit* and a data file named "Sample Edit File." *Find the file Sample Edit File and open it.* There are two ways to do this. If you double-click on the Sample Edit File icon, then the program *xSimpleEdit* will start up and will open the file. (The Sample Edit File was created using the program *xSimpleEdit*. The Macintosh remembers this fact, and when you double-click on the file, it starts up the program that created it and tells that program to open the file.) Alternatively, you can start up the program directly by double-clicking on the program icon. Once the program is started, you can use the **Open** command from the **File** menu to open a data file. The **Open** command is discussed in more detail below.

xSimpleEdit is a very simple text-editing program. It allows you to type in and edit text, to save the text as a file and to send the text to a printer. A similar type of text editing will be used in several programs in later labs. Although *xSimpleEdit* is quite limited and is very far from being useful as a word-processing program, it does show many of the features common to Macintosh programs. You will use it to learn about these features.

When you open the file, a window appears on the screen that looks much like the one in Figure 0.2. This window displays the information from the file. In this case, the window contains information about ***scroll bars***, the ***zoom box***, the ***grow box***, the ***title bar*** and the ***close box***. *You should read all this information and experiment with each of these features.* If you do what the text in the window tells you to do, you will end up by closing the window.

Next, you should work with *xSimpleEdit* to learn the basics of text-editing and of using menus. Almost every Macintosh program has a **File** menu and an **Edit** menu. The commands in these menus can vary somewhat from program to program, but some commands are used in almost all programs. The **File** and **Edit** menus for *xSimpleEdit* are shown in Figure 0.3.

Try selecting the **New** *command from the* **File** *menu.* The **New** command opens up a new, empty window into which you can type some text. Notice the letter "N" to the right of the word "New" in this menu. This letter is a ***command-key equivalent*** for the **New** command. This means

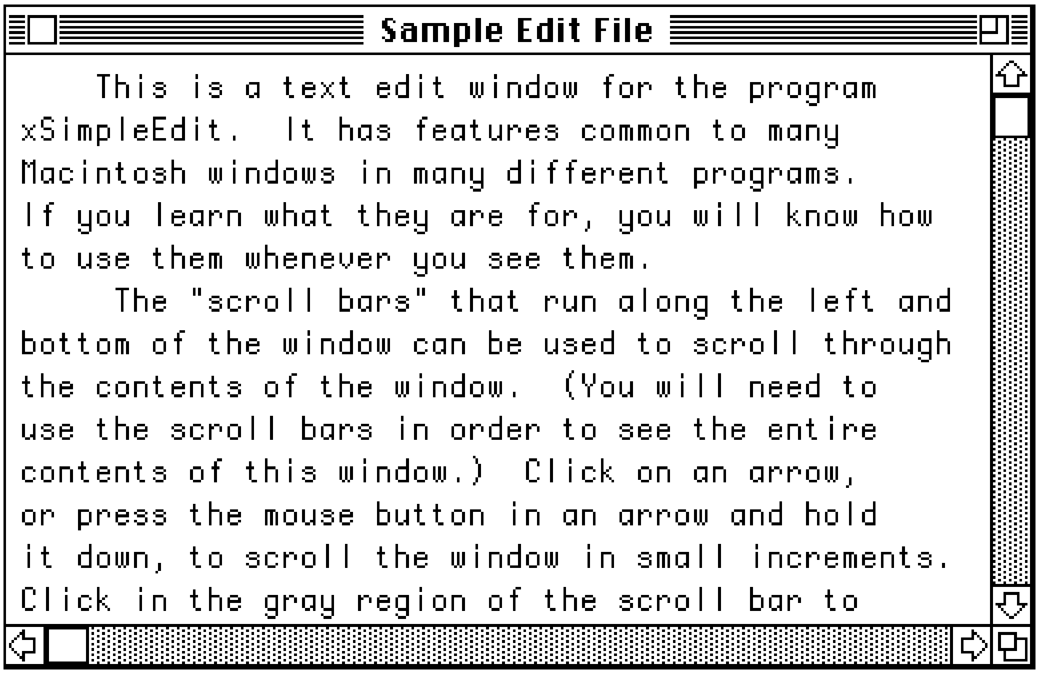


Figure 0.2. *A typical Macintosh window containing some text to be edited. You can type text into such a window. You can then edit the text, print it, or save it as a file.*

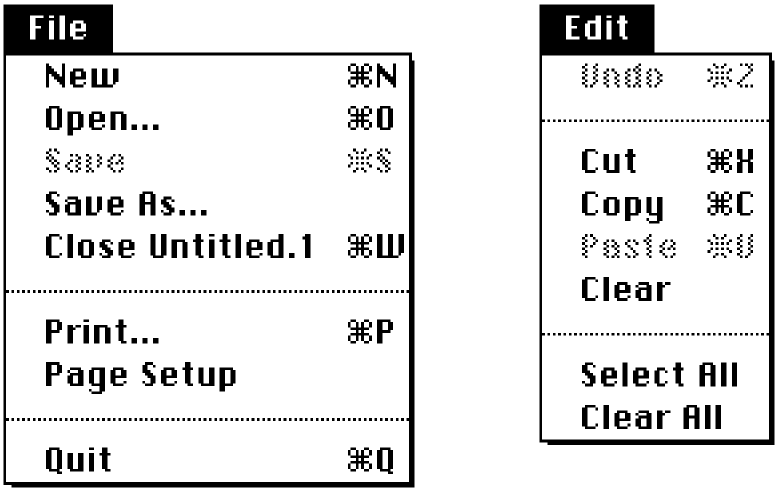


Figure 0.3. *The File and Edit menus used in the program xSimpleEdit. The commands in these menus are available in many Macintosh programs.*

that instead of selecting the command from the menu, you can simply hold down the command key on the keyboard and type the letter "N." (The command key is the key with an apple and/or a cloverleaf symbol on it.) You should learn how to read menus to find out the command-key equivalents for common commands, since using them is easier than using the mouse to select the commands. (This is especially true for the **Cut**, **Copy** and **Paste** operations in the **Edit** menu.)

Text editing is easy on the Macintosh. As you type characters, they appear on the screen at the position of a blinking ***insertion point***. You can move the insertion point with the arrow keys or with the mouse. (To use the mouse, just click at the position where you want the insertion point.) The delete key will erase the character to the left of the insertion point.

You can hilite a range of text by dragging the mouse across it. That is, press the mouse button at some position in the text, then hold the button down while moving the mouse. Text will be hilited as you move the mouse. If you move the mouse outside the window, the text in the window will scroll. Release the mouse button when you are done. Once the text is hilited, you can use the **Cut**, **Copy** and **Clear** commands from the **Edit** menu.

The **Cut** command will remove the hilited text from the window. However, it will save a copy of the text in a special place called the ***Clipboard***. Once some text is in the Clipboard, you can use the **Paste** command to place a copy of that text at the current insertion point. Thus, you can move text from one place to another by cutting it from one place and then pasting it at a different place. (If some text is hilited when you use the paste command, the text from the Clipboard will *replace* the hilited text.)

You can cut and paste from one window to another and even from one program to another. After you paste some text, a copy of it is still in the Clipboard, so that you can paste the same text into several different places if you want. The **Copy** command is similar to the **Cut** command except that it does not remove the hilited text; it merely puts a copy of it in the Clipboard so that it can be pasted elsewhere.

The **Clear** command will remove hilited text from a window without putting a copy of it in the Clipboard. Hitting the delete key while some text is hilited will have the same effect.

The **Undo** command is used to cancel the effect of the most recent command or typing. For example, it might be used to "un**Clear**" text that you have unintentionally erased. Once the undo command has been applied, the command in the menu changes to **Redo**, which can be used to "undo the undo."

Try typing and editing some text, including opening a second window and copying text from one window to another. (The exercises at the end of the lab will give you more opportunities to do some text editing.)

The **Open**, **Save** and **Save As** commands in the **File** menu allow you to work with files saved on a disk. **Open** allows you to open a file that already exists. A new window is opened in which the contents of the file are displayed. When you edit the contents of that window, you don't change the file itself—just the copy of the file's contents that is displayed in the window.

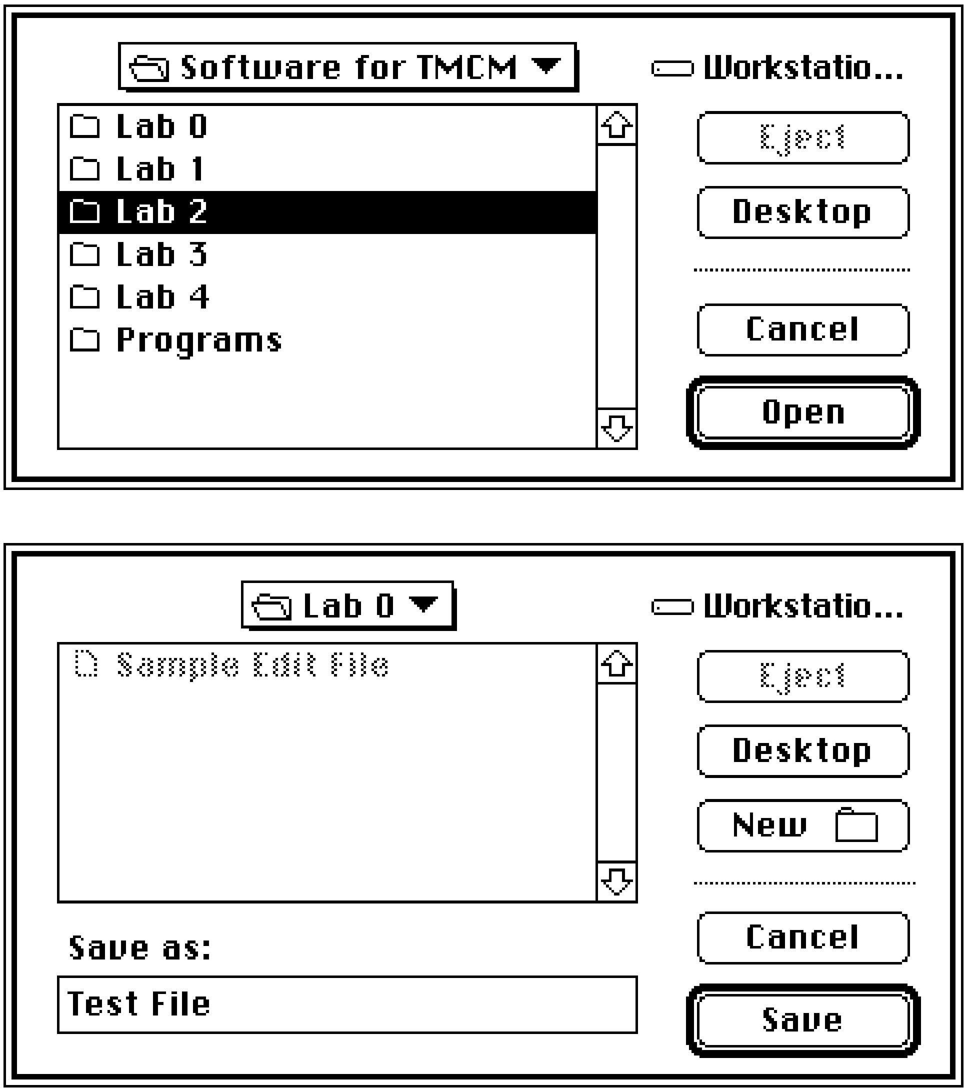


Figure 0.4. *Dialog boxes for opening and saving files. The boxes you see on your computer might look different from these if your Macintosh is running "System 6" instead of "System 7," but the only real difference is the method used to move from one disk to another. In the System 7 boxes as shown, you can click on the Desktop button and then double-click on the name of the disk you want to select. The System 6 boxes contain a Drive button instead. Clicking on this will move you from one disk to another; keep clicking on it until you get to the disk that you want.*

To make the changes apply to the file itself, use the **Save** command. If you want to store the changed text in a new file, leaving the original unaltered, use the **Save As** command. Remember that the changes that you make don't become permanent until you save them! (Note that the first time you apply **Save** to a window that you created with the **New** command, a new file will be created, just as if you had used the **Save As** command instead.)

When you use **Open**, you will see a ***dialog box*** that allows you to choose the file you want to open. A dialog box is a special type of window that the

Macintosh uses to ask you a question or allow you to enter information. Once a dialog box is on the screen, you have to do something to make it go away before you can do anything else. (Often there is a Cancel button that you can click to make the box go away and cancel any operation in progress; for example, when using **Open**, click the cancel button if you decide you don't want to open a file after all.) When you use **Save As**—or **Save** on a new window—you get a dialog box that allows you to name the new file you are creating and to specify the folder in which it should be stored. The **Open** and **Save As** dialog boxes are shown in Figure 0.4.

At the top of each dialog box is the name of a folder or disk. Beneath the name is a scrolling list of the items stored in that folder or disk. (In the case of the **Open** command, a file will appear in this list only if the program knows how to open it; for *xSimpleEdit*, only files containing text will appear.) If you double-click on any folder in this list, that folder will be opened and will replace the name at the top of the dialog box. The name at the top is actually a ***pop-up menu***, which can be used to move back up to folders or disks containing the current folder or disk. (Point at the name with the mouse and press and hold the mouse button; it should be obvious what to do.)

When opening a file, locate the file you want and then double-click on its name or click on the Open button. *Try opening a file or two.* Feel free to explore the disk and open files that look interesting.

When saving a file, you should make sure that the folder or disk at the top of the dialog box is the place where you want the file to go. (If you don't know which folder the file is saved in, you might not be able to find it again!) Type a name for the file into the rectangle at the bottom of the dialog box. You might have to click in the rectangle first, if you don't see the blinking insertion point there. (Hint: Whenever some text is hilited, if you just start typing, the hilited text will be deleted and replaced by what you type; there is no need to get an insertion point and erase the text by hand.) After typing the name, click on the Save button or just press the return key. (Another hint: Whenever a button has a heavy border around it, pressing the return key is equivalent to clicking on the button.) *Try entering some text in a new window, saving that text in a file, and then reopening that file.*

The other thing that you can do with the text in a window is print it. Usually, this just requires choosing the **Print** command from the **File** menu and then clicking on the Print button in the dialog box that appears on the screen. *Try this.*

However, there can be some complications. The print dialog box contains some options you can set (although some of these will have no effect in *xSimpleEdit*). Further options are available in a Page Setup dialog box, which you will see if you choose the **Page Setup** command; if you want to use this command, you must do so before you print. Finally, before you print successfully, your computer must be connected to a printer and that printer must be "selected." You can select a printer using the **Chooser** command, which should be available in the **Apple** menu. If you need to use the Chooser and don't know how to do so, ask someone or look in your computer manual.

xSimpleEdit has two additional menus, a **Font** menu and a **Windows** menu. When you have several windows open, the **Windows** menu can be used to move easily from one window to another. (Of course, you can also move to a new window by clicking on it, but that only works if at least some part of the window is visible.) Many of the programs you will use in later labs have a **Windows** menu.

The **Font** menu controls the appearance of the text displayed in a window. It is divided into two parts. The upper part controls the size of the text. (The size is specified in "points." The more points, the bigger the text.) The current size of the text is checked; when you choose a new size, the check mark will move. The bottom half of the **Font** menu controls the font, that is, the style of character used to display the text. You can try out different fonts and different point sizes. Although the other programs you will use do not have a **Font** menu, many of them do have similar menus that you will use to select one option from among several.

There was a lot to learn in this lab, especially if you have not used the Macintosh before. You should continue to work until you feel reasonably comfortable with the computer and with the *xSimpleEdit* program. Also, try running other programs that are available to you. You should find that you can figure out a lot about how to use them, since most Macintosh programs work in similar ways.

Exercises: This lab ends with a number of exercises for you to work on, think about and write up. You should do them *after* you have gained experience with Macintosh computers by working through the lab. Some of them require further work on the computer. Your responses might be part of a lab report, if you are doing this lab as part of a course.

Exercise 1: *Look at the commands for the* **File**, **Edit** *and* **Windows** *menus in xSimpleEdit. Sometimes the commands are active, and sometimes they are grayed out and therefore unavailable. Try to figure out the conditions under which each command is grayed out, and write up your observations.*

Exercise 2: *Make a file—not just a window!—containing the line "This is a stupid file" repeated 128 times. You only have to type the line once, and then use* **Copy** *and* **Paste** *commands. Explain how you made the file. What is the smallest number of* **Copy** *and* **Paste** *commands needed to make the file? (You need far fewer than 128.)*

Exercise 3: *Explain in detail how you could make a new file containing just the last paragraph from the file Sample Edit File used in this lab (without retyping the paragraph!). There are at least two essentially different ways to do this. Try to think of both of them.*

Exercise 4: *The Macintosh is designed to have a* ***consistent user interface****. This means that most programs work in the same way, so that much of what you learn about one program also applies to other programs. Discuss several examples of this that you have seen during your exploration of the Macintosh in this lab. Why is a consistent user interface is useful?*

Lab Number 1 for

The Most Complex Machine

by David J. Eck

Data Representation

About this Lab: All the different types of information in a computer are represented by strings of just two basic symbols: zeros and ones. The same string of zeros and ones can represent many different items of data—it all depends on how it is interpreted. In this lab, you will see the same string of thirty-two bits interpreted in six different ways. You will also learn something more about the "feel" of using a Macintosh computer program.

The topic of data representations is covered in Section 1.1 of *The Most Complex Machine*. The relevant material is reviewed here, but it would be very useful to have read that section for more perspective about what is going on.

Background: Bits, Bytes, Etc: The fundamental unit of information is a ***bit***. A bit is a quantity that can take on either of the two values `zero` or `one`. By taking on one of these two values, it can represent "information" by answering a single yes-or-no question, with the value `one` meaning "yes" and `zero` meaning "no." Of course, for it to *really* represent

---

This lab and the software used in it accompany *The Most Complex Machine: A Survey of Computers and Computing*, an introductory computer science textbook by David Eck (published 1995 by AK Peters, Ltd., 289 Linden Street, Wellesley, Massachusetts 02181; ISBN 1-56881-054-7). Support for the text, software and labs is available on the World Wide Web at http://math.hws.edu/TMCM.html. David Eck can be reached at the e-mail address eck@hws.edu.

information in the usual sense, you have to know what question is being answered. The "question that is being answered" is the ***interpretation*** of the bit. The same bit can have almost any meaning, depending on how it is interpreted.

By stringing together a sequence of bits, we can represent more than just two possible values. A string of two bits can represent four possible values, corresponding to the bit-patterns 00, 01, 10, and 11. The values might be the numbers 0, 1, 2, and 3; or they might be the four different things that can occupy a square in a game of checkers: red piece, black piece, red king and black king; or, they might have any one of indefinitely many other interpretations. Again, the meaning depends on the interpretation.

A string of eight bits is called a ***byte***. A byte has 2^8—that is 256—different bit-patterns. These 256 different bit-patterns can be used to encode 256 different values. In one common application, the bit-patterns are used as codes for letters, digits and other characters. Although such an encoding could be done in many ways, the most common association of bit-patterns to characters is the ***ASCII code***, which is used in most computers.

By stringing together bits (and keeping their interpretation straight!) we can represent any data that the computer might have to deal with. In this lab, you will look at just six of the possible interpretations of a string of thirty-two bits. The interpretations considered here are, in a sense, built into the design of the computer itself. For example, the computer can—under the direction of a program—take a thirty-two bit value from a certain memory location and treat it as an ***integer***, that is a positive or negative whole number. The computer can do things with that string of bits that are appropriate things to do with an integer, such as add it to another number or divide it by seventeen. It can also display the integer on its screen in standard "human-readable" form.

The six interpretations you will see for a string of thirty-two bits are: a binary number, an integer, a hexadecimal number, a real number, a string of four characters and an eight-by-four grid of pixels. You should remember that you will be looking at *the same* string of thirty-two bits interpreted in different ways. You should also remember that the same bit-patterns could be interpreted in other programs in an endless variety of ways: as a bar of music, or the chemical ingredients in a bar of soap, or your tab at your favorite bar, or….

The Program: To begin the lab, start up the program *Data Reps*, which you will find in the folder named "Files for Lab 1." You will see a window like that in Figure 1.1, but with different values in the rectangular boxes. This is a very simple program, with just one functional command—**Quit** in the **File** menu. (You can also quit this program simply by closing the window.)

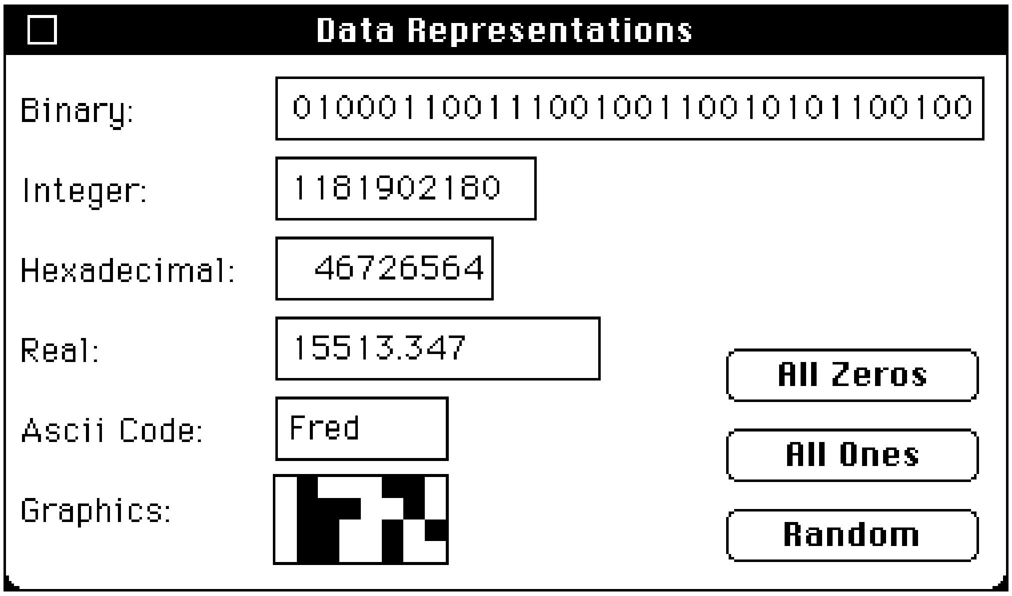


Figure 1.1. *Window from the program Data Reps. The window displays six values of different types, but each value is represented in the computer by the same thirty-two bits.*

The window contains six rectangular boxes, labeled with the type of data each contains. The first five boxes are ***text-input boxes***. Many Macintosh programs use text-input boxes to allow you to type in data to be used by the program. Some Macintosh hints: If all or part of the contents of a box are hilited, you can simply start typing to erase and replace the hilited characters. If a box contains a blinking insertion point, anything you type will be inserted at that that point. You can move the insertion point with the arrow keys or by clicking with the mouse at the new location. To move the insertion point to a different text-input box, just click in that box. (Actually, the easiest way to move from one text box to another is to press the tab key; this will hilite the contents of the next box so that you can just start typing the new value.)

You might find that you are only allowed to type certain characters into a text-input box, or that the number of characters or the contents of the box are restricted in some other way. In the program *Data Reps*, when you violate one of these restrictions, the computer will beep at you. (Most of the restrictions are obvious; for example, the box labeled "Binary" can only contain the binary digits 0 and 1.)

The sixth box in the window, labeled "Graphics," is not a text input box. This box is divided into an eight-by-four grid of small squares which can be either black or white. Each of the thirty-two squares corresponds to one of the bits in the binary number. The square is white if the corresponding bit is 0 and is black if that bit is 1. If you click the mouse on one of these squares, it will change from black to white or vice versa. If you click and drag, you can change several squares at once. This is the easiest way to change specific bits in the number.

Try changing the contents of each of the six boxes in the window. Note that as soon as you make any change, the contents of all the other bases are changed as well. This is because each box is really displaying the same string of thirty-two bits, interpreted in different ways.

The three ***buttons*** in the lower right corner of the window provide another way to change the contents of the boxes. *Click on each button to see its effect.* The button named "Random" will produce a different bit-pattern each time it is pressed.

Types: Here is a description of what you are seeing in each of the six boxes in the *Data Reps* program window, along with some suggested exercises. Note that you are certainly *not* expected to learn all the details of the different representations. The important idea here is just that different representations are possible.

• **The Binary Box.** This is the most direct display of the thirty-two bit binary number, showing a zero or one to represent each individual bit. If you type fewer than thirty-two bits in this box, the computer will fill in with extra zeros on the left.

• **The Integer Box.** A binary number can be interpreted as a positive integer (0, 1, 2, 3, 4, ...) in a natural way, as explained in Section 1.1 of *The Most Complex Machine*. Representing the negative integers (-1, -2, -3, ...) is trickier. However, it is always true that a binary number whose leftmost bit is 1 represents a negative number. This is explained in Subsection 2.2.3 in the text. *Type in some integers and observe the bit-patterns that represent them. Include some negative numbers. Can you say anything about how negative numbers are represented?*

• **The Hexadecimal Box.** It is difficult (for humans) to read long strings of zeros and ones. Hexadecimal numbers are a kind of shorthand for writing such strings. A hexadecimal number is written using the sixteen ***hexadecimal digits*** 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F. The letter A stands for 10, B for 11, and so forth. A hexadecimal number is a number written in the "base 16," just as ordinary integers are in the "base 10" and binary numbers are in the "base 2." For example, the hexadecimal number $2B7_{16}$ would be written in decimal as:

$$\begin{aligned} 2B7_{16} &= 2 \times 16^2 + 11 \times 16^1 + 7 \times 16^0 \\ &= 2 \times 256 + 11 \times 16 + 7 \times 1 \\ &= 695 \end{aligned}$$

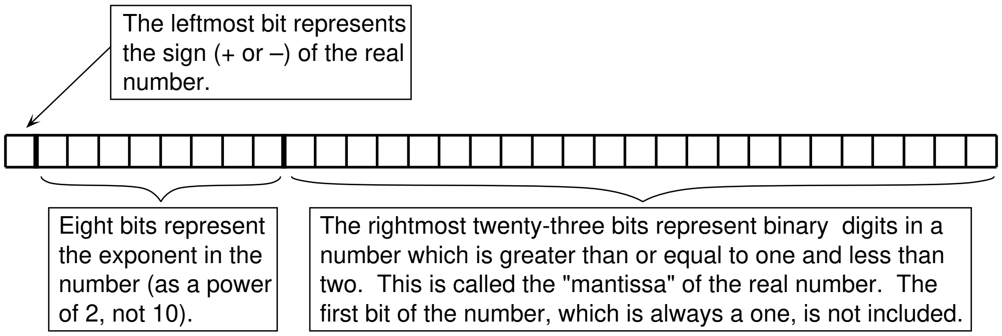


Figure 1.2. *The representation of a non-zero real number as a string of thirty-two bits. Special bit patterns are used for the number zero, as well as for the values infinity and minus infinity. The representation has three parts: a "sign," an "exponent," and a "mantissa." In the binary number system, a mantissa consists of a 1, followed by a decimal point, followed by a string of zeros and ones; in the representation used for real numbers in a computer, the 1 before the decimal point is omitted, and only the bits following the decimal point are represented. The eight-bit exponent is used in a rather odd way: First of all, if all eight bits of the exponent are 1's and if all the other bits in the number are 0's, then the corresponding real number is "infinity"; "minus infinity" is obtained from this by turning on the left-most bit. Aside from these cases, the eight bits represent a power of two; the power is given by converting the eight-bit binary number to decimal and then subtracting 127. For example, 01111111 represents the power 0; 10000000, the power 1; and 01111110, the power -1.*

The conversions from hexadecimal to binary and vice versa are much easier. Each hexadecimal digit corresponds to a string of four binary digits according to the rules $0_{16} = 0000_2$, $1_{16} = 0001_2$, $1_{16} = 0010_2$, $3_{16} = 0011_2$, $\ldots$, $\mathrm{F}_{16} = 1111_2$.

