A Short Course In Programming by Tom Pittman

Chapter 0 -- How To Read This Book

Programming a computer is like driving a car -- you learn how to do it by doing it, not by reading a book. But, also like learning to drive, it helps to have someone show you which knobs to twist and how to point the car (or program) in the direction you wish to go. That is the function of this book. I will try to show you how your computer works, so you can learn to write programs yourself.

Driving a car takes muscles (but not much) to turn the steering wheel, push the pedals, and so on. Programming a computer takes brains (but not much) to select the right instructions, anticipate the computer's response, and so on. Very few people do not have the necessary muscles to drive a car. Very few people do not have the necessary brains to program a computer. You probably can do both (at least after you finish learning what is in this book and if you already know how to drive).

You do not learn how to drive a car by sitting in your easy chair and waving your pinky in the air. You cannot learn how to program a computer by sitting in your easy chair and "speed-reading" this book. Plan on taking a couple hours or even a couple evenings on each chapter (except the first two, which are not very difficult). When you finish this book, you will know more about programming your computer than most computer hobbyists.

Beginning with Chapter 3, each chapter has a number of little programs which illustrate some feature of your computer. Put the program into your computer and run it (Chapter 2 tells you how to do this, in case you do not already know). The best way to understand each concept is to see it in use. Try to understand what is happening and why. To help your thinking, I will ask questions about what you see. Try to answer the questions before you go on, but if you are confused (and in this business we all are confused at some time or another), read a couple more sentences and I will try to give you some clues, or answer the question myself, but in different words. You should not try to go on to the next topic before you understand the previous one -- you will only become more confused.

I have tried to arrange this book so that every concept is built on the previous concepts. There is no need for "forward references". Each time I introduce a new

term or jargon word, I give a definition. The word is printed in **boldface** so you can find it easily.

There is one more thing you should keep in mind when doing the problems in this book. The computer is not as smart as you are, and it will do just exactly what you tell it to do, even if you did not mean to tell it to do that. If the computer does not do what you think it should have done, 99.99% of the time it is because you did not tell it to do what you wanted. All of the programs in this book have been tested, and they work correctly. If you think a program is not doing what it should, check to be sure you put it in correctly. It is all too easy to read the letter "B" as the numeral "8" in the book (and vice versa), or to confuse the letter "B" for a "6" on the computer's output. Check your program in the computer against the book. Then check the book against the program in memory. If they match, it may be that you do not understand what is happening, so re-read the previous section(s).

Finally, don't let me scare you away. The computer is a lot of fun, once you get the hang of it. It is a thrill without equal to be able to command a machine to do some complicated task and to have that machine obey your every command. So relax, and read Chapter 1.

Chapter 1 -- What Is A Computer?

First, what is not a computer? The $10 calculator you can get at the local variety store is not a computer. That super sewing machine that Aunt Mary just got is not a computer. The noisy teletypewriter in the Realtor's office is not a computer. The spinning tapes you see on TV are not a computer. There may be computers related to these things, but what you see is not the computer. What is?

A **computer** is a device which automatically performs an arbitrary sequence of operations on Data. The teletypewriter in the Realtor's office automatically performs a sequence of operations on data, but it is not an arbitrary sequence; you cannot cause it to perform a different sequence of operations tomorrow. The pocket calculator performs an arbitrary sequence of operations on data, but it is not automatic; you have to push a button for each operation. Note, however, that the more expensive "programmable" calculators in the strictest sense are computers, because you can set up a sequence of steps for it to follow, and it will do so automatically. The sewing machine can automatically perform a sequence of operations, but it operates on cloth and thread, not on data. What then is Data?

Data are the answers to questions. The answer may be numerical (Question: What is the result of adding 7 to 3, then dividing by 8? Answer: 1.25. Question: What is the fourth counting number? Answer: 4.) or logical (Question: If all men are mortal and Socrates is a man, is Socrates mortal? Answer: True.) or the answer

may be a sequence of letters of the alphabet (Question: How is my name spelled? Answer: "Tom".). The answer may be long (Question: How did Columbus come to America? Answer: In three ships, the Pinta, the Nina, and the Santa Maria, which were paid for by the Spanish royal government in 1492, etc...) or very short (Question: How many apples are in an empty paper bag? Answer: 0.). A single Datum may be the answer to several questions at once (Questions: Who was the 38th President of the United States? What is the name of the second largest automobile maker in the world? How do you cross a shallow stream when there is no bridge? Answer to all three: Ford.) or it may be the answer to a question nobody thought of asking: (Question: ? Answer: "@+!*=".). It is data if there could be at least one question for which these are the answers.

When the computer is not actually operating on it, the data is kept in **memory**. This is just as you might keep all the answers you know in your mind's memory. We often say the data is **stored** in the memory. For convenience, the memory is organized into a number of **words**, all the same size. A word may contain one answer or it may contain several small answers; large answers often require many words in memory. The smallest answer we can possibly have in a computer is either a "yes" or a "no". This is the same size as 0 or 1, True or False, Hot or Cold, or any other answer to a question which has only two possible answers. The amount of memory needed to store one of these smallest answers is called a **bit**. Each word in the computer memory is some fixed number of bits, say 8 or 16 or 60; the size of the computer word is determined by the person who designed the computer.

Most computers have several different kinds of memory. One kind is called **RAM** and will have hundreds or (usually) thousands of words. Another kind is called **file storage** and may have a million or more words, but it is harder for the computer to use this memory; more about this later. A third type of memory which most computers have is very easy for the computer to use. It is called **registers**, and there are usually only a few words of these. Often we tell computers apart by the number and kinds of registers they have.

Among the computer's registers, one (or sometimes several) is particularly important, because it holds the answer the computer is working on at any particular instant. Such a register is usually called an **accumulator**. It is normally a single word in the computer, and almost everything the computer is able to do to its data, it does to the data in the accumulator. I will mention some other useful registers shortly.

With thousands of words of data in memory (and we usually mean RAM when we say "memory"), one very important question to be answered is, "Where is the word I want?" We answer this question by numbering the words in memory,

starting with zero (this turns out to be easier for the computer than starting at one). The number of each word is called its **address**, something like the number on your house. Obviously, every answer stored in memory has another answer associated with it, namely, "What is its address?" Often there are special registers in the computer that are particularly suited to holding this kind of data, that is, the addresses of other data stored in memory. We sometimes call these **address registers**. The computer is able to operate on the data in the address registers also.

Another very important kind of data in the computer memory answers the question, "What operation will the computer perform next?" This is important because we control the sequence of operations the computer performs by the proper arrangement of this particular kind of data. Each datum which answers the question, "What operation should the computer perform?" is called an **Instruction**, because it "instructs" the computer in its task. The whole sequence of instructions is called a **Program**. Of course there is an address register that points to the next instruction the computer will obey. This register is called the **Program Counter** or **PC**. Notice that the only difference between the data which you consider to be the answers to your questions and the data which are the answers to the computer's questions about what to do next is this: Who is doing the asking? When the computer asks, "What next?" the answer that comes out of memory is, by definition, an instruction. When the computer asks for data to operate on according to your program, the answer that comes out of memory is not an instruction, but "real data". (People usually prefer to say that the program is not data so we can use the word "data" to mean the answers to our questions, not those of the computer.)

When a computer is operating normally, we say it is **running**. It continually goes through the following pair of steps: First, the Program Counter is used to address an instruction; that is, the computer looks at the word in memory whose address is in the PC. This is called the **instruction fetch**. If the instruction occupies more than one word, all the words of this instruction are fetched. The PC is **incremented** (added to) by the number of words in the instruction, so that it now points to (i.e. addresses) the next sequential instruction in the program. Second, the computer performs or **executes** the operation specified by the instruction just fetched. The computer repeats this fetch-execute cycle for the next instruction, and so on, many thousands and millions of times.

The continuous repetition of the fetch-execute cycles in the computer may seem a little monotonous, until you realize that the instructions being executed are quite varied, and their sequence can make for programs having widely differing results. There are, for example, instructions to copy a datum from someplace in memory into the accumulator and others to copy the datum in the accumulator into some

memory word; we call these **loads** and **stores**. There are instructions to add data to or subtract data from the accumulator, or to perform certain logical operations on it (which we will describe in detail in Chapter 5). There are instructions to manipulate the data in the other registers. Perhaps most important, there are instructions which can change the contents of the Program Counter; we call these **branches**. Chapter 3 covers these in some detail, but for now it is important to realize that the computer is able to alter its own sequence of program execution. Finally, there are instructions which interact with the outside world.

Up to now we have not mentioned how data gets into or out of the computer. I suppose it would still be a computer if it had no way to pass data out to the rest of the world or to accept new data to operate on, but it would not be worth much (a few of the really fast super-computers work this way, but they cheat). We mortals need what we call **Input/Output** or **I/O**.

Input refers to the way data gets into the computer. Usually the computer will have special registers which can be loaded with data from some external source, and perhaps some instructions by which the computer is able to copy this data into its internal memory. **Output** is the reverse operation: Some type of instruction enables the computer to copy data from its internal memory into special output registers, which the outside world is able to examine. Often these special Input and Output registers are called **I/O ports**. Input might come from a human being (such as some buttons or switches or a typewriter-like keyboard), or it may come from special equipment which operates the file storage. Similarly, output may go to lights for humans to look at, or perhaps to control the picture on a TV set; it may also go to the file storage.

Most file storage, as I mentioned earlier, is more difficult for the computer to use than its internal memory (RAM and registers). This is because it tends to be relatively slow, and the computer must pick off (on input) or send (on output) the words one at a time, at the right speed. If the file consists of magnetic tape (such as used with audio cassette systems) the speed may also depend on the human who is operating the controls.

In summary, then, we have seen that a computer consists of some kind of control mechanism (we really did not talk much about it, only assumed that it existed), and a memory consisting of registers and RAM, and some I/O. Perhaps there may be some file memory attached to the I/O. The computer operates on some of the data in memory, according to instructions in another part of the memory called the program.

In the next chapter we will look at a particular computer, and see how it fits with this generalized picture.

Chapter 2 -- The ELF II

A **microprocessor** is an electronic device which is physically very small (hence the name "micro-" from the Greek word meaning small) which contains the heart of a computer. In fact, it is the complete computer except for the RAM and the I/O. There are many different microprocessors around, but we will be looking only at one: the RCA 1802. Everything I said in Chapter 1 is true of all computers, including all microprocessors. Most of what I say in this chapter is specific to the 1802, and much of it is limited to a particular computer, the Netronics ELF II, which contains an 1802 microprocessor.

The 1802 has a word size of 8 bits. All data in this computer comes in 8-bit pieces, which we call **bytes**. One byte can contain a single letter of the alphabet (or other typewriter character) or a small number (between 0 and 255). Most of the instructions for the 1802 will also fit into one byte each.

The basic ELF II comes with 256 bytes (or words) of RAM. You can buy more memory to expand this to 65,536 bytes. Inside the 1802 there are 20 registers, more or less. The lack of precision here is due to the fact that some of these registers are not accessible to you as the programmer and you may as well assume they do not exist; while others are not exactly one byte in size and might be counted once or twice, depending on your inclination. Instead of counting them, let us see exactly what the registers are, and what they can do in the computer:

D

This is the Accumulator. It is one byte (eight bits) and as we mentioned in Chapter 1, all the data that the computer manipulates for us is operated on in the D register. Just how important this register is will be seen in the following chapters as we look at the instructions.

R0-RF

There are sixteen 16-bit Address registers in the 1802. 16 bits can store any number between 0 and 65,535 so an address register is able to point to any single byte in the maximum amount of memory that the ELF II can handle. These are general-purpose address registers, and there are few restrictions on what they can address (i.e. point to) in memory. The Program Counter can be any one of them (you decide which). Others may point to data being computed, while still others may simply be temporary data storage and not used as addresses at all. We number these registers R0, R1, R2, ... RE, RF (using hexadecimal notation: 10=A, 11=B, 12=C, 13=D, 14=E, and 15=F; for a full description of hexadecimal notation see Appendix B). Thus, we can also address the address registers.

P

This is a 4-bit register which is the address of one of the address registers. Whichever register P points to is the Program Counter. Note that P is not the PC; it only points to the PC. The PC register in turn points to some byte in RAM, which is the next instruction to be executed. Since there are 16 address registers, four bits is exactly the right size to uniquely identify one of them. There are a few instructions in the 1802 which can modify P, but there are many instructions which affect the address register P points to.

X

This is another 4-bit register that points to an address register. The address register pointed to by X is used by the 1802 to address a byte in memory for most of the arithmetic and logical instructions, as well as the I/O instructions and certain others involving loads and stores.

T

This 8-bit register is for temporary storage of P and X. It is built into the 1802, but its use is very limited. There is only one instruction which stores T into memory, and no good way to load data into T. For now, just assume that T does not exist.

I, N

These two 4-bit registers together are sometimes called the Instruction Register, and they always hold the instruction the computer is currently executing. Since you, the programmer, have no say in how the instruction register is used, you may as well forget it exists. I only mention it because all the reference manuals for the 1802 make a big deal about the I and N registers; just ignore them.

DF

This is a single bit register in which is stored part of the result of arithmetic and some logical operations. It is normally called the Carry bit and I usually like to abbreviate it "C"; but RCA calls it "DF", so DF it is. We will discuss how DF is used more fully in the following chapters.

IE

This is a another one-bit register in the computer which remembers whether the computer will allow I/O devices to affect its operation without executing I/O instructions. When something like this happens (intentionally) we say the computer has been **interrupted**, and the bit that says whether the computer will let that happen is called the **Interrupt Enable flag**. The 1802 has instructions to set this flag to "yes" or "no", and to ask what its current state (answer) is. The use of this flag is discussed more fully in Chapter 6, when we talk about interrupts.

There are several varieties of instructions the 1802 can execute. All begin with a single byte, which defines what the computer is to do when it executes this instruction. We call this byte the **Opcode**. Recall that a single byte can contain a number between 0 and 255; it is often convenient to number all the possible instructions, so that each one is associated with some number in that range. This is in fact exactly the same as looking at the opcode of an instruction and thinking of

it as a number instead of as an instruction. For example, a particular byte in memory might have the bit pattern "01010011". If we think of it as a number, we would say it is 53 in hexadecimal (which is 83 in decimal). If we think of it as a letter, we would say it is the ASCII code for the letter "S". If we think instead of it as an instruction opcode, we would say it is the "STR 3" instruction which tells the computer to store D into the RAM byte pointed to by R3 (address register 3). There are many other questions this one-byte datum could be the answer to, but they do not concern us here. The point is that instruction number 83 is a particular store instruction which uses address register 3.

Some of the bits in the opcode tell the computer what kind of instruction it is, while others may select one of the registers or further specify the operation. In the case of the 01010011 byte, the first four bits tell the 1802 that this is a store instruction, and the last four bits point to an address register. Thus all of the opcodes from 80 through 95 are store instructions, using different address registers.

Some of the instructions come with data attached. The 1802 can tell which these are by the particular bits in the opcode, so it knows to go back to memory for another byte. Other opcodes contain bits that tell the computer that there is an address (or a piece of an address) attached to the instruction, so the 1802 goes back to RAM for one or two more bytes as part of this instruction. When I describe each instruction, I will tell you exactly how many bytes the 1802 will expect to come with its opcode.

In a similar way, some of the instructions depend on X to indicate which address register to use, while others (like the store instruction mentioned) have that information as part of the opcode. Still others do not directly use any address register. Here also I will give the details when I describe the instructions.

In Chapters 3-6 we will be looking at exactly how the 1802 in the ELF II executes each instruction in its repertoire (the list of instructions the computer knows how to execute). But first let us get a general notion of how the computer works.

Look at your ELF II. You will notice three toggle switches (or possibly locking pushbuttons) on the right hand side near the edge of the board; the bottom switch (closest to you) is labeled "M/P" for "Memory Protect". When this switch is on (positioned away from you in the direction of the little arrow, or locked down), the RAM memory is protected from being accidentally changed by you or the computer; in other words, while the switch is on, nothing new can be stored into memory, even by those instructions or operations which normally would store into it.

The next switch is labeled "LOAD", and is used to load data (and programs) into the computer before running. This switch works with the third switch, labeled "RUN". When both the LOAD and RUN switches are off (towards you), the computer is **reset**, that is, put into a known initial state or condition. This is so that the computer will always start executing the first instruction of the program. Specifically, when the 1802 is reset, X and P are both set to point to R0, which is also set to zero so that it will point to the first byte in memory. IE is also set to the enabled state, but that does not concern us now.

When the switches are in the combination RUN off, LOAD on, M/P off, the computer is able to manually load bytes from the hex keypad. You push the keys corresponding to the hex code of the byte you want entered, then push the "I" (for "Input") key once. The byte that was entered into memory shows up on the two-digit hex display just above the keypad. For the next byte you repeat the sequence. We will call this combination of the switches the **Load mode**. The 1802, when operated in this way, uses address register R0 to point to the memory byte to be loaded next. Since this register is inside the 1802 you cannot see it, so you have to count the number of times you push the "I" key to know which memory byte will be loaded next. Later I will show you a few tricks to make it a little easier to know where you are. If you entered the Load mode from the **Reset state** (both RUN and LOAD off), you know that R0 points to memory address 0, which is the first location in memory. Therefore the first byte you enter will go into memory location 0, the second byte will go into location 1, the third into location 2, etc. If you ever lose count, you probably have to reset the computer (turn the LOAD switch off, then back on) and start over. I have done this many times myself, so you should not feel badly if this happens to you rather often.

You recall I said that the M/P switch protects memory from both the computer and you. Let's talk about you. If you enter the Load mode and then turn the M/P switch on, you will find that the computer thinks it is loading bytes into memory, but since memory is not changing, what you see on the hex display is not what you loaded, but what was there before. This is a convenient way to examine the contents of memory, so we'll call it the **Examine mode**. You can turn the M/P switch on and off as often as you like, thus switching between the load mode and the examine mode. Doing this does not affect R0 (you have to reset the computer for that), so you can examine a few bytes, store something new into the next byte, then go back to looking at what is already there after that. Thus if you make a mistake on the 13th byte of your program, you can correct the mistake without re-keying the whole program by resetting, examining the first 12 bytes (you already know they are right, but you can look at them anyway), then loading the 13th byte, and continuing. You will probably find that you make more than one mistake in a single program, so it is best to save all your fixups for a single pass.

This way you only need to step through memory three or four times; once to load the original program, once to examine it and see where all the errors are, once to correct those errors, then finally one more time to make sure you got them all right. Obviously it is very important that you write down on a piece of paper exactly what hex codes you are going to put into each memory byte. More on this later.

After loading your program into the ELF II you probably want the computer to execute it. This is done by setting RUN on, and LOAD off. We will call this the **Run mode**. It is generally best to enter the Run mode from the Reset state. That is, turn both switches off, then turn the RUN switch on. Since P is 0 and address register R0 is also 0, the first instruction to be executed will be the byte you loaded into location 0. The second instruction will be the byte you loaded into location 1 (unless the first instruction was more than one byte; in that case the second instruction may come from memory location 2 or 3). Chapters 3-6 are a detailed description of how each instruction is executed, so I will say no more at this time.

The fourth possible combination of the RUN and LOAD switches is both on. This is called the **Wait state** because it is used to stop the computer from executing without resetting it. If you put the computer back in the Run mode, it will continue where it left off. Wait will also stop the computer from proceeding in the Load mode, but that is not of much value, since it is normally waiting for you to push the "I" key anyway. You cannot depend on switching the computer from Load to Run, or from Run to Load by going through Wait, because whatever the computer is doing when it enters the Wait state is not finished, and so it may not be able to properly start up in the other mode.