Test this correspondence by entering some hexadecimal numbers and observing how they translate into binary.

• **The Real Box.** In mathematics, a real number is one that can include a decimal point and any number of digits—even an infinite number—after that decimal point. On a computer, there is a limit on the number of digits that can be represented, so that real numbers can only be represented approximately. In *Data Reps*, if you type in more digits than can be represented, the extra digits will be ignored and will be erased when you move to another input box. (If you type more than about eight digits in the Real box, you will notice that the entries in the other input boxes stop changing.)

The representation for real numbers is complicated by the fact that real numbers can be written in ***scientific notation***. In standard scientific notation, a number can be multiplied by a power of ten. For example, 3.75×10^4 means 3.75 times 10000, or 37500, since ten raised to the fourth power is 10000. On a computer, 3.75×10^4 is written as 3.75e4. The "e4" means "times 10 raised to the exponent 4." This notation is used in the Real input box in the program *Data Reps*.

The bit patterns used to represent real numbers are quite complicated. They are based on a binary-number version of scientific notation in which powers of two are used instead of powers of ten. Some strings of bits don't even represent legal real numbers; such bit strings are shown as "(invalid)" in the Real input box. There is even a special bit pattern that is assigned to represent infinity (denoted "INF" in the real box). Some of the details of the representation are explained in Figure 1.2.

Type in some real numbers and observe the bit patterns that represent them. (Don't worry about all the details of the representation; they are not important.) Check that turning on the leftmost bit will change a positive number to its negative. Make the numbers INF and −INF, based on the description in the caption of Figure 1.2.

• **The Ascii Box.** With eight bits needed for each character, a thirty-two-bit binary number can represent a string of four characters. Some of the possible characters, such as the tab character, are "non-printing." Such characters are drawn in *Data Reps* as small squares. *Try typing in some words to see what numbers they correspond to. Note that each character corresponds to a two-digit hexadecimal number. Why?*

• **The Graphics Box.** This is explained above. *Try drawing some pictures, to see what letters or numbers the bit-patterns represent.*

* * *

Exercise 1: *Use the program Data Reps to find out which character has ASCII code 120. Use the program to find the ASCII code of the character '@'. To do the latter, start with "All Zeros" and then replace the last character in the ASCII Code box with @. What happens if you just type @ by itself into the ASCII box? Why?*

Exercise 2: *Type the following integers into the Integer input box, and observe the corresponding pictures in the graphics box: 1, 2, 4, 8, 16, 32, 256, 4096, 65536. What pattern do you see in the pictures? Explain why these numbers produce this pattern.*

Exercise 3: *Type a four-letter word, such as "TIME," into the ASCII Code text input box of the program Data Reps. Then, in the graphics box, turn the bits on and off in the third column from the left. What have you discovered about the meaning of the third bit from the left in the eight-bit ASCII code of a letter?*

Exercise 4: *You could type "1000" into any of the five text input boxes. Explain carefully the meaning of this string of symbols when it is typed into each box. How is it possible that the same string of symbols can have different meanings in different circumstances? (Recall the definition of a "symbol.")*

Inside a working computer, the interpretation of a given string of bits has to be kept straight by someone or something. At this point, you don't know enough about computers to understand exactly how this is done, but you might be able to make some good guesses. Try it! Think about the analogy with information stored in books.

Lab Number 2 for

The Most Complex Machine

by David J. Eck

Logic Circuits

About this Lab: It is possible in theory to construct a computer entirely out of transistors (although in practice, other types of basic components are also used). Of course, in the process of assembling a computer, individual transistors are first assembled into relatively simple circuits, which are then assembled into more complex circuits, and so on. The first step in this process is to build ***logic gates***, which are circuits that compute basic logical operations such as AND, OR and NOT. In fact, once AND, OR and NOT gates are available, a computer could be assembled entirely from such gates. In this lab you will work with (simulated) circuits made up of AND, OR and NOT gates. You will be able to build such circuits and see how they operate. And you will see how simpler circuits can be combined to produce more complex circuits.

This lab covers much of the same material as Chapter 2 in *The Most Complex Machine*. The lab is self-contained, but many of the ideas covered here are covered in much more depth in the text, so that it would be useful for you to read Chapter 2 before doing the lab.

Remember that you should *always* read through the lab worksheet in advance of starting a lab. In addition to exercises for you to do, the lab

---

This lab and the software used in it accompany *The Most Complex Machine: A Survey of Computers and Computing*, an introductory computer science textbook by David Eck (published 1995 by AK Peters, Ltd., 289 Linden Street, Wellesley, Massachusetts 02181; ISBN 1-56881-054-7). Support for the text, software and labs is available on the World Wide Web at http://math.hws.edu/TMCM.html. David Eck can be reached at the e-mail address eck@hws.edu.

worksheet contains background material for you to read, and you should not have to spend your time at the computer reading this material.

Logic and Circuits: A logic gate is a simple circuit with one or two inputs and one output. The inputs and outputs can be either `on` or `off`, and the value of the output is completely determined by the values of the inputs (with the proviso that when one of the inputs is changed, it takes some small amount of time for the output to change in response). Each gate does a simple computation. Circuits that do complex computations can be built by connecting outputs of some gates to inputs of others. In fact, an entire computer can be built in this way.

Circuits are drawn using standard symbols for AND, OR, and NOT gates, as shown in Figure 2.1. This figure shows a window from *xLogicCircuits*, the program that you will use in this lab. The window shows a "circuit board" containing an AND gate, an OR gate, and a NOT gate. Each gate has one or two inputs and one output, with the inputs on the left and the output on the right.

In *xLogicCircuits*, a circuit board window is used as a base on which you can construct complex circuits from simpler components. The circuit board has eight input leads on the left and eight output leads on the right. These can be connected to the inputs and outputs of the circuit components. Every input or output can be either ON or OFF. You have control over the inputs on the left of the circuit board; just click on the word ON or OFF on the left of the screen to change it. Aside from these inputs, which you control, all the other values are determined by the simple rules that govern the behavior of gates. These rules are explained in *The Most Complex Machine* and are summarized by the following tables:

AND gate:

Input 1	*Input 2*	*Output*
OFF	OFF	OFF
OFF	ON	OFF
ON	OFF	OFF
ON	ON	ON

OR gate:

Input 1	*Input 2*	*Output*
OFF	OFF	OFF
OFF	ON	ON
ON	OFF	ON
ON	ON	ON

NOT gate:

Input	*Output*
OFF	ON
ON	OFF

Find the data file Basic Gates, and open it. (It should be in a folder named Files for Lab 2.) On the screen, you will see the window shown in Figure 2.1. Try turning the inputs on and off to check that the gates have the correct behavior for all possible inputs.

This all becomes more interesting when we start building more complex circuits. *Open the file named Majority. Play with its inputs and try to discover the rule that determines the output in terms of the input. The name of the circuit is a hint.*

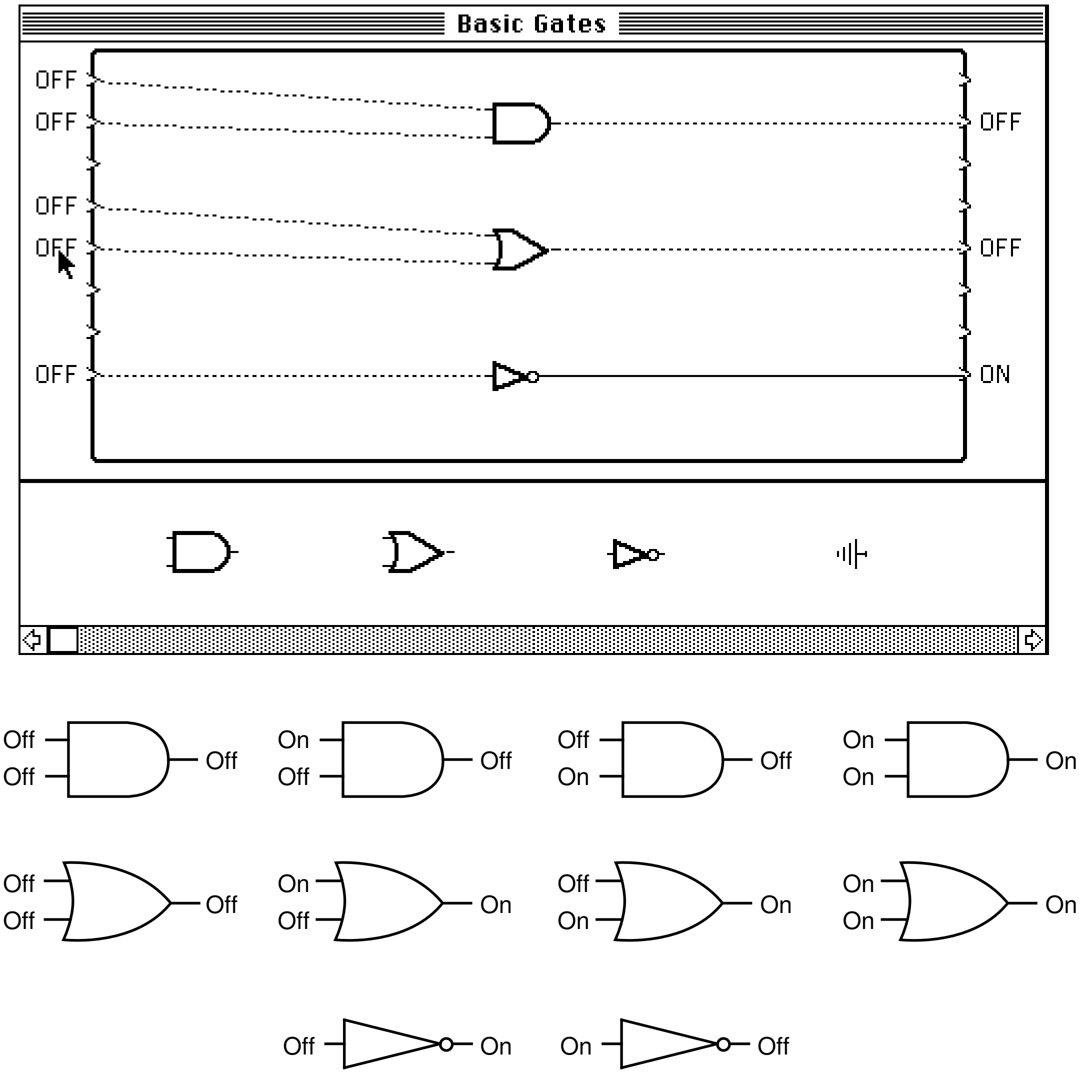


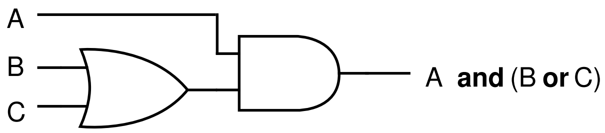
Figure 2.1. *At the top of this figure is a window from the program xLogicCircuits, showing the circuit stored in the file* Basic Gates. *This circuit contains, from top to bottom, an* AND *gate, an* OR *gate and a* NOT *gate. Inputs to the circuit are shown along the left edge of the window; outputs along the right edge. Inputs can be turned on and off by clicking on the word ON or OFF. Circuits are constructed by dragging gates and other components from the bottom of the window into the circuit and then drawing lines to represent wires. Below the window are* AND, OR, *and* NOT *gates shown in all their input/output configurations.*

Now, there is a branch of mathematics called ***Boolean algebra*** that is intimately connected with logic circuits such as *Majority*. Boolean algebra studies expressions made up of letters—representing inputs that can be either `true` or `false`—and the operators **and**, **or** and **not**. For example, here are some expressions of Boolean algebra:

A **and** (B **or** C)
(**not** (A **or** B)) **or** (A **and** B)
(A **and** B **and** C) **or** (**not** B)

Like logic circuits, such expressions represent computations. Once the values of the inputs are known, an output value for the expression can be computed according to the laws of logic. In fact, if we associate the logical value `true` with "`on`" and the logical value `false` with "`off`," then the same computation can be done by a properly wired logic circuit. Each different letter in the expression corresponds to an input wire for the circuit, and the expression serves as a blueprint for building the circuit.

For example, the simple expression A **and** (B **or** C) corresponds to the circuit:



The *Majority* circuit that you looked at previously corresponds to the more complicated expression: (A **and** B) **or** (B **and** C) **or** (A **and** C). (The text discusses the correspondence between Boolean algebra and logic circuits in more detail.)

Building Circuits:

To create a new circuit with *xLogicCircuits*, you should start with the **New** command to get an empty circuit board. Drag gates from the strip along the bottom of the screen onto the circuit board. To draw a connecting wire, move the mouse to the *source* of the wire you want to draw. (The cursor will change from an arrow to a plus sign when it is at a legal position.) Press the mouse button; hold it down while you move the mouse; and release it at the other endpoint of the connection. *You must position the cursor carefully over the endpoint of an input wire; again, the cursor changes to a plus sign when it is in a legal position.*

While drawing a connecting wire, you can leave a little black dot along the way by quickly releasing and pressing the mouse button. These dots represent "junctions" that can themselves be used as sources for connections. If you make a mistake, you can use the Undo command from the Edit menu to undo it, provided you do so immediately. (Undo can only undo one action.) To protect yourself against bigger mistakes, you should save your circuit frequently. Then, if you make a major mistake, you can open the saved version and discard the version on the screen.

As a warm-up exercise, use xLogicCircuits to construct the circuit that computes A AND *(B* OR *C). Then use your circuit to make a table of the value of this expression for each of the eight possible combinations of inputs.* Try building more complicated circuits. (Exercise 1, below, has other Boolean algebra expressions for you to work on.)

Subcircuits and Arithmetic: In order to have circuits that display "structured" complexity, it is important to be able to build on previous work when designing new circuits. Once a circuit has been designed and saved, it should be possible to use that circuit as a component in a more complex circuit. The program *xLogicCircuits* allows you to do this. The **Include** command from the **File** menu is used to add a previously-created circuit to the strip of components along the bottom of the circuit board. From there, it can be dragged onto the circuit board like any other component.

Start another new circuit. Use the **Include** *command to add the Majority circuit to the component strip. (The* **Include** *command works much like the* **Open** *command.) Then drag a copy of the Majority circuit onto the circuit board. Use this as a basis for building a "Minority circuit," that is, a circuit with exactly the opposite behavior from the Majority circuit. (All you have to do is add a* NOT *gate and some wires.)*

To see a more useful example of constructing circuits from sub-circuits, open the file *Three-bit Add.* This is a circuit that can add binary numbers. Previously in this lab, we have considered the inputs and outputs of a circuit to have the values `on` and `off` or else the values `true` and `false`. When a circuit is meant to work with binary numbers, the values 1 and 0 are used instead, with `off` standing for 0 and `on` standing for 1.

The circuit *Three-bit Add* adds two three-bit binary inputs. (See Figure 2.2.) Examples of such sums are:

$$\begin{array}{r} 001_2 \\ +\ 001_2 \\ \hline 0010_2 \end{array} \qquad \begin{array}{r} 111_2 \\ +\ 001_2 \\ \hline 1000_2 \end{array} \qquad \begin{array}{r} 010_2 \\ +\ 101_2 \\ \hline 0111_2 \end{array} \qquad \begin{array}{r} 011_2 \\ +\ 110_2 \\ \hline 1001_2 \end{array} \qquad \begin{array}{r} 111_2 \\ +\ 111_2 \\ \hline 1110_2 \end{array}$$

As in ordinary addition, each column is added separately, from the right to left, with a possible "carry" from one column to the next. The rules for adding columns of three bits (including a carry from the previous column) are $0_2 + 0_2 + 0_2 = 00_2$, $1_2 + 0_2 + 1_2 = 10_2$, $1_2 + 1_2 + 1_2 = 11_2$, and so forth. When three-bit numbers are added, the answer contains four bits, since there can be a carry from the leftmost column.

The structure of the addition circuit reflects the structure of the problem it solves. When binary numbers are added by hand, they are added one column at a time, using the same procedure for adding each column; a circuit that does the addition contains several identical subcircuits, and each of these subcircuits does the sum for one column. The sum for each column is done by a *Full Adder* subcircuit, which adds two digits from the input numbers and a carry digit from the previous column. (See Section 2.1 in the text. If you want to see the inside of *Full Adder*, a file containing that circuit is available. A *Full Adder* is made from two *Half Adder* circuits, and a file containing a *Half-Adder* is also available to you.)

Check that the three-bit addition circuit gives the correct answer for each

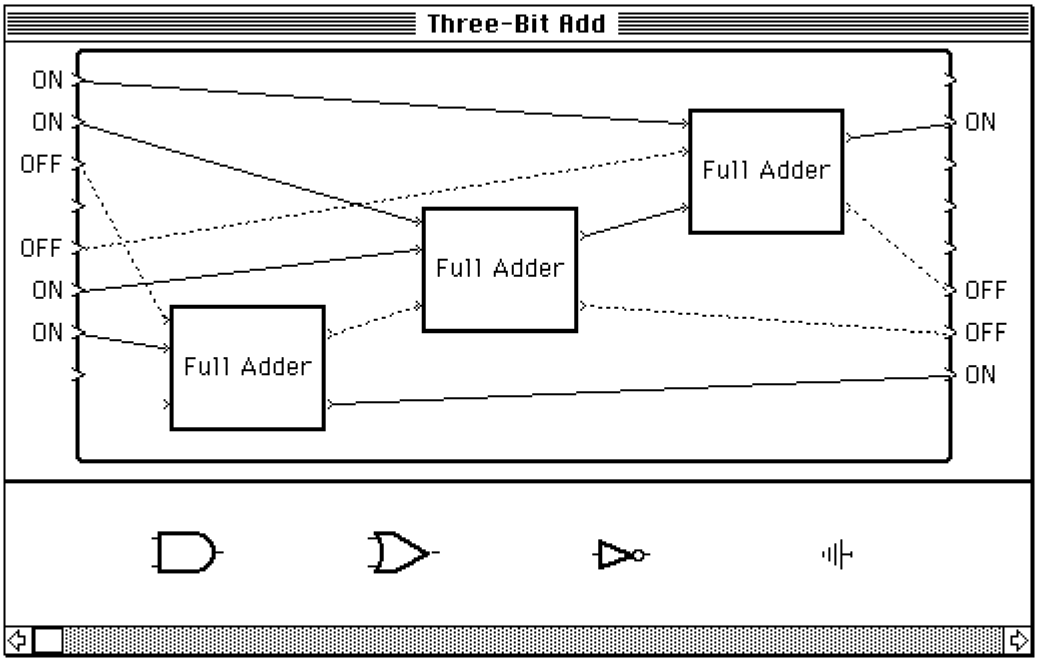


Figure 2.2. *The circuit Three-bit Add has six inputs and two outputs. The inputs, arranged into two groups of three, represent two three-bit binary numbers. These numbers are written from top to bottom with ON representing 1 and OFF representing 0, so that in this example the two numbers being added are 110 and 011. The four outputs represent the three-digit sum of the two input numbers and the carry-out from the leftmost column. Here, the sum is computed correctly to be 001 with a carry of 1.*

of the example sums given above. Make sure that you understand how the circuit is used and how it could be extended to add numbers with more bits.

Introduction to Feedback Loops: If the output of a gate is used as input to that same gate, either directly or through a sequence of other gates, the result is called a ***feedback loop***. Circuits with feedback loops do not have the nice relationship with Boolean algebra that exists for circuits without feedback loops. However, feedback is absolutely essential for building ***memory circuits*** that can store values. In this section of the lab, you will look at two kinds of circuits with feedback loops. The circuits are in the files *Clocks* and *Simple Memory*.

Open the file Clocks. It contains two very simple circuits. One consists of a single NOT gate with its output connected to its input. This circuit rapidly oscillates between being on and off. (The behavior of this circuit is discussed in the answer to Question 5 in Chapter 2 of the text.) The other circuit includes a ***delay*** component, that simply delays the signals passing through it by a certain fixed amount. This slows down the oscillation. *You might want to try similar circuits with different delay times.* Do you see why the name of this file is *Clocks*?

Next, open the file *Simple Memory*. This file contains the circuit from Figure 2.14 of the text. This circuit has a feedback loop that can store either a zero or a one. If you turn the first input to this circuit `on` and then back `off`, the output will stay on. The circuit is "remembering" the value one. If you turn the second input `on` and `off`, the output will turn `off`, and the circuit remembers the value zero. Essentially, the circuit remembers which of its inputs was most recently turned `on`. *Try it!*

* * *

Exercises: You should be able to do most of the following exercises after working through the lab. (If you plan your lab time in advance, you might find it more natural to work on some of these exercises as you work though the lab.) However, in some cases (especially Exercise 5), you will need additional help from *The Most Complex Machine*. Note that Exercises 1 and 3 can be done with or without a computer.

Exercise 1: *Build circuits to compute each of the following expressions of Boolean algebra. You can build them all on one screen if you want. Make a printout or drawing of the result.*

$$\textbf{not } ((\textbf{not } A) \textbf{ and } (\textbf{not } B))$$
$$(C \textbf{ and } D \textbf{ and } E) \textbf{ or } (\textbf{not } D)$$
$$(\textbf{not } (F \textbf{ or } G)) \textbf{ or } (G \textbf{ and } F)$$

Exercise 2: *In the previous exercise, you went from logical expressions to circuits. It is also possible to go in the other direction. That is, given a circuit that does not contain feedback loops, it is possible to describe its output as a logical expression of its inputs. Open the file named FindExpression.1. Assuming that its three inputs are named A, B, and C, find the expression that describes the output in terms of the input. Then do the same with the file FindExpression.2, assuming that its inputs are A, B, C, and D. Explain in words how you got your answers.*

Exercise 3: *One of the major points of Section 2.1 of the text is that Boolean algebra can be used to build a circuit with any specified input/output behavior. Construct a circuit with the following specification:*

A	B	**Output**
`off`	`off`	`on`
`off`	`on`	`on`
`on`	`off`	`off`
`on`	`on`	`on`

To apply the method given in the text, your first step should be to write down a Boolean algebra expression corresponding to this table.

Make a printout or drawing of your circuit.

Exercise 4. *Explain why the ability to make and use subcircuits is so important for the construction of complex circuits.*

Exercise 5: *The final exercise of this lab is to construct a mini-ALU. This is not a trivial exercise, and you should not be surprised if you have to ask for some hints.*

As explained in Chapter 2 of the text, the purpose of an ALU is to do basic arithmetic and logical calculations. The ALU is constructed so that it actually does all of the computations all the time; control wires are used to control which of the answers gets through to the output wires of the ALU. (See Section 2.2.6 of the text.)

For this exercise, you should construct a mini-ALU that can compute either the sum or the logical-AND of two three-bit binary numbers. That is, you want a circuit that can imitate either the addition circuit from the file Three-bit Add or the circuit from the file Three-bit AND. Your mini-ALU will have eight inputs and four outputs. The first three inputs represent one three-bit number to be used as input for the ALU's computation. The next three inputs represent the second three-bit input to that computation.

The seventh and eighth input for your ALU control which answer will be output—the answer from Three-bit Add or the answer from Three-bit AND. If input seven is on and input eight is off, the ALU should output the three-bit sum of the two input numbers and the carry-out from the addition. If input eight is on and input seven is off, the ALU should output the three-bit logical-AND of the two input numbers; in this case, the fourth output should be off.

The file Hidden ALU contains a circuit that works in this way, but you can't see what's inside, so you will have to make your own. Start with the two existing circuits Three-bit Add and Three-bit AND. (Add them to the circuit using the **Include** *command and then drag them onto the circuit board.) Your problem, then, is to add the gates and wires that are needed to complete the ALU. The circuitry necessary to compute the output (for a much more complicated ALU) is discussed in Section 2.2.6.*

The easiest way to do this exercise is to write down a Boolean algebra expression for each output wire. The expression should describe when the wire should be turned on, in terms of the seventh and eighth inputs to the ALU and the outputs of the Three-Bit Add and Three-Bit AND circuits. (The expressions for three of the output wires are essentially identical; the fourth is simpler.)

Make a printout of your finished circuit.

Lab Number 3 for

The Most Complex Machine

by David J. Eck

Memory Circuits

About this Lab: This lab looks at circuits that can be made when feedback loops are allowed. The primary use for circuits with feedback is in ***memory circuits***. In this lab you will see how complex memory circuits can be built up starting from a simple one-bit memory. You will construct multi-bit registers and an addressable memory that stores several numbers in a sequence of numbered locations.

This lab is based on Sections 2.3 and 3.1 of *The Most Complex Machine*. It will be useful for you to be familiar with this material before you begin the lab.

A One-bit Memory: When circuits are allowed to contain feedback loops, then circuits can be built that have an ***internal state***. The output of such a circuit depends not just on the values of its inputs but also on its internal state. The state is a kind of memory—it allows the output of the circuit to depend on what happened in the past, not just on what is happening now.

This lab and the software used in it accompany *The Most Complex Machine: A Survey of Computers and Computing*, an introductory computer science textbook by David Eck (published 1995 by AK Peters, Ltd., 289 Linden Street, Wellesley, Massachusetts 02181; ISBN 1-56881-054-7). Support for the text, software and labs is available on the World Wide Web at http://math.hws.edu/TMCM.html. David Eck can be reached at the e-mail address eck@hws.edu.

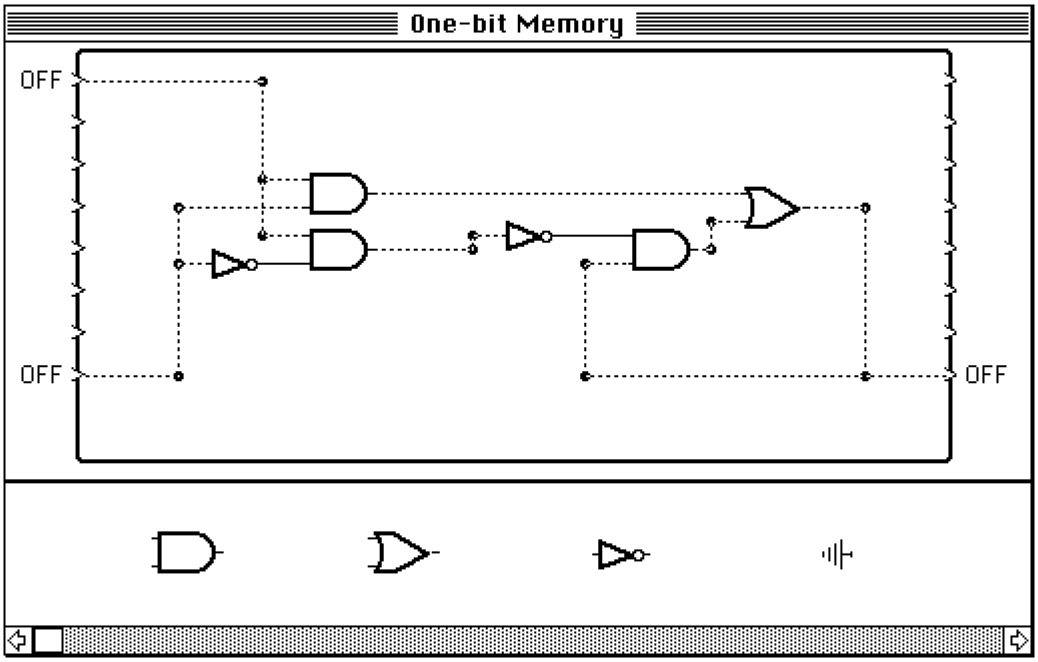


Figure 3.1. *A one-bit memory circuit. The output value represents a number (0 or 1) stored in the circuit. The output wire is named* Data-out. *The two inputs on the left are used when a value is to be stored. The top-left input is the* Load-Data *wire, while the bottom-left input is* Data-in. *In effect, the one-bit value stored in the circuit is actually held in the feedback loop at the right. This loop will remain in one of two stable states as long as the* Load-Data *wire is kept turned off.*

It particular, it is possible to build a one-bit memory circuit, as explained in Section 2.3 of *The Most Complex Machine.* This circuit is shown in Figure 2.15 of the text, and a functionally identical circuit from the file *One-bit memory* is shown in Figure 3.1 in this lab worksheet.

Assuming that both its inputs are `off`, the *One-bit memory* circuit can be in one of two states. The state it is in depends on which value, 0 or 1, was most recently stored in the circuit. The stored value can be read on the output wire of the circuit.

If you are to understand the rest of this lab, it will be important that you understand exactly how to use this circuit, that is, how to store values in it and how to read the stored value. *Open the file One-bit Memory so that you can get some hands-on experience doing so.* Note that the bottom wire on the left is used to specify a value to be stored in the circuit. That value is stored by turning the top input wire `on` and back `off`. These two input wires are called, respectively, Data-in and Load-data. The value stored in the circuit is safe as long as the Load-data wire is kept turned off. (That is, the state of the circuit can change only when the Load-data wire is `on`.

Try storing values in the one-bit memory circuit. Verify that switching Data-in *on and off has no effect as long as* Load-data *is off. If you are not* *clear on the details, consult Subsection 2.3.1 in The Most Complex Machine.*

A Multi-bit Memory: With the one-bit memory as a starting point, you can build more complex memory circuits. Just as you can line up three *Full Adder* circuits to get a three-bit addition circuit, you can combine k one-bit memory circuits to get a k-bit memory. Such k-bit memories are used as components in central processing units. A k-bit memory that is part of a CPU is called a ***register***.

If you open the file *Hidden 3-bit Register*, you will see a three-bit memory (with its internal construction hidden). This circuit is used exactly like a one-bit memory, except that it stores a three-bit number instead of a single bit. It has three Data-in wires and three Data-out wires, but has only a single Load-data wire. When you open the file, you will see that the circuit currently stores the value 011_2.

As a simple exercise, change the stored value to 101_2. Exercise 1 at the end of the lab asks you to construct a three-bit register.

Addressable Memory: The main memory circuit of a computer is made up of a large number of k-bit memories. Each k-bit memory is called a ***location*** in the main memory, and each location has an assigned number, called its ***address***. For the model computer introduced in Chapter 3 of the text, for example, main memory consists of 1024 sixteen-bit locations. Modern computers generally have millions of locations, each holding an eight-bit number.

In this section of the lab, you will construct a four, eight, or sixteen-location memory, depending on how far you want to carry things. Each location will hold a three-bit number; in fact, each location will actually be a "three-bit register" of the type you have just been looking at. (Properly speaking, the circuit here should be called a "three-bit memory" instead of a "three-bit register," since it's not being used as part of a CPU. But that is really beside the point.)

The memory circuit will have three Data-in wires, three Data-out wires and one Load-data wire. In addition, it will have one or more Address wires. The purpose of the Address wires is to pick out one particular location in the memory. The Data-in, Data-out and Load-data wires apply only to that location. As long as the Address wires have some particular value, the memory circuit behaves identically to a simple three-bit memory; it will be possible to read the value in that memory by looking at the Data-out wires or to store a value using the Data-In and Load-Data wires. However, changing the value on the Address wires will make the memory circuit behave as a *different* three-bit memory, which independently stores a different three-bit number.

Open the circuit Hidden 16-location Memory. This circuit can store a three-bit number in each of 16 locations. It therefore contains 48 one-bit

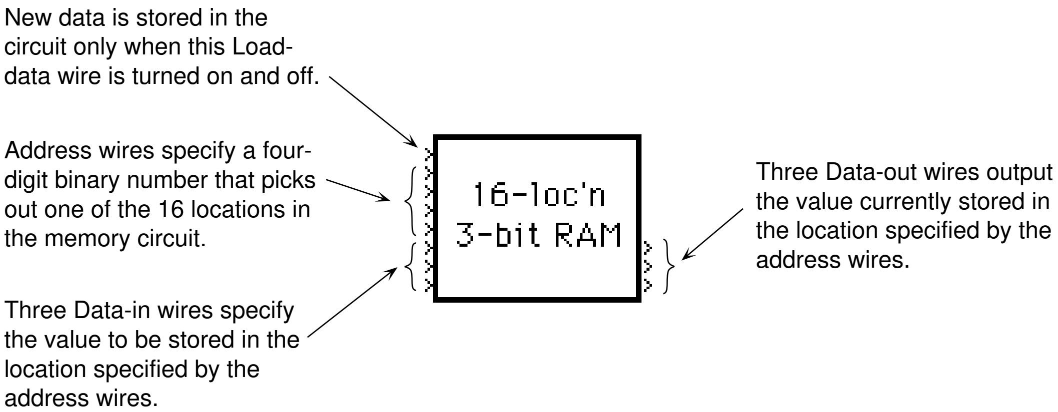


Figure 3.2. *A 16-location memory circuit. The four address wires make it possible to access* 2^4*, or sixteen, locations.*

memories, plus all the extra circuitry necessary to support the operation of the address wires. *One important aspect of the complexity of this circuit is the time it takes to operate.* For example, when you change the values on the Address wires, it will take a few seconds until the value on the Data-out wires will change. Similarly, when you want to store a value in the circuit, you will have to wait several seconds from the time when you turn on the Load-data wire before you turn it off again. (On slower computers, the time you have to wait can be unpleasantly long, but failure to wait the necessary time can have surprising effects, such as putting the circuit into an oscillating state where the outputs blink on and off.)

Try out this circuit, testing how it can be used to store a value in a location and then to retrieve that value at a later time. Exercise 2, below, contains some specific exercises for you to do with this circuit.

As for actually building this memory circuit, we can proceed one address wire at a time, as explained in Section 3.1 of *The Most Complex Machine.* The first step is to combine two three-bit registers into a memory circuit that has only two locations (numbered zero and one), and one address wire. This circuit is in the file *2-location 3-bit RAM.* The subcircuit, *Three-bit Select*, that it uses is also available in a file. You will need to include copies of each of these files as subcircuits in the circuit that you build.

Open the file 2-location 3-bit RAM. Experiment with the circuit. Make sure that you understand how to use it to store and retrieve values in its two memory locations. Make sure that you understand how it works, especially how its address wire is used to select between the two locations.

The next stage in building the memory is to take two copies of this two-location memory and combine them to make a four-location memory. The new circuit will have two address wires. One of the address wires simply

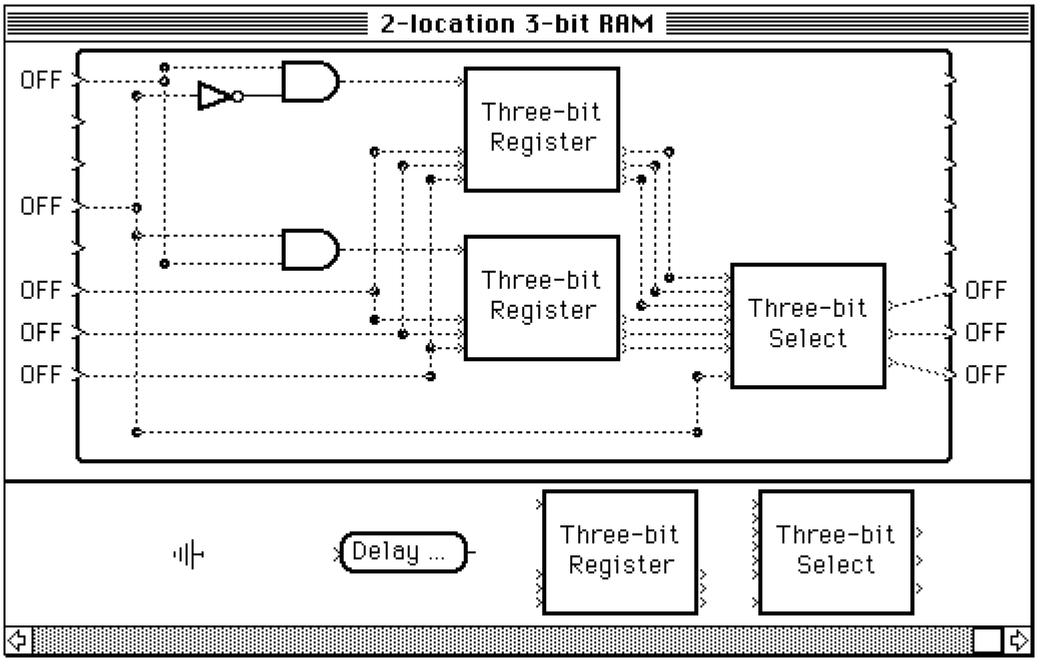


Figure 3.3. *A two-location memory, in which each location stores three bits. The subcircuit Three-bit Select selects between the outputs of the two registers. If the* Address *wire is off, then the output of the Select circuit is equal to the output of the top register; if* Address *is on, then Select lets the output of the bottom register through instead.*

connects to the address wires of the two-location memories; the other selects between the two two-location memories in the same way that the single address wire in Figure 3.3 selects between the two registers. In fact, the design of the four-location memory is very similar to the design of the two-location memory. (See Figures 3.2 and 3.3 in Chapter 3 of the text; you might need to refer to these figures for help with Exercise 3 below.)

* * *

Exercise 1: *Construct a three-bit register that functions in the same way as the Hidden 3-bit Register that you looked at earlier in the lab. Start with a new circuit and include the One-bit Memory circuit using the* **Include** *command. Drag three copies of the One-bit Memory onto the circuit board. The* Load-data *wire of the three-bit register should connect to the* Load-data *wires of all three one-bit memories. Make a drawing or a printout of your circuit.*

Exercise 2: *Open the file Hidden 16-bit Memory. (If it already open, reopen it so that it is in its original state.) By manipulating the* Address *wires, read the values stored in locations 1001, 0111, and 0100 of the memory circuit. Record your answers. Change the value stored in location 1001 to*

111. Explain exactly what steps you used to read and to store values in this circuit.

Exercise 3: *Build a four-location memory, as described above. Start with a new circuit and include copies of Three-bit Select and 2-location 3-bit RAM. If you like and if you have time, you can go on from there to build an eight-location and even a sixteen-location memory. Save your circuit in a file.*

Exercise 4: *Determine exactly how many logic gates of each type (AND, OR and NOT) are contained in the 16-location memory circuit. You will have to do some thinking and some work. Start by opening some circuits (like Three-bit Select) to find out how many gates they contain.)*

Lab Number 4 for

The Most Complex Machine

by David J. Eck

Introduction To xComputer

About this Lab: This lab introduces the program *xComputer*, which simulates the simple model computer—also called xComputer—that is discussed in Chapter 3 of *The Most Complex Machine*. The xComputer consists of a ***Central Processing Unit*** (CPU) and a ***main memory*** that holds 1024 sixteen-bit binary numbers. The CPU contains an ***Artihmetic-Logic Unit*** (ALU), like the one designed in Chapter 2, for performing basic arithmetic and logical computations. It also contains eight ***registers*** that hold binary numbers being used directly in the CPU's computations, a ***Control circuit*** that is responsible for supervising the computations that the CPU performs, and a ***clock*** that drives the whole operation of the computer by turning a single output wire on and off.

The *xComputer* program that you will use in this lab lets you load programs and data into the memory of the simulated xComputer. You can then watch while those programs are executed, and you can observe how numbers stored in the computer change as a program runs.

This lab contains some information about xComputer and its machine language. It demonstrates how instructions are fetched from memory and executed by the CPU. And it will explain the features of the simulation

---

This lab and the software used in it accompany *The Most Complex Machine: A Survey of Computers and Computing*, an introductory computer science textbook by David Eck (published 1995 by AK Peters, Ltd., 289 Linden Street, Wellesley, Massachusetts 02181; ISBN 1-56881-054-7). Support for the text, software and labs is available on the World Wide Web at http://math.hws.edu/TMCM.html. David Eck can be reached at the e-mail address eck@hws.edu.

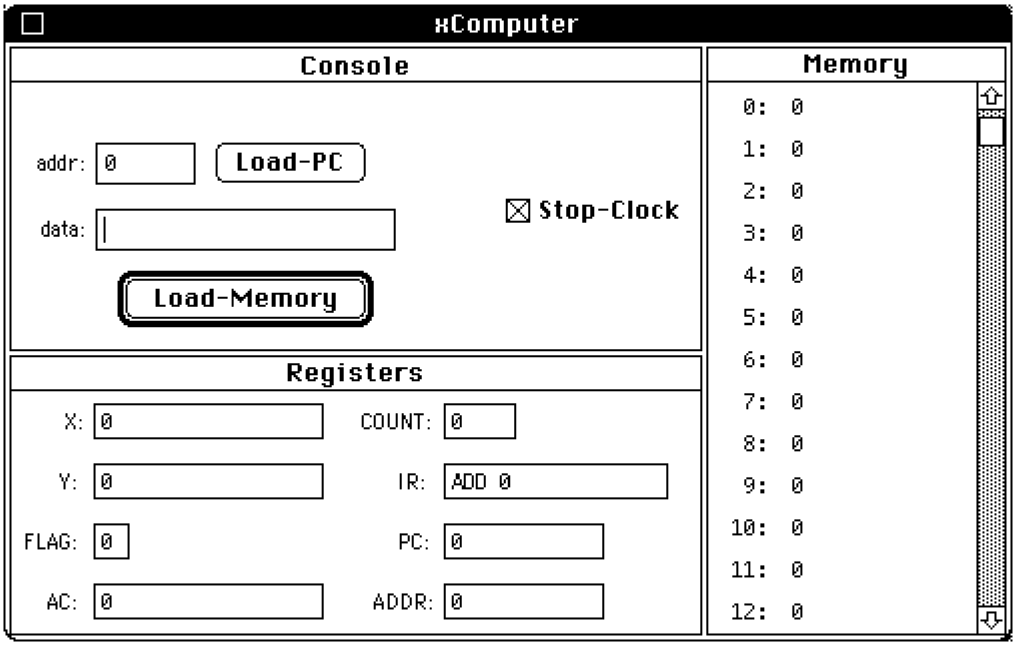


Figure 4.1. *The main window of the xComputer program, which allows you to control the computer and see the contents of its registers and memory*

program, *xComputer*. Lab 5 will be a continuation of this lab that covers the programming process in more detail.

It would be useful for you to read through Chapter 3 of the text before doing this lab. Chapter 3 is rather technical, and you might find that you need to work through both this lab and that chapter before you really understand either of them.

The Program: Begin the lab by starting up the program *xComputer*. You will see the window shown in Figure 4.1.

The "Memory" section of this window shows the 1024 locations in xComputer's memory. These locations are numbered from 0 to 1023. Each line in the memory shows a location number and the value stored in that location. When the program first starts up, the memory contains only zeros. The scroll bar can be used to view any part of memory.

The "Console" section of the window is used to interact with and control the xComputer. The Stop Clock checkbox turns the simulated clock on and off. Since the computer is running only when this clock is on, the Stop Clock checkbox really turns the computer itself off and on. The other items in the Console section are used to load values into memory and into the PC register. This will be discussed below (along with a much easier way to load values into memory).

The "Registers" section of the window shows xComputer's eight registers. Remember that a register is simply a memory unit in the CPU that holds data being used directly in the CPU's computations. Each register plays a particular role in the execution of programs by the CPU. These roles are described in detail in the text and will be illustrated during the course of this lab, but here for your reference is a brief summary:

- The X and Y registers hold two sixteen-bit binary numbers that are used as input by the ALU. For example, when two numbers are to be added, they are first put into the X and Y registers.
- The AC register is the accumulator. It is the CPU's "working memory" for its calculations. When the ALU is used to compute a result, that result is stored in the AC. For example, if the two numbers in the X and Y registers are added, then the answer will appear in the AC. Also, data can be moved from main memory into the AC and from the AC into main memory.
- The FLAG register stores the "carry-out" bit produced when the ALU adds two binary numbers. Also, when the ALU performs a shift-left or shift-right operation, the extra bit that is shifted off the end of the number is stored in the FLAG register.
- The ADDR register specifies a location in main memory. The CPU often reads values from memory or writes values to memory. Only one location in memory is accessible at any given time. The ADDR register specifies that location. So, for example, if the CPU needs to read the value in location 375, it must first store 375 into the ADDR register.
- The PC register is the program counter. The CPU executes a program by fetching instructions one-by-one from memory and executing them. (This is called the ***fetch-and-execute cycle***.) The PC specifies the location in memory that holds the *next* instruction to be executed.
- The IR is the instruction register. When the CPU fetches a program instruction from main memory, this is where it puts it. The IR holds that instruction while it is being executed.
- The COUNT register counts off the steps in a fetch-and-execute cycle. It takes the CPU several steps to fetch and execute an instruction. When COUNT is 1, it does step 1; when COUNT is 2, it does step 2; and so forth. The last step is always to reset COUNT to 0, to get ready to start the next fetch-and-execute cycle. This is easier to understand after you see it in action.

You will learn how the xComputer works by giving it a short program and watching it execute that program. To enter the program, follow these steps:

1. *Make sure that the "addr" input box in the Console contains a zero. Type the instruction "lod-c 17" into the "data" input box in the Console and press return. (Don't type the quotes!) The number 25617 will appear in* *location zero in main memory.* Here's what happened: Lod-c 17 is an ***assembly language*** instruction. When it is eventually executed by xComputer, it causes the computer to load the number 17 into the AC register. When you pressed return, this instruction was translated into machine language and stored in the location indicated by the "addr" box (that is, location zero). It so happens that in machine language, the instruction "lod-c 17" is represented by the number 25617, so that is what appeared in location zero. (Even this is not really true since the instruction is actually stored in memory as a binary, not a decimal, number.)

2. *Type "add-c 105" into the "data" input box and press return. The number 16489 appears in location 1 in memory.* The instruction add-c 105 translates into 16489 in machine language. That number is stored in location 1 because there is a 1 in the "addr" box. (For your convenience, the "addr" is incremented automatically when you press return.) Keep in mind that the instruction add-c 105 is *not* being executed when you press return—just entered into memory. Later, when you run the program, it will be executed. At that time, it will tell xComputer to add the number 105 to whatever number is already in the AC.

3. *Type "sto 10" into the data input box and press return. The number 10250 appears in location 2 in memory.* When it is executed later, this instruction will make the CPU store whatever number is in the AC into location number 10 in main memory.

4. *Type "hlt" into the data input box and press return. The number 11264 appears in location 3 in memory.* When this instruction is executed, the computer will stop running.

You have now entered a very simple program into memory. When the computer executes this program, it will start by loading 17 into the AC. Next, it adds 105 to the AC. Then, it stores the answer into memory location 10. Finally, it executes the "hlt" instruction and stops. After the program runs, you can look in memory location 10 to find the result of the computation.

How do you make the computer run the program? If you click on the Stop-Clock checkbox in the Console, the computer will start running. That is, it starts fetching instructions from memory and executing them (which is all that it ever does). You have to make sure that it begins with the first instruction of the program, which is in location zero in memory. What tells the computer where to get the next instruction? The number in the PC register. So, running a program requires two steps: First, make sure that the PC contains the starting location of the program in memory; and second, click on the Stop-Clock checkbox to start the computer running.

Try this now. Make sure that the PC contains a zero. If it does not, set the PC to zero using the **Set PC=0** *command in the* **Assembler** *menu. (An alternative method for entering a number into the PC is to type the* *number into the "data" input box and then click on the "Load PC" button.) Then, click on the Stop-Clock checkbox to start execution of the program (or, equivalently, use the* **Run** *command from the* **Assembler** *menu).*

If you have done everything correctly, the program will run. You will see things happening, but will probably not really understand them at this point. But you will notice that the instructions in the program appear one by one in the IR register as they are executed. Eventually, the HLT instruction will be executed and the computer will stop running. The correct answer to the computation, 122, will be in memory location 10. This gives you the general idea of how programs are executed by xComputer. Your goal in the rest of the lab is to understand the details.

Program Windows: It should be clear that entering a long program into xComputer by typing it into the data-input box would be very tedious and error-prone. If you accidentally leave out one instruction, for example, you would have to retype all or most of the program. Fortunately, you can type your program in a separate window and then load that window into memory. This has three advantages: You can edit the program in the window, for example by inserting a new instruction. You can save the contents of a window in a file, so that you'll never have to retype it again. And you can use ***labels*** in your program. Labels are a powerful programming technique; they are described in the Postscript to Chapter 3 in the text. They are not covered in this lab, but they will be an important part of Lab 5.

Use the **New** *command in the* **File** *menu to open a new program window. Enter the following program into that window:*

```
lod-c 17
sto 12
lod 12
inc
sto 12
jmp 2
```

This program *counts.* It starts by putting the number 17 into memory location 12, and then it adds one to the number in that location over and over, forever. (You'll see this in action in a moment.) There are several new instructions here. Lod 12 tells xComputer to copy the number from memory location 12 into the accumulator. (Note that this differs from Lod-c 12, which would put the number 12 itself into the AC, rather than the number stored in memory location 12.) The inc instruction adds one to the value in the accumulator. And jmp 2 is a jump instruction that sends the computer back to location 2, that is, back to the lod 12 instruction.

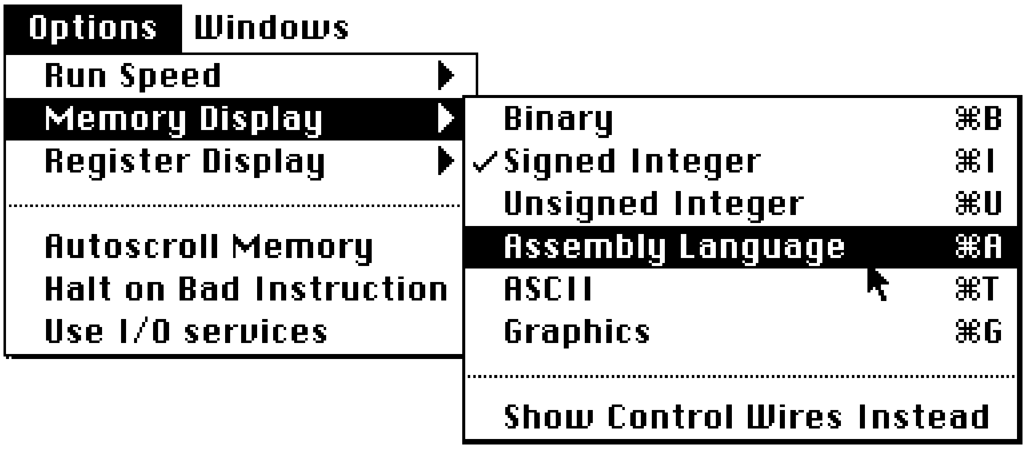


Figure 4.2. *Using a sub-menu, or "hierarchical menu," to change the style of display for the memory of xComputer. The memory display menu appears when you move the mouse over "Memory Display" in the Options menu. It lists six different ways of displaying the contents of memory, and also allows you to see xComputer's control wires instead of the memory. Here, the memory display style is being changed to assembly language. Run Speed and Register Display are also sub-menus. By the way, note that the command-key equivalents listed at the right edge of the menu are often more convenient to use than the menu itself.*

After typing this program in a window, load it into xComputer's memory using the **Load** *command from the top of the* **Assembler** *menu. (This command is available only if the front window on the screen is a program window and if a program is not already running.) When you load the program, you will see the equivalent machine language program appear in locations zero through five in memory. After loading the program, run it to see how it operates. Don't forget to set PC to zero before starting the computer.*

You can watch as the PC counts off the instructions in the program. You can see the assembly language instructions themselves as they are loaded into the IR. And you can observe that the value in memory location 12 changes from 17 to 18 to 19 to 20 and so on. This program will run forever, if you let it. You can stop the program by clicking on the Stop-Clock box, or, equivalently, using the **Stop-Clock** command from the **Assembler** menu.

You will be working with this little program throughout the remainder of the lab. Your objective is to understand how xComputer operates and to appreciate the fetch-and-execute cycle.

Options: The **Options** menu in *xComputer* gives you some control over the speed at which the computer runs and how numbers in the computer are displayed. You should find the various speeds and display styles helpful in understanding the way computers operate. The speed and display commands in the **Options** menu are contained in sub-menus. Figure 4.2 shows how to use such menus.

Computers work only with binary numbers, but those numbers are used to encode machine language instructions and data of various kinds. The program *xComputer* allows you to view the contents of the main memory and the registers in different ways. When you first start the program, it is set up to display the contents of memory as ordinary decimal numbers in the range −32768 to 32767. The contents of the registers are displayed as decimal numbers, except for the IR, which displays an assembly language instruction. Keep in mind that what is really there are binary numbers, and what you see is just one interpretation or view of those numbers. You can use the **Memory Display** sub-menu of the **Options** menu to change the display style of the numbers in main memory.