All of the programs given as examples in this book follow a fixed format. After you have used this format for a while you will begin to find that reading programs written in this format is much easier than when they are some other format. I strongly urge you to use the same format, at least until you get used to programming and find something better. Appendix C in the back of this book is a blank page ruled for coding this format; you may wish to use it to make several photocopies, or you may prefer to buy one or more pads of ruled forms from Netronics.

The Coding Form is divided into six columns, identified as: Location, Code, Label, Mnemonic, Operand, and Comments.

Location

The Location field is a four-digit hexadecimal number corresponding to the memory address of the byte on this line. On the forms, only the first line has all

four digits, since the first three digits will be the same for every byte on the page. Also, the fourth digit is pre-numbered on the form to save you some writing.

Object

The Code field is a two-digit hexadecimal value representing the byte to be stored into that location in memory. Sometimes we will collapse several bytes into a single line if they are part of a single instruction.

Label

The Label field is used to give names to certain memory locations. This is for your benefit, so you can choose names to remind you what is important about this address.

Mnemonic

The Mnemonic (pronounced "knee-MAW-nick") field contains the two- three- or four-letter name of the instruction which is in this position. This is also for your benefit. The word "mnemonic" is derived from the name of the Greek goddess of memory, and the instruction mnemonic is a short, easily remembered, abbreviation of the full name of the instruction. When we talk about instructions in the following chapters we will normally use the mnemonic rather than the hex code. The only purpose of the hex object codes is to load the program into memory.

Operand

The Operand field is used to write down any information about this instruction that is not already implied in the mnemonic. We write it on the same line as the mnemonic, even though it may correspond to a byte in the next location of memory. This way all of the human-readable information about one instruction is on the same line.

Comments

The last field on the form is the largest, and you are expected to do the most writing in that part. It is called the Comments field, because here you explain to the world (or to yourself next week) what the program is supposed to do and why. Once you fully understand the operation of each instruction you should not be merely repeating in the comments what is obvious from the mnemonic. Instead you should use this space to explain what you think is in the registers, or where the data came from, or what you expect to do with the data three pages later. Write it as if to explain how the program works to someone who understands how the computer works (unless you are dealing with some obscure feature of the hardware that nobody understands), but who does not know what this program is supposed to do with the data or why; that person will be you, in six weeks. I know!

Those of you who are familiar with computers will immediately recognize that the format I am recommending is that printed out by computers when they run an Assembler program. An Assembler is a program that takes as input data, the

human-readable version of a program (labels, mnemonics, operands, and comments) and outputs the appropriate machine language code with its addresses. When you start writing large programs, you may find one of these helpful.

EXERCISES

1. Reset your ELF II, then put it into the Load mode. Key in Program 2.1, remembering to enter only the hex object code, not the addresses or mnemonics or labels. Do not worry about what is in memory after the end of the program (though you may want to examine some of it, just to see what it looks like). Reset the computer and put it into the Examine mode and check your work. When you have it in correctly, reset the computer again. Leaving the M/P switch on, put it in the Run mode for a few seconds, then reset the computer again and examine memory. Did anything change? (Hint: Take another look at the part of memory after your program.)

   Now reset it, turn the M/P switch off, then run the program. Examine memory again. What happened to the program? What about the rest of memory? We call this program a **Memory Clear**, because it clears (almost) all of memory to zeros. Notice that most of the program is still there! Occasionally I will suggest that you clear memory before trying some experiment; by that I mean, put Program 2.1 in and run it.

   By the way, don't worry about trying to understand these programs right now. I will explain how they work in Chapter 5.

2. Now key in Program 2.2. Run it, then examine memory. How is this different from the Clear? If you do not particularly care what is in the part of RAM you are not using, this gives you a convenient way to step through memory without losing count. It also may help you to find bytes that are stored in strange parts of RAM (this often happens if you make a mistake in writing or keying in a program). We call this program a **Memory Sequencer** because it stores a sequence of numbers in (most of) memory.

3. This exercise will help you to understand the Wait mode. Key in Program 2.3 and run it. Once every three or four seconds the Q light (next to the hex display) should blink briefly. After you have a feel for the timing of the blink, put the computer into the Wait state. Notice that the blinking stops. Return to the Run mode, and see that the time remaining to the next blink is finished out, but the blinking continues. This is easier to see if you flip the switch (into Wait) just before the next blink. You might, just for fun, try to hit the Wait switch while the

blink is still on. This is quite difficult because it is only on for 1/80th of a second or so. In Chapter 3 you will learn how to turn the Q light on and off yourself, or rather how to instruct the computer to do it when you want it to.

```
        ..  PROGRAM 2.1 -- MEMORY CLEAR
        ..
0000 90 CLEAR:  GHI 0   .. REGISTER 0 HAS 0001
0001 AE         PLO 14  .. MAKE RE=0000
0002 BE         PHI 14
0003 EE LOOP:   SEX 14  .. EACH TIME, R14 IS -1
0004 73         STXD    .. D STILL HAS 00
0005 30         BR LOOP .. GO BACK FOR ANOTHER
0006 03
```

```
        ..  PROGRAM 2.2 -- MEMORY SEQUENCER
        ..
0000 90 SEQ:    GHI 0   .. THIS PART IS
0001 AE         PLO 14  ..   JUST LIKE CLEAR
0002 BE         PHI 14
0003 EE LOOP:   SEX 14
0004 8E         GLO 14  .. THIS IS ADDRESS VALUE
0005 73         STXD    ..   SO DATA=ADDRESS
0006 30         BR LOOP .. REPEAT UNTIL DONE
0007 03
```

```
        ..  PROGRAM 2.3 -- SLOW BLINK
        ..
0000 91 BLINK:  GHI 1   .. LOOK AT TIMER IN R1
0001 CE         LSZ     .. IS ZERO ONLY 1/256
0002 7A         REQ     .. IF NOT 00, Q OFF
0003 38         SKP
0004 7B         SEQ     .. WHILE ZERO, Q ON
0005 11         INC 1   .. BUMP COUNTER
0006 30         BR BLINK..   THEN REPEAT
0007 00
```

Chapter 3 -- I/O and Branches

Beginning in this chapter, every program I introduce will consist only of instructions you know about and at most one new instruction (which will be marked with a double asterisk **). This way you can be sure you understand exactly what the new instruction is doing. Each program will be designed to give you a feel about what this instruction does, and how it might be expected to work together with other instructions. Do not advance to the next program until you fully understand all the material that goes before it.

Each instruction is presented in a standard format. First I will give you a test program which you can key into your ELF II to demonstrate the operation of the particular instruction. Associated with this program will be a discussion of just what you should look for when running this program, and perhaps some hints about further experiments. Following this will be a formal definition of the instruction for future reference. This will have a title line in bold face with the instruction mnemonic, hex opcode, the type of operand (if any), and a short descriptive name. The following are the operand types which the 1802 uses, but do not be overly concerned about understanding them now; I will explain each one when you first use it:

-- (No operand)

r Address register, R0-RF

p I/O Port number, 1-7

b Immediate data byte

a One-byte address

aa Two-byte address

Let us start with a very simple program. It has one byte:

```
        ..  PROGRAM 3.1 -- TEST IDL
        ..
0000 00         IDL     .. **
```

Key this program into your ELF II, then switch it to Run mode. Did you see anything happen? Try pushing some of the keys. Any results? If you put this instruction in carefully, the computer should just sit there doing nothing, idle. In fact, the instruction is called the IDLE instruction, because it stops the computer. Since this program does nothing else, there is nothing to see. How can you tell the computer is actually stopped? You can't; you have to take my word for it. It is important that you realize that the ELF II does not tell you it has stopped. If you had an oscilloscope you could look at the SC0 line on the CPU chip; it toggles up and down while the computer is running; you might even want to wire up an LED to tell you that it is toggling, so that you will know when your computer has stopped. That is outside the scope of this book, so for now, you have to believe

that the IDL instruction really does what it is supposed to. Later on we will try some experiments to see how it differs from the other instructions.

IDL Idle 00

Stop the execution of instructions, and wait for an interrupt or DMA to resume.

Now that you understand all about the IDL instruction, we will use it at the end of the next test program:

```
        ..  PROGRAM 3.2 -- TEST SEQ
        ..
0000 7B         SEQ     .. **
0001 00         IDL     .. STOP
```

Key this program in, and run it. What happened? Obviously, since there is only one more instruction than in the previous program, that instruction is what causes the Q light to go on. The Q is a kind of funny bit of memory inside the 1802 which is connected to the outside world. Therefore we can say that SEQ is an Output instruction: It outputs a "1" or "yes" or "on" to that part of the circuit that lights the little LED.

SEQ Set Q 7B

Set the Q flip-flop in the 1802 on, and set the Q output pin high.

You will notice that this next program looks very similar:

```
        ..  PROGRAM 3.3 -- TEST REQ
        ..
0000 7A         REQ     .. **
0001 00         IDL     .. STOP
```

When you run this program, what happens to Q? You say you cannot tell the difference between this program and the first one? That is because Q is being set to "0" or "off" by the REQ instruction, but it is already off because Reset turns it off. Moral of the story: If an instruction commands the computer to do something that has already been done, it does it again, but you have no way of being sure of the fact. Try this program instead:

```
        ..  PROGRAM 3.4 -- TEST REQ AFTER SEQ
        ..
0000 7B         SEQ     .. TURN IT ON FIRST
```

```
0001 7A         REQ     .. TURN IT OFF AGAIN
0002 00         IDL     .. THEN STOP
```

When you run this program, turn the lights out in the room and put your thumb over the hex digit display, then flip the Run switch on while watching the Q light carefully. You might see a very faint blink of light. Computers are fast! If you do not see anything, try 50 or 100 SEQ's followed by one REQ and the IDL; a 1 millisecond flash of light is easier to see than 11 microseconds. The point is, the SEQ turned the Q light on, then the REQ turned it off. Thus the REQ is also a sort of output instruction.

REQ **Reset Q** **7A**

Turn the Q flip-flop in the 1802 off, and set the Q output pin low.

Now the computer is able to tell us about what it is doing. Let's complete the loop by looking at an input instruction.

```
        ..  PROGRAM 3.5 -- TEST B4
        ..
0000 7B         SEQ     .. TELL THE WORLD
0001 37         B4  4   .. **
0002 04
0003 7A         REQ     .. TURN Q OFF!!
0004 00         IDL     .. STOP
```

When you key in this program and run it, what happens to Q? Does it look any different from program 3.4? Now reset the computer, and while holding the "I" key depressed, flip it into Run mode again. What happened to Q this time? Why did it not go off? To convince yourself that the "I" key has no effect on the REQ instruction, put program 3.4 back in, and run it with the "I" key depressed. Obviously, the new instruction we added in program 3.5 is somehow looking at the "I" key, and making the REQ instruction not work. For the next experiment, clear memory (i.e. run program 2.1, or key in 256 00's by hand). Then key in this program:

```
        ..  PROGRAM 3.6 -- TEST B4 DESTINATION
        ..
0000 7B         SEQ     .. TURN IT ON
0001 37         B4  18  .. **
0002 12
0003 7A         REQ     .. TURN Q OFF
0004 00         IDL     .. STOP HERE
```

```
0005 00         IDL     .. (SPACE FILLER)
0006 00         IDL
0007 00         IDL
0008 00         IDL
0009 00         IDL
000A 00         IDL
000B 00         IDL
000C 00         IDL
000D 00         IDL
000E 00         IDL
000F 00         IDL
0010 00         IDL
0011 00         IDL
0012 7A         REQ     .. Q OFF HERE TOO
0013 00         IDL     .. STOP ALSO
0014 00         IDL     .. ETC...
```

Be sure you count the IDL's very carefully when you key this program in. Now when you run it, Q should blink on and then off (you may not see it blink in bright light) whether you start with the "I" key depressed or not. Now go in and modify the third byte of this program (the hex 12 in memory location 0002) to hex 13, and try again. Notice that now the Q light stays on if you run the program with "I" depressed. Now change memory location 0013 (this is the 20th byte, which is the next one after the REQ in location 0012) from IDL (00) to REQ (hex 7A), and try the program again. Finally, change the REQ in memory location 0003 to an IDL (00) and run the program to see how pressing the "I" key affects it. By now you should have noticed that when the "I" key is not pressed, the next instruction after the B4 is executed in sequence, but if the "I" key is pressed the next instruction to be executed is the one whose address is in the byte immediately following the B4 opcode. If you are not yet convinced of this, try putting an SEQ at any place in the first 256 bytes of memory you like, with zeros (IDL instructions) elsewhere, and a B4 instruction (hex 37) in the first byte. When the second byte has the hexadecimal equivalent of the location where you put the SEQ, then running the program with "I" depressed turns Q on; otherwise Q stays off.

We call the B4 a **branch** instruction because the flow of program execution takes one branch of a fork when B4 is executed. That is, depending somehow on the "I" key, the computer either takes the next instruction in sequence, or it starts executing instructions from some other part of memory. The exact place in memory where it goes for that next instruction is defined in the byte immediately following the opcode, and the two bytes together are one instruction.

We also say that the B4 instruction is an Input instruction, because it causes the internal memory of the computer to change depending on conditions external to the computer itself. In this case the memory that is changed is the Program Counter register. The external condition that the 1802 sees is called **EF4** (for External Flag 4). If you look on the schematic for the ELF II computer, you will see that the "I" key is connected (somewhat indirectly) to the EF4 pin on the CPU. As you might guess, there are also EF1, EF2 and EF3 pins, but on the basic ELF II some of these are not connected to anything, and others are hard to observe. I hope you can assume that B1, B2, and B3 work the same way as B4, but look instead at their own EF lines.

```
B1 a    Branch on External Flag 1                       34 aa
B2 a    Branch on External Flag 2                       35 aa
B3 a    Branch on External Flag 3                       36 aa
B4 a    Branch on External Flag 4                       37 aa
-------------------------------------------------------------
If the corresponding external flag line is True (i.e. the
pin is electrically low), then take as the next instruction
the one found at the address which is in the second byte of
the instruction. If the flag is False, advance to the next
instruction in sequence and ignore the address in the second
byte of this instruction.
```

We say "the branch is taken" when the address is used to determine the location in memory of the next instruction. You will recall that the PC is actually a 16-bit register, but there are only 8 bits in the address part of the B4 instruction. We cheat a little bit here; the low 8 bits of the PC get replaced by the branch address, and the high 8 bits are left unchanged. Later in this chapter we will look at branch instructions that replace the entire PC with a new address.

All of the conditional branches in the 1802 have a matched set of branches which test for the reverse condition. In other words, instead of testing for EF4 true as B4 does, we can also test for EF4 false. If you want to see how this works, put program 3.5 or 3.6 back into the computer, but change the 37 to 3F and see how it runs differently. As you become familiar with the 1802 instruction set, you will discover that the branch instructions are paired in such a way that the true version differs from the false version in only one bit position of the opcode; that bit is 0 for the branch if true, 1 for the branch if false.

```
BN1 a   Branch on Not External Flag 1                   3C aa
BN2 a   Branch on Not External Flag 2                   3D aa
```

BN3 a Branch on Not External Flag 3 3E aa

BN4 a Branch on Not External Flag 4 3F aa

If the corresponding external flag line is False (i.e. the pin is high electrically), then take as the next instruction the one found at the address which is in the second byte of the instruction. If the flag is True, advance to the next instruction in sequence and ignore the address in the second byte of this instruction.

Well you might say, if the 1802 can branch when a condition is true, and it can branch when the condition is false, is there some way we can convince it to branch always? The answer is obviously yes, as the next program shows:

```
        ..  PROGRAM 3.7 -- TEST BR
        ..
0000 7A         REQ     .. Q OFF
0001 3F         BN4 4   .. TEST EF4
0002 04
0003 7B         SEQ     .. Q ON IF EF4
0004 30         BR  0   .. **
0005 00
```

When you first run this program, it will not look particularly different from the other programs that do nothing. But now, push the "I" key without resetting the computer, then release it. What happened to the Q light? The computer did not stop! When the program counter reached location 0004, it simply branched back to location 0000 and repeated what it had just done. We call this a loop. Notice that every time through the loop Q is turned off, but only if EF4 is true (i.e. the BN4 does not branch) is it turned on. If Q is turned off every time through the loop, why is it on (unblinking) when the program is running? To help answer that question, try flipping the computer into the wait state a few times while you hold the "I" key down. Notice how sometimes the Q goes off and sometimes it gets just a little brighter. What you are doing is stopping the computer at various points in the loop. Can you predict which times Q will be on when you stop it? (I can't).

BR a Branch unconditionally 30 aa

Take as the next instruction, the one whose address is the second byte of the BR instruction.

As I said earlier, the two-byte branch instructions only have one byte of address with which to modify the PC. The 1802 also has three-byte branch instructions which replace the entire PC with the contents of the two bytes following the

opcode; the first byte is the most significant eight bits, and the second is the least significant eight bits. Try to convince yourself this is true by clearing memory, then storing a single SEQ somewhere. Put a hex C0 in location 0000 and the address of the SEQ (in hex) in bytes 0001 and 0002. When you run the program, Q should come on if and only if the address of the SEQ exactly matches the address in those two bytes. Notice that it does not depend on the "I" key.

LBR aa Long Branch unconditionally C0 aaaa

Take as the next instruction, the one whose address is the second and third bytes of the LBR instruction.

Exercise: What happens when a branch instruction branches to itself? See if you can think of a test program to show this happening. Hint: try a BN4.

We have seen an example of an output instruction and an input instruction in the 1802. Yet neither is called "input" or "output". This is because there are other instructions which also do input or output. Key in the following program:

```
        ..  PROGRAM 3.8 -- TEST OUT
        ..
0000 64         OUT 4   .. **
0001 00                 .. (DATA TO BE OUTPUT)
0002 3F         BN4 0   .. LOOP BACK
0003 00
0004 64         OUT 4   .. **
0005 11                 .. (DATA TO BE OUTPUT)
0006 30         BR  2   .. LOOP BACK
0007 02
```

When you run this program, watch the hex display. Then push the "I" key and see what happens to it. By changing the contents of locations 0001 and 0005, convince yourself that the OUT 4 instruction puts the contents of the next byte out on the hex display. Later on we will see that this is true only if P=X, which in this case is true (both are set to zero by reset). Try as an experiment, OUT 2 (hex 62) in location 0000; what happened to the display? If you try OUT 1 you may find that the whole program goes crazy; I will explain why in Chapter 7.

Now clear memory (run program 2.1) and key in program 3.9:

```
        ..  PROGRAM 3.9 -- TEST INP
        ..
0000 3F         BN4 *   .. WAIT FOR "I"
0001 00
```

```
0002 6C         INP 4    .. **
0003 7B         SEQ      .. WATCH THIS!!
0004 00         IDL      .. STOP
0005 00
```

Before you run this program (if you already ran it, go back and verify that the program is still there) turn on the M/P switch. This is important, because this program modifies itself! I will have more to say about programs that modify themselves later. Run the program once with the M/P switch on, so you can see how it works. Notice that Q does not come on until you push the "I" key. (Why?) Now push the "0" key twice and run the program again (with M/P still on). Is it any different? Reset the computer and convince yourself that the program is still all there (i.e. go into the Examine mode and look at it). Now you are ready to run the program with the M/P s witch off. While the computer is still reset, push the "0" key twice, and turn off the M/P switch. Now run the program: When you push the "I" key, what happens to Q? To find out why, (be sure to turn the M/P switch back on and) examine your program. What happened to the SEQ instruction in location 0003? Run the program again, and before pushing the "I" key, be sure the M/P is off again, and push the "7" key, then the "B" key. Now what happened? Try running the program a few more times, but each time push a different sequence of keys: try 7,A; 6,4; 3,0. Reset the computer each time and examine 0003 to convince yourself that whatever pair of keys you push shows up in the byte following the INP 4 instruction. Try another experiment: change the INP 4 to INP 2 (hex 6A); does the keypad have any effect on what goes into location 0003 now?