Try out the various Memory Display styles. Make sure that the little counting program from earlier in the lab is still in the computer's memory.

The Binary display shows a 16-bit binary number in each memory location; this display style is closest to the actual physical contents of the memory.

The Signed Integer and Unsigned Integer displays show ordinary decimal numbers. The difference is that signed 16-bit integers are in the range −32768 to 32767, while unsigned 16-bit integers are in the range 0 to 65535. (In either case, there are 2^{16} different possible values—it's just a question of how they are interpreted. See Subsection 2.2.3 in the text.)

The Assembly Language display shows the contents of each memory location as an assembly language instruction. In this display style, you should see the original counting program in memory locations 0 through 5. Most of the other locations contain "Add 0," which just happens to be the assembly language instruction encoded by the 16-bit binary number 0000000000000000. (Since not every 16-bit binary number corresponds to a legitimate assembly-language instruction, you might see some funny things in this display style.)

The ASCII display interprets each sixteen-bit number in memory as made up of two eight-bit ASCII character codes, and shows the two characters.

The Graphics display is very different from the others. It shows the *entire* memory at once. Each bit in memory—all 16 times 1024 of them—is represented by one pixel on the screen. That pixel is white if the bit is zero and is black if the bit is one. If you choose the Graphics display now, the memory will be almost entirely white, except for a few black dots at the top that represent the program you entered into memory.

I should note that when you enter information into Memory using the "data" input box in the Console, you can type the information in several of the above display styles, as well as in assembly language. You can, for example, enter ordinary numbers in the range −32768 to 65535. You can enter a binary number, but you must precede it by the letter B. For example: B1011010111. Finally, you can enter a single ASCII character, provided that you precede it by a quote mark. For example: '@ or 'A. You will need to do this for Exercise 1 at the end of the lab.

The **Memory Display** menu contains one additional option: **Show Control Wires Instead**. If you choose this option, the Memory display will disappear and will be replaced by a list of ***control wires***. These control wires are the key to understanding how the xComputer really works. The basic idea is that turning wires on and off makes things happen in the computer. An instruction is fetched from memory and executed simply by turning on the right wires in the right sequence. All the wires are connected to the outputs from a ***Control circuit*** which is responsible for turning them on and off. All this is explained carefully in *The Most Complex Machine*. In the next section of the lab, you will get to see it in action, at least in simulation.

The **Options** menu contains another sub-menu that controls how fast the simulated xComputer runs. Speed number 1 is fastest. At this speed, the computer's registers disappear from the screen so that no time is wasted writing their contents. It can be particularly useful to use this speed with the Graphics memory display style—if you look closely, you can see individual bits changing in memory as the program runs. This is, in a way, the most realistic view of the computer's memory that you will ever see.

Speeds 2, 3, 4 are successively slower. At these speeds, you can watch the contents of the registers change. At the Slow speed, number 4, you will have time to notice each little step in the fetch-and-execute cycle.

Speeds number 5 and 6 are a bit different. At these speed settings, the computer does not proceed automatically from step to step. Instead, a button labeled "Next" appears when you run the computer, and you have to click on this button every time you want the computer to execute a step. The difference between speed 5 and speed 6 is what is meant by a "step." At speed 5, a step is a complete fetch-and-execute cycle, that is, the execution of one machine language instruction. Each fetch-and-execute cycle is really made up of a number of simpler, lower-level steps. It is these lower level steps that you see at speed 6.

Experiment with different speeds as the counting program runs. Try to understand how the computer works. When you feel comfortable with the program, you are ready for the next exercise.

Stop the computer. Set the **Speed** *to number 6,* **Manual, by step.** *Set the* **Memory Display** *to* **Show Control Wires Instead**.

The control wires are the key to understanding how the computer works. They are turned on and off by the Control Circuit, and they control the operation of other components of the CPU. Each control wire has a function. Turning that wire on causes something to happen, such as moving a number from main memory into the AC register or adding the numbers in the X and

Y registers and putting the answer into the AC. Executing a program is just a matter of turning the right wires on and off in the right sequence.

How does the control circuit know which wires to turn on at each step? It only needs two pieces of information: (1) Which step of the fetch-and-execute cycle is currently being performed. This information is available in the COUNT register. And, during the "execute" part of the cycle, (2) what machine language instruction is being executed. This information is in the IR register.

At speed 6 and memory display **Show Control Wires Instead**, you can step through the fetch-and-execute cycle and see how each step is performed.

Reset the PC to zero and then restart the computer to run the program again starting from the beginning. The "Next" button appears. Watch as the first instruction of the program, Lod-c 17, is fetched and executed:

- First click on the Next button: COUNT becomes 1, indicating that the first step in the fetch and execute cycle is being performed. The Load-addr-from-PC control wire is turned on, so the value in the PC register is copied into the ADDR register. (The PC register tells which memory location holds the next instruction; that location number must be copied into the ADDR register so that the computer can read that instruction from memory.)

- Second click: COUNT becomes 2; Load-IR-from-Memory control wire is turned on; the next program instruction is copied from memory into the IR. (The ADDR register determines which instruction is read.) In this case, the instruction is Lod-c 17

- Third click: COUNT becomes 3; Increment-PC control wire is turned on; the value in the PC register is incremented by 1, changing in this case from 0 to 1. This prepares the PC for the *next* fetch-and-execute cycle: the *next* instruction will be read from location 1. This completes the "fetch" portion of the fetch-and-execute cycle. The remaining steps depend on the particular instruction that is begin executed.

- Fourth click: COUNT becomes 4; Load-AC-from-IR control wire is turned on; 17, the data value in the instruction in the IR register, is copied into the accumulator. This is the only step necessary to execute the Lod-c 17 instruction. Other instructions are more complicated.

- Fifth click: COUNT becomes 5, but only briefly; Set-COUNT-to-Zero control wire is turned on and immediately the value of COUNT is reset to 0. One fetch-and-execute cycle is over. On the next click, COUNT becomes 1 again, and the next cycle begins.

As you click on the Next button in this exercise, you are actually simulating the role of the xComputer's clock. Each click has the same effect as one tick of the clock, and you are driving the computation at your leisure in the same way as the ticking of the clock usually drives the computer with its regular ticking.

Continue clicking through a few more execution cycles. Exercise 2 at the end of the lab asks you to follow the execution of another instruction in detail, as we have done here for lod-c 17.

Count and Store: As a final exercise, enter the following program into xComputer's memory. The easiest way to do this is to type it into a window, and then use the **Load** command. (Remember that the **Load** command won't work if a program is running.) This program is similar to the previous counting program, except that the number for the second sto command has been changed, and six new instructions have been inserted before the jmp command:

```
lod-c 0
sto 12
lod 12
inc
sto 13
lod 2
inc
sto 2
lod 4
inc
sto 4
jmp 2
```

The instructions "lod 2; inc; sto 2" add 1 to the number stored in memory location 2. Similarly, "lod 4; inc; sto 4" adds 1 to the number in location 4. But if you look at what's in locations 2 and 4, you'll see the instructions "lod 12" and "sto 13." This seems odd. What happens when you "add 1" to an instruction?

Remember that machine language instructions are "really" just numbers. There is no problem with adding 1. However, the meaning of the instruction is changed. If you add 1 to the number that encodes "lod 12," the meaning of the answer is "lod 13." Similarly, if you add 1 to "sto 13," you get "sto 14." If you want to understand exactly why this is true, check the format of machine language instructions given at the beginning of Section 3.2 of *The Most Complex Machine.*

Run this program and see what it does. Try running it at highest speed with the memory display set to "Graphics." Exercise 4 at the end of the lab asks you to figure out what this program does and why.

* * *

Exercise 1: *It was noted above that you can use xComputer to translate from one type of data to another by entering it in one form in the "data" input box and viewing it in memory in another form. Use this method to do the following conversions, and explain briefly how you do each part:*

a) *Find the ASCII code for the character $.*

b) *Find the character whose ASCII code is 103.*

c) *Find the binary representation of* -233.

d) *Find the unsigned integer that has the same binary representation as the signed integer* -233.

e) *Find the unsigned integer that represents the assembly language instruction "sto 1023." Add 1 to that number, and then find the assembly language instruction represented by the resulting number. Why do you get a completely different instruction? (Note: Do the addition yourself; you don't have to program the computer to do it!)*

Exercise 2: *The instruction lod 17 tells the computer to copy a number from memory location 17 into the accumulator. Use xComputer to watch as a lod 17 instruction is executed step-by-step, just as you did above for the instruction lod-c 17. To do this, enter a lod 17 instruction into memory location zero. Set the run speed to speed number 6, set the display style to* **Show Control Wires Instead**, *and make sure that the PC contains a zero. Then step through the fetch-and-execute cycle as the lod instruction is executed. Write down what happens during each step. Carefully explain the purpose of each step in the execute phase of the cycle. What differences do you find between the execution of a lod-c instruction and the execution of a lod instruction?*

Exercise 3: *Modify the first counting program used in this lab so that it will count just from one to sixteen, stopping when it reaches sixteen. To do this, you need to test whether the number is sixteen and, if it is, jump to a HLT instruction at the end of the program. You can test whether a number is equal to sixteen by subtracting it from 16 and testing whether the answer is zero. And you can do that with a JMZ instruction. This is like a JMP instruction, except that the jump only occurs if the number in the AC is zero.*

Exercise 4: *Describe what is done by the Count and Store program you encountered at the end of the lab, and discuss in detail how it works.*

Lab Number 5 for

The Most Complex Machine

by David J. Eck

Assembly Language Programming

About this lab: The machine language for xComputer consists of thirty-one different instructions, each of which performs a very simple task. Nevertheless, very complex programs can be built up from these instructions. This lab is an introduction to programming in xComputer's machine language. In fact, though, you will actually be using assembly language rather than the nearly unreadable (to humans) machine language itself.

You should be familiar with the assembly language of xComputer, as covered in Chapter 3 of *The Most Complex Machine*, including the idea of ***labels*** that is introduced in the postscript to the chapter. You should also be familiar with the workings of the xComputer simulation program that was introduced in Lab 4.

The Assembler: In Lab 4, you saw how individual instructions are executed in xComputer in a step-by-step fashion. That lab used only a few different instructions in very small programs, but it was noted that larger programs can be entered in windows, saved in files, and loaded into xComputer's memory when needed. When such a program is loaded into

---

This lab and the software used in it accompany *The Most Complex Machine: A Survey of Computers and Computing*, an introductory computer science textbook by David Eck (published 1995 by AK Peters, Ltd., 289 Linden Street, Wellesley, Massachusetts 02181; ISBN 1-56881-054-7). Support for the text, software and labs is available on the World Wide Web at http://math.hws.edu/TMCM.html. David Eck can be reached at the e-mail address eck@hws.edu.

memory to be executed, it must be translated into machine language. Recall that a program that translates assembly language into machine language is called an ***assembler***. The *xComputer* simulation program includes an assembler that permits the use of labels and other neat things in assembly language programs.

To see some of the possibilities, start up *xComputer* and open the file *3N+1 Sequence*. It should be in a folder called *Files for Lab 5*. This file contains an assembly-language program that computes a "3N+1 sequence." (This is a problem that you will see several times in *The Most Complex Machine*.) Given a positive integer N, the program applies the rule: "If N is even, then replace N by $N/2$; if it is odd, then replace it by $3N + 1$." It applies this rule over and over until the number N becomes equal to 1. For example, if the starting value of N is 7, then the program generates the sequence of values: 7, 22, 11, 34, 17, 52, 26, and so on.

Load the program from the window 3N+1 Sequence, using the first command in the **Assembler** *menu. (You can only do this if the 3N+1 Sequence window is the front window on the screen, and the xComputer is not already running.) Run the program. To see the numbers as they are generated, watch memory location number 115, which contains the successive values of N starting with 3. You will probably want to bump the speed up to speed number 2,* **Fast**.

The assembler for *xComputer* understands all of the data-entry formats explored in Lab 4: assembly language instructions, base-ten integers in the range -32,768 to 65,535, binary numbers of from 1 to 16 bits (provided the number is preceded by a "B"), and individual characters (which must be preceded by single quotes and which are translated into into their ASCII code numbers). The program can contain at most *one* of these things on each line. When a line in a file contains one of these items, that item is translated into a binary number and loaded into the next available location. However, an assembly language program can also contain other things.

A program can contain ***comments*** meant purely for human readers. A comment starts with a semicolon (;) and ends at the end of the line. These comments are completely ignored by the assembler. There are lots of comments in the *3N+1 Sequence* file. *You should read them.*

A program can use ***labels*** to refer to memory locations. A label is a name that is defined in a program to stand for some specific number. That number is the address of some location in memory. For the most part, a label can be used any place a number can be used. For example, if "Lbl" is a label in a program, that program could use the instruction "JMP Lbl" to jump to the location specified by Lbl. Since Lbl is really just a number, that program could also use instructions like "LOD Lbl" or even "Add-c Lbl." Besides these ***references*** to the label Lbl, the program must contain exactly one ***definition*** of this label. A label is defined by using it to label a memory location, like this:

```
Lbl:  Lod 10
```

Here, Lbl will be equal to the address of the memory location that ends up holding the instruction "Lod 10" when the program is loaded into memory. The whole point of labels is that as you write the program, you don't need to know what the location number will be.

The program *3N+1 Sequence* contains five labels: NextN, Even, Odd, Done and N. The label Even is defined but is never referred to; it is really there just for human readers.

Look at the label N, which occurs near the end of the program. Like all labels, N is the name of a memory location, but in this case the location is loaded with a number, 7, rather than with an assembly language instruction. The 7 is meant to be used as data by the program. The name "N" can be used in the program to refer to that data. Of course, N is still really just a number indicating a location in memory. When the program is assembled, it turns out that N is a label for memory location 115. Therefore, the instruction "Lod N" in the program is equivalent to the instruction "Lod 115," which loads the contents of memory location 115 into the accumulator.

The starting value 7 in location N can be changed to any other number. Try making such a change and then re-loading and re-running the program. The program will generate a 3N+1 sequence starting with whatever number you specify.

A program can contain a few other neat things, such as the "@100" at the start of *3N+1 Sequence*. You can read about this and other features in the comments in the sample programs you will look at. Here are few final remarks on the assembler:

- The assembler makes no distinction between uppercase and lowercase letters in instruction names and labels. Thus, for example, "LOD," "lod" and "Lod" are all equivalent.
- Binary numbers and characters can be used in instructions in place of decimal numbers, as in "Add-c B1101" and "Lod-c '$".
- The **Disassemble** command in the **Assembler** menu will produce a new window containing an assembly language program representing the contents of xComputer's memory at the time the **Disassemble** command is used. This program might contain some funny-looking instructions, such as "Hlt 16," that you wouldn't write yourself. This is the consequence of trying to interpret data in memory as instructions, even though not all binary numbers stand for legal instructions.

Loops and Decisions: Programs are structured by ***loops***, in which a group of instructions is repeated over and over, and ***decisions***, where a choice is made between alternative courses of action. The 3N+1 program shows both of these structuring methods. It has a loop which computes one step in the sequence. This loop is repeated until the sequence reaches 1. Inside this loop is a decision between the actions "replace N by N/2" and "replace N by 3N+1."

There is another sample programs that you can look at, *Multiply by Adding*, which also contains a loop but is somewhat simpler than *3N+1 Sequence*. It will be important for you to understand how these programs work. You should read them carefully. A good way to get a feel for how they work is to run them at speed number 5, **Manual, by Instruction**. At this speed setting, you drive the execution of the program by clicking on a button labeled "Next." Each time you click, the xComputer goes through one full fetch-and-execute cycle, executing one machine language instruction. Click through the program and observe what happens. (As you watch, note that the instruction you see in the IR register is the one that has just been executed, *not* the one that is about to be executed.)

Make sure that you understand these programs and the general ideas of loops and decisions. You will need to understand them to complete the exercises at the end of the lab.

Dancing Bits:

There is another sample program that is provided mostly for your amusement, but also because watching it run might just give you a better appreciation of just what a computer is really doing as it computes. Open the file *Powers of Three*, read the comments at the beginning of the file, and then load the program. Set the display style to **Graphics** and the run speed to **Fastest**, and run the program. You will see the bits in xComputer's memory dance as a non-trivial computation is performed.

* * *

Exercise 1. *Modify the 3N+1 program so that it counts the number of steps it takes for N to become 1. To do this, add another labeled location at the end of the program. Call it "Count." Change the program so that it starts by storing a zero in location Count. Each time through the loop, the program should add 1 to Count. When the program ends, the value in Count is the number of times the program has gone through the loop. This is also the number of steps that were computed in the sequence. Add comments to the program to indicate the modifications you have made. Make a printout of your program.*

Exercise 2. *Look at the file ListSum. This file contains part of a program for adding up a list of numbers. The assignment, which is to complete* *the program, is described in full in that file. Make a printout of your program.*

Exercise 3. *The program for Exercise 2 uses indirect addressing. (This part is done for you!) Explain how indirect addressing works in that program and why it is used.*

Exercise 4. *Just as it is possible to multiply by adding over and over, it is possible to divide by subtracting over and over. Suppose you want to know how many times N1 goes into N2. Start with N2 and subtract N1 repeatedly until the answer is less than N1. The number of subtractions you performed is the number of times that N1 goes into N2. For example, you can compute that 4 goes into 14 three times by computing* $14-4-4-4=2$. *(The number 2 that you end up with here is the* remainder *when 14 is divided by 4.)*

Write a program to compute how many times a number N1 goes into another number N2. Your program will be somewhat similar to the sample program Multiply by Adding. *You still need a loop, and you still need to count how many times that loop is executed. However, the set-up before the loop, the action performed in the loop, and the test for ending the loop are different. Note that to test whether* $A < B$, *you can test whether* $A - B < 0$ *with a JMN instruction.*

Your program must have comments that describe what it does and how it works. Make a printout of your program.

Lab Number 6 for

The Most Complex Machine

by David J. Eck

Subroutines for xComputer

About this Lab: A subroutine is a set of instructions for performing some task, chunked together and thought of as a unit. Like loops and decisions, subroutines are useful in the construction of complex programs. The machine language for xComputer does not provide direct support for subroutines. But then again, it doesn't really provide direct support for loops and decisions, which must be implemented by the programmer with jump and conditional jump instructions. Similarly, it is possible to implement subroutines using jump instructions. They are not as easy, as neat, or as safe as subroutines in a high-level language, but they can still be a valuable tool. And you get to see how subroutines can be made to work even on a very minimal computer. (You should understand, though, that the machine languages of real computers do provide more support for subroutines than what is covered here.)

Before doing this lab, you should be very comfortable with the material from Lab 4 and Lab 5. This lab is based on the assembly language of the xComputer, as described in Chapter 3 of *The Most Complex Machine*, but it goes beyond the material presented there.

Making and Using Subroutines: The idea of subroutines is simple enough. A subroutine is just a sequence of instructions that performs some specific task. Whenever a program needs to perform that task, it can ***call*** the subroutine to do so. The subroutine only has to be written once, and once it is written, you can forget about the details of how it works. You only need to know what it does and how to call it.

In xComputer's assembly language, "calling" the subroutine means jumping to the first instruction in the subroutine, using a JMP instruction. The execution of the subroutine will end with a jump back to the same point in the program from which it was called. This is known as ***returning*** from the subroutine. (Other computers provide special commands for calling and returning from subroutines.)

There is more to it than a few jump instructions, though, because there usually has to be some kind of communication between the subroutine and the rest of the program. In particular, the subroutine needs some way of knowing the location in the program to which it must return. The address of that location is called the ***return address*** for the subroutine. Since a subroutine can be called from several different places in a program, the return address can be different each time the subroutine is called. So, whenever the subroutine is called, the return address must be provided to the subroutine as data. In a high-level language this is done automatically, but for now you have to know how to do it yourself. Before jumping to the start of a subroutine, you must explicitly store the return address in a memory location reserved for that purpose. The final instruction in the subroutine will then be a jump to the address specified in that location.

The return address is an example of data sent from the main program to the subroutine. Many subroutines require additional data from the main program in order to perform their tasks. For example, a subroutine whose task is to multiply two numbers needs to know what the two numbers are. Sometimes, a subroutine has to communicate data back to the main program. For example, after the multiplication subroutine ends, the program will want to know the answer that it computed. Data values exchanged between the subroutine and the main program are called ***parameters*** for the subroutine. Parameters can be implemented in the same way as the return address: The parameter is simply stored in a location known to both the main program and to the subroutine. (In the xComputer, a parameter value could also be passed in the accumulator register, which has some advantages but can only accommodate one parameter.)

So, when a program calls a subroutine, it goes something like this: First, the program loads any parameter values needed by the subroutine into the appropriate memory locations. Then, it loads the return address into the memory location reserved for that purpose. Next, it executes a jump to the first instruction in the subroutine. (The return address is just the address of the location that follows this jump instruction.) When the subroutine finishes, it jumps back to the return address. At that point, the program can access any parameter values computed by the subroutine.

You might ask, is it really worth all this trouble? If the subroutine is very short, the answer is probably no. But if the subroutine performs some complex task, then the work that goes into calling it is much less than the work it would take to re-write it. You should also remember that in high-level languages, much of the work of calling a subroutine is done automatically, behind the scenes.

It's time to take a look at an example to see how all this works in practice. Open the file *ListMax*, which you should find in a folder named *Files for Lab 6*. This file contains a subroutine called *find_max* and a program that uses it. The subroutine is at the end, starting at about the midpoint of the file. ("*Find_max*" is really just a label for the first instruction in the subroutine; calling the subroutine will mean jumping to this label.) The subroutine searches through a list of numbers to find the largest number in the list. *Read all the comments on the subroutine.*

The subroutine is preceded by a "data area" consisting of memory locations to hold the return address and the three parameters used by the subroutine. Each of these memory locations has a label that can be used to refer to it conveniently. (Recall that the "data" command, used for example in "`ret_adr: data`" just tells the computer to reserve a memory location that the program will use to hold data.)

In addition to the parameters, the data area includes some memory locations that are for purely internal use by the subroutine. These memory locations are not parameters—they are not used for communication with the main program. Ideally, the main program wouldn't need to know anything about them at all.

Taken all together, the return address, the parameters, and the local data form what is called the ***activation record*** for the subroutine. The activation record could in fact be anywhere in memory. It was only to make it easier to find that I included it right before the subroutine.

Finally, look at the very end of the subroutine, where you will find the instruction "`JMP-I ret_adr`". This is the instruction that jumps back to the return address. Recall that the JMP-I instruction uses indirect addressing. It doesn't jump to location *ret_adr* but to the location whose address is stored in location *ret_adr*. This is, of course, exactly what we want because location *ret_adr* contains the return address.

The main program in file *ListMax* is very simple. It just calls the subroutine and then saves the value that was computed by the subroutine in a memory location named *answer*. It is worth studying this main program carefully and in detail. It starts by loading parameter values in locations *list_loc* and *size*, which are within the subroutine's activation record. It then specifies the return address by loading the number *back* into *ret_adr*. The next instruction begins execution of the subroutine by jumping to location *find_max*. The three instructions starting at location *back* are executed after the subroutine has completed. This part of the program just saves the parameter value that the subroutine has computed and stored in location *max*.

You should run the program to see it in action. Also try changing the list of numbers and the size of the list to make sure that the program still works with the modified data. Recall that to run a program, you should fist load it into memory with the **Load** command in the **Assembler** menu. Then, make sure that the PC register contains a zero (to indicate the starting instruction of the program), and click on the **Stop Clock** box to begin execution.

* * *

Exercise 1: *Write a subroutine to add two numbers. (This is a pretty silly thing to do—the point is to demonstrate that you understand the basic concepts involved.) Your subroutine should have three parameters: the two numbers to be added and their sum. Write a main program that uses your subroutine to add 17 to 105. Turn in a listing of your program, including the subroutine. The subroutine should be carefully commented.*

Exercise 2: *The file ListSort contains a subroutine, "swap_max" that finds the largest number in a list of numbers and then exchanges that maximum value with the number at the end of the list. This subroutine is very similar to the "find_max" subroutine that you looked at earlier in the lab, except that after finding the maximum it moves it to the end of the list. It is possible to* ***sort*** *a list of numbers into increasing order by repeatedly calling swap_max. When sorting a list of 10 numbers, for example, first call swap_max with a size parameter equal to 10. This will move the largest item into position 10 in the list. If swap_max is then called with size equal to 9, it will ignore the number in position 10 and will find the next largest item and move it into position 9. Calling swap_max with size 8 will then move the third-largest item into position 8, and so forth.*

Read the comments in the file ListSort. Write a main program that will sort the list of numbers by repeatedly calling the swap_max subroutine, as described in the comments. Use a loop. It should be easy to modify your program to sort a longer list. (Note: As you are developing your program, you might find it convenient to include an extra HLT command or two. The computer stops when it gets to a HLT, but if it is restarted it will just continue with the next instruction. The extra HLTs will let you run the program at full speed but make it stop at those points where you would like to inspect what is happening more closely. A HLT instruction used in *this way is called a "breakpoint." Breakpoints are very useful for debugging a program. You can remove the extra HLTs after the program is working perfectly.)*

Exercise 3: *The file PrimeNumbers contains a subroutine that computes the remainder when one number is divided by another. (For example, the remainder when 23 is divided by 5 is 3.) In this exercise, you will use this subroutine to make a list of* ***prime numbers****.*

We say that the number n is ***evenly divisible*** *by the number d if the remainder when n is divided by d is zero. A number p, greater than 1, is prime if it is not evenly divisible by any number between 2 and $p - 1$. In fact, it is enough to check that p is not evenly divisible by any prime number between 2 and $p - 1$.*

The comments in the file PrimeNumbers explain how to write a program to make a list of prime numbers. Read the comments and write the program.

Exercise 4: *A subroutine is a "black box" that can be used without understanding the internal workings of the subroutine. The black box has an* ***interface*** *that provides for communication between the black box and the rest of the program. The "internal workings" of the black box make up its* ***implementation.*** *Look back at the subroutine "list_max." What exactly is the interface of this subroutine? What is its implementation? The xComputer assembly language does not support the black box concept very effectively. Why not? (Think about all the labels used internally in the subroutine. Ideally, when you write the main program you wouldn't need to know about these labels. But in fact, you at least need to know that they exist.) Write an essay answering these questions.*

Lab Number 7 for

The Most Complex Machine

by David J. Eck

Turing Machines

About this Lab: Turing machines are extremely simple calculating devices. A Turning machine remembers only one number, called its ***state***. It moves back and forth along an infinite tape, scanning and writing symbols and changing its state. Its action at a given step in the calculation is based on only two factors: its current state number and the symbol on the tape that it is currently scanning. It continues in this way until it enters a special state called the ***halt state***. In spite of their simplicity, Turing machines can perform any calculation that can be performed by any computer. In fact, certain individual Turing machines, called ***universal Turing machines***, can actually execute arbitrary programs, just as a computer can. You won't see any universal Turing machines in this lab, but you will experiment with Turing machines that can perform non-trivial calculations.

You should have some familiarity with the material on Turing machines from Chapter 4 of *The Most Complex Machine* before beginning this lab. Especially important is the idea that a Turing machine is described by a table that specifies what action the machine will take for each combination of state and scanned symbol. The action takes the form of writing a new (or

This lab and the software used in it accompany *The Most Complex Machine: A Survey of Computers and Computing*, an introductory computer science textbook by David Eck (published 1995 by AK Peters, Ltd., 289 Linden Street, Wellesley, Massachusetts 02181; ISBN 1-56881-054-7). Support for the text, software and labs is available on the World Wide Web at http://math.hws.edu/TMCM.html. David Eck can be reached at the e-mail address eck@hws.edu.

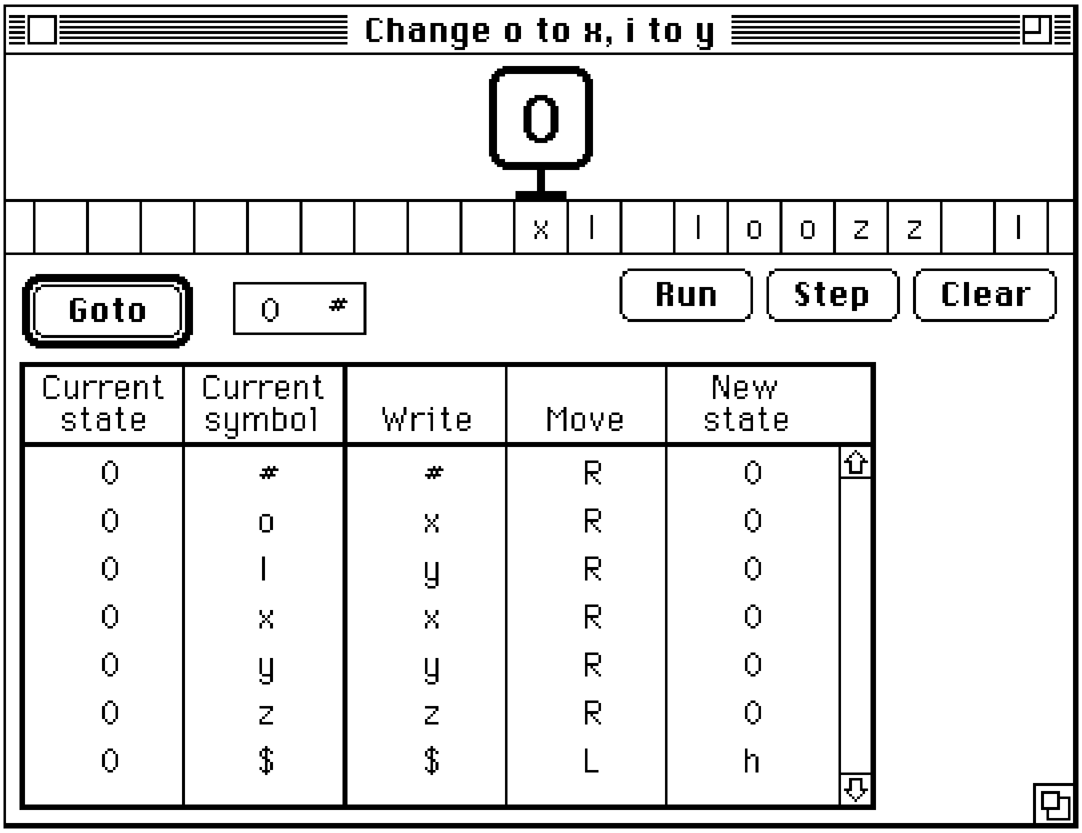


Figure 7.1. *A window from the program xTuringMachine showing a Turing machine and its tape, along with the table of rules that determine the behavior of the machine.*

the same) symbol to the current square, moving either left or right on the tape, and entering a new (or the same) state.

A Trivial Machine: In this lab, you will work with a program called *xTuringMachine*. This program simulates Turing machines with up to twenty states. The symbols that can be used on the tape are the letters i, o, x, y and z; the dollar sign $; and the blank space (sometimes written as #). The letters i and o are meant to look like the digits one and zero, and will be used to represent "binary numbers" on the tape. But remember that the *meaning* of a symbol has no effect on any calculation performed by a Turing machine; all the machine does is follow its rules.

To start the lab, you will look at a simple Turing Machine in action. *Open the file Change o to x, i to y. You should find it in a folder named Files for Lab 7. When you double-click on the file, the program xTuringMachine will start up and a window will open like the one shown in Figure 7.1.*

At the top of the window is the Turing machine itself, which looks a little like a small computer display screen. This "screen" displays the current state of the machine. This state can be either an "h", indicating the halt state, or a number between 0 and 19. The machine starts in state 0.

Just below the machine is an infinite tape made up of squares that can each contain one symbol. It extends, conceptually, an infinite distance off the right and left edges of the window. The Turing machine moves back and forth along this tape as it computes, reading and changing the symbols in the squares.

At the bottom of the window is the table of rules that defines the behavior of the machine. Each row of the table specifies one rule. In Figure 7.1, for example, the first row of the table says "If the machine is in state 0 and if the symbol in the current square is # (that is, a blank), then the machine will write a # in the square, move one square to the right, and change to state 0." All the rules for a Turing machine are of this general form. Note that in this case, the symbol it writes in the square is the same as the symbol that was already there; this is just a fancy way of saying that it doesn't change the contents of the square. Similarly when the machine "changes to state 0" in this case, it doesn't really change its state; it just stays in the same state that it was already in.

Below the tape and above the rule table in the window are some controls that are used to add new rules to the table and to control the execution of the Turing machine. If you click on the **Step** button, the machine will do a single step in its computation; it applies the appropriate rule in the table, depending on its current state and on the symbol in the square where it is sitting. (Also, to help you follow what is going on, a box will be drawn in the table around the rule which will apply in the *next* step of the computation.) If the machine enters its halt state, then its computation is finished. In that case, the **Step** button becomes a **Reset** button; clicking on the **Reset** button will restore the state of the machine to 0 so that it can begin a new computation.

Step through the execution of the Turing machine "Change o to x, i to y" by clicking on the **Step** *button as often as necessary. The name of the machine tells what it does. How does it do this? What makes the computation halt? What happens if there is no $ on the tape? What happens when the machine reaches the edge of the window?*

There are a few other aspects of the program *xTuringMachine* that you can try out on this rather trivial machine. *Try these before you go on to the next section.*

• *Modifying the tape.* You can change the contents of any square on the tape manually. If you move the mouse cursor over one of the squares, the cursor will change to a "pencil." If you then click and hold the mouse button, you will get a ***pop-up menu*** that you can use to change the contents of the square. For example, you can change some of the x's and y's on the tape back to o's and i's. It is also possible to change the contents of the entire tape at once, using the **Type in Tape** and **Clear Tape** commands from the **Control** menu.

• *Changing the state.* You can change the state of the Turing Machine by hand (but only if the machine is not running). Just click on the state displayed on the machine's screen. You will get a pop-up menu from which you can select the new state. You will find this option handy when you are creating a complex machine and you want to make it back up to a previous state.

• *Moving the Turing machine.* To move the Turing machine by hand, hold down the Option key on the keyboard and point to the machine with the mouse. When the cursor is on top of the machine, it will change from an arrow to a "hand." You can drag the machine with this hand by pressing the mouse button and holding it down as you drag. This allows you to move the machine to a different square on the tape. *Important note:* If you don't hold down the Option key while you press the mouse button, you can still use the hand cursor to move the machine, but the tape will be dragged along with it.

• *Running the machine.* As an alternative to pressing the **Step** button over and over, you can simply click once on the **Run** button. The machine will compute automatically until it reaches its halt state. The **Run** button becomes a **Stop** button while the machine is running; you can click on this **Stop** button to interrupt the computation. The speed at which the machine runs depends on the setting of the **Speed** menu. (At the highest speed, the machine's "state display screen" disappears so that it can run as quickly as possible.)

A More Interesting Machine: As another example, open the file *Find double-x.* This file defines a machine whose sole purpose is to move to the right along its tape, until it finds two x's in a row; it then halts on the leftmost of those two x's. The machine you looked at in the previous section had only a single numbered state, state 0. The machine *Find double-x* has two states, number 0 and number 1. As this machine runs, you will see it changing between these two states. *Try it. Use the* **Step** *button to step through the computation one step at a time.*

Although its states are completely meaningless to the machine, from a human point of view, we can assign a kind of meaning to each state. In state 0, the machine is "moving to the right searching for an x." In state 1, it "has found one x and needs to check the next square to see whether there is another x there." In state 1, after checking the next square, it halts if it finds an x there and returns to state 0 if not. (One might say that the state counts the number of x's in a row that the machine has encountered.)

Current State	Current Cell Contents	New Cell Contents	Direction of Motion	New State
0	x	#	L	1
1	#	x	R	4
4	#	#	R	0
0	y	#	L	2
2	#	y	R	4
0	z	#	L	3
3	#	z	R	4
0	#	#	L	5
5	#	#	L	6
6	x	x	L	6
6	y	y	L	6
6	z	z	L	6
6	#	#	R	h

Figure 7.2. *A rule table for a Turing machine that nudges a string of x's, y's and z's one square to the left. The machine must be started on the leftmost symbol in the string, and it will only work if there are a couple of blank squares surrounding the string.*

As your first exercise at the end of this lab, you will change this machine so that it searches for three x's in a row instead of two. To make such changes, you will need to know about editing the rule table. This is covered in the next section of the lab.

New Machines: Figure 7.2 shows a rule table for a Turing machine that nudges a string of x's, y's and z's one square to the left on its tape. In this section of the lab, you will create an identical machine from scratch. (You will find a ready-made copy of this machine in a file named *Nudge Left*, but since you need to learn how to build Turing machines, you should make your own.)

Begin by using the **New** *command from the* **File** *menu to open a new window. The machine in the window will have a blank tape and a blank rule table. Enter a string of x's, y's and z's, such as "xxyxzz", on the tape, and move the Turing machine to the leftmost symbol in the string. You are ready to build the rule table for the machine.*

The program *xTuringMachine* does not allow you to simply type in a rule. Instead, it has procedures for adding a new rule to the table and for modifying rules that are already in the table.

Rules are added to the table using the **Make** button, which is located just above the rule table in the window. Just to the right of this button, you can see a *state number* and a *symbol*. If you click on the **Make** button or press the return key, a new rule is added to the table that tells the machine what action to take when it is in that state and sees that symbol in the current square. When the rule is first entered, it specifies a default action: leave the same symbol in the current square, move to the right and stay in the same state. Once the rule is in the table, you can change the action it specifies. (Note: If a rule *already* exists in the table for the specified action and state, the **Make** button becomes a **Goto** button; clicking on the **Goto** button will find the existing rule and draw a box around it.)

You can change the specified action for any rule in the table by changing the symbol listed in the *Write* column, the direction listed in the *Move* column, or the state listed in the *New State* column. Such changes can be made using the mouse in the same way that it is used to change symbols on the tape: Point to the item you want to change; the cursor changes into a pencil; then, when you click and hold the mouse button, you get a pop-up menu from which you can select the value you want. (Note that you *cannot* change the entries in the first two columns of the table.)

As you might have guessed by now, you can also change the state number and the symbol next to the **Make** button by using the mouse in the same way. This allows you to specify which rule you want to create.

All this allows you build any desired rule table, but there are a few other convenient features that you might want to explore: First, note that there is usually a *box* drawn around one of the rules in the table or around the state and symbol next to the **Make** button. (On color monitors, the box is red.) You can move the box around with the arrow and tab keys on the keyboard, or by clicking with the mouse (when the cursor is an arrow, not a pencil). We say that the boxed item is ***selected***. You can use the keyboard to change values in the selected item. For example, if a rule is selected and you type an "x", then the symbol in the *Write* column will change to an x. If you type "17", then the state in the *New State* column will change to 17. You can delete a selected rule from the table with the **Delete Rule** command from the **Control** menu. And you can drag a selected rule to a new position in the table.

Finally, there is one more feature to make it easier to build a rule table from scratch. If you use the **Step** button to run a machine step-by-step, you will notice that at each step, the rule that will be applied on the next step is boxed. If there is no rule for the current combination of state and symbol, then the current state and symbol will appear in a box next to the **Make** button. At this point, just click on **Make** to make a rule to handle the current state and symbol, and set the "Write," "Move" and "New State" data in that rule to perform the action you want. You can build an entire rule table in this way, stepping through a computation and making rules as you need them.

Create the table for the Nudge Left machine shown in Figure 7.2. Run *the machine and make sure you understand how it works. This machine is the basis for Exercises 2 and 3 at the end of the lab. If you are not sure your version is correct, compare it to the ready-made version in the Folder Files for Lab 7. (Note that the order in which rules appear in the table is not important.)*

Binary Arithmetic: The operations of incrementing (adding one to) or decrementing (subtracting one from) a binary number are simple enough to be done easily by Turing machines.

The procedure for adding one can be described as follows: If the last digit of the number is a zero, then simply change that digit to a one. If the last digit is a one, change it to a zero, and "carry" a one to be added to the next digit to the left. Finally, to add one to a blank space, simply change that blank to a one. (This can occur when a one is carried beyond the leftmost digit of the number; the blank should be treated just like a zero.) For example, $110_2 + 1_2 = 111_2$, $1011_2 + 1_2 = 1100_2$ and $11_2 + 1_2 = 100_2$.

A simple Turing machine for incrementing a binary number is found in the file *increment*. (This is the machine specified in Figure 4.2 in the text.) This machine must be started on the rightmost digit of the number to which it will add one. After it is finished adding one, it returns to this position and halts. *Open the file named Increment, and run the machine a few times to see how it works.*

If you modify this machine so that instead of halting, it returns to state zero, then it will add one to the tape over and over again forever (or until you halt it manually). It is, in effect, counting. *Make this modification and let the machine run. Change the speed to maximum to see just how fast it can go. Why does each digit in the number change twice as fast as the number to its left?*

As described in the text, it is possible to combine an "incrementing" and a "decrementing" machine into a machine that can add two binary numbers. Given two binary numbers as input, this machine will repeatedly subtract one from the second number and add one to the first until the second number becomes zero. At that point, the first number will be equal to the sum of the two original numbers. A machine to do this can be found in the file *Add (by repeated add-1)*. This machine only works on a tape containing two binary numbers, separated by one blank space, and it must be started on the rightmost digit of the second number.

It is also possible to add two numbers in the ordinary way, by adding digits in corresponding "columns" in the two numbers. The file *Add (digit by digit)* contains a machine that adds two input numbers in this way. As it adds the second number to the first, this machine keeps track of which columns have already been added by replacing o's and i's with x's and y's.

When it finishes, it erases the second number from the tape and changes all the x's and y's in the first number back to o's and i's.

Finally, remember that two numbers can be multiplied by repeatedly adding the first number to itself. The second number tells how many times the addition is to be performed. The file *Multiply by repeated add* contains a Turing machine that can multiply two binary numbers in this way. Like the addition machines, it requires two binary input numbers on its tape. After it computes the answer, it erases the two original input numbers. The number remaining on the tape at the end of this process is the product of the two inputs.

To convince yourself that Turing machines can perform non-trivial tasks, watch these three machines in operation. They can all be found in the folder, Files for Lab 7. You don't need to understand every detail of their operation, but you should get a general idea of how they work.

* * *

Exercise 1. *Modify the Turing machine Find double-x to make a machine Find triple-x that will move right until it finds three x's in a row. To do this, you will have to add another state—state number 2—with the meaning "found two x's in a row." In state 1, if it finds a second x, it will enter this new state instead of halting. Make a printout of your rule table.*

Exercise 2. *The Nudge Left machine that you constructed in this lab moves a sequence of symbols one square to the left. Explain carefully how it works. In particular, discuss the meaning that you, as a human observer, can see in each of its states.*

Exercise 3. *The Nudge Left machine halts after moving its input string one square to the left. If you tell it to enter state 0 instead of halting, then it enters a loop in which it moves its input string to the left forever. It's fun to watch this at full speed. Try it! Next, suppose you want the machine to stop when it hits a $. This is a simple modification that requires only one more rule. Make it so.*

Now, here is the real assignment: Suppose you want a machine that will move a string of x's, y's and z's back and forth between two $'s on the tape. Such a machine can be made (partially) from a "nudge left" machine like the one you already have and a "nudge right" machine that is very similar.

Construct a "back and forth" machine of this type. Save your machine in a file. Turn in both a printout of your rule table and the file in which your machine is saved. This problem is not easy! Ask for help if you need it!

Exercise 4. *The multiplication machine you looked at in this lab is rather complicated. Trying to understand its operation is very difficult if you don't think in terms of "structured complexity." From a low-level point* *of view, this machine goes through a long, complicated sequence of individual steps. From a high-level, "black-box" point of view, it just multiplies two numbers. But to really understand the operation of the machine, you have to look at it on some intermediate level, where you can see a sequence of meaningful operations, structured perhaps by loops and decisions. Write a description of the operation of the "Multiply by repeated add" machine on an "intermediate level." When thinking about what is meant by this, consider the relationship of this machine to the "Add (digit by digit)" machine.*

Lab Number 8 for

The Most Complex Machine

by David J. Eck

Interrupts and I/O in xComputer

About this Lab: The xComputer, which you have used in some previous labs, is a very simple model computer, but it does demonstrate many aspects of the basic operation of real computers. This lab introduces two additional features of the xComputer: the ability to do a limited form of input/output processing, and the ability to respond to interrupts.

You will have to use a bit of imagination here, since I will not discuss all the wiring, circuitry, and programming that would be needed to add these features to the version of the xComputer discussed in the text. In this lab, you will work on the assembly language level rather than the level of circuits and control wires.

This lab is based on the xComputer, as discussed in Chapter 3 of *The Most Complex Machine*, but it uses features of the xComputer that are not mentioned in the text. You should be thoroughly familiar with the material covered in Lab 4 and Lab 5. This lab is meant to illustrate some of the aspects of real computers that are covered in Section 5.2 of the text. You

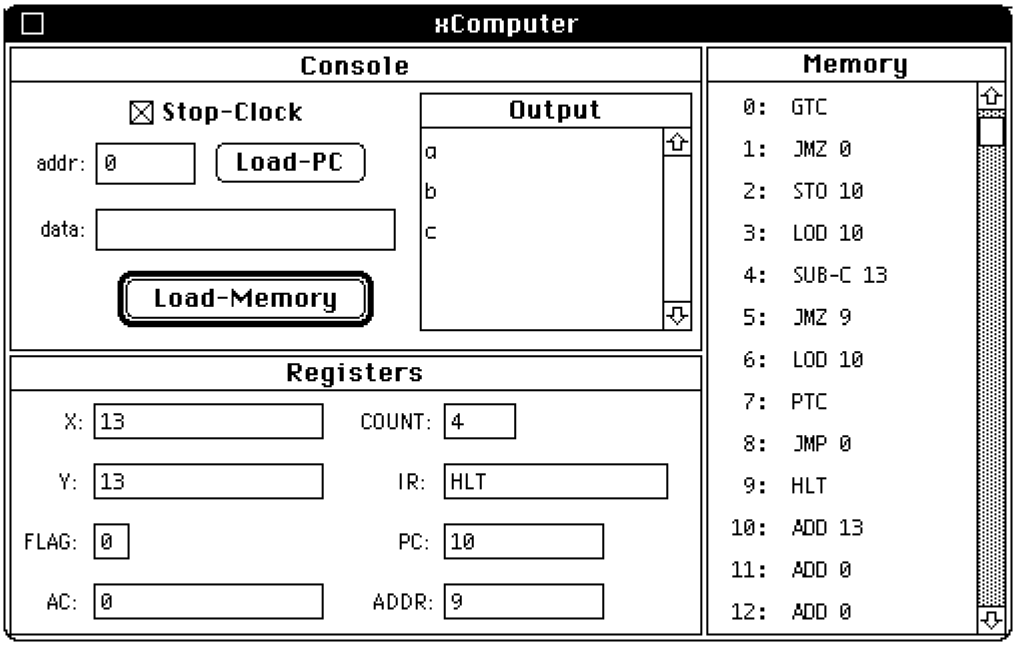


Figure 8.1. *The main xComputer window as it appears when the "Use I/O Services" option is turned on. For this picture, the sample program* EchoChars *was loaded and run. As it was running, the user typed in "abc" followed by a carriage return. The characters a, b, and c were echoed to the output, and the carriage return caused the program to halt.*

will find background material for the lab there, but for the most part the lab is self-contained.

Input/Output: To begin the lab, start up the program *xComputer*. Select the **Use I/O Services** command from the **Options** menu. You will notice some changes in the Console portion of the xComputer window, as shown in Figure 8.1. Turing on the I/O Services option makes the simulated xComputer into an essentially different machine that understands several new assembly-language instructions. To see some of these instructions in action, open the sample file *EchoChars* and load it into xComputer's memory, using the **Load** command in the **Assembler** menu. You should find *EchoChars* in a folder named *Files For Lab 8*.

The program *EchoChars* uses two new assembly language instructions: Get-Character (GTC) and Put-Character (PTC), which are used for input and output. These instructions will only be recognized by xComputer when I/O Services are turned on. (To see this, set the **Memory Display** (under the **Options** menu) to **Assembly Language**, so that you can see program *EchoChars* in memory in assembly language form. Then turn I/O Services on and off. You will see the GTC and PTC instructions in memory only when I/O Services are on. Make sure I/O services are on, and leave them on for the rest of the lab.

Run the program EchoChars. Set the **Speed***, under the* **Options** *menu, to speed number 2,* **Fast***. While the program is running, type in some characters and watch what happens. Press return when you want the program to end. Read the comments in the file EchoChars.*

Here is what happened: When you started the program, a box labeled "Input Queue" appeared in the Console portion of the xComputer window. When you type a character, it is entered in that box. (You should think of the input queue as an additional register or a special area of memory, and imagine that pressing a key on the keyboard turns on some wires that load the character you typed into that memory.) When the computer executes a GTC instruction, the first character in the input queue is removed from the queue, and its ASCII code is placed in the accumulator. If there are no characters in the queue, then a zero is placed in the accumulator as a signal to the program that there is no user input waiting to be processed. The program can then do whatever it wants with the character. In this case, a PTC instruction is used to output the character. (Output items are simply added to a list, labeled Output, in the xComputer Console. Think of this list as a roll of paper on which the computer types the output.)

The *EchoChars* program illustrates a typical method for processing user input. The computer executes a loop to read the user input. The first command in the loop is a GTC instruction, which retrieves an input character from the queue of characters typed by the user. If no character is found, then GTC retrieves a zero instead, and the program uses this in the instruction "`jmz get_ch`" as a signal to jump back to beginning of the loop with no further processing. Otherwise, it processes the character. This method of sitting in a loop, waiting for user input, and processing it when it occurs is known as ***polling***. It is an effective and simple means of organizing a program, but at full computer speeds, the program spends most of its time just waiting for the user to press a key.

In the *EchoChars* program, most user input is simply echoed to the output list. However, a carriage return is treated differently. When the program reads a carriage return, it jumps to a Halt instruction at the end of the program, and the program ends. The program implements this feature with the three instructions

```
LOD ch
SUB-C 13
JMZ done
```

If the character, *ch*, is a carriage return—with an ASCII code of 13—then the JMZ instruction transfers control to location *done*, where there is a HLT instruction.

Similar tests for other special characters can result in a sophisticated program. In the second sample program for this lab, *EchoNums*, ch is first tested to see whether it is a return, then to see whether it is a period, then to see whether it is in the range or characters from "0" to "9". Each of these cases results in a different action. This program allows the user to type in an integer as a sequence of digits. When the user presses the period key, the number is written to the output list.

Open the sample program EchoNums. Read the comments, then load the program and run it. (Don't forget to set the PC to zero before trying to run it!) Try typing in "123.", including the period. It would be useful to run this program at various speeds in order to trace how it works. If you run it at full speed and watch location 35 (num), you can see how the number changes as you type. When you run the program at lower speeds, it will take some time for each character to be retrieved from the input queue and processed.

EchoNums uses a Put-Integer (PTI) command for output. The PTI command writes the number in the accumulator to the output list in the form of a signed integer (in the range −32768 to 32767). In addition to PTC and PTI, there are two more commands for output: Put-Unsigned (PTU), which outputs an unsigned integer (in the range 0 to 65535), and Put-Binary (PTB), which outputs a number in binary form. GTC is the only command for input; whatever the user types, you have to read it character by character and process it one character at a time.

Many modern computer programs, including all the Macintosh programs used in these labs, are based on a more sophisticated version of the polling loop used in *EchoChars* and *EchoNums*. The main difference is that in addition to user input in the form of typed characters, there are several other types of ***events*** to which the program must respond. An event is generated, for example, when the user clicks the mouse button or when the user chooses a command from a menu. The operating system maintains an ***event queue*** that contains all the events that have occurred and that are waiting to be processed. The program executes a loop that repeatedly performs the actions: Get the next event from the event queue, and process that event. This type of loop is called an ***event loop***.

Interrupts and Interrupt Handlers: When you turn on I/O Services, in addition to enabling input/output instructions, you also turn on interrupt handling. The idea of an ***interrupt*** is that a signal on some designated control wire causes the CPU to interrupt what it is doing in order to handle whatever condition caused the interrupt to occur. After handling the interrupt, the CPU returns to whatever it was doing when the interrupt was signaled. Interrupts are covered briefly in Subsection 5.2.2 of *The Most Complex Machine*.

For the xComputer, you should imagine that an interrupt control wire is connected to the keyboard and that anytime the user types a character, an interrupt is signaled. Ordinarily, the xComputer responds to this interrupt by adding the typed character to the input queue and then immediately returning to its regular processing. However, it is possible to set up an ***interrupt handler*** to respond to keyboard interrupts.

An interrupt handler is simply a sequence of instructions, stored in memory like any other program. The difference is that the interrupt handler is executed ***asynchronously***. (This just means that it is executed at unpredictable times, whenever the user happens to type a character. The timing of the interrupt handler is not "synchronized" with the timing of the rest of the program.)

You have to tell the computer where the interrupt handler is located in memory. This is done with an Interrupt-Handler (INH) instruction. For example, the instruction "`INH 500`" specifies that there is an interrupt handler starting at location number 500. When an interrupt occurs, the xComputer will first put the typed character in the input queue. It will then jump to the location specified by the INH instruction. The interrupt handler must retrieve the character from the input queue with a GTC instruction. The interrupt must end with a Return-from-Interrupt (RTI) instruction. When the computer executes this instruction, it returns to whatever it was doing when the interrupt occurred.