As you should have noticed by now, both the INP and OUT instructions have the left hex digit 6. The right digit is between 1 and 7 for OUT and between 9 and F for INP. As it turns out, the same bit which determines whether a conditional branch tests the true or false of a condition also determines whether an I/O instruction with left digit 6 is OUT or INP. You can think of it this way: If the bit is "0" it is "OUT"; if the bit is "1" it is "INP" (relating 0=O, 1=I). We do not yet have any way to prove it on your computer, but input and output use the X register to select an address register to point to the memory byte being input or output. In Chapter 4 we will try to nail that aspect down. Meanwhile you should notice that the OUT instruction increments R0 when the byte is output (go back and look at Program 3.8: Did the computer ever use the byte to be output as an instruction? Convince yourself by looking at OUT 4 followed by 00= IDL or 7B=SEQ). Notice that in our examples R0 was both Program Counter and data pointer. In the case of the INP instruction, however, whatever was input into memory following the INP 4 instruction was also executed as another instruction; in other words, when the INP used R0, R0 was not incremented, so the next

instruction fetch (which is also using R0 as PC) will fetch out the same byte which was just input. Be sure you understand this difference.

```
OUT p   Output from memory (p = 1 to 7)                    6p
-----------------------------------------------------------
Output on the port p, the memory byte pointed to by the
address register pointed to by X, then increment the address
register.

INP p   Input to memory and D (p = 9 to F)                 6p
-----------------------------------------------------------
Input from the port p, a byte to be stored into the memory
location pointed to by the address register pointed to by X,
and also place the byte into D.
```

For both the OUT and INP instructions, the low three bits of p come out on the "N" lines of the 1802 chip; these may or may not be decoded into seven separate ports; if not, then only three ports are distinguishable (1, 2, and 4).

Program 3.10 is rather complicated, and I would recommend that you sequence memory before keying it in. This will help you to place the two pieces which are located at some distance from the beginning. You will also notice that I have arranged it so that each OUT instruction outputs a byte with its own address. This can serve as a check on your keying: After keying in the 64, turn on the M/P switch and examine the next byte; it should already be correct. When you get to location 000F, switch into Examine mode and step through memory to location 002F, then go back to the Load mode for location 0030. Do the same for 0034 to 0079. The first byte after the end of your program should be at 007E.

```
        ..  PROGRAM 3.10 -- CONDITIONAL BRANCHES
        ..
0000 37         B4  *   .. WAIT FOR "I" RELEASE
0001 00
0002 64         OUT 4   .. ANNOUNCE OURSELVES
0003 03         ,03
0004 7B         SEQ     .. ALSO TURN ON Q
0005 3F         BN4 *   .. WAIT FOR "I"
0006 05
0007 6C         INP 4   .. GET INSTRUCTION
0008 7A         (REQ)   .. DO IT **
0009 30         BR 122  .. GOTO 7A
000A 7A         (REQ)   .. Q OFF IF EXECUTED
```

```
000B 64         OUT 4   .. ANNOUNCE "HERE"
000C 0C         ,0C
000D 30         BR  0   .. DO IT AGAIN
000E 00

        ..
0030 64         OUT 4   .. ANNOUNCE "HERE"
0031 31         ,31
0032 30         BR  0   .. GO BACK
0033 00

        ..
007A 64         OUT 4   .. ANNOUNCE "HERE"
007B 7B         ,7B
007C 30         BR  0   .. GO BACK TO FRONT
007D 00
```

Examine this program carefully before running it. Notice that there are four OUT instructions, but each one should output a different value. Therefore you should always know where the program was executing by what you see on the hex display. The program contains no instructions which you do not understand, so you should be fully convinced about what it will do when you run it, before you run it. In particular, you should be convinced that as coded, there is no way for the program to output either "0C" or "31"; it should alternate between "7B" and "03" as you push and release the "I" key. Turn on the M/P switch and verify that this is so by running the program. Notice that as long as the M/P switch is on, none of the other keys have any effect. Notice also that the REQ instruction in location 0008 turns off the Q when you push the "I" key.

Now turn off the M/P switch. The program should be waiting for you to push the "I" key, which you know because the hex display shows "03". First push the keys "7", "B", and then "I". Push and release "I" a few times so you know what is happening. What happened to Q? Is anything else different? Now push "7", then "A". Convince yourself that you are able to key in the opcode that the computer will execute when you push the "I" key. Remember, the INP instruction is inputting a byte into the memory location pointed to by the PC, so that byte input from the keypad becomes the next opcode. Experiment with a few opcodes that you understand, like 00=IDL. What happened? Notice that the computer is no longer running, so it cannot respond to the "I" key. Reset and try again with 30=BR. What happened this time? Why? Look again at the program, and try to convince yourself that the computer did exactly what it should have done. First it executed an INP instruction which input the 30 into memory location 0008 (if you don't believe it, reset the computer and examine location 0008). Then it executed the byte in 0008, which is the opcode of a BR instruction. But this is a two-byte instruction, so location 0009 is fetched to be used as an address. 0009 also

contains 30 (it is the opcode of another BR instruction, but the computer does know that at this time). So the computer branches to location 0030, which is an OUT 4 instruction.

Now let's try some new opcodes. Remember I said that each branch instruction had a paired opcode which branches on the false of the same condition? It is bit 3 of the opcode that makes the difference (that is an 8 in the second digit: 0-7 has a zero in bit 3; 8-F has a one in bit 3). You already know about the difference between 37=B4 and 3F=BN4. Does it work for 30 (which is branch always)? Is there a "branch never"? Try it. If the opcode in 0008 is "branch never" then what is the next instruction to be executed? No, it is not in 0009, because all branch instructions are two bytes, even if the branch is not taken (to convince yourself of that, review what you learned about the B4 instruction). We thought that location 000A was the second byte of a BR instruction, but it seems that the computer thinks it is the next opcode. So off goes Q, and pretty soon the OUT 4 at 000B gets executed.

There is a moral to this lesson, besides what you learn about the new instructions. That is, the computer is the final judge of what your program is going to do. It does not matter much if you think that such and such a location in memory is part of a two-byte instruction; when the computer fetches the second byte during an opcode fetch then it is an opcode. As in religion, if you want to avoid being surprised at the end, you have to follow the rules laid down by the final judge.

But what good is a "branch never"? Not much, as such. However, do notice that the second byte has no effect on its execution (it never uses the branch address), so we can put anything we want there. In fact, if it is a one-byte instruction, then the 38 instruction effectively skips over it, so we call this instruction SKP for "skip". It does not do anything the BR instruction could not do, but it does it in one byte instead of two.

```
SKP     Skip one byte                                   38
-------------------------------------------------------------
The one-byte instruction following the SKP opcode is skipped.
```

Program 3.10 is not set up to demonstrate the operation of long branches, but you can try the "long branch never". What would you expect to happen? In other words, what will be the next instruction after a C8 opcode? Try it. How is this different from 38=SKP? If Q does not go off, then obviously the REQ "instruction" in 000A is not being executed. Since you see "0C" on the hex display, you have to assume that the BR in 0009 is not being executed either, but that the OUT 4 in 000B is being executed. We call the C8 opcode a "Long Skip" because it skips two bytes.

```
LSKP    Long Skip                                          C8
-------------------------------------------------------------
Skip the two bytes following the opcode.
```

Let me quickly introduce you to the instruction that does nothing. Try the opcode "C4" in Program 3.10. What happens? Obviously it is not a skip or branch, because the branch in 0009-000A is executed. It does not turn Q off; it does not output anything that you can see. You will have to take my word for it that it does nothing else. We use it when we want the computer to kill some time, or we want to leave some space in a program for later additions, or we want to remove an instruction without moving the whole program back down.

```
NOP     No Operation                                       C4
-------------------------------------------------------------
This instruction does nothing, except take three major cycles
to execute.
```

Let's try a new conditional branch. Enter the value "32" into Program 3.10. What happens? How is this different from the 38=SKP? If you agree that it does not seem to be different, turn on the M/P switch (so that the 32 in location 0008 will not be changed by subsequent entries), then try entering various other pairs of digits. What happens if you enter "00"? Can you find any other pair of digits that is distinctive? Convince yourself that this only works with the 32 opcode, and not with the 38 or 30 opcodes. Why did I tell you to turn on the M/P switch for this test? If you enter 00 with the M/P switch off, what happens to the program? Evidently, after an INP instruction in which the byte that was input is 00 the 32 opcode branches, but for all other bytes, the 32 opcode does not branch. If you go back and review the formal definition of the INP instruction, you will notice that it not only inputs to memory, but the same byte is also placed in the accumulator, D. When memory is protected, memory cannot be altered so only D is changed. The 32 opcode branches if D is exactly equal to zero, and not otherwise.

```
BZ a    Branch on Zero                                 32 aa
-------------------------------------------------------------
Branch if D is zero; otherwise execute the next instruction
in sequence (after the branch address).
```

If there is a Branch on Zero, is there also a Branch on Not Zero? What would be its opcode? Try it.

```
BNZ a   Branch on Not Zero                             3A aa
```

```
------------------------------------------------------------
Branch if D is not exactly 00; otherwise take the next
instruction in sequence (after the branch address).
```

If you think about it for a minute you will realize that there might also be a Long Branch on Zero and a Long Branch on Not Zero. As I said, Program 3.10 is not set up to test long branches, so you cannot use it to observe these instructions in action.

Exercise: Write a program to test LBZ and LBNZ. Convince yourself that these are three-byte instructions that can branch anywhere in memory.

```
LBZ aa  Long Branch if Zero                          C2 aaaa
------------------------------------------------------------
If the accumulator D is exactly zero, replace the contents of
the Program Counter with the second and third bytes of this
instruction; otherwise proceed to the next instruction in
sequence.

LBNZ aa Long Branch if Not Zero                      CA aaaa
------------------------------------------------------------
If the accumulator D is not exactly zero, replace the
contents of the Program Counter with the second and third
bytes of this instruction; otherwise proceed to the next
instruction in sequence.
```

There are two more instructions which you can learn about using Program 3.10. This time run it once with M/P off and enter "C", "6". Notice that it behaves like an LSKP instruction. Now turn on M/P so you can feed the new opcode various data bytes. Did you try 00? What happened? Notice that opcode C6 skips when D is not zero; when D is exactly zero the instruction in 0009 is executed. Now try the opcode "CE". We call these "Conditional Skips". Notice that the condition being tested seems to be the reverse from the conditional branches. If you think about it, you will also see that C4=NOP is really a "Skip Never". Do you think the CC opcode would be a "Skip Always" instruction? What about C8=LSKP? Try CC. We will come back to this one.

```
LSZ     Long Skip if Zero                              CE
------------------------------------------------------------
Skip two bytes if D is zero; otherwise execute the next byte
in sequence after the LSZ opcode.
```

LSNZ Long Skip if Not Zero C6

Skip two bytes if D is not zero; otherwise execute the next byte in sequence.

Let us look at one more condition before moving on to the register operations. Key in Program 3.11:

```
        ..  PROGRAM 3.11 -- TEST Q CONDITIONALS
        ..
0000 7A         REQ     .. Q OFF
0001 6C         INP 4   .. GET NEXT OPCODE
0002 7B         (SEQ)   .. MAYBE Q ON AGAIN
0003 C1         LBQ 10  .. **
0004 00
0005 0A
0006 64         OUT 4   .. NOTE NO BRANCH
0007 11         ,11
0008 30         BR  0   .. REPEAT
0009 00
000A 64         OUT 4   .. NOTE BRANCHED
000B 33         ,33
000C 30         BR  0   .. REPEAT
000D 00
```

First run this program with M/P on. What shows on the hex display? Notice that the program does not wait for or depend on EF4, so it is ignoring the "I" key. Now push the keys "7", "A", and flip the M/P switch off. What happened to the hex display? What is the difference between the program as it is now running and how it was running a few minutes ago? Flip the M/P switch back on and try entering other data: Is there any change? Clearly, the program is different depending on whether it is executing 7A=REQ or 7B=SEQ in location 0002. Obviously the new instruction in 0003-0005 depends on Q. As you probably guessed by now, it is a Long Branch if Q is on. Change location 0003 to hex C9. What do you think this will do to the program? Try it.

LBQ aa Long Branch if Q is on C1 aaaa

Branch to the location whose two byte address follows this opcode if Q is on; otherwise ignore the address and take the next instruction in sequence.

LBNQ aa Long Branch if Q is off C9 aaaa

Branch to the location whose two byte address follows this opcode if Q is off; otherwise ignore the address and take the next instruction in sequence otherwise.

As you might have guessed, there is also a Short Branch on Q. To test this, put a NOP=C4 into location 0003, then a hex 31 into location 0004. Leave the "0A" in location 0005; it is the second byte of the branch. Try it. Then change the 31 to hex 39. Notice that after entering 7B to the running program, the "31" opcode branches; after entering any other pair the "39" opcode branches.

BQ a Branch If Q is on 31 aa

Branch to the location whose one-byte address follows this opcode if Q is on; otherwise ignore the address and take the next instruction in sequence.

BNQ a Branch if Q is off 39 aa

Branch to the location whose one-byte address follows this opcode if Q is off; otherwise ignore the address and take the next instruction in sequence.

And of course, there is a "Skip if Q". To test this, replace 0003-0005 with the three bytes CD, 64, FF. Now when you run the program, if Q is on what do you see? When Q is off, what happens? Where is the OUT 4 instruction followed by "AA"? To help you answer this question, try switching the computer into Wait a few times. If your eye sees "FF" and "11" in rapid succession, what will it look like? Now change the CD in location 0003 to C5 and run the program again, alternately pushing 7B and 7A on the keys. Do you think you understand this skip instruction? If not, reset the computer and write down the contents of memory from 0000 to 0009. See if you can think like the computer as it would to execute the bytes in the program. Does that help?

LSQ Long Skip if Q is on CD

If Q is on, skip two bytes; otherwise execute the next byte after this opcode.

LSNQ Long Skip if Q is off C5

```
If Q is off, skip two bytes; otherwise execute the next byte
after this opcode.
```

Now we have learned all of the 1802 instructions which do input or output; these are INP, OUT, B4, BN4 (also B1, B2, B3, etc.), SEQ and REQ. We have learned most of the conditional skips and branches: BZ, BNZ, LBZ, LBNZ, LSZ, LSNZ, BQ, LBQ, LSQ (and BNQ, etc.), and B4 etc. and all of the unconditional skips and branches: BR, LBR, SKP, and LSKP. And we know how to use NOP and IDL. If you look at the opcodes for these instructions, you will see that they include all the opcodes that begin with 6 except 60 and 68. Opcode 68 is not a defined instruction for the 1802 (though it is for the 1804, 1805, and 1806). We will discuss 60 in the next chapter. You will see that we have covered all the opcodes that begin with 3 except for 33 and 3B, and all those that begin with C except C3, C7, CB, CC, and CF. Opcode CC has to do with the interrupt system, which we will cover in Chapter 6. All the others are conditional skips and branches which test the DF or carry flag. Since I have not yet shown you how to set or clear the carry flag, you have no way to see how these conditionals work. For now, please accept that they look similar enough to the other conditional skips and branches that you do understand, so that when we get to DF, you will have no particular trouble with these instructions.

There is one form of input and output which your computer is capable of that does not use input or output instructions. We call it **DMA** or **Direct Memory Access** because it goes directly to memory without affecting the program (except for the time it takes to cycle memory). We will discuss this in some detail in Chapter 7.

Chapter 4 -- Register Operations

When I introduced the 1802 in Chapter 2, I mentioned that it had 16 general-purpose registers. In this chapter you will see how to manipulate these registers. So far we have used only R0 since Reset forces P and X both to point to R0. I am going to have to presume on your patience just a little for the first few instructions we study in this chapter: Since R0 is the only register we know anything about, we will try out some of the new instructions on R0. This will necessarily leave you with a little bit of an empty feeling, since you will not yet have a good feel for the power of these instructions. Bear with me, and you will become more comfortable with them as you see them used in the later examples.

Put Program 3.10 back into your ELF II. With M/P off, run the program and key in "10". What happens? How is this instruction different from a SKP? What happened is that this instruction incremented R0 (by increment we mean "add one"); since P=0, the next instruction byte was skipped. Try it with the instruction "11" or "14". Now what happens? You will have to believe me for now, that R1 or

R4 was incremented. At the moment we cannot tell if it was or not. Later you can prove it in a better program.

```
INC r   Increment register                                  1r
-----------------------------------------------------------
Increment (add one to) the address register specified in the
right digit of the instruction.
```

Still in Program 3.10, try the opcode "20". What happened? Remember, if you keyed in the program correctly, all the instructions you know about result in some output when you push the "I" key. Or do they? Try IDL. Is 20 different from 00? If you wired up a Run/Stop light, you would see that with 20, the computer did not stop, but it still refused to go to the next instruction. (If you do not have a light to tell you when the computer stops, take my word for it; it didn't.) Suppose I tell you that the 20 instruction backs up R0 by one (we call this a decrement). Since R0 is also the PC, it now points to the same instruction again. The fetch cycle fetches the opcode and increments R0; the execute cycle decrements R0. It is stuck in a loop just as surely as if you had coded a branch that jumped to itself. Reset the computer and try it with "21" or "24". As you can see, you can't see what it did. We will come back to this later.

```
DEC r   DECrement register                                  2r
-----------------------------------------------------------
Decrement (subtract one from) the specified register.
```

I will assume you have a vague notion about the operation of INC and DEC. Key in Program 2.3. How many instructions are there in this program that you have not yet met? Let's think about this program a little. Remember that R1 is 16 bits. That means that it can have any value between 0 and 65535. It also means that only half of it will fit into the accumulator (D) at a time. Suppose R1 is initially zero. As soon as you execute the INC R1 instruction it will be one, not zero. Look at it in binary:

00000000 00000000 initially
00000000 00000001 after increment

Notice that most of the register is still zero. In fact, the whole left half is zero, even if you increment it again:

00000000 00000010

How many times can you increment it before the left half is not zero? Here is what it looks like after 255 increments:

00000000 11111111

The next increment takes it to

00000001 00000000

How about after 511 increments (in all)? You see, after each 256 increments, the left half goes up one. How many increments will be required to take the left half up to all ones? Would you believe that it takes 65535 (=255x256+255) increments to go from all zeros to all ones? Then what happens? It takes only one increment to go from all ones to all zeros. Notice however, that for 65280 of the 65536 increments that it takes to go from all zeros back to all zeros, the left half of the register is not zero. Now look at Program 2.3 again. If the accumulator is zero, the LSZ instruction will skip over the REQ and the SKP and execute the SEQ: the Q light comes on. But if the accumulator is not zero, the LSZ falls through to execute the REQ (turning Q off) and the SKP (which skips over the SEQ). Since the Q light obviously blinks, you have to assume that sometimes the accumulator is zero, and sometimes it is not.

I have not said much about instruction execution time (that is one of the topics of Chapter 7), but let me remark for this discussion, that each instruction in your ELF II requires 11 microseconds (except for those that have a left digit of "C", which require 16 microseconds). This means that the loop will go through once in about 70 microseconds, that is, it can repeat the loop about 15000 times per second. How does this compare with how often the Q light blinks? Think of it as a Monopoly game; every time R1 passes "GO" (in this case all zeros) Q blinks. Thus you can deduce that the zeros in R1 are probably making the accumulator zero. Take a guess at how long the Q light stays on. Does it seem more like 1/50th of a second than, say, one second or 1/15000th of a second? About how many times can your ELF go through the loop in 1/50th of a second? Would you believe 256? By now you should be ready to believe that the "91" opcode is copying the left half of R1 into the accumulator.