(Actually, that is not precisely true: If the user types a second character while the interrupt handler is executing, the typed character will be added to the queue, but the computer will *not* jump to the beginning of the interrupt handler—the interrupt handler cannot itself be interrupted. When the computer executes an RTI instruction, it checks whether there is another character waiting in the queue. If so, it jumps to the start of the interrupt handler to process that character. When the queue is empty, the RTI instruction sends the computer back to the main program.)

Open the sample program Simple Interrupt Handler. Read the comments, load the program, and run it with the **Speed** *set to* **Fast**. *(Don't forget to set the PC to zero before trying to run it!) The program "counts" by repeatedly adding 1 to memory location number 5. When you type a character, an interrupt handler will get the character you typed and write it to the output list. Try running the program at speed number 6. You will have to click on the* **Next** *button (or press return) to execute each step of the fetch-and-execute cycle. Watch carefully to see what happens when an interrupt occurs and when an RTI instruction is executed.*

When an interrupt occurs, the Count register is set to 0 and the PC is set to the address of the interrupt handler (6 in this case). Although you won't see it, the computer also saves the values of all the registers. When the interrupt handler ends, the saved values of all the registers are restored, so that the program can continue as if the interrupt had never occurred. (The numbers in the registers constitute the ***state*** of the computer, as defined in the text.)

Interrupts are a very important part of computer processing. In real computers, there can be many different types of input. Programmers don't usually need to deal with interrupts, since they are generally handled by the operating system. For event loop programs, the operating system adds an appropriate event to the program's event queue when an interrupt occurs. The program can retrieve that event and do any necessary processing at its leisure. The only disadvantage to this is that a user action that generates an interrupt might not get an immediate response, if the program is busy doing something else.

Background Processing: As noted above, a program that uses a polling loop often wastes most of its time just waiting for user input. It would be nice if there were a way to use that wasted time. Interrupts provide a method for doing so. Instead of sitting in a loop waiting for user input, the program can use an interrupt handler to process user input when it occurs. When there is no user input to process, the computer can work on some other task. From the point of view of the user, that task is secondary and is being performed in the ***background***. The ***foreground*** task—that is, processing the user's input—is given priority, since it can interrupt the background task at any time.

You can see this foreground/background behavior in the *Simple Interrupt Handler* program: The background task is counting, while the foreground task is copying user input to the output list. It is, of course, possible to do more interesting things both in the foreground and the background.

Open the sample file EchoNums and Background Process, read the comments, load the program and run it. (Don't forget to set the PC to zero before trying to run it!) Try running it at **Fastest** *speed with the memory display set to* **Graphics**. *The foreground processing is identical to the EchoNums program, except that it is implemented using an interrupt handler instead of a polling loop. Try typing some numbers, ending each number with a period. The background process is a complicated program that computes arbitrarily large powers of 3; the details are not important.*

In the exercises for this lab, you will write programs that do input/output, interrupt handling, and background processing.

* * *

Exercise 1: *Modify the EchoNums program so that when the user presses "d", the number in location num is divided by 2, and when the user* *presses "m", the number in location num is multiplied by 2. (You can use an SHL instruction to do the multiplication and an SHR instruction to do the division.) The program should still write num to output when the user presses ".". Write a short essay discussing how a polling-loop program can do any desired processing for each possible typed character.*

Exercise 2: *A program can do output even if it doesn't have any input. Write a program that will write the letters of the alphabet—A, B, C, ..., Z—to the output list. Your program should use a loop.*

Exercise 3: *Write an essay explaining what the xComputer does when you run the program EchoNums with Background Processing and type in "123.".*

Exercise 4: *Write a program that does background processing while it uses an interrupt handler to perform "binary to decimal conversion" as its foreground task. The user can type in a binary number as a sequence of zeros and ones. When the user types a period, the program should output the number twice—once as a binary number and once as an unsigned integer. The user can then start entering the next number. If the user types any character except for zero, one, or period, the program should output an exclamation point (!) as a signal that the user has entered an illegal character. Your program will be very similar to EchoNums and Background Process, but the computation that it does when it reads a zero or a one will be much simpler.*

As your background task, you can use the same program used in the file EchoNums and Background Process. You'll find a copy of this program in the file Background Process. But if you did Lab 6, you might want instead to use the program you wrote for that lab to compute a list of prime numbers as your background process.

Lab Number 9 for

The Most Complex Machine

by David J. Eck

Introduction To xTurtle

About this Lab: This lab is an introduction to a high-level programming language called ***xTurtle***. This is a language created to be used with *The Most Complex Machine*, but it is in the mainstream of high-level languages, along with Pascal, Ada and C. It incorporates some ideas common to all these languages: variables, assignment statements, loops, if statements and subroutines. (You will find that you are already familiar with the basic ideas because of your work in previous labs.) The xTurtle language also contains special-purpose commands for doing "turtle graphics." These commands can be used to draw pictures on the computer screen. In this lab, you will learn about the basic xTurtle commands, about loops and if statements, and a little bit about variables. Future labs will cover programming in more detail, including the use of subroutines.

This lab covers some of the same material as Chapter 6 of *The Most Complex Machine*. However, the lab is meant as a self-contained introduction to this material. It would be useful but not essential to read Chapter 6 before doing the lab.

This lab and the software used in it accompany *The Most Complex Machine: A Survey of Computers and Computing*, an introductory computer science textbook by David Eck (published 1995 by AK Peters, Ltd., 289 Linden Street, Wellesley, Massachusetts 02181; ISBN 1-56881-054-7). Support for the text, software and labs is available on the World Wide Web at http://math.hws.edu/TMCM.html. David Eck can be reached at the e-mail address eck@hws.edu.

XTurtle Commands: For this lab, you will use a program called *xTurtle*. Find the program and start it up by double-clicking on the program icon. You will see a window like that shown in Figure 9.1. The "turtle" is shown as a small black triangle. The turtle has a ***position*** and a ***heading***. Its heading is the direction it is facing, given as a number of degrees between −180 and 180; the turtle has a heading of zero when it is facing to the right, a heading of 90 when it is facing upwards towards the top of the screen, and a heading of −90 when it is facing downwards. Its position is given by two numbers: an ***xcoord***, or horizontal coordinate, and a ***ycoord***, or vertical coordinate.

The window in Figure 9.1 shows a twenty-by-twenty square in which the turtle can move and draw. This square has horizontal coordinates from −10 on the left to 10 on the right, and it has vertical coordinates from −10 at the bottom to 10 at the top.

The turtle starts out in the center of the screen—at the point (0,0)—facing to the right. It can obey commands such as "forward(5)," which tells it to move forward five units, and "turn(120)," which tells it to rotate in place through an angle of 120 degrees. (It turns counterclockwise if the number of degrees in positive and clockwise if the number of degrees is negative.) The number in parentheses is called a ***parameter*** for the command; you can substitute any number you want. The parameter in a "forward" command tells the turtle *how far* to move forward, while the parameter in the "turn" command tells it *how many degrees* to turn.

The turtle can actually move outside its basic 20-by-20 square. You can expand the window to see a larger region, and the the scroll bars allow you to view any part of a 60-by-60 square. But if the turtle moves outside of that 60-by-60 square, you won't have any way of seeing it.

The turtle can draw a line as it moves. You can think of it as dragging a pen that draws as the turtle moves. The command "PenUp" tells the turtle to "raise the pen." While the pen is raised, the turtle will move without drawing anything. The command "PenDown" tells the turtle to lower the pen again.

As an exercise, you should try to make the turtle draw two separate, parallel lines, like this:

If you make a mistake, you can use the command "clear" to clear the screen and the command "home" to return the turtle to its original position and orientation (at the center of the screen, facing right).

Here are the other basic turtle graphics commands (in addition to *forward*, *turn*, *PenUp*, *PenDown*, *clear* and *home*). In these commands, x and

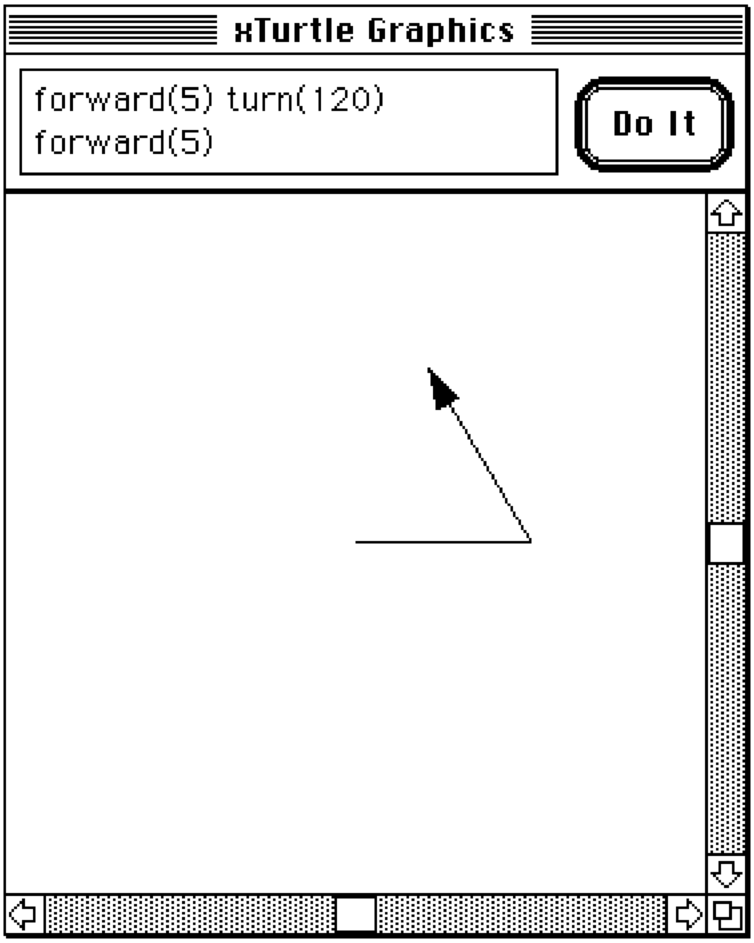


Figure 9.1. *The graphics window from the program xTurtle. The turtle draws in the lower part of the window. Commands for the turtle can be typed into the box at the top of the window. The turtle executes these commands when you press return or click on the "Do it" button. In this illustration, the turtle has already executed the commands shown in the input box.*

y are parameters that can be replaced by any number when you use the command.

$back(x)$ tells the turtle to back up x units, that is, to move x units in the direction *opposite* to its current heading. For example, "$back(3)$ tells the turtle to back up three units. Negative numbers are allowed as parameters for both *forward* and *back*. $Back(x)$ is provided only as a convenient shorthand for $forward(-x)$.

$face(x)$ makes the turtle turn to a heading of x degrees from heading zero. For example, $face(90)$ points the turtle straight up, $face(-90)$ points it straight down, and $face(180)$ points it to the left. Note the distinction between *turn* and *face*: *turn* specifies a change in direction from the current heading, while *face* specifies a new heading without any reference to whatever the old direction might have been.

moveTo(x,y) just tells the turtle to move from wherever it happens to be, to the point with coordinates x and y.

move(x,y) is related to *moveTo*(x,y) in the same way that *turn*(x) is related to *face*(x). That is, while *moveTo*(x,y) says "move from the current location, whatever it is, to the point with coordinates (x,y)," *move*(x,y) says "move x units horizontally and y units vertically *from the current location*." Note that these commands do not depend upon or change the heading of the turtle.

circle(x) draws a circle of radius x. You should think of the turtle moving in a circle starting from its current position and returning to that position at the end. Note that the turtle position is on the circle. If x is positive, the turtle curves to its left as it draws the circle, and the center of the circle is x units to the left of the original turtle position. If x is negative, the turtle curves to the right, and the center of the circle is to the right of the original position.

arc(x,y) draws part of a circle of radius x. A full circle would be 360 degrees; *arc*(x,y) draws an arc of y degrees. As with *circle*(x), the turtle curves to the left if x is positive and to the right if x is negative. If y is negative then the turtle will "back up" along an arc. Note that the turtle changes position and heading as it draws.

HideTurtle and *ShowTurtle* make the turtle (the small black triangle) invisible and visible. The turtle can still draw while it is invisible. (Note also that there is a menu command **Always Hide Turtle** in the **Options** menu. When this option is set, the turtle will be invisible whenever a program is running, regardless of whether the program uses any *HideTurtle* or *ShowTurtle* commands.)

Draw some pictures using these basic commands. Here are a few things you can try, for example:

1) `circle(7) circle(5) circle(3) circle(1)`
2) `arc(5,90) arc(-5,-90) arc(5,90) arc(-5,-90)`
3) `forward(5) turn(120) forward(5) turn(120) forward(5)`
4) `forward(7) turn(90) forward(5) back(10)`

(Note, by the way, that you can type in several commands at once, or you can type in one command at a time, pressing return after each command. Note also that after a command is executed, the contents of the command-input box are hilited, so that as soon as you start typing, the previous command will be erased and replaced with what you type. And finally, note that you can change the speed at which the turtle follows a sequence of commands using the **Options** menu.)

Programs: The power of a computer comes from its ability to execute a program. A program is a sequence of instructions that can include features such as loops, decisions and subroutines. The program *xTurtle* allows you to write programs, to save programs in files, and to reuse previously saved programs. As an example, open the file *Necklace*.

This file contains a short program that shows how a loop can be used. The program itself is as follows:

```
LOOP
  arc(5,20)
  circle(-0.5)
  EXIT IF heading = 0
END LOOP
```

In addition, the file contains a lot of ***comments***. In an xTurtle program, a comment is anything enclosed between braces, { and }. Comments are meant for human readers of the program and are completely ignored by the computer.

In an xTurtle program, a loop consists of the word **loop**, then a sequence of instructions, then the words **end loop**. One of the instructions must be an **exit** statement, which gives a condition for ending the execution of the loop. When the program is executed, the computer will execute the statements in the loop repeatedly. Each time the **exit** statement is executed, the computer tests the condition specified by that statement. If the condition is satisfied, the computer jumps out of the loop. (Ordinarily, after exiting from a loop, the computer jumps to whatever statement follows the **end loop**. In this example, that just means jumping to the end of the program.)

Run the program using the **Run** command from the **Run** menu. You will find that part of the picture is drawn off the screen. This would be fixed if the turtle started at the point (0,−5) instead of at the point (0,0). *Add commands to the program, before the beginning of the loop, to move to the point (0,−5), without drawing a line.* Then, run the program again. Use the modified version of this program as your starting point for doing Exercise 1 at the end of the lab.

As another example, open the file *random walk*. The program in this file shows several new features of the xTurtle programming language, including **if** statements, variables and assignment statements. This program makes the turtle do a "random walk" in which it repeatedly moves in a randomly chosen direction. *Run the program several times.* Note that you can change the speed of the turtle using the **Options** menu, and that the **Run** menu contains commands to interrupt or terminate a program that is being executed. Read the comments on the program, which will give you some idea about how it works.

A ***variable*** is just a memory location with a name, which can be used to store a value. In xTurtle, you give a name to a memory location with a **declare** statement, as illustrated in the program *random walk*. Once you have declared a variable, you can store a value in it with an assignment statement, which has the form

$$\langle \textit{variable name} \rangle := \langle \textit{value} \rangle$$

The value on the right can be given as a number, as another variable, or as a mathematical formula. For example:

```
x := 17
newAmount := oldAmount
cost := length * width * costPerSquareFoot
```

An **if** statement, which is used to decide among alternative courses of action, begins with the word **if** and ends with the words **end if**. The exact rules for using **if** statements are rather complicated, and are covered in detail in the text, but you should be able to get the basic idea by looking at the example in this sample program. The random walk program is used as the basis for Exercise 2 at the end of the lab

Input/Output: Any real programming language needs to provide some way for a program to communicate with the person who is using the program. The xTurtle programming language provides only minimal support for input and output, but what it provides is enough for a program to have a simple dialog with the user.

There are two commands for output (sending information from the computer to the user), and one command for input (getting information from the user into the computer). All of these commands use ***strings***, which are sequences of characters enclosed in quotes, such as: `"Hello"`. The command

```
DrawText("Hello")
```

will print the string `Hello` in the xTurtle graphics window, at the current turtle position. The command

```
TellUser("Hello")
```

will display `Hello` in a standard Macintosh "dialog box," along with an OK button. The user reads the string and then presses return or clicks on the OK button to get rid of the dialog box. The *TellUser* command has no effect on the picture in the graphics window. Finally, there is a more complicated command, *AskUser*, that can be used to allow the user to enter a number; the number entered by the user will be stored in a variable. For example,

```
AskUser("How much do you want to bet?", betAmount)
```

will display a dialog box with the string "`How much do you want to bet?`" and a box where the user can enter a number. The number entered by the user will be stored in the variable *betAmount*, so that the program can use it. Of course, you have to declare the variable *betAmount* before you can use it in an *AskUser* statement.

All of these commands have a nice feature that allows you to use the value of a variable inside a string. If a string includes the character #, then that # must be followed by the name of a variable. When the string is displayed, the # and the name will be replaced with whatever value is stored in that variable at that time. For example, if *betAmount* and *winnings* are variables, then

```
TellUser("You bet $#betAmount; you win $#winnings.")
```

might actually display "`You bet $25; you win $75.`".

All of this is illustrated in the sample program *I/O example*, which you should open, read, run and understand. Exercise 3 below is based on the I/O capabilities of xTurtle.

* * *

Exercise 1: *Earlier in the lab, you modified the program necklace so that it draws a picture starting at* $(0, -5)$ *instead of at (0,0). For this exercise, start with the modified version of that program. Try changing the radius in the circle command and the number of degrees in the arc command to various values. For example, try "*`arc(5,2) circle(-2)`*" and "*`arc(5,5) circle(7)`*". Also try replacing the circle command with any other command or sequence of commands that will leave the turtle with the same position and heading that it starts with, such as "*`forward(5) back(5)`*". You might also want to change the radius used in the arc command. Turn in a printout or hand-written listing of the program that makes the prettiest picture you come up with. (Hopefully, this will use something other than circles as the "beads" on the "necklace.")*

Exercise 2: *In the sample program Random Walk, which you used earlier in the lab, the computer chooses one of the four directions 0, 90,* -90 *and 180 at random. Modify the program so that it chooses one of the three directions 0, 120 and* -120 *instead. It should have an equal chance of choosing any of these directions. Turn in a print out or hand-written listing of your program. (Make sure you test it!)*

Exercise 3: *With what you have learned in this lab, you can now write a simple guessing game program (which will use none of the graphical capabilities of xTurtle). Write a program in which the computer chooses a random integer between 1 and 100, and the user tries to guess the number. Each time the user makes a guess, the computer should (honestly) tell the user "Sorry, your guess is too high," "Sorry, your guess is too low" or "You got it." You can use an* **if** *statement to pick out the appropriate message.*

Use a **loop** *to allow for repeated guesses. The loop will end when the user guesses correctly. Your program can begin like this, before you begin the loop:*

```
DECLARE answer
DECLARE guess
answer := randomInt(100)
```

(If you want to improve your program, you might count up the number of guesses that the user takes, and report that number at the end. You might also want to allow the user the option of playing another game. In that case, you will need another I/O command that I didn't mention above. The command

```
YesOrNo("Do you want to play another game?", response)
```

will store a zero in the variable response if the user answers no and will store a one in that variable if the user answers yes.)

Your program should include comments. Like the sample programs, it should use indentation to show the structure of the program. (The **Indent** *command in the* **Edit** *menu can be used to automatically indent a program; this feature is also useful for finding certain types of errors in a program, such as a missing* **end if***.)*

Turn in a printout or hand-written listing of your program.

Exercise 4: *Write a short essay comparing the assembly language of xComputer with the high-level language xTurtle. For example, you could: compare the way loops are constructed in each language; compare labels in assembly language to variables in xTurtle; and compare the way computations are done in assembly language with the way they are done by assignment statements in xTurtle.*

Lab Number 10 for

The Most Complex Machine

by David J. Eck

Thinking about Programs

About this Lab: This lab continues the study of programs. The emphasis here is on how complex programs can be developed to perform specified tasks. An organized approach to programming is necessary for all but the most simple programs. Complex tasks can be broken down into simpler tasks, and complex programs can be built up out of simple components. The problem is how to determine what components are needed and how to piece them together.

Before beginning this lab, you should be very familiar with the material in Chapter 6 of *The Most Complex Machine*, especially Section 6.3. The idea of "preconditions and postconditions" introduced there is particularly important, and the example of "nested squares" from that section is used in the lab. This lab also briefly introduces ***subroutines***, which are covered in Chapter 7 of the text.

Preconditions: The file *Nested Squares*, in the folder *Files for Lab 10*, contains a program for drawing a set of squares nested inside one another, as given in Figure 6.10 of the text. *Open this file now, read the*

This lab and the software used in it accompany *The Most Complex Machine: A Survey of Computers and Computing*, an introductory computer science textbook by David Eck (published 1995 by AK Peters, Ltd., 289 Linden Street, Wellesley, Massachusetts 02181; ISBN 1-56881-054-7). Support for the text, software and labs is available on the World Wide Web at http://math.hws.edu/TMCM.html. David Eck can be reached at the e-mail address eck@hws.edu.

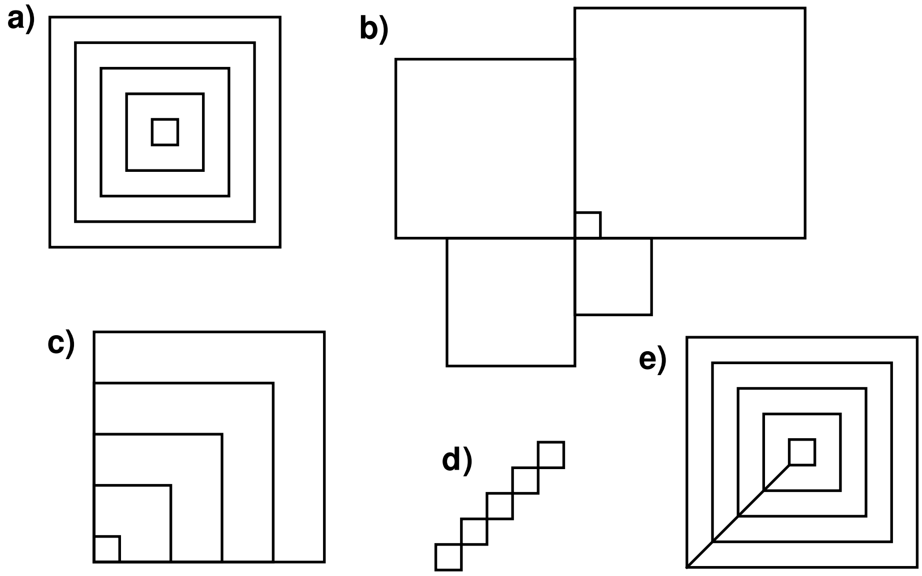


Figure 10.1. *Five nested squares and the results of four incorrect attempts to draw them.*

comments (which were not in the text), and run the program. For this example and for most of this lab, I strongly suggest that you use the **Options** menu to reduce the speed of the turtle; this will make it easier for you to see what is going on.

As explained in the text, the key to writing this program was making sure that the ***preconditions*** for drawing each square were set up properly. The preconditions are the things that must be true at a given point in time for the next section of the program to be executed correctly and to have the desired effect.

Figure 10.1 (taken from Figure 6.9 in the text) shows the correctly drawn squares and the results of five incorrect attempts to draw them. In each case, the error can be traced to the failure to set up one or more of the preconditions correctly. Exercise 1 at the end of the lab asks you to determine what the error is in each case. Exercise 2 also deals with preconditions.

Postconditions: Preconditions are things that must be true at a given time for the program to continue correctly. ***Postconditions*** are things that are *actually* true at a given time because of what has been done by the program so far. A common way for programmers to think about programs is to ask "At this point in the program, do the postconditions from what comes before match up with the preconditions for what is done next."

Figure 10.2 shows a drawing of a simple "staircase." Suppose that you want a program to draw such staircases. Let's add some specific requirements for the program: (1) The user will be able to enter the number of steps that

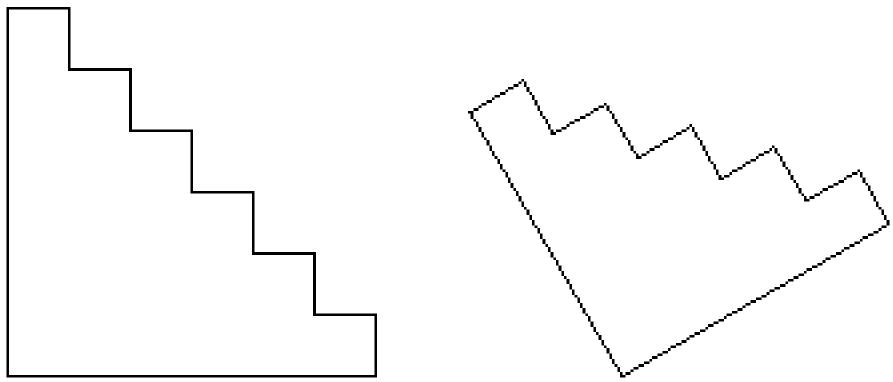


Figure 10.2. *Two staircases, one with six steps and one with five. The staircase at the right was drawn after the turtle had turned to a heading of thirty degrees.*

the staircase will contain. (2) Each "step" in the staircase is one unit wide and one unit high. (3) The orientation of the staircase will depend on the heading of the turtle (as shown in the second example in the figure); this means that it should be drawn using only the commands *forward*, *back*, *turn* and *move*, and avoiding the commands *face* and *moveTo*. (4) After the staircase is drawn, the position and heading of the turtle will be the same as they were when the drawing begins. (Note that condition number 4 is actually a *postcondition* for the program as a whole.)

The program will include a loop. Each execution of the loop will draw one of the staircase's steps. Before drawing each step, the turtle must be facing in the right direction; this is a precondition. After drawing the step, the turtle has changed direction; this is a postcondition. You have to include commands that will provide "splicing" from the actual postcondition to the desired precondition.

In addition to the steps, the staircase contains two long lines that must be drawn outside the loop. At the end of all this, the turtle must return to its initial position and heading. It's not easy to get all the details right unless you keep careful track of postconditions and preconditions. Exercise 3 asks you to write this program.

Preview of Subroutines: A subroutine is—more or less—a small program, made into a black box and given a name. Some subroutines, such as *forward* and *PenUp* are predefined; others are written by a programmer as part of a larger program. They are an essential tool for organizing complex tasks.

Most subroutines have ***parameters***, such as the 9 in "*forward*(9)" or the 30 in "*turn*(30)." Parameters allow subroutines to receive information from the rest of the program or to send information back. Suppose that we want to turn the staircase program described above into a subroutine.

Then—*and this remark is absolutely vital for understanding subroutines*—it would no longer make sense to get the input from the user, since that would greatly limit the generality of the subroutine. Instead, the number of steps would probably be provided as a *parameter*. From the "point of view of the subroutine," the parameter is like input coming from "somewhere outside," just as input from the user comes from outside the program.

The file *Spirals Subroutine* defines a sample subroutine named *spiral*. The file includes comments that give some information about the syntax of subroutines in xTurtle. *Open this file and read the comments.* If you apply the **Run** command to this file, it will look like nothing has happened! This is because all the file does is *define* a subroutine; it doesn't *use* that subroutine. Before the file is run, the computer has no idea what the word "spiral" means; after the computer compiles the subroutine definition, it will know what to do with commands like "*spiral*(61)" and "*spiral*(89)." Such commands can be added to the file after the subroutine definition, or they can be typed directly into the data-input box of the xTurtle Graphics window after the file is run. *Try it; some of the pictures you can make are rather pretty.*

Exercise 4 below asks you to convert the staircase program you will write for Exercise 3 into a subroutine.

* * *

Exercise 1: *Consider each of the pictures* **b**, **c**, **d** *and* **e** *in Figure 10.1, and determine what small change in the program Nested Squares will produce that picture. In each case, it's a question of removing one or more statements from the original program. For each of the pictures, record the change that you made, and determine which of the preconditions are causing the problems. Turn in your answers.*

Exercise 2: *The file Quadratic contains a program that solves the "quadratic equation." Open this file and run the program. It runs fine, but there can be a problem if the values of the variables A, B and C are changed. Try changing C from* -1 *to* 1*, and running the program again; the program will crash with an error. When it does, the blinking cursor will be moved to the point in the program where the error occurred. Why does the program crash? What is the precondition that is not properly checked in this program? How can the program be modified so that it does something more reasonable than crashing when the precondition fails to hold? Turn in your answers to these questions and a printout of the modified program.*

(Note: You are **not** *being asked simply to write a program that handles the case* $C = 1$*. Your program should handle any possible values of A, B and C.)*

Exercise 3: *Write a program to draw a staircase, as described above. The program must satisfy the four requirements that are listed there. Start* *your program with these three lines:*

```
DECLARE NumberOfSteps
AskUser("How many steps?", NumberOfSteps)
DECLARE count
```

The variable named count should be used as a counting variable in the program, just as the variable of the same name is used in the Nested Squares program. (It turns out to be a little easier to start drawing the staircase at the top rather than at the bottom.)

You should think about preconditions and postconditions as you write the program. The comments that you include in your program should discuss specific preconditions and postconditions for various parts of the program, and explain how they were used—or could have been used—in developing the program. Turn in a printout of your program.

Exercise 4: *Convert the program you wrote for Exercise 3 into a subroutine. To do this, remove the first two lines of the program, as they were given in Exercise 3. Replace them with:*

```
SUB stairs(NumberOfSteps)
```

Add the line

```
END SUB
```

at the end of the program. That's it! (That is, that's it provided that your solution to Exercise 3 was correct—including leaving the turtle in its original position and heading at the end of the program.) After running the modified program, you will be able to use commands like stairs(5) to draw a staircase with five steps. To make a more interesting picture, add the following lines at the end of your modified program, after the **end sub** *that ends the subroutine, and then run the program:*

```
loop
  stairs(3)
  stairs(5)
  stairs(7)
  turn(30)
  exit if heading = 0
end loop
```

Turn in a printout of the picture you produce. Also turn in a brief essay explaining: (1) why the line "SUB stairs(NumberOfSteps)" replaces the first two lines from the original program, including the declaration of the variable NumberOfSteps, (2) why the variable count remains as an internal part of the subroutine, and (3) what this exercise teaches you about subroutines.

Lab Number 11 for

The Most Complex Machine

by David J. Eck

Subroutines and Recursion

About this Lab: Subroutines were introduced briefly in the previous lab. This lab will continue the study of subroutines. The lab concentrates on the idea of a subroutine as a black box and on recursive subroutines that call themselves, either directly or indirectly.

You should be familiar with the material from Chapter 7 of *The Most Complex Machine*, especially with the material on recursive subroutines in Section 3. The Koch curve and the binary tree introduced in that section will be used in the lab.

Boxed Symmetry: You are familiar with the idea of a subroutine as a black box. When you use predefined subroutines such as *forward* and *moveTo*, you don't need to know exactly how they work. All you need to understand is how to use them and what they will do. User-defined subroutines can also be used as black boxes, provided that someone else has written them for you. (But remember that "not having to know what's inside" is only half of the black box story! When you write a subroutine yourself, you are working inside the box, trying to make it work the way it's

---

This lab and the software used in it accompany *The Most Complex Machine: A Survey of Computers and Computing*, an introductory computer science textbook by David Eck (published 1995 by AK Peters, Ltd., 289 Linden Street, Wellesley, Massachusetts 02181; ISBN 1-56881-054-7). Support for the text, software and labs is available on the World Wide Web at http://math.hws.edu/TMCM.html. David Eck can be reached at the e-mail address eck@hws.edu.

supposed to, without worrying for the moment exactly what role it will play in a larger program. The real point of black boxes in program design is that they split the problem into manageable subproblems.)

To start this lab, open the file *Symmetry Subs* in the folder *Files for Lab 11.* This file contains the definitions of six subroutines for drawing symmetric pictures. These subroutines are meant to be used in the same way as the usual drawing subroutines such as *forward* and *circle*. For example, the subroutine *multiForward* will draw the same line that would be drawn by *forward*, and it will leave the turtle in the same position and heading at the end. However, in addition to this line, it will also draw the seven additional lines that can be obtained by reflecting the original line horizontally, vertically and diagonally.

To see how this works, run the program *Symmetric Subs*. Nothing appears on the screen, since all the program does is define some subroutines, but after running the program, you can use the subroutines

$$\begin{array}{l} multiForward(dist) \\ multiBack(dist) \\ multiArc(radius,\ degrees) \\ multiCircle(radius) \\ multiMove(dx,\ dy) \\ multiMoveTo(x,\ y) \end{array}$$

in addition to all the usual built-in subroutines of xTurtle. (When using these subroutines, you can use actual numbers, variables or formulas as parameters.)

Run the program Symmetry Subs. Then type the following commands into the input box of the xTurtle Graphics window, pressing return after you enter each command:

```
multiArc(5,40)
turn(-40)
multiforward(3)
multicircle(2)
```

Try some other commands. For a more complicated picture, you can add commands at the end of the Symmetry Subs window, and then run it again. Try this with the simple loop:

```
LOOP
  multiforward(1)
  face(randomInt(360))
  EXIT IF 1=2
END LOOP
```

It is a good idea to run this with the option "Autoscroll to Show Turtle" in the **Options** *menu turned off. You will have to end the program with the* **Kill Program** *command. (Do you see what the line "***exit if** *1=2" does in this program?)*

You can try other sets of commands using the "multi" subroutines. You might try converting the random walk exercise from Lab 9 (the one that produced triangles on the screen) to use *multiForward*. Exercise 1 at the end of this lab asks you to turn in one of the pictures you create.

Recursive Trees and Recursive Walks: Section 7.3 in the text introduces the idea of recursive subroutines using the example of a tree. The program for that example is in the file *Trees* in the folder *Files for Lab 11*. *Open and run this file.* Nothing will happen, since the file only defines some subroutines. The main subroutine defined in the file is *TestTree*. If you type this into the command-input box and press return, you will be asked to specify a complexity level, and then a tree will be drawn with that many levels of branching. *Be sure to read about this example in The Most Complex Machine, or you won't understand what is going on!*

The other example from Section 7.3 in the text is the Koch curve, which is a way of getting from one point to another (with a lot of detours). *Again, you should read about this in the text!* A subroutine for drawing Koch curves can be found in the file *Koch Curves*. The main subroutine in this file is called *TestKoch*. *Open the file, run it, and try out TestKoch for complexity values of 0, 1, 2, 3, 4, and 5. The file also has a subroutine called Snowflake that you should try.*

Exercises 2, 3 and 4 deal with these recursive subroutines.

* * *

Exercise 1: *Turn in a printout of one of the pictures you made using the subroutines from the Symmetry Subs file. Along with the picture, turn in your answers to the following questions: The mathematics used in the symmetry subroutines is not trivial; how much did you need to know about this mathematics to produce your picture? What point about subroutines does this illustrate?*

Exercise 2: *Open and run the file Trees so that you can use the subroutine TestTree. Use this subroutine to draw trees of complexity 0, 1, 2, ..., 8. (Hide the turtle for the larger complexity values.) Given any complexity value, n, determine how many different straight line segments there are in a tree with that complexity. For small values, you can just count the lines. For example, if the complexity is 1, the number of line segments is 3—each branch is a single line, and the trunk is the third line. However, you are being asked for a formula that will give the number of line segments for any value of n. Whatever form your answer takes, you should be able to use it to predict the number of line segments for any given complexity level.*

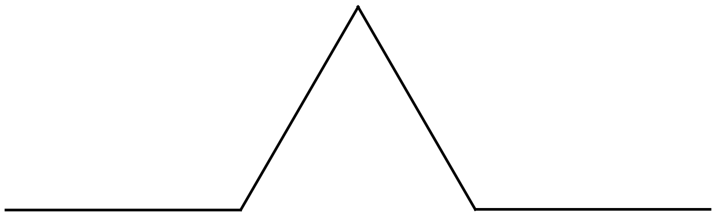
One way to approach this is first to determine how many new lines are added to the tree when you go from a tree of complexity n to one of complexity $n+1$. Then use that to figure out the total number of line segments.

Another approach is to "think recursively": What exactly is a tree of complexity $n + 1$?

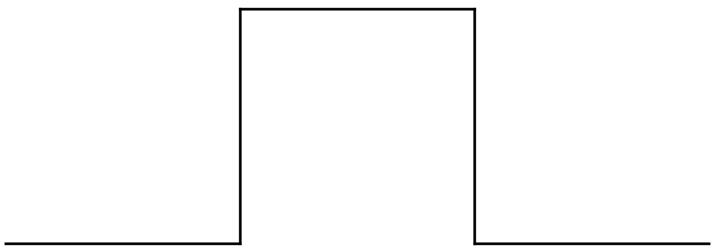
Turn in your answer together with explanation of your reasoning. If you don't come up with an answer, you should talk about how you approached the problem, what ideas you had and what partial results you obtained.

Exercise 3: *The text notes that you can add randomness to a Koch curve by programming the subroutine to decide randomly whether to detour to the left (using turns of 60, -120 and 60) or to the right (using turns of -60, 120, and -60 instead). Make this change to the Koch subroutine in the file Koch Curves, and try it out. Turn in a printout or hand listing of the modified program. (When you have made the change, the SnowFlake subroutine will produce a "Koch Island" instead. Try it!)*

Exercise 4: *The idea of "detouring" used in making Koch curves can be used to make other interesting fractal pictures. In a Koch curve, the idea is to replace a straight line with a line containing a "triangular detour," like this:*



Suppose that a "square detour" were used instead, looking like this:



What would the resulting picture look like, for higher degrees of complexity? Find out by rewriting the subroutine Koch to use square detours instead of triangular detours. (Start with a fresh copy of the file Koch Curves.) Turn in a printout or hand-listing of your subroutine.

Lab Number 12 for

The Most Complex Machine

by David J. Eck

Sorting

About this Lab: One of the most common operations performed by computers is that of ***sorting*** a list of items. An example of this would be sorting a list of names into alphabetical order. This lab deals with two natural questions: How can sorting be done? And how can it be done efficiently?

Sorting is discussed as one example in Section 9.3 of *The Most Complex Machine*, which deals with the ***analysis of algorithms***. Although some of the material from that section is repeated in this worksheet, the background material and motivation is not repeated here. So, you should read section 9.3 before doing the lab.

Sorting Algorithms: Recall that an ***algorithm*** is an unambiguous step-by-step procedure for solving a problem, that is guaranteed to terminate after a finite number of steps. For a given problem, there are generally many different algorithms for solving it. In this lab, you will see five remarkably different algorithms for sorting a list. Each algorithm solves the sorting problem in a different way. You will see how each algorithm

---

This lab and the software used in it accompany *The Most Complex Machine: A Survey of Computers and Computing*, an introductory computer science textbook by David Eck (published 1995 by AK Peters, Ltd., 289 Linden Street, Wellesley, Massachusetts 02181; ISBN 1-56881-054-7). Support for the text, software and labs is available on the World Wide Web at http://math.hws.edu/TMCM.html. David Eck can be reached at the e-mail address eck@hws.edu.

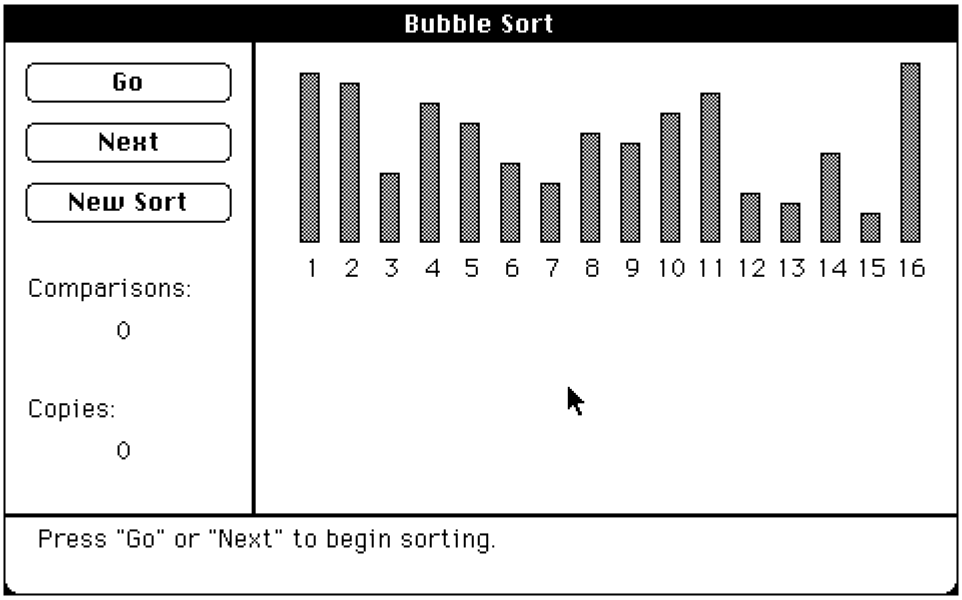


Figure 12.1. *The main window from the program* xSortLab. *The sixteen bars are in a random order that will be different each time the program is started and each time a new sort is begun. The buttons on the left of the window are used to control the sorting procedure. There is space on the bottom of the window for two informational messages that describe each step in the sort. The sorting algorithm to be used is indicated in the title bar of the window*

works, and you will also see that some algorithms are much more efficient than others.

The program you will use in this lab is called *xSortLab*. This program has two different modes of operation, a "visual" mode in which you can watch as bars of different lengths are sorted, and a "timed" mode in which you can measure the efficiency of an algorithm as it sorts large numbers of items.

When the program first starts up, it is in visual mode, displaying a window like that shown in Figure 12.1. The visual mode is used in the first part of the lab.

The five sorting algorithms you will be looking at are: Bubble Sort, Selection Sort, Insertion Sort, Merge Sort and QuickSort. The sorting method to be applied by the program is chosen using the **Method** menu. (If you select a new method while a sort is in progress, the new method will be applied to the *next* sort; the sort that is in progress will continue using the previously selected method.)

When you click on the button labeled "Next" (or choose **Next** from the **Control** menu), the program performs one step in the sort. The step is described by one or two lines of text at the bottom of the window. The bottom line, if present, describes an individual step; the top line describes the overall goal of a sequence of steps.

There are two basic operations performed by the program: ***compare*** two items to see which is largest, and ***copy*** an item from one place to another. Sorting consists of these two operations performed over and over (plus some "bookkeeping," such as keeping track of which step in the sort the computer is currently performing). Sometimes, the program has to exchange or ***swap*** two items. It takes three copy operations to perform a swap: (1) Copy the first item to a special location called "Temp", (2) copy the second item into the first location, and (3) copy the item from Temp into the first location.

Whenever the program compares two items, it draws a dotted box around each of them. When it copies an item, you will see that item move. Furthermore, the program counts the number of comparison and copy operations that it performs. The number of comparisons and copies is one way of comparing different sorting algorithms.

Start up the program xSortLab and change the sorting method to "Selection Sort." Use the "Next" button repeatedly, to completely sort the sixteen bars into increasing size. Read enough of the comments at the bottom of the screen to understand what is going on.

Selection Sort is one of the easiest sorting algorithms to understand. The idea is simple: Look though all the items to find the largest one, and put it aside. Then look through all the remaining items to find the next largest, and put it aside. Repeat this until all the items have been processed.

xSortLab describes Selection Sort as a sequence of "phases." The goal of the first phase is to find the largest item and "set it aside" by moving it to the end of the list; the goal of the second phase is to find the second largest item, and move it into position second from the end of the list; and so forth.

It is important to remember that the program can't just "look at all the bars and pick the biggest one" in one step, as you can. It is restricted to comparing two items at a time. To find the largest item in a list, the computer searches through the list one item at a time, keeping track of the largest item it has seen so far. After looking at every item in the list, it knows which is the largest item overall, and it can then move it into position at the end of the list.

Once you have begun to understand Selection Sort, it might be useful to watch it at higher speed. Instead of pressing the Next button for each step of the sort, you can press the Go button to have the computer perform the sort automatically. There are two visual speeds for you to watch; you can change speeds by using the third and fourth commands in the **Speed** *menu. Try watching Selection sort using the* **Visual/Fast** *speed. The automatic mode of xSortLab is meant to help you get an overall "feel" for a sorting algorithm, once you have already begun to understand it by going through the sort step-by-step.*

There are five sorting algorithms altogether. They can be divided into two groups. Bubble Sort, Selection Sort and Insertion Sort are fairly straightforward, but they are relatively inefficient except for small lists. Merge Sort and QuickSort are more complicated, but also much faster for large lists. You will need to look at Insertion Sort and Merge Sort in some detail for the exercises at the end of the lab. You should also look at Bubble Sort and QuickSort if you have time. (QuickSort is, on average, the fastest sorting method available. Bubble Sort is the slowest. Bubble Sort is often the first and sometimes the only sorting method that students learn.) Here are brief descriptions of the remaining four sorting methods (but of course brief descriptions are no substitute for watching them in action):

Bubble Sort: The basic idea is to compare two neighboring items and, if they are in the wrong order, swap them. The computer moves through the list comparing and swapping. At the end of one pass, the largest item will have "bubbled up" to the last position in the list. On the second pass, the second largest item moves into position, and so forth.

Insertion Sort: The basic idea is to take a sorted list and to insert a new item into its proper position in the list. The new sorted list is one item longer than the old list. You can start with a (trivially) sorted list of one item and repeat this process until all items have been inserted into the sorted list.

Merge Sort: The basic idea is that if you have two sorted lists, you can easily "merge" them into one combined sorted list. Start with (trivially) sorted lists of length one, merge them into sorted lists of length two, then into lists of length four, then eight, and so on until all the items are in one sorted list.

QuickSort: The basic idea is "QuickSortStep," an operation that works like this: Remove one item from the list. Then divide the remaining items into two parts: items bigger than the removed item and items smaller than the removed item. Move all the smaller items to the beginning of the list and all the bigger items to the end, leaving one space in between for the item that was removed at the beginning. Note that when that item is placed in that empty space, it is in its correct final position. To finish the sort, it is just necessary to sort the smaller items to the left of it and, separately, to sort the bigger items to the right. These two smaller sorting jobs can be done by recursively applying QuickSort to each group of items. The great cleverness of QuickSortStep is in the efficient way in which it separates the smaller from the bigger items in the list—but that is easier seen than described. (See Figure 12.2)

The Question of Time: Now that you understand how some sorting algorithms work, the next step is to investigate how efficiently lists of items can be sorted. In this part of the lab, you will use the "timed" mode of the program *xSortLab*. To use this mode, choose one of the first two

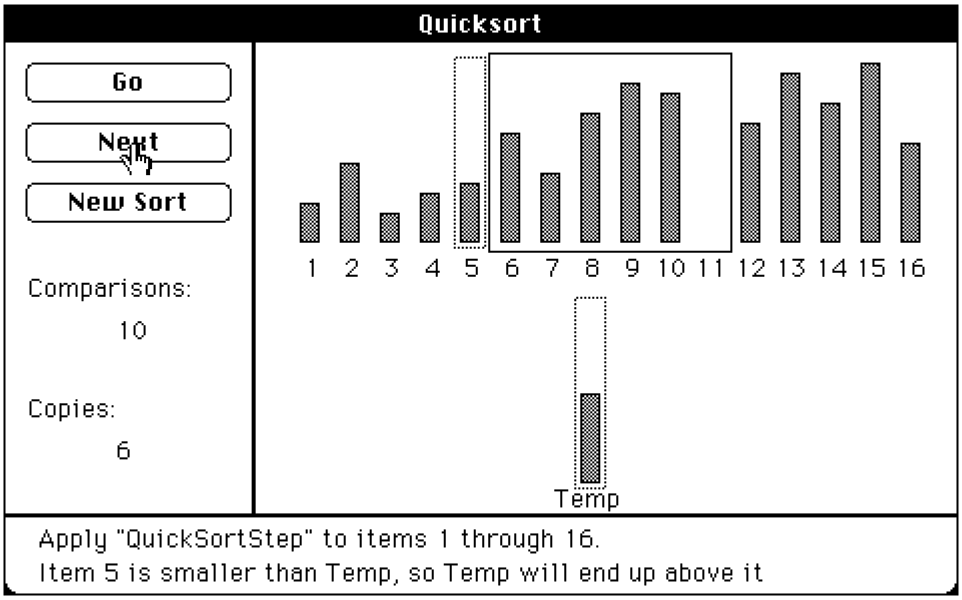


Figure 12.2. *xSortLab applying QuickSortStep to the full list of sixteen bars. One item has been removed from the list and stored in Temp. A box encloses items that have not yet been compared to Temp; as far as the program knows, Temp could end up in any of the locations enclosed by the box. Items to the left of the box are known to be smaller than Temp. Items to the right are known to be larger. Each step shrinks the box by one location, moving an item from that location to the other side of the box if necessary. In the end, there is only one location in the box, and that is where Temp belongs.*

commands in the *Speed* menu. In this mode, the computer works behind the scene to sort arrays of randomly generated numbers. (An ***array*** is just a numbered list of items; the ***size*** of the array refers to the number of items in the list.) The computer reports the amount of time it spent and, for speed number 2, how many comparison and copy operations it performed. This information will appear in the Log Window after the sort is completed.

At speed number 1 or 2, the Sort Window will contain an input box where you can type the size of the arrays to be sorted. There is also a box where you can type the number of arrays to sort. This is absolutely necessary for small arrays, where the time it takes to sort a single array is a small fraction of a second. Since *xSortLab* measures times in ***ticks***, where each tick is 1/60 second, the only way to get an accurate idea of the sorting time for a small array is to measure the total time to sort a large number of arrays, and then to divide that total time by the number of arrays to get the average sorting time for a single array. Even for larger arrays, you can get a more accurate measurement of the sorting time by sorting several arrays and taking the average time.

To get a feeling for how the program works, as well as for how the different sorting algorithms compare, try sorting arrays of various sizes using

Array Size, N	Number of Arrays	Total Sorting Time	Running Time Per Array, R	$R/(N^2)$

Figure 12.3. *A table to be filled in with data you collect for either Bubble Sort, Selection Sort, or Insertion Sort. The value is the third column is measured by the xSortLab program. The values in the third and fourth columns are to be computed by you. The Running Time Per Array is obtained by dividing the Total Sorting Time by the Number of Arrays.*

various sorting algoritms. Before you begin, clear the Log Window, using the appropriate command from the **Control** *menu.*

Set the speed to speed number 1 (Timed/Uninterruptible). Set the Array Size to 10 and the Number Of Arrays to 1000. Click the Go button to perform the sort, and do this for each sorting method. Check the results in the Log Window. [Depending on the speed of your computer, you might want to adjust the number of arrays; this is not supposed to take too long. On an old Macintosh Plus, for example, you might want to divide the number of arrays by 10. Use your judgment.]

Next, repeat the exercise with the Array Size set to 100 and the Number Of Arrays set to 100 [or 10]. Apply each of the five sorting methods. Finally, repeat the exercise again with Array Size set to 1000 and Number Of Arrays set to 10 [or 1].

When you have finished, print the Log Window using the command in the **File** *menu. The printout should contain the results from fifteen experiments: five sorting methods applied to each of three array sizes. You are asked to turn in this printout as part of Exercise 4 at the end of the lab.*

(You might also want to try speed number 2, but that is not required for this lab. On this speed setting, the computer spends most of its time counting copies and comparisons, updating the screen, and checking to see whether you have pressed Command-Period. At speed number 1, the computer does none of these things; the time reported is the time it actually spends sorting the arrays.)

Array Size, N	Number of Arrays	Total Sorting Time	Running Time Per Array, R	$R/(N * \log(N))$

Figure 12.4. *A table to be filled in with data you collect for either QuickSort or Merge Sort. The value is the third column is measured by the xSortLab program. The values in the fourth and fifth columns are to be computed by you. (In computing the fifth column, you can use either the common log or the natural log,* $\ln(N)$*, as long as you are consistent. Both types of log can be found on most calculators.)*

In the remainder of the lab, you will investigate two of the sorting algorithms in greater detail.

Choose one of the three methods Bubble Sort, Selection Sort, or Insertion Sort. Gather data on sorting time for this algorithm on arrays of various sizes, and use it to fill in the table in Figure 12.3. (Or make a similar table by hand.) The data in the last column of the table is for use in Exercise 6. You can use the data you've already collected for arrays of size 10, 100 and 1000. You should collect data for other array sizes ranging from 2 to 10,000. For very large array sizes, use just one array. (Don't try to do arrays much larger than 10,000, since it will take too long. On older computers, even 10,000 items might be too many. *Be careful to copy down any data you need from the Log Window before sorting a very large array,* in case it takes too long and you decide to abort the program.)

Finally, choose either Merge Sort or QuickSort, and gather data on sorting time for arrays of various sizes. Use your data to fill in the table in Figure 12.4. (Or make a similar table by hand.) The data in the last column of the table is for use in Exercise 7. Besides the data you've already collected for arrays of size 10, 100 and 1000, you should collect data for other array sizes ranging from 2 to 100,000 or more.