Just for fun, change the 91 to "81". Would you believe it is still blinking? Try stopping the computer (put it into the Wait state). Take a guess at how fast it is blinking now. Does it seem reasonable to you that there should be two instructions in the 1802; one for copying the left half of R1 into D and one for copying the right half? Try another experiment: Change the right digit in location 0000 to some other number (e.g. "9B") and also change the INC in location 0005 to increment the same register (i.e. make it "1B"). What happens if you change only one of them? Try some other numbers. Why does it not work for zero?

```
GLO r   Get LOw byte of register                    8r
```

```
------------------------------------------------------------
Copy the least significant eight bits of the specified
register into D.

GHI r   Get HIgh byte of register                          9r
------------------------------------------------------------
Copy the most significant eight bits of the specified
register into D.
```

You should also try changing the INC in Program 2.3 to a DEC. How does that affect the program operation? Try thinking of DEC as "un-INC". If you decrement a register containing all zeros, what is the result? I will say more about negative numbers in Chapter 5.

You may have guessed by now that the 1802 also lets you copy data from D into the address registers. Key in Program 4.1, and with the M/P switch on, run it.

```
        ..  PROGRAM 4.1 -- TEST PHI AND PLO
        ..
0000 6C         INP 4   .. GET A COUNT
0001 BE         PHI E   .. **
0002 7A         REQ     .. Q OFF
0003 2E         DEC E   .. DELAY:
0004 9E         GHI E   .. LOOK AT IT
0005 3A03       BNZ *-2 .. FALL THRU ON 00
0007 6C         INP 4   .. DITTO, LO BYTE
0008 C4         NOP     .. OR PLO E
0009 7B         SEQ     .. Q ON
000A 2E         DEC E   .. DELAY
000B 9E         GHI E   .. N COUNTS MORE
000C 320A       BZ  *-2
000E 3000       BR  0   .. REPEAT
```

Notice that it does not wait for the "I" key (EF4). Push various digit keys. See if you can see a relationship between the time to the next blink and the last two keys you pushed. (Hint: try "11" and "FF"). Now change the NOP in 0008 to "AE" and the "BE" in 0001 to a NOP ("C4"). Don't forget the M/P switch! How do the keys affect it now? Can you see a difference in the length of the blink? Try "11" then "FF". As you can see, opcodes "BE" and "AE" are changing the amount of the count in RE. You should know what is in D (from the INP 4 instruction, since you pushed the digit keys); does that seem to affect the timing of the program in a reasonable way? Are you ready to believe that these two instructions copy D into the two halves of register 14? If you are not yet sure, study the program some

more. You see, it is not all that different (in principle) from Program 2.3. What happens if you key in "00"? Can you guess why?

```
PLO r   Put D into Low byte of register                    Ar
-----------------------------------------------------------
Copy D into the least significant eight bits of the specified
register.

PHI r   Put D into High byte of register                   Br
-----------------------------------------------------------
Copy D into the most significant eight bits of the specified
register.
```

Now you know how to set up an address register; but what is it good for? Well, maybe we can make the OUT instruction a little more useful. Key in Program 4.2:

```
        ..  PROGRAM 4.2 -- TEST SEX
        ..
0000 90         GHI 0   .. COPY R0 TO R8
0001 B8         PHI 8
0002 80         GLO 0
0003 A8         PLO 8
0004 3F04 WOW:  BN4 *   .. WAIT FOR "I"
0006 E8         SEX 8
0007 64         OUT 4   .. OUTPUT TO DISPLAY
0008 C4         NOP     .. SPACE FILLER
0009 3709       B4 *
000B 3004       BR WOW  .. REPEAT
```

Notice that the first thing this program does is copy R0 to R8. What address do you think will now be in R8? Obviously, the whole program is in the first page (256 bytes) of memory, so you would expect the left half of R0 to be zero (remember that Reset puts all zeros into R0). What about the right half? No, it's not zero. R0 was zero before executing the first instruction; after fetching it, R0 was incremented to 0001. Let's try another one; what is the value of R0 when the GLO 0 instruction is executed (i.e. after it is fetched; think about the INP instruction as we have been using it). Before running this program, take a guess at what the output will be when you push the "I" key. Why was it not "C4" That's right, blame the new instruction, "E8". You will recall that the OUT instruction actually outputs the byte pointed to by the address register pointed to by X. Reset sets both X and P to zero, and we depended on that. Now, "E8" changed X to point to R8 (you know what was in R8, right?), so the OUT instruction used R8 instead of R0. Do you understand why you saw "A8"? Where is that byte in the

program? Now guess what will happen when you push the "I" key again. Review the description of the OUT instruction in Chapter 3 again if you do not understand what happened. Which register is incremented by OUT? (Hint: what does X point to?) Now change the NOP in 0008 to a SEQ. Do you think this byte will be executed or skipped? Try it. Why does Q come on here but not if you run Program 3.8 with 7B in 0001 or 0008? Have you looked to see what X points to in each case? What does OUT do to that register? Which register does P point to? Experiment with different registers for the PHI, PLO, and SEX instructions. Can you figure out what happens if you use R0?

SEX r Set X Er

Set register X to point to the specified register r.

We know of one way to put data into the D register (the INP instruction); is there some way which does not inconvenience the operator (you) quite so much? Obviously there is. To see how it works, first sequence memory (i.e. run Program 2.2), then key in Program 4.3:

```
        ..  PROGRAM 4.3 -- TEST LDI
        ..
0000 90         GHI 0   .. SET PAGE 00
0001 B7         PHI 7
0002 F833       LDI #33 .. **
0004 A7         PLO 7   .. LOW BYTE OF ADDRESS
0005 E7         SEX 7   .. SELECT FOR OUT
0006 64   LOOP: OUT 4   .. DISPLAY CONTENTS
0007 27         DEC 7   .. BACK UP R7
0008 3F08       BN4 *   .. WAIT FOR "I"
000A 6C         INP 4   .. GET KEYIN
000B 64         OUT 4   .. ECHO IT
000C 370C       B4 *    .. WAIT FOR RELEASE
000E 3006       BR LOOP .. REPEAT
```

Run this program and look at the display. What do you see? Where in memory do you suppose the 33 came from? Key some value into the hex keypad and push the "I" key; did your input show up on the display? Now release the "I" key and look again. Repeat this a few times so you know what the program is doing. Then reset the computer and examine memory. See if you can find where your inputs are stored. Now think about the program; which address register did the OUT and INP instructions use? (Hint: Look at the SEX instruction). Remember that when the computer executes a SEX, X continues to point to the specified address register until for some reason the computer changes the contents of X again. In

this case you understand all of the instructions in the loop that follows the SEX, and there is no way out of that loop except resetting the computer. So all that time X=7. What is in R7? Notice at the beginning of the program that the program puts 00 into the left half of R7 (it got the 00 from the left half of R0). Recall also that the OUT instruction increments the address register (R7 in this case). Inside the loop there are two OUT instructions, and one DEC 7; that is two times R7 is incremented, and once it is decremented. What happens to R7 if you add one twice and subtract one once? Does this correspond to what you noticed the program doing? Finally, can you tell what was in R7 at the beginning of the loop? Clearly it must depend on whatever is in D when the 1802 executes the PLO 7 instruction, but what is it? Do you see a correspondence between the second byte of the LDI instruction and the starting value of R7? Convince yourself that it is not a relationship with the previous contents of memory by running the program again. This time what is displayed? Does it remind you of what you previously keyed in? Change the byte in location 0003 to something else and try it again. When the data that the instruction uses (in this case, that it loads into D) is immediately following the opcode, we call it the **Immediate addressing mode**.

LDI b Load D Immediate F8 bb

Copy the second byte of the instruction into the D register.

Now that we know how to put specific addresses into an address register, things will be a little easier to test. In Program 4.4 we set up two address registers with different values. One of them we will use for inputting data; the other will demonstrate the next instruction. First sequence memory, then key in Program 4.4:

```
        ..  PROGRAM 4.4 -- TEST STORE
        ..
0000 90         GHI 0   .. SET UP R6 AND R7
0001 B6         PHI 6
0002 B7         PHI 7
0003 F833       LDI #33 .. DISTANCE IN RAM
0005 A6         PLO 6
0006 F81E       LDI #1E .. CLOSE FOR EASY LOOK
0008 A7         PLO 7
0009 3F09 LOOP: BN4 LOOP.. WAIT FOR "I"
000B E6         SEX 6   .. INPUT A BYTE
000C 6C         INP 4
000D 370D       B4 *    .. WAIT FOR RELEASE
000F 26         DEC 6   .. MOVE OFF INPUT
```

```
0010 57         STR 7   .. **
0011 3009       BR LOOP .. REPEAT
```

There is only one instruction here you have not yet met. Ignoring that, what would you expect the program to do? Notice there are no OUT instructions, so you cannot expect the display to change. Run it, and key in a few numbers, remembering what you enter. Now reset the computer and look at memory. What is in memory location 001E? Why should it be the last number you keyed in instead of the "1E" that the sequencer put there? Look also at the few bytes before location 0033; do they correspond to what you expected the program to do? If not, think through the program again. Notice that after each input, R6 was decremented. Now change the 57 in location 0010 to a 56. Before you run the program again, see if you can guess what will be different. Try it. Were you right? Did you notice that the memory R7 points to is now unchanged, but the memory R6 points to is now the same as what was last input? Change the DEC 6 instruction to INC 7, and put the 57 back in 0010 (i.e. put 17 in 000F, 57 in 0010) and run it again. When you examine memory this time, what is it you see near 001E (I hope you saw a sequence of input bytes)? Do you think you understand the STR instruction?

STR r STOre D into memory 5r

Using the specified address register, store (copy the contents of) the accumulator into memory.

The 1802 has two store instructions. The second one is a little tricky to understand. Resequence memory, and put Program 4.4 back in, but this time change the STR instruction to a hex "73". Run the program and examine the part of memory near location 0033. How does it differ from what you expected? Change the DEC 6 to a SEX 7 (hex E7 in 000E) and run it again, but be careful not to enter more than 5 or 6 data bytes. Where did the data show up? Do you see a relationship this time between R7 and where the data was stored? What can you guess about which address register the 73 opcode uses? Does anything special happen to that register? Compare this to the STR instruction; did it change its address register? If you are not sure you know what happened, read the instruction summary for STXD (below) and review the program again.

STXD STore D via R(X) and Decrement R(X) 73

Store the accumulator into the memory location pointed to by the address register pointed to by X, then decrement that address register.

As you might have guessed, when a computer has an instruction to copy data from one kind of memory to another, it usually also has an instruction to copy data in the other direction. The opposite direction from a Store is a Load, and you have seen one of these already (LDI). Let's look at a few others. Program 4.5 is a little larger than the others, but we hope to make it do the work of two or three. First sequence memory, then key in Program 4.5:

```
        ..  PROGRAM 4.5 -- TEST LOADS
        ..
0000 90         GHI 0   .. SET UP R8 AND R9
0001 B8         PHI 8
0002 B9         PHI 9
0003 F880       LDI #80 .. R8=0080
0005 A8         PLO 8
0006 F890       LDI #90 .. R9=0090
0008 A9         PLO 9
0009 E9   AGAIN:SEX 9
000A 7B         SEQ     .. ANNOUNCE US
000B 3F0B       BN4 *   .. WAIT FOR INPUT
000D 6C         INP 4   .. GET OPCODE
000E 64         OUT 4   .. SHOW IT
000F 7A         REQ
0010 50         STR 0   .. STORE INTO PROGRAM
0011 C4         NOP     .. EXECUTE IT HERE
0012 29         DEC 9   .. POINT TO SCRATCH
0013 59         STR 9   .. STUFF D THERE
0014 3714       B4 *    .. WAIT FOR RELEASE
0016 64         OUT 4   .. SHOW DATUM
0017 29         DEC 9   .. NOW
0018 88         GLO 8   .. SHOW R8,
0019 59         STR 9
001A 3F1A       BN4 *   .. AT THE SIGNAL
001C 64         OUT 4   .. SHOW DATUM
001D 89         GLO 9   .. SHOW R9,
001E 29         DEC 9   .. NOW,
001F 59         STR 9
0020 3720       B4 *    .. AT THE SIGNAL
0022 64         OUT 4   .. SHOW DATUM
0023 3009       BR AGAIN.. REPEAT
```

Before running this program, look at it carefully and be sure you understand exactly what it will do. There are no instructions that you have not met. There are, however, a few tricky spots. At location 0010 there is a STR 0; what do you think

is in R0 when this instruction is executed? (Remember, R0 is the PC, which points to the next instruction). What will happen to the NOP in 0011? What will be the next instruction to be fetched (and executed)? Think: does STR change its address register? If R0 points to 0011 before the STR is executed, then R0 also points to 0011 after the STR is executed, but before the next instruction is fetched. How does this compare with Program 3.9 and 3.10?

Turn on the M/P switch the first time you run this program, just so you can get a feel for its operation. Notice that nothing you key in has any effect, since memory is protected. Get comfortable with what happens as you push the "I" key: First Q comes on, and it waits for the "I" key; when you push it, Q goes off, and the display shows what you keyed in (but of course, since memory is protected, you only see the previous contents of 0090, which is 90); when you release the "I" key, you see the results in D of executing the opcode that was input (or in this case, the previous contents of 008F); when you push the "I" key again it tries to display the right half of R8, then R9 when you release it, both in the same memory location. (If I seem to dwell excessively on understanding what the programs are doing, it is because you cannot expect to write computer programs if you cannot understand what a program is doing.)

Now reset the computer, turn off M/P, and run it again. First key in "C4" (NOP), and push "I" twice slowly. Do you understand why you see what is on the display? Without keying in any new datum, push "I" twice again. Notice that R8 does not change, but R9 is incremented. Now key in "19" (INC R9), and watch it. Try "18" (INC R8), "29" (DEC R9), and "28" (DEC R8). Since you understand these instructions already, you should be able to follow what you see. Try "73" (STXD). Try GHI and GLO for various registers: what is in the registers you have not been using? What happens with PLO 9 or PLO 8? Now that you have clobbered your registers, Reset the computer and Run again. Before trying it, try to analyze what will happen when you key in "F8" (LDI). Remember LDI is a two-byte instruction; what is the second byte? What did that do to R9? If "29" is the second byte of an LDI at location 0011, do you understand why it is not executed as a "DEC 9"?

Once you are convinced you understand how this program works, and you have tried some of those previous instructions that you did not exactly understand (I hope it helped to run them through this program), you are ready to try some new opcodes. Key in "48". What happened to R8? Did this remind you of the INC 8 instruction? How about D? Do you know what was in memory where R8 pointed? Perhaps it would help to resequence memory, and put Program 4.5 back in, so that you know exactly what is in memory; that is a choice up to you, if you feel unsure about what this opcode did. Remember, the only way you can be sure you know

what an instruction did is to know what the complete state of the computer was both before and after executing it. Now try "49". How is this different from 48? Try "40". Do you know what happened? If not, compare it to LDI ("F8").

```
LDA r   Load D and Advance                                  4r
-----------------------------------------------------------
Copy the contents of the memory byte pointed to by the
specified address register r into the Accumulator, and
increment the register.
```

Still using Program 4.5, try keying in "08" and "09". How is this different from the LDA instructions 48 and 49? Is "00" different from 40 in the same way? What happens when you try it? You do remember that 00 is IDL, don't you?

```
LDN r   Load D via N (r = 1 to F)                           0r
-----------------------------------------------------------
Copy the memory byte pointed to by the specified address
register r into the Accumulator.
```

Now try three other opcodes in Program 4.5: "60", "F0", and "72". How do these compare respectively with INC 9, LDN 9, and LDA 9? What is in X at the time these opcodes are executed by the computer? Does this suggest how these might be related to X? Can you think of some way to modify Program 4.5 to see if your guess is correct?

```
IRX     Increment R(X)                                      60
-----------------------------------------------------------
Increment the register pointed to by X.

LDX     Load D via R(X)                                     F0
-----------------------------------------------------------
Load the accumulator from the memory byte pointed to by the
address register pointed to by X.

LDXA    Load D via R(X) and Advance                         72
-----------------------------------------------------------
Load the accumulator from the memory byte pointed to by the
address register pointed to by X, then increment that
register.
```

Now you have a good understanding of the register operations in the 1802. This includes all opcodes with a left digit of 0, 1, 2, 4, 5, 8, 9, A, B, and E, as well as

the opcodes 60, 72, 73, F0, and F8. In the next chapter we will use them to help us study the arithmetic and logical instructions.

Chapter 5 -- Arithmetic and Logic

In this chapter we will be using only one program, which will enable us to observe the various arithmetic and logical operations of the 1802 computer. Before we get into the details of Program 5.1 and these operations, let's quickly review grammar school arithmetic (adding and subtracting) and some of the logical operations a computer can do.

When you were in grammar school, you added two numbers (say 43 + 25) one digit at a time, starting from the right: 3+5=8; 4+2=6; result = 68. Now you know that 3+5=8, but when you were first learning how to add, you had to memorize a big addition table. Exactly the same kind of process goes on when your computer adds, except that its table is for binary numbers, not decimal, and looks like this:

```
 + | 0  1
 0 | 0  1
 1 | 1 10
```

Your ELF II, however, does not let you look at the binary numbers directly, so it helps to imagine that the addition table that the computer has memorized is actually for hexadecimal, not binary. It turns out to have the same results. I will not reproduce the hex addition table here.

Subtraction works by the same principles as addition, as you will recall from way back when. But if you try to subtract a larger number from a smaller number, things get a little tricky. I think the best way to introduce you to most of these problems is by seeing what the computer actually does.

If you want your computer to multiply or divide, you are almost out of luck. The 1802 and most other 8 bit microprocessors can only multiply or divide by two. The operation is called a **shift**. If you shift a number left by one bit position, you have doubled it (i.e. multiplied it by 2). Similarly, shifting a number right one bit position divides the number by two. More on this later.

There are three logical operations which most microprocessors can do with a single instruction. They operate on the individual bits (remember, one bit is a single true-false answer) in the computer words (or bytes). I will give you the tables now, so you can compare these operations to the instructions later.

```
AND| 0  1
 0 | 0  0
 1 | 0  1
```

```
OR | 0  1
 0 | 0  1
 1 | 1  1

XOR| 0  1
 0 | 0  1
 0 | 1  0
```

Now key in Program 5.1. As you can see, it requires three inputs; an opcode, which is executed at location 0029, and two data bytes, one stored in location 0060 (via R6) and one stored in location 0061 (same address register, but incremented). It does its operation, then puts the result back into location 0060 to display it. If the result is zero, the program also turns Q on. So that you will know which input is expected, the display will show "00" when it is waiting for an opcode, "01" when waiting for the first operand, and "02" when waiting for the second operand. After it performs the operation, the result also becomes the first operand of the next time through the loop, so you only have to key in the second operand for each successive time through. Of course, any time you want to put a new opcode or first operand, you can reset the computer and run the program again. Look over the program and be sure you understand it. Notice that it contains only instructions you have met in Chapters 3 and 4.

```
        ..  PROGRAM 5.1 -- TEST ALU OPS
        ..
0000 90         GHI 0     .. SET UP R6
0001 B6         PHI 6
0002 F829       LDI DOIT  .. FOR INPUT OF OPCODE
0004 A6         PLO 6
0005 E0         SEX 0     .. (X=0 ALREADY)
0006 6400       OUT 4,00  .. ANNOUNCE US READY
0008 E6         SEX 6     .. NOW X=6
0009 3F09       BN4 *     .. WAIT FOR IT
000B 6C         INP 4     .. OK, GET IT
000C 64         OUT 4     .. AND ECHO TO DISPLAY
000D 370D       B4 *      .. WAIT FOR RELEASE
000F F860       LDI #60   .. NOW GET READY FOR
0011 A6         PLO 6     .. FIRST OPERAND
0012 E0         SEX 0     .. SAY SO
0013 6401       OUT 4,01
0015 3F15       BN4 *
0017 E6         SEX 6     .. TAKE IT IN AND ECHO
0018 6C         INP 4     .. (TO 0060)
```

```
0019 64          OUT 4      .. (ALSO INCREMENT R6)
001A 371A        B4 *
001C E0          SEX 0      .. DITTO SECOND OPERAND
001D 6402        OUT 4,02
001F E6          SEX 6
0020 3F20 LOOP:  BN4 *      .. WAIT FOR IT
0022 6C          INP 4      .. GET IT (NOTE: X=6)
0023 64          OUT 4      .. ECHO IT
0024 3724        B4 *       .. WAIT FOR RELEASE
0026 26          DEC 6      .. BACK UP R6 TO 0060
0027 26          DEC 6
0028 46          LDA 6      .. GET 1ST OPERAND TO D
0029 C4    DOIT: NOP        .. DO OPERATION
002A C4          NOP        .. (SPARE)
002B 26          DEC 6      .. BACK TO 0060
002C 56          STR 6      .. OUTPUT RESULT
002D 64          OUT 4      .. (X=6 STILL)
002E 7A          REQ        .. TURN OFF Q
002F CA0020      LBNZ LOOP  .. THEN IF ZERO,
0032 7B          SEQ        .. TURN IT ON AGAIN
0033 3020        BR LOOP    .. REPEAT IN ANY CASE
```

When you first run this program, try entering "C4", "11", "22", then "33". Notice that after the first time through, the only display is the "11" alternating with your input. If you run it again, entering "00" instead of "11", you will notice that Q comes on and stays on. The program loads the first operand into D, executes the NOP you keyed in, then displays the contents of D (which has not changed).

Restart the program, but this time enter an opcode of "F0" (LDX). Now what data is displayed? Be sure you understand what you are seeing for the NOP and the LDX entered into this program before proceeding any further. It is essential to have a good feeling for the prior contents of D (the accumulator) and the byte in memory that the address register pointed to by X points to; the NOP shows you the former, LDX shows you the latter. The **ALU (Arithmetic and Logical Unit)** instructions combine these two data bytes to give some result; if you do not know what the operation starts with, you will not know how it got its answer. Notice however, that the first time through they are your inputs; after that they are the previously displayed result and the next input, respectively.

Now you should be ready to try a new opcode. Start with "F1", followed by "33", then "55". What was the result? Does repeating the operation with the same second operand (i.e. push the "I" key again) change the result? Try giving it "01". Can you figure out how to get a result of "FF"? Can you figure out which

operation this opcode does (ADD, AND, OR, or XOR)? (Hint: try giving it 00,00; 00,01; 01,00; and 01,01 and then compare the results to the four tables I gave you).

```
OR      Logical OR                                              F1
-------------------------------------------------------------
The datum in D and the datum in the memory byte pointed to
by the address register pointed to by X are combined in a
bit-by-bit inclusive OR, and the result is left in D.
```

Try the same experiments with the opcode "F2". Can you figure out how to get a result of "FF"? Have you figured out which operation this is?

```
AND     Logical AND                                             F2
-------------------------------------------------------------
The datum in D and the datum in the memory byte pointed to
by the address register pointed to by X are combined in a
bit-by-bit logical AND, and the result is left in D.
```

There is one more logical operation you have not yet tested. Try opcode "F3". Notice that repeated operations with the same datum in the second operand is not the same as for AND and OR. Do you see a pattern?

```
XOR     eXclusive OR                                            F3
-------------------------------------------------------------
The datum in D and the datum in the memory byte pointed to
by the address register pointed to by X are combined in a
bit-by-bit exclusive OR, and the result is left in D.
```

It might help you to understand how these instructions are used if you consider them in the following way:

AND forces to zero those bits in the accumulator which are zeros in the second operand.

OR forces to one those bits in the accumulator which are ones in the second operand.

XOR complements (sets zeros to ones, ones to zeros) those bits in the accumulator which correspond to ones in the second operand, leaving the other bits in the accumulator unchanged.

Also, XOR forces to zero those bits in the accumulator which match the corresponding bits of the second operand, and forces to one those bits which do

not match.

We have met before (in the branches and the I/O instructions) a family of opcodes with "positive" and "negative" versions, distinguished by a single bit. The ALU opcodes are something like them, but the bit distinguishes the Immediate addressing mode (two byte instruction, with the operand being the second byte) from the "R(X)" or **register indirect** addressing mode (the datum is in memory, an address register points to it; in the 1802 ALU operations that address register is selected by X). Consider the difference between LDX (F0) and LDI (F8). Try LDI in Program 5.1. What datum is displayed as a result? Where in memory did it come from? (Hint: look at "the second byte of the instruction"). When you have a good feeling for the difference, you are ready to try "F9". Does the bit pattern of the opcode give you a clue at to what to expect? Was your guess right?

```
ORI b   OR Immediate                                    F9 bb
-------------------------------------------------------------
All the bits in D corresponding to ones in the second byte
of the instruction are set to ones.
```

Try also "FA" and "FB".

```
ANI b   ANd Immediate                                   FA bb
-------------------------------------------------------------
All the bits in D corresponding to zeros in the second byte
of the instruction are set to zeros.
```

```
XRI b   eXclusive oR Immediate                          FB bb
-------------------------------------------------------------
All the bits in D corresponding to ones in the second byte
of the instruction are complemented.
```

Now you are ready to try the opcode "F4". For the first two operands, use something like "12" and "34". Obviously the result is the sum of these. Then enter a larger number like "58". Is the result what you expected? Remember your computer adds in binary (or hexadecimal), not the familiar decimal. Look at the sum in binary:

```
  binary        hex
 0100 0110       46
+0101 1000      +58
 1001 1110       9E
```

Notice that 6+8 (which is 0110 + 1000) is 14, but 14 decimal is 1110 binary, which is E in hexadecimal. Do you understand why 4+5=9? Notice here that the

second bit sum from the left is 1+1, which our table tells us is 10 (in binary). So the sum is 0 with a carry into the next digit to the left. Before you enter it, try to anticipate what adding "27" will result in. Start your paper sum with the rightmost bit, which is 0+1=1. The next bit sum is 1+1; what digit do you write down? What carry? The third digit is trickier: 1+1+carry=? Our addition table does not show what to do with three ones, but your recollection from grammar school should remind you that 1+1+1=3 which is 11 in binary. In other words, write 1, carry 1. Finish the sum, then try entering the number. Did the computer get the same result you did? If not, do you know what you did wrong? (Notice that I assume the computer's answer is the correct one. You can depend on the fact that any mistakes are yours, not the computer's. Computers these days, and your ELF II in particular, are more reliable than the cars we depend on every day.)

How much do you think you need to add to hex C5 to get FF? You can figure this out by looking for results of one in the addition table; notice that this happens for 0+1 and 1+0, so if the first operand has a zero, add one; if it has a one, add zero. Do you believe that C5+3A=FF? Try it. Do you think you can guess what FF+01 is? Try it. What happened to the carry out of the left bit of the sum? We will come back to that carry later. For now let us assume it vanishes (which is all you can see anyway).

If you add 01+02 you get 03, which is reasonable since 1+2=3. We think of subtracting A-B as "the number which added to B results in A." In other words, 1+2=3 is the same as 3-2=1. What number do you add to 1 to get 0? In high school algebra we called these "negative numbers": 1+(-1)=0. Now you already know what your computer will add to 01 to get 00. Obviously, hex FF is equal to -1. It is a strange sort of mathematics where 255=(-1). Can you guess what is -2? Does it seem obvious that -2 must be one less than -1? Try adding FE+02. Notice also that (-1)+(-1) = FF+FF = FE. Aside from the peculiarity that -2=254, negative numbers seem to do all the things we want them to. What is -4 (try adding -2 to itself)? Or -8? Can you figure out -9?

Negative numbers are simply the positive numbers subtracted from 256, which we observe to be indistinguishable from zero. You saw how to decide how much to add to a number to get hex FF; you add a number with each bit complemented (replace ones with zeros, zeros with ones). If A+B=-1, then A+(B+1)=0. Given A, you know how to find B; add one to that and you have computed -A. Let's try an example: What is the negative of (hex) 64?

```
  binary        hex
 0110 0100       64
+1001 1011      +9B (complement)
```

```
        +1      +1
 1001 1100       9C = -64
```

Add 64 and 9C on your computer to prove that this is correct. Try a few other examples. A little later, when we let the computer subtract, you will see that it does it the same way we did.

```
ADD     Add                                                    F4
-------------------------------------------------------------
Add the contents of the memory byte pointed to by the
register pointed to by X to the contents of D, and put the
sum in D. Put the carry out of the sum into DF.
```

I slipped you a ringer in the above description of ADD. The last sentence describes something you cannot see. Let's remedy that situation. At the end of Chapter 3, I mentioned that there was a set of conditional branches and skips that tested the DF register, but we could not see how they worked yet. Now you can. Change the opcode in location 002F of Program 5.1 from CA to CB. Then run it, giving it the ADD opcode with operands "11" and "22". Notice that Q stays off. Keep adding 22, and watch Q as the results go up. Notice what happens when you add FF+22=21. You see, there really is a carry out of the left bit of the sum. When that carry is one, the DF register is set to one; when the carry is zero (that is, there is no carry), DF is set to zero.

Do some addition in which the carry is zero, then restart the program so you can try another opcode, such as LDX, OR, AND, or XOR. Does the DF ever get set (as evidenced by Q coming on)? Now do an addition which sets DF to one, and repeat the experiment. Why does Q come on this time? Does this convince you that none of the other instructions we have met so far affect DF? Notice that not even Reset has any effect on it. To get a feel for the conditionals which test DF, change the opcode in 002F to C3 and run a few more sums through it. Try the short branch version by putting a NOP in 002F and a 33 in 0030. Try 3B. Can you think of how to change 002F-0031 to demonstrate the skip version of this test? (Hint: See what happens when you put LSZ in 002F and BR in 0030.)

```
BDF a   Branch if DF is 1                                33 aa
LBDF aa Long Branch if DF is 1                          C3 aaaa
-------------------------------------------------------------
Branch to the location whose address is in the second byte
(or second and third bytes) of the instruction if DF=1.

BNF a   Branch if DF is 0                                3B aa
```

```
LBNF aa Long Branch if DF is 0                          CB aaaa
-------------------------------------------------------------
Branch to the location whose address is in the second byte
(or second and third bytes) of the instruction if DF=0.

LSDF    Long Skip if DF is 1                              CF
-------------------------------------------------------------
Skip the next two bytes if DF=1.

LSNF    Long Skip if DF is 0                              C7
-------------------------------------------------------------
Skip the next two bytes if DF=0.
```

Be sure Program 5.1 is correctly modified to reflect the status of DF on Q (for example, CF in 002F and 30 in 0030). There are three other opcodes in the 1802 which do addition. Let's explore them a little.

Try the opcode FC. What does it do? Can you figure out why it ignores the second operand you are entering? What does it use instead? How does that compare with the difference between LDX and LDI? By trying different first operands, convince yourself that the operation is the same, only that the second operand is the C4 in location 002A. You could even try modifying that byte to verify this, but be sure to put the NOP back afterwards.

```
ADI b   ADd Immediate                                   FC bb
-------------------------------------------------------------
Add the value of the second byte of the instruction to the
accumulator, and put the sum back into the accumulator. Put
the carry out bit in the DF register.
```

The next one is a little more tricky. Do some addition which leaves DF=0, then enter the opcode "74" with small operands such as "11" and "22". Convince yourself that it still adds the same as ADD. Continue adding 22 until you get to the sum FF+22=21+carry. What result would you expect from the next addition if it were ADD instruction (F4)? If you are sure that 21+22=43, press "I" again and see if you know why you do not see 43. Try various other numbers. Do you see any relationship between the previous value of DF (as seen in Q) and the way this instruction adds? See if you can convince yourself that this instruction adds the two operands plus the value of DF (either 0 or 1). Try the immediate version of the same opcode (hex 7C). Do you understand it?

```
ADC     ADd with Carry                                  74
```

```
ADCI b  ADd with Carry Immediate                          7C bb
-------------------------------------------------------------
Add the memory byte pointed to by the address register
pointed to by X (or the second byte of the instruction) plus
the value of the DF register to the accumulator. Put the
result in the accumulator, and put the carry out of the sum
back into DF.
```

When I talked about negative numbers, I carefully introduced them before you knew about DF. As it turns out, DF is a little confusing when you think of your numbers as positive and negative. For example, if you add 3+5=8, the carry out is 0. If you add (-3)+(-5) = -8, the carry out is one. That does not mean the result is too big for your eight bits, only that there was a carry out. Well, it tells you a little more: (-3)+5=2 produces a carry=1; 3+(-5)=(-2) has a carry=0. If you know you are adding a positive and a negative number, the sum will have a carry out of zero if the result is negative. So what? That's not very helpful, yet.

We would like the computer to compute negative numbers for us, so we can do subtraction. A-B = A+(-B), so if we want to subtract some number from another, we need only find its negative and add. Remember the XOR? We can use it to find the negative of the number in the accumulator, by complementing every bit, after which it is easy to add one. Do you believe Program 5.2 will find the negative of whatever is in D? By then adding, this does an effective subtraction.

```
..  PROGRAM 5.2 -- SUBTRACT D FROM MEMORY
..
        XRI #FF    .. COMPLEMENT D
        ADI 1      .. ADD ONE
        ADD        .. ADD MEMORY (=SUBTRACT)
```

I did not include the object code for this program because I don't think you actually need to see it run. Especially because there is a single instruction in the 1802 which does the work of all three of these. Still using Program 5.1 (modified to reflect DF on Q as before), key in the opcode F5 with operands "01" and "03". What is 3-1? What does the computer say? Suppose you subtract the result from 3 again (hit "I" again); what do you get? Try subtracting 1-3. Notice DF; when subtracting positive numbers, DF=1 if the difference is also positive, and DF=0 if the difference is negative. If you consider all numbers positive (in the range 0-255) then 253-3=250 is positive, so DF=1; 237-241=(-4) is negative, so DF=0. Remember, it is only your opinion that matters in deciding whether a number is positive or negative. The number 237 equally well answers the questions "What is the 237th positive number?" and "What is the 19th negative number (as represented by counting down from 256)?" The computer does not know which of these two questions is answered by a hex ED.

SD **Subtract D from memory** **F5**

Add the memory byte pointed to by the register pointed to by X, to the complement of the accumulator plus 1. The sum is put back into the accumulator, and the carry out of the sum is placed in DF. This is equivalent to a Two's Complement subtraction of the accumulator from the memory byte.

As you might expect, the 1802 also has a subtract immediate. In Program 5.1 the immediate byte is hex C4. Try the opcode FD with various first operand values (such as C4,00 or FF,01, etc.). Notice that the second operand has no effect (Why?). In each case you should be satisfied that the first operand is subtracted from C4. Notice also that if you subtract the result from C4 again (by pressing "I" again), you get back your original datum. Notice the carry out each time (in Q).

SDI b **Subtract D from Immediate byte** **FD bb**

Complement the accumulator and add it plus one to the second byte of the instruction. The sum is put back into the accumulator, and the carry out is placed into DF.

As you probably guessed by now, any instruction which modifies DF has a version which adds the previous value of DF into the result. The purpose of this is so that you can do arithmetic on larger numbers than will fit in one byte. For example, to add two 16-bit numbers (say, one in RA that we'll call OP1, the other in memory pointed to by R8 which we'll call OP2), something like Program 5.3 would be used:

```
..  PROGRAM 5.3 -- ADD TWO 16-BIT NUMBERS
..
        SEX 8     .. R8 POINTS TO OP2
        GLO A     .. GET LOW BYTE OF OP1
        ADD       .. ADD LOW BYTE OF OP2
        PLO A     .. PUT SUM IN RA
        GHI A     .. GET HIGH BYTE OF OP1
        DEC 8     .. POINT TO OP2
        ADC       .. ADD WITH CARRY
        PHI A     .. PUT RESULT IN RA
```

It should not be hard to imagine that this would also work for a subtract. In this case we are concerned about borrowing rather than carrying (remember your grammar school subtraction), but it turns out that the same mechanism works for

both. Remember that subtraction consists of complementing the number to be subtracted, then adding one. Let's try a 16-bit example:

```
        binary            hex
 0100 0001  0011 0110    4136
-0001 0010  1011 0100   -12B4
```

First we find the negative of hex 12B4. To do this, complement all the bits, then add 1:

```
        binary            hex
 1110 1101  0100 1011    ED4B   complement of 12B4
                    +1     +1   add 1
 1110 1101  0100 1100    ED4C   is -12B4
```

Notice that I split the number into two bytes so you can see how the computer might compute it. You see, when the rightmost two bits were added, there was a carry into the next bit position to the left, but there was no carry out of the eighth bit across the byte boundary into the next byte (the ninth bit from the right). On the other hand, if I take the negative of hex 4000:

```
        binary            hex
 0100 0000  0000 0000    4000

 1011 1111  1111 1111    DFFF   complement of 4000
                    +1     +1   add 1
 1100 0000  0000 0000    C000   is -4000
```

Here the carry propagated; that is, each carry resulted in another carry, all the way across both bytes to the 14th bit; in particular the carry out of the low byte was 1, which had to be added into the high byte. Now if this 16-bit number is in one of the address registers and you instruct the computer to increment it, it will propagate the carry all the way across as it should. But the accumulator is only 8 bits, so the carry out of the low half is carried by DF into the high half.

Returning to our subtraction problem, we would like the computer to do the complement, add 1, and add all in the fewest machine instructions. Let's look at how this should work:

```
        binary            hex
 0100 0001  0011 0110    4136
-0001 0010  1011 0100   -12B4
```

Do the right byte first:

```
        binary            hex
 0100 0001  0011 0110    4136
           +0100 1011     +4B   complement of B4
```

```
                +1     +1
 1000 0000      80
```

Notice there is no carry out of this sum, so we do not want to use the same instruction to subtract the left byte, because the SD instruction adds that 1! Instead, we just want to add the complement of D to the memory byte:

```
        binary            hex
 0100 0001  1000 0000    4136
+1110 1101              +ED    complement of 12
 0010 1110  1000 0000    2E80
```

Notice the carry out of the sum, indicating that the result is positive. What happens if there is a carry out of the low byte? Let's try subtracting zero from our result:

```
        binary            hex
 0010 1110  1000 0000    2E80
-0000 0000  0000 0000   -0000
```

which is:

```
        binary            hex
 0010 1110  1000 0000    2E80
+1111 1111  1111 1111   +FFFF
                   +1     +1
 0010 1110  1000 0000    2E80
```

Here we got the same answer (which we should expect), and the carry out is 1 (which we should expect since the result is not negative). The carry across the byte boundary is also 1, not 0. If we knew this ahead of time, we could use the SD instruction on the left byte as well as the right byte. It is better, however, to have an instruction which will add the DF register into the sum, just as the ADC does. The opcode for this instruction is 75 (or 7D for the immediate addressing mode). Try using it with a few numbers until you see exactly what this instruction does. Don't forget to notice what is in the DF register the first time through (you might start with an ADD which leaves DF in a known state before testing this instruction).

```
SDBI b  Subtract D w.Borrow from memory Immediate byte  7D bb
SDB     Subtract D with Borrow from memory                75
-----------------------------------------------------------
Complement the accumulator and add it plus the value of the
DF register to the second byte of the instruction (or the
memory byte pointed to by the address register pointed to by
X). Put the sum into the accumulator and the carry out of the
```

```
sum back into DF. This instruction is used to extend a borrow
from a previous SD or SDB (or SDI or SDBI).
```

You should be aware that when DF=1, there is no borrow; when DF=0, the SDB effectively borrows from the next byte. If you are not convinced of this, try a few paper examples, or compare the meaning of DF after a subtraction (i.e. whether the result is positive or negative).