* * *

Exercise 1: *The basic idea in Insertion Sort is to insert an item into its correct location in a sorted list. Describe Insertion Sort in more detail. What is the exact sequence of phases it goes through when sorting a list of 16* *items? What sequence of steps does it go through to complete each phase? (The information you need can be found by running xSortLab.)*

Exercise 2: *Suppose that Selection Sort is applied to the following list of numbers. Show what the list will look like after each phase in the sort:*

73 21 15 83 66 7 19 18

Exercise 3: *Suppose that Merge Sort is applied to the following list of numbers. Show what the list will look like after each phase in the sort:*

73 21 15 83 66 7 19 18 21 44 58 11 91 82 44 39

Exercise 4: *Turn in the printout of the Log Window that you were asked to make. Add to it, in writing, your observations on the data it contains.*

Exercise 5: *Turn in tables, like those in Figures 12.3 and 12.4, showing your measurements of sorting time for two sorting algorithms on arrays of various sizes. One table should show times for Bubble Sort, Selection Sort or Insertion Sort. The other should show times for Merge Sort or QuickSort. Be sure to collect data for a wide variety of array sizes, as specified earilier in the lab. State the conclusions you draw from your data. How do the two methods compare? What happens as the size of the array increases? (You might want to present the data as a graph, as well as in tables.)*

Exercise 6: *The first table that you turned in for Exercise 5 showed running times for a $\mathcal{O}(n^2)$ algorithm. This means that the average running time for an array of size n is approximately kn^2 for some constant k, at least for large values of n. For more details, see Section 9.3 of The Most Complex Machine. (Note that k depends on the computer you are using, as well as the algorithm.) Thus, if the running time is R, then the value of k is approximately R/n^2, and this approximation tends to get better as n gets bigger. Use the data in your first table to estimate the value of k for the particular sorting method you used to generate the table. Once you have a value for k, use it to estimate how long it would take to sort an array of size 1,000,000 using that sorting method. Explain your reasoning.*

Exercise 7: *The second table that you turned in for Exercise 5 showed running times for a $\mathcal{O}(n * \log(n))$ algorithm. This means that the average running time for an array of size n is approximately $k * n * \log(n)$ for some constant k, at least for large values of n. Thus, if the running time is R, then the value of k is approximately $R/(n * \log(n))$, and the approximation tends to get better as n gets bigger. Use the data in your second table to estimate the value of k for the particular sorting method you used to generate the table. Once you have a value for k, use it to estimate how long it would take to sort an array of size 1,000,000 using that sorting method. Explain your* *reasoning. Comment on the comparison between the results of Exercises 6 and 7.*

Lab Number 13 for

The Most Complex Machine

by David J. Eck

Multitasking in xTurtle

About this Lab: A central processing unit executes a program one step at a time, fetching each instruction from memory and executing it before going on to the next instruction. In many cases, though, a problem can be broken down into sub-problems that could be solved at the same time. In ***parallel processing***, several CPUs work simultaneously on a problem, each one solving a different sub-problem. This is one of the major techniques for speeding up the execution of programs.

Even when only one processor is available, it is sometimes natural to break down a program into parts that can be executed simultaneously. ***Multitasking*** can be applied to divide the single processor's time among the various parts of the program. The program won't be executed any more quickly, but the use of parallel processing "abstractions" might make the program easier to write.

In this lab, you will use the multitasking capabilities of the xTurtle programming language. In this language, it is possible to split (or ***fork***) a process into several processes that will all execute simultaneously and independently. Each process will have its own turtle visible on the screen, so you can actually see what is going on. Although you will be seeing only

---

simulated parallel processing, it would be at least theoretically possible for each process to run on its own CPU.

The background material for this lab is covered in Section 10.1 of *The Most Complex Machine*. It will be useful for you to read that material before working on the lab.

Multiple Turtles: Start up the program *xTurtle*, and open the file *Bugs*. (You should find this file and all the other files for this lab in a folder named *Files for Lab 13*.) Run the program. You will see ten turtles wandering around on the screen. This is a very simple program, but it illustrates the basic multiprocessing command in xTurtle, the *fork* statement.

The statement *fork*(10) causes a single process to split into ten processes. Any commands in the program that follow the *fork* statement will be executed by each process independently. In the program *Bugs*, each of the ten processes goes into a loop that sends its turtle on a random walk. Exercise 1 at the end of the lab asks you to add a few modifications to this program.

When a process is forked, all of the processes that are created start out in exactly the same state, with one small exception. The xTurtle language has a predefined variable named *ForkNumber*. This is a read-only variable; that is, you can test the value of this variable but you can't change its value. Each of the processes created by a *fork* statement gets its own value for *ForkNumber*. For the first process, the value of *ForkNumber* is 1, for the second it is 2, and so forth. To test this, try executing the following commands by typing them into the input box in the "xTurtle Graphics" window and then clicking on the **Do it** button:

fork(3) *TellUser*(`"ForkNumber = #ForkNumber"`)

Make sure you understand what happens. (You will have to type both of these commands on one line. If you execute "*fork*(3)" by itself, three processes will be created, but all three processes will "die" before you get a chance to type in the next command. Then, when you execute the command "*TellUser*(`"ForkNumber = #ForkNumber"`)", you will just get a value of zero—the default value of *ForkNumber* when no processes have been forked.)

The file *TwoTasks* contains a sample program illustrating one way in which *ForkNumber* can be used. Open the file and run the program that it contains.

In this program, two processes are created with a *fork* command. Each process then executes an **if** statement of the form:

if *ForkNumber* $= 1$ **then**
  { do one thing }
else
  { do something else }
end if

The first turtle, with a *ForkNumber* of 1, does one task while the second turtle does another completely different task. You are asked to do something similar in Exercise 2 at the end of the lab.

The sample program *Parallel Snowflake* uses *ForkNumber* in a different way, in statements such as "$turn(60 * ForkNumber)$". When the first turtle, with a *ForkNumber* of 1, executes this command, it turns through 60 degrees; the second turtle turns through 120 degrees; the third, 180 degrees; and so forth. Even though every turtle executes the same command, they each do something different because each has a different value of *ForkNumber*.

The *Parallel Snowflake* program also shows that several *fork* statements can be used in the same program. The first *fork* creates several processes. Each of these processes splits into several new processes when it executes the second *fork* statement. You should note that each *fork* command resets the value of *ForkNumber* for all the processes it creates.

You can see two more examples of multiple *fork* commands in the sample programs *Circles* and *Tiling by Pentagons*. (The tiling program was written by my colleague, Kevin Mitchell, who is interested in the mathematical properties of tilings. Several more of his tiling programs are included in the folder *More Tiling Examples* inside the *Files for Lab 13* folder. They all produce pretty pictures that would be more difficult to make without multitasking.) The sample program *Circles* illustrates the fact that **declare** statements that occur after a fork are treated just like other statements. That is, every process executes the **declare** statement and creates its *own* copy of the variables that are declared. You will need to understand this program to do Exercise 3.

By the way, you might want to use *Circles* and *Parallel Snowflake* to investigate the effect of the **Random Scheduling for Forks** option in the **Options** menu. Try running these programs with the option turned off as well as with the option turned on. Remember that your computer, with its single CPU, executes multiple processes by switching its attention from one process to another, executing one process for a while, then moving on to the next process, and so forth. When the **Random Scheduling** option is on, it devotes some random amount of time to a process before moving on to the next. Therefore the processes quickly get "out of sync." When the option is off, the processes stay "in sync" because the computer devotes exactly the same processing time to each process. Random scheduling is a more realistic simulation of parallel processing, but equal-time scheduling can be prettier to watch.

Shared Variables: In all the examples you have seen so far, the multiple processes are completely independent. The various turtles go about their business without interacting with the other turtles in any way. (This is not quite true, since the turtles have to share the same screen. You might have noticed that one turtle will sometimes wipe out the image of another turtle temporarily.)

Things can get more interesting when the processes have to communicate with each other. In xTurtle, processes communicate through ***shared variables***. When a variable is declared *before* a *fork* command, there is only one copy of that variable, which is shared by all the forked processes. If any of those processes changes the value of the variable, then all the other processes can see the new value. This is the only form of communication between processes that can occur in xTurtle.

As explained in *The Most Complex Machine*, great care must be taken when shared variables are used for communication, so that one process does not change the value of a variable while another process is using that value. A process must obtain exclusive access to a shared variable while it is using that variable. This is the ***mutual exclusion*** problem. In xTurtle, the **grab** statement is provided to make mutual exclusion possible. A **grab** statement takes the form

grab ⟨global variable name⟩ **then**
  ⟨statements⟩
end grab

Only one process at a time is allowed to "grab" a given variable. When a process comes to a **grab** statement, the computer checks to see whether another process has already grabbed the variable. If so, then the second process must wait until the first process gets to the end of its **grab** statement. Only then is the second process allowed to grab the variable and execute the statements in its own **grab** statement.

The statements inside a **grab** statement are called a ***critical region***. As long as access to shared variables is confined to critical regions, processes can use the variables to communicate in relative safety.

Communication can still be very complicated, but a fairly straightforward example is available in the sample file *Synchronized Random Walk*. Open this file, read the comments, and run the program. You will see two turtles executing identical random walks. One of the turtles selects a random angle to be used in the random walk and records it in the shared variable *angle*. The other turtle reads the value from the shared variable and uses it. A second shared variable, *control*, is used in a **grab** statement to control access to *angle*. Exercise 4 asks you to modify this program so that more than two processes are involved.

Another example of shared variables is given in the file *3N+1 Max*. Read the comments and run the program. It will take a few seconds to run, and you won't see anything happening until the end, when the program reports the value that it has computed. This program also illustrates what happens when a *fork* command is used inside a subroutine: At the end of the subroutine, all the processes are "rejoined" into one process before the subroutine terminates. Therefore, after the subroutine finishes, there is only one process to execute the *TellUser* statement. In Exercise 5, you will work with a similar example.

* * *

Exercise 1: *In the sample program Bugs, ten "bugs" wander around on the screen. Real bugs, though, are born and die. Add "birth" and "death" to the Bugs program. Add birth by programming a one-in-twenty-five chance that a bug will split in two, each time it moves. (You can program a one-in-twenty-five chance by checking* **if** $RandomInt(25) = 1$.*) To program death, you will need a new command: KillProcess. When a process executes a KillProcess statement, it dies. (This is an easy exercise.)*

Exercise 2: *The sample program DrawGraphs draws the graphs of two functions, one after the other. Modify this program so that the two graphs are drawn simultaneously by two turtles. (This is even easier than Exercise 1.)*

Exercise 3: *Write a program that uses two* $fork(9)$ *statements to draw a multiplication table like this one on the screen:*

1	2	3	4	5	6	7	8	9
2	4	6	8	10	12	14	16	18
3	6	9	12	15	18	21	24	27
4	8	12	16	20	24	28	32	36
5	10	15	20	25	30	35	40	45
6	12	18	24	30	36	42	48	54
7	14	21	28	35	42	49	56	63
8	16	24	32	40	48	56	64	72
9	18	27	36	45	54	63	72	81

The entry in row number row and column number col is $row * col$. *Your program will be similar in outline to the sample program Circles. Recall that if* x *is a variable, then you can write the value of* x *on the screen with the command* `DrawText("#x")`.

Exercise 4: *Modify the program Synchronized Random Walk so that instead of showing two turtles moving in identical random walks, it shows* **six** *turtles moving in identical random walks.*

Exercise 5: *The program Sum of Squares is a failed attempt to write a program that computes the value of the sum*

$$1^2 + 2^2 + 3^2 + 4^2 + \cdots + 25^2$$

Run the program several times. You will see—provided that the **Random Scheduling in Forks** *option is turned off—that it gives different answers,* *none of them correct. Use a* **grab** *statement to fix the program so that it gives the correct answer. Write a short essay explaining carefully what goes wrong when the* **grab** *statement is omitted, and how adding the* **grab** *statement fixes the problem.*

Lab Number 14 for

The Most Complex Machine

by David J. Eck

Graphics and Geometric Modeling

About this Lab: Images are often created on computers in a two-step process. First, a ***geometric model*** of a scene is created, then the scene is ***rendered*** using realistic coloring and lighting effects. This lab deals with the model-construction stage of image creation.

Complex geometric models are built up out of simpler components which are scaled, rotated and positioned in the scene using ***geometric transformations***. Simple geometric shapes like circles and lines are used as a starting point in the modeling process. These shapes can be combined to form more complex figures that can then be combined to form even more complex scenes.

Scenes constructed from objects in this way can be used in a natural way to produce ***animations***. An animation is just a sequence of frames in which objects move slightly from one frame to the next. When the frames are quickly displayed one after the other, the viewer perceives objects in motion.

In this lab, you will use two programs, *xModels-2D* and *xModels-3D*, to construct two-dimensional and three-dimensional geometric models. You will also construct simple animations.

This lab and the software used in it accompany *The Most Complex Machine: A Survey of Computers and Computing*, an introductory computer science textbook by David Eck (published 1995 by AK Peters, Ltd., 289 Linden Street, Wellesley, Massachusetts 02181; ISBN 1-56881-054-7). Support for the text, software and labs is available on the World Wide Web at http://math.hws.edu/TMCM.html. David Eck can be reached at the e-mail address eck@hws.edu.

The background for this lab is Section 11.1 in *The Most Complex Machine*, which covers geometric modeling, geometric transformations and the basics of animation. Questions 11.1 and 11.2 in the text—and their answers in the back of the book—are also relevant to this lab. You should be familiar with this material before starting the lab.

Basic Objects and Basic Transformations: To begin the lab, start up the program *xModels-2D*. You will get a standard text-editing window in which you can type a specification of the scene you want to model. The specification is written in a ***scene description language***—something very much like a programming language. A scene description consists of a list of objects in a scene, together with the geometric transformations to be applied to them. It can also contain definitions of objects to be used later, perhaps repeatedly, in the scene. Such object definitions are very much like subroutines in a programming language.

For example, suppose you want a rather boring scene consisting of a single square. You can specify this scene by typing the single word

```
square
```

in the text-edit window. To see the scene, use the **Render** command from the **Control** menu.

Try this now. Another window, titled "Drawing Window," will appear, with a small square drawn at the center.

The Drawing Window represents a region of the xy-plane with x ranging from -10 at the left edge of the window to 10 at the right and y ranging from -10 at the bottom to 10 at the top. The *square* command specifies a one-by-one square, centered at (0,0), with corners at (-0.5,-0.5), (0.5,-0.5), (0.5,0.5) and (-0.5,0.5).

Suppose you want a larger square, or a square at a different position? You have to start with a standard small, centered square and apply geometric transformations to get the square you want. This is not the most obvious way of doing things, but it turns out to be remarkably powerful and even, after some experience, intuitive. The geometric transformations that you can use are ***scaling***, ***translation*** and ***rotation***. Scaling changes an object's size. Translation moves it. Rotation pivots it about the point (0,0) or about some other specified point.

To transform an object, simply list the transformation after the object to which it applies. For example, the sequence of commands

```
square scale 5
square rotate 45
square translate −3,7
```

specifies three squares with different transformations. In the first of these commands, the transformation *scale* 5 magnifies the square by a factor of 5, producing a five-by-five square. To shrink an object, you would use a scaling operation with a factor less than 1, as for example in "*square scale* 0.5". The scale command can also be used with two numbers. The first number gives a horizontal scaling factor and the second a vertical scaling factor. For example, "*square scale* 2,5" specifies a rectangle that is 2 units wide and 5 units tall.

The transformation *rotate* 45 pivots the square about the point (0,0) through an angle of 45 degrees in a counterclockwise direction. A negative angle would rotate an object in a clockwise direction. It is possible to specify a different pivot point. For example, "*square rotate* 45 *about* 0.5,0.5" would pivot the square about its upper right corner, (0.5,0.5), instead of (0,0).

In the third example, the transformation *translate* −3,7 moves the square 3 units to the left and 7 units up. (All commas in this and other examples are optional, by the way; they can be included for human readability). There are also commands *xtranslate* and *ytranslate* for moving an object horizontally or vertically only. For example, "*square xtranslate* 5" produces a square translated five units to the right.

You can apply a sequence of transformations to the same object, simply by listing them all after the object's name, in the order in which they are to be applied. For example,

```
square scale 3 rotate 30 translate 5,5
```

specifies a square that is first magnified by a factor of 3, then rotated through 30 degrees, then translated 5 units horizontally and 5 units vertically.

There are other basic objects besides squares. A *circle* is a circle of diameter one, centered at (0,0). A *line* is a line of length one that extends from the point (−0.5,0) to (0.5,0). A *polygon* can be specified by listing its vertices. For example,

```
polygon 0,0 0,5 3,4
```

specifies a triangle with vertices at the points (0,0), (0,5) and (3,4). To make things a bit more interesting, a few "filled-in" versions of squares, circles and polygons are also allowed. These are given by the commands *blackSquare*, *graySquare*, *whiteSquare*, *blackCircle*, *grayCircle*, *whiteCircle*, *blackPolygon*, *grayPolygon* and *whitePolygon*. (The difference between a *whiteSquare* and a plain *square* is that the white square will hide objects behind it while the plain square is transparent.)

Here is a simple scene description. You will find a copy of this in the file First Examples. Open the file and render it. Then try modifying and adding to it. You should try to get a feel for the basic objects and the transformations that can be applied to them:

```
graySquare scale 2 translate 5,5
blackCircle scale 5,2 rotate 30
square translate -5,5
polygon 0,0 0,5 3,4 translate -7,-7
```

Animation: Let's face it, static scenes are not all that exciting. Once we have animation, things get a lot more interesting. This is fairly easy to do in *xModels-2D* and *xModels-3D*. Here is an example (which can also be found in the file *Simple Animation*):

```
animate 15
graycircle scale 1.5 ytranslate -2:2
whitepolygon -1,-1  1,-1  -1:1,1  scale 1:3
blacksquare scale 8,0.3 rotate 0:130
```

Any scene description that specifies an animation must begin with the word *animate*, followed by the number of frames in the animation. In this example, 15 frames will be rendered. The remaining lines are a standard scene description, except that in some cases "number ranges," such as -1:1 or 0:130, appear instead of single numbers. Where such a range appears, the first value in the range is used in the first frame, the second value is used in the last frame, and intermediate values are used in intermediate frames. If a range is used with *translate*, the object moves during the animation; if with *scale*, the object grows or shrinks; and if with *rotate*, the object rotates through a range of angles.

Open and render the file First Animation. As the individual frames are rendered, they are stored in the computer's memory so they will not have to be redrawn later. Depending on the speed of your computer and the complexity of the scene, the animation might run slowly until all the frames have been rendered. After that, the playback rate can be controlled with the **Speed** *menu. (There is only room in memory for a limited number of frames; if you exceed that limit, you can still run the animation by turning off the* **Save Frames to Memory** *option in the* **Control** *menu, but the animation might then run unacceptably slowly.) This particular animation will look better if you use the* **Control** *menu to change the playback style from* **Loop Back to Start** *to* **Loop Back and Forth**. *Try some of the various menu commands. Try modifying the animation.*

It is possible to write a "segmented animation," which is like two or more simple animations spliced together. An example would be

```
animate 15 25
square scale 0:5:8 rotate 0:0:90
circle scale 0::12
```

Here, there are two segments, with 15 frames in the first and 25 in the second, indicated by the two numbers listed after the word *animate*. Instead of simple number ranges like "0:5", double ranges like "0:5:8" are used instead. For 0:5:8, the value ranges from 0 to 5 during the first segment of the animation and from 5 to 8 during the second. For 0:0:90, the value stays constant at 0 during the first segment, then ranges from 0 to 90 during the second segment. For the double range 0::12, the value ranges from 0 to 12 over the entire animation. Try this if you want.

Model Building and Structured Complexity: It's a long way from simple geometric shapes to complex scenes. As usual, this complexity is handled by tackling it level-by-level, with reasonable jumps in complexity from one level to the next. In *xModels-2D* and *xModels-3D*, new objects can be defined on one level that can then be used on higher levels. An object definition takes the form of the word **define** followed by a scene description enclosed in square brackets ("[" and "]"). For example:

```
define wheel [
    circle
    line
    line rotate 60
    line rotate 120  ]
```

Once this definition has been made, a "wheel" can be used like any other object. For example:

```
wheel scale 2 xtranslate -2.5
```

The word *wheel* becomes part of the language of *xModels-2D*, on an equal basis with *square* and *circle*. It can even be used in the definitions of other objects.

The file Wagon contains an example in which wheels are defined and used. Open the file, read it, and render it to see what it looks like. Note that the wheels on the wagon rotate.

The sample file *Houses* gives another example of defining and using objects in a scene.

Three-Dimensional Modeling: Models can be constructed in three dimensions, as well as in two. Even for three-dimensional models, of course, the image still has to be displayed on a two-dimensional computer screen, but the model of the scene exists in three dimensions, at least in our imagination and the computer's memory. Once you understand the basic ideas of geometric modeling in two dimensions, the step up to three dimensions is not so hard. Just remember that in addition to the x and y-coordinates that you are used to, there is also a z-coordinate. Think of z as measuring distance in front of the computer screen (or, if z is negative, behind it).

The basic building blocks of three-dimensional models in the program *xModels-3D* still include *line*, *circle*, *square* and *polygon*, which exist in the xy-plane but which can now be translated or rotated out of that plane. (The filled-in versions of these objects are not valid in *xModels-3D*.) There is also a *cube* object, which represents a one-by-one-by-one cube centered at the point (0,0,0). And there is *polygon_3D* which constructs polygons from a list of three-dimensional points instead of two-dimensional points.

Three-dimensional translations include *xTranslate*, *yTranslate*, *zTranslate* and plain old *translate*, which now requires three parameters. *Scale* can also take three parameters, giving magnification factors in the x, y and z directions. There are now three different rotation commands, since an object can be rotated about the horizontal x-axis, about the vertical y-axis, or about the z-axis that points directly at you out of the screen. The three rotation commands are: *xRotate*, *yRotate* and *zRotate*. The *rotate* command is still available, but its effect is identical to *zRotate*.

The only way to understand all this is to look at some examples. Start up the program xModels-3D. Type in the following simple scene description:

```
animate 30
square scale 5 yrotate 0:180
```

You will see the square rotate around its vertical axis. Note that the edge of the square that is farther from you looks shorter, as it should. This is a question of "three-dimensional viewing," which is explained in Section 11.1 in The Most Complex Machine. (If you are familiar with that section, you might want to play with the **View** *menu. Otherwise, leave the* **View** *menu set to its default value.) Try changing the yrotate to xrotate and to zrotate to see the different effects.*

Next, try the following example, which illustrates a translation in the z direction:

```
animate 30
square scale 5 ztranslate 8 yrotate 0:360
```

The ztranslate command moves the square 8 units forward towards you. The yrotate command then sends it circling away from you and back. Try xrotate instead. Also try xtranslate instead of ztranslate, and make sure you understand what is going on.

The files Flaps, SpinningBows and NestedSquares-3D contain some other examples for you to look at. Note especially that xModels-3D lets you define and use new objects, just as xModels-2D did. This is most striking in the example NestedSquares-3D.

There are two more commands in *xModels-3D* that can be used to easily produce certain types of complicated objects. The commands are *lathe* and *extrude*. These are standard operations in three-dimensional graphics. The idea is similar in each case: a specified figure is copied several times, and the vertices of the copies are connected with line segments. For lathing, the copies are obtained by rotating the original around the y-axis. For extrusion, the copies are obtained by translating the original in the z-direction.

This is easy to understand if you see it. *Type in the commands:*

```
animate 20
lathe  4  3,3  7,-3
    yrotate 0:90
```

and render the image. You will see a truncated pyramid. If you change the 4 to a 12, you will see something that looks rather like a lampshade instead. The first parameter of *lathe* specifies how many copies are to be made. (In the example, this tells how many slanted edges there are: 4 for the pyramid, 12 for the lampshade.) The remaining parameters specify a sequence of one or more line segments in the xy-plane, in this case the single line segment from (3,3) to (7,−3). Try adding the point (10,−3) onto the command.

Next, as an example of extrusion, try the commands:

```
animate 30
extrude  2  -3,3  -2,3  0,-2  2,3  3,3  0,-4
    yrotate 0:90
```

and render the image. Again, the first parameter gives the number of copies to be made. (Two is actually the most likely value for the parameter.) The remaining parameters specify a polygon lying in the xy-plane. Copies of this polygon are made, separated by a distance of one unit in the z direction. In this case, the result is a solid V-shaped figure.

The sample file *Goblet* has a rather neat example of the *lathe* command.

*   *   *

Exercise 1: **(a)** *Use xModels-2D to draw five squares of different sizes, all with their centers at (0,0), nested inside one another.* **(b)** *Starting from your solution to part (a), construct an animation consisting of five squares of various sizes rotating around their common center. They should rotate at different speeds. Some should rotate clockwise and some counterclockwise. Turn in a printout or hand-listing of your scene description.*

Exercise 2: *The file Bounce contains an animation in which a circle seems to bounce back and forth between two edges on a square. Try it. Add a second circle bouncing between the other two edges, and then make the square plus the two circles into an object by putting them inside a* **define** *command. Finally, make an animation that shows the combined object rotating and changing size. What point does this illustrate about object definitions?*

Exercise 3: *Construct a three-dimensional wagon, using xModels-3D. Its body and handle should be made from transformed cubes. It should have*

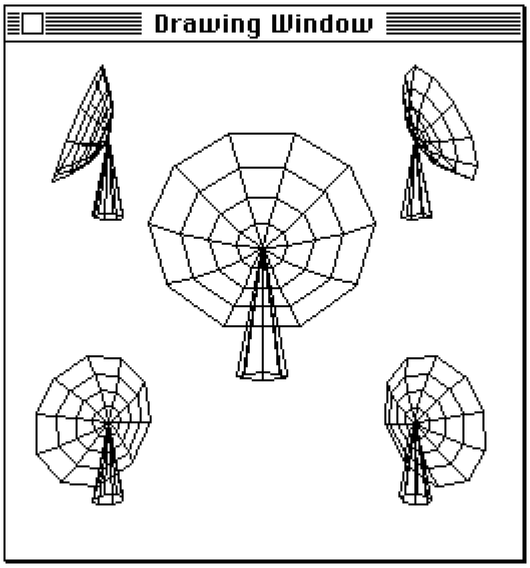


Figure 14.1. *A "radio telescope" constructed using xModels-3D. The telescope is shown from several different viewpoints to give a better idea of its three-dimensional structure.*

four wheels. Try out your wagon in this animation:

```
wagon xtranslate -10:10 yrotate 60 xrotate 15
```

Exercise 4: *Figure 14.1 shows a "radio telescope" that was constructed with xModels-3D. This telescope is made of two pieces. The base can be obtained by lathing a single line segment. The dish can be made by lathing a curve—consisting of several line segments—that connects (0,0) to (4,2). The dish has to be rotated and translated into position after it is created. Define a "telescope" object in xModels-3D. Then create an animation that shows three different telescopes rotating about their axes.*

Exercise 5: *Play with xModels-2D or xModels-3D until you make an image you really like. Print out the image and turn it in.*