Sometimes it is more convenient to subtract the byte in memory from the accumulator. We would like to be able to do this without exchanging the two bytes, so by golly, there is a set of subtract instructions which goes the other way (but still leaves the result in the accumulator and DF). Convince yourself that you understand how the opcodes F7, FF, 77, and 7F work by trying them in Program 5.1. Notice that opcodes that start with "7x" add DF; and the ones whose second digit is greater than seven (78 to 7F) are immediate addressing mode.

```
SM      Subtract Memory byte from D                      F7
SMI b   Subtract Memory from D, Immediate              FF bb
SMB     Subtract Memory with Borrow from D               77
SMBI b  Subtract Memory with Borrow from D, Immediate  7F bb
------------------------------------------------------------
Complement the memory byte pointed to by the address register
pointed to by X (or the second byte of the instruction) and
add it plus 1 (or the DF register) to the accumulator. Put
the sum in the accumulator, and the carry out of the sum into
DF. This is equivalent to subtracting the immediate or memory
byte from D, with or without consideration of a borrow
incurred by a previous subtraction.
```

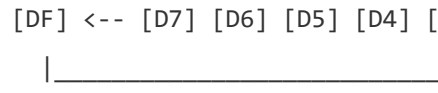
What happens when you add a number to itself? You should know that this is the same as multiplying it times two. Look at an example in binary:

```
  binary        hex
 0100 0011       43
+0100 0011      +43
 1000 0110       86
```

Notice that the bits are almost exactly the same, just shifted over one bit position to the left. For your convenience the 1802 has a special instruction which does this without requiring the memory location that an ADD needs. Try opcode FE in Program 5.1, giving it various first operands. Notice what happens to DF; do you believe it is exactly the same as if you added the number to itself? If not, try the ADD instruction with the same value as both first and second operand. What does the FE opcode do with the second operand? Convince yourself that this is not an

immediate addressing mode instruction (try changing location 002A to something else like IDL or SEQ. If you change it to SEQ, you will need to look very carefully for a blink or else put NOPs in 002E and 0032.

What happens when you continue to repeat the shift opcode (by pushing "I" repeatedly)? As you probably guessed, there is a version of the shift that "adds" in the previous contents of DF. Try 7E in Program 5.1. How many times do you have to repeat the operation at any particular datum before you get the same value back? Do you understand why? Try thinking of the accumulator and carry (D and DF) as a 9-bit circular register:



SHL SHift D Left FE

SHLC SHift D Left with Carry 7E

Move each bit in the accumulator D one bit position to the left. The leftmost bit of D is moved into the DF register. The rightmost bit of the accumulator is set to zero (SHL) or to the previous contents of DF (SHLC).

As you might suppose in a nice symmetrical machine like the 1802, there is also a shift right to divide by two. Try opcodes F6 and 76 in Program 5.1. Be sure to watch DF (as reflected in Q). Do you understand what is happening? Look at a paper version, shifting hex 43 right:

```
 binary        hex
0100 0011       43
0010 0001       21     (and DF=1)
```

Of course if a shift right with carry were executed next, you would get

```
 binary        hex
1001 0000       90     (DF=1)
```

What is next, if you continue? Try it.

SHR SHift D Right F6

SHRC SHift D Right with Carry 76

Move each bit in the accumulator D one bit position to the right. The rightmost bit of D is moved into DF. The leftmost bit of the accumulator is filled with zero (SHR) or with the previous contents of DF (SHRC).

Notice that if shifting left multiplies by two, shifting right divides by two. The bit shifted into DF can be thought of as a remainder; if a number is even, the remainder is zero when dividing by 2; if odd, the remainder is one. Also notice that dividing (shifting right) large numbers requires that you start with the leftmost byte instead of the rightmost byte as in adding and subtracting. If you recall your grammar school arithmetic, you will remember that long division works the same way.

By now you should have a thorough understanding of the arithmetic and logical instructions of the 1802. You already understand the register operations (from Chapter 4) and the branch and I/O instructions (from Chapter 3). All that remains are the control instructions, which are the rest of those with the left digit of "7x" and those with the left digit "Dx". These we cover in Chapter 6.

Chapter 6 -- Control Instructions

In this chapter you will learn about the six remaining instructions in the 1802 instruction set. So far we have made considerable use of the fact that Reset forces P=0 and R0=0000. However, it is often convenient to use registers other than R0 for the program counter. Chapter 7 deals with one such occasion. In this chapter, therefore, you will learn how to modify the contents of P. You will also learn about some other ways to operate on X.

You only need one program to experiment with the instructions for this chapter, so I hope you do not mind its being rather large. To check your work in loading it, sequence memory. Then after keying in the last byte of Program 6.1 flip the M/P switch on and verify that the next byte contains 4A. If not, you may have dropped a byte or inserted one too many.

```
        ..  PROGRAM 6.1 -- TEST CONTROL INSTRUCTIONS
        ..
0000 90         GHI 0     .. SET UP R1-R4:
0001 B1         PHI 1     .. ALL IN PAGE 00
0002 B2         PHI 2
0003 B3         PHI 3
0004 B4         PHI 4
0005 F842       LDI ONE
0007 A1         PLO 1     .. R1=0042
0008 F846       LDI FOUR
000A A4         PLO 4     .. R4=0046
000B F83E NEW:  LDI TWO   .. (LOOP TO HERE)
000D A2         PLO 2     .. R2=003E
000E F81B       LDI DOIT
```

```
0010 A3         PLO 3     .. R3=001B
0011 E3         SEX 3     .. KEYIN OPCODE:
0012 3F12       BN4 *     .. WAIT FOR IT
0014 6C         INP 4     .. HERE IT IS
0015 64         OUT 4     .. ECHO IT
0016 3716       B4  *     .. WAIT FOR RELEASE
0018 F83A       LDI CON   .. MOVE R3
001A A3         PLO 3
001B C4   DOIT: NOP       .. <-- OPCODE GOES HERE
001C 7B         SEQ       .. Q ON IF FELL THRU
001D A5   DISP: PLO 5     .. SAVE ACCUMULATOR
001E 64         OUT 4     .. DISPLAY M(R(X))
001F C4         NOP
0020 3F20       BN4 *     .. HOLD
0022 F0         LDX       .. BACK UP REGISTER
0023 73         STXD      ..  POINTED TO BY X
0024 7A         REQ
0025 E2         SEX 2     .. FORCE X=2
0026 64         OUT 4     .. DISPLAY M(R(2))
0027 3727       B4  *
0029 64         OUT 4     .. ALSO NEXT BYTE
002A 3F2A       BN4 *
002C 85         GLO 5     .. RECOVER SAVED D
002D 52         STR 2     .. SO TO DISPLAY IT
002E 64         OUT 4
002F 372F       B4  *
0031 E3         SEX 3     .. NOW ALSO R3
0032 64         OUT 4
0033 C4         NOP       .. IF THIS IS SKIP,
0034 300B       BR NEW
0036 7B         SEQ       .. TURN ON Q
0037 300B       BR NEW    .. GO REPEAT
0039 00         IDL
003A 24         ,#24      .. R3 POINTS HERE
003B 33         ,#33      .. (DISTINCTIVE #)
003C FF         ,#FF
003D 22         ,#22
003E EE   TWO:  ,#EE      .. R2 POINTS HERE
003F DD         ,#DD
0040 BB         ,#BB
0041 00         IDL
```

```
0042 F811 ONE:  LDI #11   .. UNIQUE # IN D
0044 301D       BR DISP   .. GO DISPLAY IT
0046 F844 FOUR: LDI #44   .. DITTO
0048 301D       BR DISP
004A
```

When you first put this program in, leave the M/P switch on and run it. This will help you understand it before trying to use it with new data. Also, if you made a keying error (like I did), you might notice it before clobbering the whole program. Be sure you understand how this program works! Let's review it:

First you will see that there are no instructions you have not already met. Also, aside from the fact that R1 and R4 point to them, there seems to be no way for the computer to execute the instructions between 0042 and 0049. R3 is set up to point to the NOP at 001B, so the first INP will replace that with what you key in (because X=3); we have done that before in slightly different ways. The program is set up to display six data bytes (on three pushes of the "I" key) before going back for a repeat. The six things are:

1. The opcode you keyed in.
2. The byte pointed to by the address register pointed to by X. This register is decremented in a tricky way after the output. Convince yourself that it works without destroying memory. Wouldn't it have been nicer to have a "Decrement Register pointed to by X" instruction? Alas, they ran out of opcodes before they ran out of good instructions. In this case, the program assumes that we do not know what is in X, and this is a way to find out (the display will be distinctive, so you can determine which register and which memory location was used).
3. The memory byte pointed to by R2. This may or may not be the same as the previous display. We assume it will be different most of the time, since X=3 before the unknown instruction.
4. The next byte in memory (still under R2).
5. The accumulator containing the results of the unknown instruction. Recall that this is saved in R5, then stored into memory for display here. If the instruction does not change the accumulator, its previous contents will be displayed, which is the "3A" loaded at 0018. Keep that in mind as you run this program.
6. The byte in memory pointed to by R3. This should help us to discover if R3 was altered, or if the memory under it was changed.

It might have been better to also display the low half of R2 and R3, but the program was already getting too big. You may add the extra instructions to do this as an exercise.

The Q register is set by this program immediately after the unknown opcode, so if for any reason that instruction does not proceed to the next, Q will not come on. It is turned off again the next time you push "I". If you were to change the NOP in 0033 to a LSKP, Q would also come on when the last datum is displayed. We will do that a little later.

At locations 0042 and 0046 there are LDI instructions which then branch to the display routine starting at 001D (notice this misses the SEQ instruction). If you see "11" or "44" when the accumulator is displayed, you will know which instructions were executed.

Now, on with the show. After trying it once or twice with M/P on (notice that it says that D="DD" because it could not store the actual contents into memory), try the program with M/P off and key in a NOP (C4) to see the difference. Try "SEX 2" (E2) to see how the second display is changed. Try something to modify the accumulator; say, GLO 0 (80). You might find it worthwhile to try GLO 1 and GLO 4 and remember the results. What happens if you key in "F8"? Do you know why? If not, review the LDI instruction in Chapter 4. Notice what happens if you key in an instruction to increment or decrement R2 or R3. Try INC 4, followed by GLO 4. How about LDXA (72)? What I want you to do is become familiar with this program. You might even try some instructions I have not suggested, but be careful! Some instructions (such as branches) will cause the program to blow up, forcing you to key the program back in; be sure you understand what to expect from each experiment before doing it.

Now let's try some new opcodes. Just to be safe, quickly scan the program (examine mode) to be sure you have not clobbered anything; if so, fix it. Now try the opcode "D1". Did Q come on? What was in the accumulator? Can you guess why the D1 opcode caused the computer to execute the instructions at 0042 instead of the SEQ at 001C? Reset the computer and try the same opcode, but this time enter a GLO 1 (81) opcode after it. Did you notice that Q came on this time? What is in R1? Why is it not 42? Key in GLO 4 (just to see what is in R4) then key in "D4", followed by "84" again. What happened to R4? Now what happens if you repeat "D1" or "D4" again? Why do you suppose the instructions at 0042 or 0046 are no longer executed? Do you understand that since R1 and R4 both point to 001C, they can no longer get to 0042 or 0046?

If I tell you that the "D1" opcode puts a "1" into P (making R1 the Program Counter), does that help you to understand what happened? R0 is still pointing to the next byte after the "D1" opcode, i.e. R0=001C. But R1 is now used to fetch instructions, and the next one loads "11" into the accumulator. It then branches to the display routine (all the time incrementing and modifying R1, because R1 is the PC). The second time through the instruction at 001B, R1 is already the PC, so

nothing new happens, and the next intruction is the SEQ at 001C. R0 still has not changed. You can see that by keying in INC 0. If you do this first, the SEQ instruction will be skipped, because P=0. But after a D1 or D4 you can increment R0 as much as you like, because it is not being used any more (you can see how much it has been incremented with a GLO 0). In the same way you can increment R4 or R1 (whichever is not the PC); just be careful not to set P back to the altered register if you do not know that it points to a valid instruction, or you may wipe out your program.

```
SEP r   SEt P                                                Dr
-------------------------------------------------------------
Copy r (the low four bits of the instruction) into P, making
the designated address register the new Program Counter.
```

Because the SEP instruction leaves the old PC register pointing to what was going to be the next instruction, we have a very convenient way to jump to a subroutine. A **subroutine** is a little program that does some function. A main program normally **calls** a subroutine by jumping to its beginning; the subroutine does its job, and when it is finished it **returns** to where it left off in the main program that called it. Most computers have special instructions (called **Call** instructions) which remember somehow where they came from, so that when the subroutine is ready to return, another special **Return** instruction returns execution to the instruction immediately following the call. In the 1802 the SEP instruction is used for both Call and Return.

To get an idea how this might work, change the BR instruction in 0044 of Program 6.1 to a NOP and a SEP 0 ("D0"). Run the program and key in a SEP 1 (D1). Do you know why the Q came on? What is in P when the display routine was executed? Test your answer by keying in INC 1 and INC 0. Remember, Q will not come on after incrementing the PC.

Now reset and restart the computer, then key in SEP 1 again. Do it a second time. Why does the D register show "44" and not "11"? To be sure that R4 has not been affected, try SEP 4. You see, what happened is that after the LDI #11 instruction, we set P back to 0 (which continued with the SEQ instruction), leaving R1 pointing to location 0046; the next SEP 1 instruction returned to load "44" into the accumulator. In essence, the main program "Called" the subroutine at 0042 by executing the first SEP 1; this subroutine "Returned" to the main program by the SEP 0. Or you can think of the SEP 0 as a Call to the display routine, which returns to the LDI #44 instruction with a SEP 1. There is really much more subtlety here than I can explain in this chapter. It turns out that this capability

makes the 1802 a much more powerful computer than any other eight bit microprocessor.

Let's move on to some other instructions which modify P (and X). Quickly verify that Program 6.1 is still in your computer (i.e. that you have not destroyed part of it by your experiments). Then key in the opcode "70". This instruction does several things; you will want to compare it to a NOP (as an example of an instruction that does nothing), to SEP 4, SEX 2, and IRX (instructions that each do one of the many things that the "70" opcode did). Are you satisfied that you know what happened?

Now try to find a logical basis for all of these things. Since R3 was incremented and X=3, it seems that this instruction operates on the register pointed to by X. Furthermore, the opcode "70" has neither a "2" nor a "4" in it, so we need to look elsewhere to know how X and P got set. But notice: R3 pointed to a byte in memory with both a 2 and a 4. Check out this hypothesis by changing the "24" in 003A to "14"; does the second display now show "F8" (the byte pointed to by R1)? Change 003A to "21"; now does P come out to be 1? Do you think you know what this instruction does?

RET **RETurn** **70**

Read the byte from the memory location pointed to by the register pointed to by X; then increment that register. Copy the right 4 bits of the byte read into P, and the left four bits into X. Then enable interrupts (set IE to 1).

This instruction is intended to be used when the computer is finished servicing an interrupt, and in that sense it "returns from the interrupt" to the program which was interrupted. What I mean by interrupts will be explained more completely in Chapter 7. Notice that this instruction enables them (whatever they are). There is another instruction which disables interrupts. Try opcode "71" in Program 6.1 and convince yourself that you cannot tell the difference between it and the RET instruction.

DIS **Return and DISable interrupts** **71**

Read the byte from the memory location pointed to by the register pointed to by X; then increment that register. Copy the right 4 bits of the byte read into P, and the left four bits into X. Then disable interrupts (set IE to 0).

The only difference between RET and DIS is the Interrupt Enable flag inside the 1802. Fortunately, we have a way of looking at that flag in our program. We can test whether interrupts are enabled by that one skip instruction that we have not yet analysed. Change the NOP in location 0033 to an LSZ (CE) and convince yourself that if and only if the instruction you key into the computer results in a zero accumulator, Q comes on with the sixth display and stays on through the entry of the next opcode. You may test this by keying in GLO 0 (80) for non-zero and GHI 0 (90) for zero results. When you are satisfied that you understand when and how the Q comes on, change that LSZ to a "CC" (in 0033). Now compare the results of Q after the RET and DIS instructions. Do you see that this instruction skips when interrupts are enabled?

```
LSIE    Long Skip if Interrupts are Enabled               CC
-------------------------------------------------------------
If the internal IE flag enabling interrupts is True=1, skip
the next two instruction bytes; if False=0, take the next
instruction in sequence.
```

Exercise: Devise an experiment to see whether interrupts are enabled or disabled (or unaffected) by Reset.

The RET and DIS instructions are convenient any time you want to load both X and P. This is expecially true when the program itself needs to compute the value for one or both of these four-bit registers. For example, you might write a subroutine which will return to any of three different program counters, depending on what was in P when the subroutine was called. Subroutines are most useful when they can be called from (almost) anywhere in the rest of the program. So the program will save its own value for X and P in memory somewhere, then call the subroutine with a SEP instruction. When the subroutine is ready to return, it sets X to the register which points to that memory byte, and executes a DIS (or RET) instruction, taking the program back to the instruction following the SEP, with the appropriate values in X and P.

But how does the calling program know its own X and P? I'm glad you asked. Obviously, since you wrote the program, you know what is in X and P, so you can let the computer know with an LDI instruction. But suppose you wrote a clever program that may have different values in X and P for different times through the program (something like Program 6.1, where P may be 0, 1, or 4, depending on what instructions you previously keyed in). How then can we know what to save? Well, obviously I would not bring up the subject if there were not an instruction to do just that, so try opcode "79" in Program 6.1.

If you tried the "79" and are not now thoroughly confused, either you already understand this new instruction as well as Program 6.1 (and have no need to be reading this book), or you do not understand the problem at all. Do not reset and try another opcode! If you already did you may have clobbered the whole program; go back and verify it. Fix any bytes that are not correct, then run once with the opcode "79". Why does the program no longer respond to the the "I" key? Did the computer stop? Reset it and try IDL (00). How is that different? Reset and try DEC 0 (20); is that like the IDL or the new opcode? So you can assume the computer did not stop. The first display is supposed to be the byte pointed to by the register pointed to by X. What evidence is there that this instruction was really executed? (Hint; look at Q.) So which register points to a "C4"? You already know there is no C4 anywhere near where R2 and R3 point; the same is true for R1 and R4. You might guess that R7 or R9 might be involved but none of the other instructions in the "7x" series have register numbers attached to them. What about R0? Where does R0 point? Be careful, remember R0 is the PC.

Reset the computer and try SEX 0 (E0). Aha! But why did the computer quit responding? Look at the program again, especially around 0022-0023. If X=0 and you decrement the register pointed to by X, are you not also decrementing the PC? Let's get out of this trap by changing the STXD in 0023 to a NOP. Notice that there is an STXD (73) in location 0024 also, not a REQ (7A)! Why? Think about what the STXD in 0023 did (What is in X? Where does R0 point? What is in D? When was D last loaded?). Put the REQ back into 0024. Also put your constants back into locations 0039-0041. Reset and run the program again, first with a NOP so you see how it is different without the STXD. Now try SEX 0. Finally try the "79" again. You already know this instruction sets X to 0 in our test. Does it change the accumulator? What about the register X previously pointed to (R3), or the memory that register points to? What happened to R2? How about the memory it used to point to?

This instruction is so complicated that I am going to give you another experiment to test it. Fix location 003E (or notice what its new content is) and reset then run Program 6.1 again (with a NOP instead of the STXD). This time key in a SEP4 (D4) first, so that you know that P=4 for the second keyin. This is important. Now key in "79" and look at the results. What was displayed first, that is, what evidence is there as to the contents of X? If X=0. what would you have seen? Remembering that P is still 4, key in SEX 0 for the third entry. Are you convinced that this time the "79" did not set X=0? If not 0, then what did it set X to? Where did the "C4" come from? Which register points to a NOP when the OUT 4 instruction in 001E is executed? Would you believe that this instruction copies P into X? Now, remembering that X is set to 3 before the keyed-in opcode is

executed, why do you suppose neither R3 nor the memory under R3 is affected by this instruction, but rather R2 and the memory it points to?

I'll tell you. There are three address registers with special (hardware) significance in the 1802: R0, as you know, is forced to 0000 and becomes the program counter at Reset; it is also used for DMA as you will see in Chapter 7. R1 is used for interrupts, as you will also see in Chapter 7. R2 is also significant in the servicing of interrupts (yes, Chapter 7 again!), but this instruction, and only this instruction, uses R2 and the memory pointed to by R2 regardless of the contents of X and P. As you can see, something was stored into the memory location pointed to by R2, then R2 was decremented. What was stored? Remember the first time you tried this instruction, "30" was stored; the second time it stored "34". The first time P=0 and the second time P=4, so it is a good guess that the low four bits are a copy of P. What about the high four bits? Recall that the RET and DIS instructions expect to see a value for P in the low four bits of the memory byte pointed to by X and a value for X in the high four bits. Thus it is reasonable to assume that the "79" instruction copies X into the high four bits of the byte it stores, and in fact, X was 3 in both cases.

```
MARK    Save X and P in T                                   79
------------------------------------------------------------
Copy X and P into the T register, store T in the memory byte
pointed to by R2, then decrement R2; finally copy P into X.
```

What's this about a "T register"? The only function of the T register in the 1802 is to receive a copy of X and P. The MARK instruction does this, and an interrupt also does it (these pop up all over this chapter, don't they?). As you will see in the next chapter, we need some way to save and restore what the computer is doing in the case of an interrupt; the T register helps us do this. But of course you have to save the contents of T, so you can put X and P back at the end of the interrupt. The "78" instruction does this. Try it, but first you might want to put the STXD back into location 0023 and fix up the constants in 003A-0040. Notice that now when you run the program and key in 78, the first display (which is the contents of the memory byte pointed to by the register pointed to by X) shows "34" (assuming your last experiment was the MARK instruction after the SEP 4). This is the same as that saved in T by the previous MARK. Notice that T is not affected by Reset. R2 is not involved in this instruction, and D is not changed. If you wanted, you could execute another MARK (don't forget to change the STXD to a NOP!) to put a different value into T, which a subsequent "78" will then reflect.

```
SAV     SAVe T                                          78
-------------------------------------------------------------
Copy the contents of the T register into the memory location
pointed to by the address register pointed to by X.
```

Now you have learned all the instructions your computer is capable of executing. If you look carefully, you will notice that we never studied the opcode "68". That's because it is not a defined 1802 instruction. It has the form of an INP instruction, but 0 is not a defined input port, so if you execute it (try it!) nothing is input. "Nothing" is the answer to a question; it is data, and something will be put in the accumulator and memory (so now you know what the computer uses to mean "nothing").

However, since the result of the "68" opcode is unpredictable, it should not be used in your programs. In fact, "68" is the first byte of a series of additional instructions for the 1804 and 1805 microprocessors.

In case you had not noticed, the "60" opcode (IRX) is in the form of an OUT instruction, but again, since there is no port 0, the output data goes nowhere. However all other functions of the OUT instruction (i.e. incrementing the register pointed to by X) are still performed. So it turned out to be a useful instruction after all. Clever, these Yankees.

In Chapter 7 we will study some of the special hardware features of the 1802 which enable it to do certain kinds of input and output. Be sure you understand all of the instructions covered in Chapters 3-6 so that you will be able to follow the programs presented in Chapter 7.

Chapter 7 -- Interrupts and DMA

Now that you understand all of the 1802 instructions (go back and review them if you have any doubts), you are ready to use them in a real program. You are going to build this program in pieces, and so learn about some of the particular hardware features of the ELF II one step at a time. In the process you should come to understand how to program for interrupts and DMA, and how to watch for timing problems. Incidentally, you will also learn a little about writing diagnostics for your computer (more about what those are later). This chapter is not as detailed about the individual programs as Chapters 3-6, so be prepared to do more of your own thinking.

First let's talk about **DMA**, or **Direct Memory Access**. All of the I/O instructions transfer data into or out of the computer at the time the program commands it; that is, when the I/O instruction is executed. DMA runs independently of the program

(to a certain extent), and the data is transferred into or out of the computer when the outside world (as represented by the other circuits in the computer) decide to do it, and the program in the computer may not even be aware it is happening. This is because the program counter, accumulator, and the register pointed to by X are not involved in the DMA transfer. All that happens is that the hardware "steals a cycle" from the CPU to move a byte into or out of memory directly (thus the name "Direct Memory Access"). The designers of the 1802 made this particularly easy by setting up R0 to point to the memory location that the DMA will access. To an extent this is not true DMA but rather a "Data Channel"; but DMA is easier to pronounce and spell, so that's what we will call it.

Sequence memory, then enter Program 7.1:

```
        ..  PROGRAM 7.1 -- TEST DMA
        ..
0000 90   REST: GHI 0     .. INITIALIZE Rl, R2, R3
0001 B1         PHI 1
0002 B2         PHI 2
0003 B3         PHI 3
0004 F81B       LDI INTS  .. R1 = INTERRUPT PC
0006 A1         PLO 1
0007 F8FF       LDI #FF   .. R2 = STACK (TEMP DATA)
0009 A2         PLO 2
000A F80F       LDI MAIN  .. R3 = MAIN PC
000C A3         PLO 3
000D 71         DIS       .. X=0, R0=000E
000E 23         #23       .. SET X=2, P=3
000F 69   MAIN: INP 1     .. TURN ON TV
0010 90   LOOP: GHI 0     .. DISPLAY R0
0011 22         DEC 2
0012 52         STR 2     .. FROM MEM AT R2
0013 64         OUT 4     .. X=2!
0014 3010       BR  LOOP  .. DO NOTHING
```

To run this program, be sure your TV set is correctly connected to the ELF II video output. With the computer reset you should see a blank screen. As soon as you switch it into Run mode, the TV should start flickering at a rate of about once per second. If you look carefully, you should see a regular pattern flash by each time. What is happening is that the 1861 Video display chip in your computer is asking for bytes out of memory, and whatever R0 points to is transferred out and displayed on your TV set. R0 is incremented each time, so the video display slowly walks through memory. This program also outputs the high byte of R0 on the hex display, so you see it incrementing at the same rate the TV flickers.

If you change the INP 1 instruction in location 000F to a NOP and then run the program, the TV display will remain off and R0 will stay at 000F. The hex display shows 00 (the high byte). To see the low byte, change the GHI 0 in 0010 to GLO 0. Now change the NOP back to the INP 1 that was originally in 000F. What do you think you will see when you run the program? Obviously the TV should be no different. If you output all the hex digits in rapid succession, the hex display should look like "88", right? Run it.

Why do you see "8A"? Notice that the right vertical bars of the "A" are dimmer than the others. Flip it into Wait a few times. Notice that the right digit is most often "F" or "7". You may see other values like 80, 22, etc. but you can tell that these are glitches in the display caused by switching into Wait, because they have the lower bar on in the right digit, whereas that bar is always off when the program is running. It happens that these are the opcodes of your program at the point you stopped it. If you display "F" and "7" in rapid succession, it looks like an "A". If the display is more often "F", then that part will be brighter. So, we can conclude that R0 is being incremented by 8. Its initial value is 000F (as you saw when you NOPed the INP instruction), so the program displays 0F, 17, 1F, 27, etc. Assuming that R0 is 000F when the 1861 begins its scan, here are the values in R0 that correspond to each position on the TV screen (though what actually appears on the TV screen are the contents of the memory locations pointed to by R0):

```
          R0 gets incremented 8 times per line, so the
          contents of 8 memory locations are displayed
          ____________________________________________
         /                                            \
1st      |                                              | blank space
byte_____|___                                           | (1802 running)
         |   \                                          |
line 1   |   000F 0010 0011 0012 0013 0014 0015 0016    |
line 2   |   0017 0018 0019 001A 001B 001C 001D 001E    |
line 3   |   001F 0020 0021 0022 0023 0024 0022 0026    | DMA Active
...      |   XXXX XXXX XXXX XXXX XXXX XXXX XXXX XXXX    | (1861 running)
line 127 |   03FF 0400 0401 0402 0403 0404 0405 0406    |
line 128 |   0407 0408 0409 040A 040B 040C 040D 040E    |
         |                                       /      |
1024th___|_____________________________________/        | blank space
byte     |                                              | (1802 running)
          \____________________________________________/
               ^-- the address in R0 ends in "F" or
                   "7" at the start of each line
```

A computer's-eye view of your TV screen

On your TV set, only a small portion of the screen is used by the 1861. This represents about half of the available time for the TV display, so the rest of the time the 1861 is not asking the computer for data and R0 is not being incremented. This happens after 1024 bytes have been output, so most of the time R0 is at 000F, 040F, 080F, 0C0F or some other multiple of 400 hex (1024 decimal) plus "F". That is why "F" is brighter.

Well, this is not very interesting. However, we can also control R0 in the program. Enter the following code on top of Program 7.1 (i.e. use this code to replace the bytes at 0010-0014):

```
0010 F880 LOOP: LDI #80   .. FIX R0
0012 A0         PLO 0
0013 3010       BR  LOOP  .. CONTINUOUSLY
```

Now when you run the program, you will see a nice clean set of vertical bars of various widths across your TV screen. Do you know what they are? Try different values in location 0011, and see if you can find the relationship between what is in the memory location R0 points to and what appears on the TV screen.

Each horizontal line displayed on the TV screen is eight bytes of eight bits, or 64 bits wide. The leftmost byte is M(R0), i.e. the contents of the memory location pointed to by R0. Next comes M(R0+1), then M(R0+2) and so on until M(R0+7) at the right end of the line. The most significant bit in each byte (bit 7) is displayed on the left, and the least significant (bit 0) on the right. A "1" in a bit is displayed as a white spot, and a "0" as a black spot. Like this:

```
byte: M(R0)    M(R0+1)  M(R0+2)  M(R0+3)  M(R0+4)  M(R0+5)  M(R0+6)  M(R0+7)
bits: 76543210 76543210 76543210 76543210 76543210 76543210 76543210 76543210
```

Let's be sure you understand what this program is doing. First it sets up R1, R2 and R3. We use R2 to point to temporary storage in memory. Don't worry about R1 yet; I only included it to save work later when you will expand Program 7.1 into Program 7.6. R3 will be the PC. How does it get that way? (Hint: look at the comments in the program). If you still do not understand what the 23 in location 000E is doing, go back to Chapter 6 and review the DIS instruction.

After executing the DIS instruction, R0 contains 000F (you saw that on the display) and the next instruction does an input from "Port 1". This happens to be recognized by the 1861 to mean "Turn on the TV display" so it starts up, requesting bytes from the computer and displaying them. The TV screen consists of a whole bunch of horizontal lines (you can see them if you look closely). The 1861 uses 128 of them in the middle of the **raster** (which my dictionary defines

as "a pattern of scanning lines covering the area upon which the image is projected"). For each line, the 1861 asks for eight bytes of memory. While the computer is pumping these eight bytes out to the 1861 it has no time for executing the program (all the cycles have been "stolen"), but during the black spaces at the left and right ends of each line there is time for exactly three 2-cycle instructions. Finally, after the 1024th byte on the 128th line, the 1861 goes to sleep (the black space at the top and bottom of the raster) and the computer is able to do some 941 more 2-cycle instructions before the 1861 starts asking for bytes again.

When we modified the program to force R0 to be some particular value, then every time the 1861 asked for a block of eight bytes it got the same block, so every line was the same. Instead of 1024 bytes, we used just 8 bytes for the entire screen. We can take advantage of this trick to save memory, so instead of needing 1024 bytes to hold the picture being displayed we can make do with only 48. The result will be that instead of being only one raster line high, each picture element (also called a **pixel**) will be 25 lines high. That is, we will display the first 8-byte line 25 times, then the second 8-byte line 25 times, etc. so that by the time we reach the 128th line, we have only used 48 bytes.

The next program is added on top of Program 7.1. In other words, if Program 7.1 is still in memory, then just step over it (in Examine mode) to location 000F and then start entering Program 7.2. If not, key in the first fifteen bytes of Program 7.1, and then begin entering Program 7.2. Notice that three bytes are missing at 0021-0023. Don't worry about putting anything there. (Why?)

```
        ..  PROGRAM 7.2 -- TV DISPLAY DRIVER
        ..
000F 69   IN:   INP 1     .. TURN ON TV
0010 3410       B1  *     .. SYNCHRONIZE EF1
0012 F814 SYNC: LDI #14   .. =20 DECIMAL
0014 3C14       BN1 *     .. WAIT FOR NEW EF1
0016 FF02       SMI 02    .. MEASURE SIZE
0018 3416       B1  *-2   .. 12 OR 28 INSTRUCTIONS
001A FE         SHL       .. 20-28 IS NEGATIVE;
001B 3312       BDF SYNC  .. TOO LONG=NO DMA
001D 93         GHI 3     .. FIX R0
001E B0         PHI 0
001F 3024       BR  DONT
0021    ..
0022    ..  DISPLAY REFRESH
0023    ..
0024 3C24 DONT: BN1 *
```

```
0026 F8C8       LDI BUFF  .. SET UP R0
0028 3428       B1  *     .. WAIT FOR DISPLAY
002A A0   ROW:  PLO 0
002B            ...       .. DMA HERE
002B A0         PLO 0     .. RESET R0
002C B7         PHI 7
002D F80B       LDI 11    .. =(RASTER COUNT - 3)/2
002F            ...
002F A7         PLO 7     .. R7 LOW IS COUNTER
0030 97         GHI 7     .. KEEP FIXING R0
0031 A0         PLO 0
0032            ...
0032 27   REPT: DEC 7     .. COUNT RASTERS
0033 97         GHI 7     .. KEEP FIXING R0
0034 A0         PLO 0
0035            ...
0035 A0         PLO 0
0036 87         GLO 7     .. CHECK THE COUNT:
0037 3A32       BNZ REPT  .. TWO RASTERS PER LOOP
0039            ...
0039 80         GLO 0     .. IF LAST TIME,
003A 3C2A       BN1 ROW   .. (NO, DO IT 6 TIMES)
003C A0         PLO 0     .. JUST BLANK IT
003D            ...
003D 343C       B1  *-1   .. (3 RASTERS)
003F 3024       BR  DONT  .. NEXT FRAME
```

This program is a little tricky. First, you need to know that the 1861 sets EF1 during the last four lines of the display on the TV, and also for an equal time just before the first line of the display. So this program uses the B1 and BN1 instructions to keep track of where the 1861 thinks it is. Unfortunately, the "first 4 lines" and "last 4 lines" flags look the same to the 1802, so it is possible that the program might think the TV is in the display part of its cycle when it is actually in the vertical blanking (the black at the top and bottom), and vice versa. The display will not work right if this is so, so we cannot let it happen.

Since the EF1 flag is true during the last four lines of the display, that means that on that occasion the computer can execute only 12 instructions (three for each line). On the other hand, before the display starts, the same amount of time is good for 28 instructions (7 each, for four lines). So we (that is, the computer, at our command) will count the number of instructions that can be executed while EF1 is true. We start the counter at 20 (decimal) in location 0012. Then each time we look and see EF1 true we subtract two, for the two instructions in the loop.

When EF1 goes false, we look to see if the counter in D went negative. If so, we go back and try again to get the other end of the display. To convince yourself that this works you might try the program a few times after putting a Branch Never (hex 38) in place of the BDF at 001B.

The main part of this program waits for the signal from EF1 telling that the display is about to start, then loads R0 with the address 00C8. The low byte of R0 is also saved in the high byte of R7. R7 also contains, in its low byte, a counter of the number of lines left to be repeated. The program continually copies the same address into R0, counting the number of times; when the count reaches zero, we let R0 advance to the next block of eight and repeat. After five of these, EF1 comes on again, so we hold the last line until EF1 goes away, then go back to the beginning.

Each place in the program with three dots ("...") marks where the 1861 is going to ask for a block of eight bytes. By carefully counting the number of instructions in the loops you can be sure that the program and the hardware play together in synchronism. We have to be sure that R0 gets reset (backed up by 8) once for each block except the last line in a group: You will notice a PLO 0 between every pair of dots, but after the last in the group is also a GLO 0 (so you put back what was already there). The timing is tricky; reading the 1861 data sheet may help you to understand what is required.

In between these PLOs are the instructions that set up the counter in R7, and that count the lines. You should have no trouble understanding this part of the program. We are breaking the 128 lines into five blocks of 25 lines each. The few instructions we have between the first three lines are used in putting an 11 in the counter, then every two lines following them we count down by one. This accounts for 125 of the 128 lines, so we know that we are in the last four (i.e. EF1 is on) when the counter hits zero.

When you run this program, you will see several tall blocks or vertical bars with notches. You can put blocks of black and white (like, say, a checkerboard pattern) on the display by storing patterns of "FF" and "00" in memory between 00C8 and 00F7. Try it to get a feel for what the next program needs to do.

As you can see, this is still not very interesting. Key in Program 7.3 on top of Program 7.2. Notice that it replaces the last branch in 003F, and continues from there.

```
        ..  PROGRAM 7.3 -- SECONDS CLOCK
        ..
003F 90         GHI 0
0040 B7         PHI 7
```

```
0041 F8C7       LDI FRCT  .. POINT TO FRAME COUNT
0043 A7         PLO 7     .. R7 IS AVAILABLE
0044 07         LDN 7
0045 FC01       ADI 1     .. BUMP COUNTER
0047 57         STR 7
0048 FF3D       SMI 61    .. MOD 61
004A 3B24       BNF DONT  .. NOT OVER
004C E7         SEX 7
004D 73         STXD      .. ROLL OVER
004E F0         LDX       .. TO SECONDS
004F FC03       ADI 3
0051 57         STR 7
0052 3B69       BNF UNIT  .. GO DISPLAY
0054 F8E2       LDI -30   .. ROLL OVER
0056 73         STXD
0057 F0         LDX       .. TO TENS
0058 FC03       ADI 3
005A 57         STR 7
005B FC0C       ADI 12    .. (OVERFLOW AT 60)
005D 3B62       BNF TENS
005F F8E2       LDI -30   .. ONE MINUTE!
0061 57         STR 7
0062            ...       .. COULD DO MINUTES, HOURS...
0062 F8C8 TENS: LDI BUFF  .. POINT TO LEFT DIGIT
0064 306B       BR UNIT+2
0066 F8C6       LDI SECS  .. (POINT TO COUNTER)
0068 A7         PLO 7
0069 F8CC UNIT: LDI BRIT  .. OR RIGHT DIGIT
006B A0         PLO 0     .. R0 POINTS INTO BUFFER
006C 47         LDA 7     .. POINT TO DIGITS
006D FCAC       ADI TABL  .. (TABLE OFFSET)
006F A7         PLO 7
0070 47   DOWN: LDA 7     .. GET DOTS
0071 52         STR 2     .. (SAVE)
0072 E2         SEX 2
0073 F0   HALF: LDX       .. CONVERT A DOT
0074 FE         SHL       .. FROM A BIT
0075 52         STR 2
0076 75         SDB       .. =00 IF DF=1, =FF IF DF=0
0077 50         STR 0     .. STORE INTO BUFFER
0078 10         INC 0
```

```
0079 80         GLO 0
007A FA03       ANI 3     .. DO THIS 4 TIMES
007C 3A73       BNZ HALF  .. (9*4 INSTRUCTIONS)
007E 10         INC 0     .. SKIP OVER THE OTHER SIDE
007F 10         INC 0     .. OF THE SCREEN
0080 10         INC 0     .. TO THE NEXT LINE
0081 10         INC 0
0082 F0         LDX       .. CHECK FOR SECOND 4 BITS
0083 3A73       BNZ HALF  .. ((36+6)*2)
0085 80         GLO 0     .. REPEAT IF THIS WAS LEFT
0086 FFF8       SMI BEND  .. CHECK AGAINST BUFFER END
0088 3B70       BNF DOWN  .. ((84+6)*3)
008A 3266       BZ UNIT-3 .. ((270+9)*2)
008C 3024       BR DONT   .. MAX TOTAL <600 INSTRUCTIONS
008E    ..
008E    ..  DOT TABLE FOR DIGITS
008E    ..
008E DAAADF     #DAAADF   .. 0
0091 D9DD8F     #D9DD8F   .. 1
0094 9EDB8F     #9EDB8F   .. 2
0097 9EDE9F     #9EDE9F   .. 3
009A EAA8EF     #EAA8EF   .. 4
009D 8B9E9F     #8B9E9F   .. 5
00A0 CB9ADF     #CB9ADF   .. 6
00A3 8EDBBF     #8EDBBF   .. 7
00A6 DADADF     #DADADF   .. 8
00A9 DACEDF     #DACEDF   .. 9
00AC    TABL=*
00C5    ..
00C5    ..  TIME COUNTERS AND DISPLAY BUFFER
00C5    ..
00C5 E2   STEN: #E2       .. MUST INITIALIZE
00C6 E2   SECS: #E2
00C7 00   FRCT: 00
00C8      BUFF=#C8        .. EMPTY BUFFER SPACE
00CC      BRIT=#CC
00F8      BEND=#F8
```

Notice that there is a large gap between the "Dot Table" and the last three bytes. Do not neglect those last three bytes at 00C5-00C7, or the program will blow up. Can you guess what this program will do before you run it? (Hint: look at the

comments!) Try it, then after you get it started and can get unglued from watching it, we will take a walk through the code.

This program is quite complicated and I will not try to explain every detail to you. But if you can figure it out, you have gone a long way toward understanding the 1802! The important thing to remember is that a TV shows 60 pictures every second. These are normally arranged in an "interlaced" fashion to give more detail, but the 1861 does not bother with that. It also does not do it in exactly 1/60th of a second. Anyway, after each display frame the program increments a counter (in location 00C7). When the counter goes over 61 it is knocked back down to 0 (61 turns out to be slightly more accurate than 60). This happens almost exactly once every second, so the program builds a "digital clock" around that counter!

When I program a counter, I can count from any number to any other number that will fit in the memory or register(s) I am using. In the display driver we counted down from 11; the frame counter counts up from zero. When counting seconds we count up from -30, but count by threes. That way, after ten seconds the counter will hit zero, which is an easy condition to test in the program, and we can set it back to -30. I chose to count up by threes because it takes exactly three bytes to specify the dots for one digit on the display (four dots wide, six dots tall, total 24 dots = 24 bits = 3 bytes). Thus at each second I can take the actual counter, add to it the address of the dot table, then use the sum as an **index** into the table (an Index is a number or a register which points to a particular entry in a table of values) to fetch the dots for that digit. The minutes works the same way, except that if it reaches -12 (by which we mean "6") we bump it back down.

Each second (except the tenth) we set up some pointers, one to point to the rightmost four bytes of the top line of the display buffer, and one to point into the dot table. There are four loops nested together in this program. The inner loop, from 0073 to 007D, shifts one bit out of the byte fetched from the dot table, then stores a whole byte of "FF" or "00" in the buffer depending on whether that bit was zero or one, respectively (note: 0 stores FF, 1 stores 00). This happens four times, or until the address in R0 is divisible by four (i.e. the right two bits are zero).

Then R0 is incremented four times and the process repeated once. This is the next level of loop, and the program recognizes the end of this loop by the fact that all the bits were shifted out of the dots byte.

The third level of loop goes back for the other two bytes in the dots table. The program can tell it is at the end of that loop if the buffer pointer (R0) is past the end of the buffer.

The outside loop takes care of the fact that we use the same part of the program to set up the tens digit as the units, only that the buffer pointer is pointing to the leftmost four bytes of each line. On every tenth second, after resetting the units to zero and incrementing the tens, the program sets up the tens digit first. Then when it finishes, if R0 is pointing exactly at the end of the buffer (i.e. the first byte past it, 00F8), the program knows it just did the tens, so it goes back for the units. When it finishes the units R0 will point to 00FC (why?), so the display formatting is finished. Take a few minutes to think through each instruction. Write down on paper what should be in the registers (and memory) at each point. In other words, pretend you are the computer and "execute" the program. It is good practice.

Did you know your computer could do two things at once? Well, not at exactly the same instant, but it can switch back and forth fast enough so it seems to be doing both at the same time. Notice the numbers in parentheses in the comments of Program 7.3. These refer to how many instructions it takes to do that part of the loop. The outside loop is a little under 600 instructions once every ten seconds, about 300 instructions the other nine seconds, and is not executed at all in between. That means there over 50,000 instructions just going to waste every second! If you have nothing to use them for you have lost nothing, but sometimes you can do things in between. If you do, however, you must be careful to get back to the display routine in time for the next frame or the picture will be garbled. You could do this by continually testing EF1, but that is a real nuisance since you have to test it every ten or so instructions. Or you can let the 1861 **interrupt** you when it needs attention.

Suppose you are entertaining guests in your home and your two-year-old daughter says "I gotta go potty!" You drop everything and tend to her needs. Then, you come back and resume your conversation where you left off. (If you do not have a two-year-old daughter, I'm sure you can imagine what it is like). You have been interrupted. Actually, when you get back the conversation may have gone on without you; but when the computer is interrupted everything stops and waits for the CPU to come back and resume where it left off. If we program it carefully, the program that was running need not even know it was interrupted. An **interrupt** then, is some external event (like the 1861 getting ready to start its DMA) which triggers the computer to set aside what it is doing so it can tend to the device which caused the interrupt, then return at a later time and resume what it was doing.

Key in Program 7.4 on top of Program 7.3 (and 7.2 and 7.1). Notice that it is in two parts. You do not have memory sequenced to help you any more, so you will have to follow the program listing carefully while skipping over the parts already entered.

```
        ..  PROGRAM 7.4 -- TEST INTERRUPTS
        ..
000F C4   MAIN: NOP       .. DON'T TURN ON TV
0010 30AC       BR TEST   .. DO TIME TEST
0012    ..
0012    ..  DISPLAY REFRESH
0012    ..
0012 12   DONE: INC 2     .. X=2!
0013 42         LDA 2     .. RESTORE DF
0014 F6         SHR
0015 42         LDA 2     .. RESTORE R7
0016 B7         PHI 7
0017 42         LDA 2
0018 A7         PLO 7
0019 42         LDA 2     .. NOW D
001A 70         RET       .. RESTORE X AND P
001B C4   INTS: NOP       .. EVEN OUT CYCLES
001C 22         DEC 2     .. PUSH STACK, TO
001D 78         SAV       .. SAVE X AND P (IN T)
001E 22         DEC 2
001F 73         STXD      .. SAVE D
0020 87         GLO 7     .. SAVE R7
0021 73         STXD
0022 97         GHI 7
0023 73         STXD
0024 7E         SHLC      .. SAVE DF
0025 73         STXD
00AC    ..
00AC    ..  FLASH Q FOR TIME TEST
00AC    ..
00AC 17   TEST: INC 7     .. USE R7
00AD 97         GHI 7     .. AS COUNTER
00AE 7A         REQ       .. RESET Q, THEN
00AF 3AAC       BNZ TEST
00B1 7B         SEQ       .. IF ZERO, TURN ON Q
00B2 30AC       BR TEST
```

Notice that this program is approximately the same as Program 2.3 -- that is, it blinks the Q light once every couple seconds. Count how many you get in one minute and make a note of it. Now change the NOP in 000F back to an INP 1 instruction, so the DMA will run. Run the program again and count how many blinks in a minute. Why do you suppose you got fewer? You see, when the CPU

takes time out to do DMA, it cannot be running its program. Cycles are being stolen.

Now you need to make three patches. The first is to change the DIS instruction in 000D to a RET (70); this will enable interrupts. The other two are the two branch instructions which branch to 0024; they must be changed to branch to 0012. In other words, at 004A there should be "3B12" and at 008C there should be "3012". When I was testing this I forgot to make those changes and wiped out half the program.

Now run it. Notice that you now get the clock back on the TV, and the Q continues to blink, although somewhat slower. Your computer is doing two things at once! You might want to time the Q blinks again, to see how much time out of the blink routine the clock routine is stealing.

Before we go any further, you should take a close look at the program your computer is running. In other words, write out on a piece of paper what is left of Programs 7.1, 7.2, and 7.3, and where Program 7.4 fits in. If it is any help, you might want to examine memory for reference. Program 7.6 at the end of this chapter is almost the same as what you have in memory (except for 00AC-00C4), and it is all in one place; you can use it for reference. I want you to particularly notice how the computer gets into the display driver and clock routine. Perhaps it would help here to draw a flowchart.

Let's follow the program sequence. The program starts with R0 as PC. It puts addresses in R1 and R3 (and R2, as you see, is just used to point to temporary data). The RET instruction in 000D sets P=3 so that now R3 is the PC; this frees up R0 for DMA. R3 points to the INP instruction in 000F, then the program branches around the display driver, to 00AC. There the computer is stuck in a tight loop, setting and resetting Q. Nowhere do we see a branch into the display driver, except those already within that part of the program. R1 points to 001B, but there is no SEP 1 instruction in the program anywhere. There is no way this program can, by executing instructions, get into the section from 0012 to 008D, and that's a fact. But it is getting there, because you know that this is the only part of the program that counts seconds on the TV screen!

It gets there without executing instructions. This is the essence of the interrupt. The Q blinking program is paying no attention to the 1861 or the TV or EF1 at all. It just sits there and counts and blinks. Bye and bye, the 1861 decides it is time to put a new picture on the TV screen, and it pulls down on the 1802's INTERRUPT pin. The 1802 finishes whatever instruction it was executing, copies X and P into that mysterious T register, and forces X=2 and P=1 just as if a DIS instruction had

been executed with X pointing to a register pointing to a byte with hex 21 in it. But there is no such instruction or byte. It happens kind of like magic.

Now all of a sudden P=1, R1 is the PC, and the next instruction is in 001B. No matter that the last instruction was in 00AE and that the next instruction was supposed to be the BNZ in 00AF (maybe; the last instruction could equally well have been the GHI or any other instruction in that loop). There are exactly 14.5 instruction times before the first line of DMA comes crashing through, and the program had better be ready. Remember, the computer has no way of knowing where it came from, except for that saved X and P in T, so first off we want to save it in RAM; later we will be able to do a RET referencing that byte to return control to where we came from. We do not know if that other program was using R2 to point to temporary data (this one was not, but generally that is not the case), so we decrement R2 first. Notice that the SAV instruction uses the register pointed to by X, but the interrupt set X=2. The display driver uses D and so does the blinking routine. If the blinker was interrupted between the GHI and the BNZ (as we supposed), then if the display routine altered D without saving and restoring it, the blinker would malfunction. The STXD in 001F saves D.

Perhaps you noticed that both the blinker and the display routine use R7. Why, you might ask, when there are so many other registers? True; I could have used separate registers, but then you would not have needed to save and restore a register. When you write larger programs, you will run out of registers and this will become a necessity. Finally, the clock routine alters DF, so we shift it into the accumulator in 0024, then save the result.

Notice that all of these bytes are saved by pushing them onto a **stack**, which is an area in memory for saving data. The **Stack Pointer** points either to the next available (unused) byte or to the last stored byte; all the bytes above the "top" of the stack contain data, and all below it are available. You "push" data onto the stack by decrementing the stack pointer and storing; you "pop" data off (i.e. recover it) by incrementing and loading. In our case, R2 is the stack pointer. There is a lot more that could be said about stacks, but you will get the idea by looking at the program.

When the display is finished, or after formatting new digits in the display buffer, the program has to get back to the blinker program. It does this by branching to 0012. All the data we saved we now recover, in the reverse order. DF is retrieved by shifting the saved byte right, R7 is reloaded from the stack, D is retrieved, and finally the RET instruction reloads X and P from the byte saved, and enables the interrupts (which were disabled when the interrupt came). At this point in time X is still =2. RET loads X and P, then increments R2. Since R2 was decremented before saving X and P, R2 is now in the same condition it was before the interrupt.

Now P=3 (because it was reloaded) so the next instruction is (you guessed it!) the BNZ (or whatever was going to have been executed before the interrupt). R1 now points to the entry point 001B, just where we found it. R7, D, and DF have also been carefully put back where they were. The system resumes running the blinker program, and is ready for the next interrupt.

What happens if you do not save everything and put it all back? I'm glad you asked! Key in Program 7.5 on top of everything else, and we shall see.

```
        ..  PROGRAM 7.5 -- MINI DIAGNOSTIC
        ..
00AC E7   TEST: SEX 7     .. SET UP R7
00AD A7         PLO 7     .. AS COUNTER
00AE 3BB1       BNF PLUS
00B0 17         INC 7
00B1 60   PLUS: IRX
00B2 7C01       ADCI 1    .. IN PARALLEL WITH D
00B4 3AB1       BNZ PLUS
00B6 E2         SEX 2     .. WHEN D=00,
00B7 87         GLO 7     .. COMPARE THEM:
00B8 22         DEC 2
00B9 52         STR 2
00BA 64         OUT 4     .. DISPLAY DIFFERENCE
00BB 32BF STOP: BZ  *+4   .. EQUAL CONTINUES
00BD 3FBD       BN4 *     .. UNEQUAL WAITS
00BF 37BF       B4  *     .. WAIT FOR RELEASE
00C1 6C         INP 4     .. GET NEW COUNT
00C2 FE         SHL       .. SET DF
00C3 30AC       BR TEST   .. REPEAT
```

As usual, try to understand what this program does before running it. There are several ways to count in the 1802, and this program starts with the same number (in D) and counts in two different ways, then compares the results. One way of counting is to increment a register; another is to add one to D. In this case, when D reaches zero, if R7 is not also zero, then something went wrong. The purpose of the program is to make it as easy as possible for something to go wrong, without actually forcing it. So in the loop we will increment R7 with the IRX instruction, and we will add one to D with the ADCI instruction. Since the initial contents of DF affect the operation of ADCI we also make a corresponding correction to the initial value of R7. At the end, the program outputs the difference (i.e. the low byte of R7), then waits for the "I" key if that was not zero. In any case a new value is input from the keyboard to become the new carry and accumulator values.

If you keyed in this program correctly, when you run it, it should display hex "00", and count time on the TV as before. You can push various keys, and while that will affect the program, you will not see it.

Now change the LDA 2 in 0019 to an INC 2. What have you done? What will our new program do if D is not the same after an interrupt as it was before? Try it. It immediately displays some non-zero value and waits for you to push the "I" key. Try entering different numbers. Does that make any difference? How about hex FF? Notice that with hex FF it may go several seconds before erroring off.

There are four LDA 2 instructions in the interrupt exit part of the program (0012-001A). When we changed one of them to an INC 2 it meant that the D register was no longer being restored at the end of the interrupt. Put that one back and change one of the others. Notice the various failures. I hope you are getting a feel for the kinds of errors that an improper interrupt service routine can cause. You should be especially aware that these kinds of errors are very intermittent: they only show up if the unrestored register is in the process of being used when the interrupt comes. You should notice that I carefully arranged that your testing did not affect R2. If the stack pointer does not get restored correctly, not only does the main program have errors, but everything in memory is likely to get wiped out! We call this condition a **stack fault**. You might try it if you feel like re-keying your program in. Notice that losing an Increment may be different than losing a Decrement.

The faults that you have been simulating resemble those that also show up when there is something wrong with the hardware. This does not occur very often, but if it does, it is nice to be able to devise a program which will isolate the fault. We call such a program a **diagnostic** because it helps us to "diagnose" the problem.

There is another characteristic of these 1861 programs that I have not yet discussed; timing. Why, for example, is there a NOP in 001B and nowhere else? The reason is that all of the instructions whose left hex digit is C take half again as long to execute as all other instructions. Specifically, all long branches, long skips, and NOP take three cycles, and all other instructions take only two cycles. The interrupt, on the other hand, takes only one cycle. The timing for the 1861 is somewhat critical; even a one cycle error causes a line of displayed data to shift or jitter to one side. So we balance out the 1-cycle interrupt with a 3-cycle NOP. You can see the consequences of a timing error for yourself by changing that NOP to something else like SEX 2 (X=2 already, so this is amounts to a two-cycle NOP instead of three). Do you see how your picture is skewed? Put the NOP back and try changing the BZ instruction in 00BB to a SEX 2 followed by LSZ. See what that does to your picture? (Try pushing various keys). When using the 1861, you

should avoid all 3-cycle instructions except for those at certain points in the interrupt service routine.

The program you now have in memory should be the same as Program 7.6.

EXERCISES

1. Program 7.6 counts up. There are several ways you might modify this to count down. One would be to subtract three from the units and tens counters instead of adding three. What else must be done to make this idea work? Is there room in the program to insert the additional add or subtract instruction to compare for wraparound in the units digit? How could you change the program so that it counts down from +30? Would this be longer or shorter? Try it.
2. Another way to display seconds counting down is rename the digits so that when the counter has a value corresponding to "0" you display "9". Likewise, you'd interchange "1" and "8", "2" and "7", and so on. What does this do to the tens digit? How can you change the program so that the tens will count down from 5 to 0, and not 9 to 4? Try your suggestion.
3. There are three dots at location 0062 representing what must be done to count minutes. Modify the program to do this, and to display minutes instead of seconds. Hours?
4. What changes could you make to this program to make a "stopwatch" out of it? That is, when you push the "I" key it counts and when you release it it stops. When it is stopped it should continue to display the last count, but the internal registers should be set to some value which corresponds to zero. I would suggest forcing them to: 59 seconds, 60 frames each time the display routine ends with EF4 false, and counting normally if EF4 is true.
   Real stopwatches have a double-action button: The first time you push it the watch is started; when it is pushed a second time the watch stops. Can you think of a way to do this? Hint: Assign a memory location, one bit of which has the same significance as EF4 used to. Every time you push "I" the bit is complemented. Caution: You cannot just complement it every time you go by with EF4 true, because you may go by several times for a single button push. Do you need more memory to do this?
5. Ideally we would like our digital clock to display hours, minutes, and seconds. 256 bytes may not be not enough for such a program. Also

you will need to figure out how to display the numbers smaller, such as by making the dots only two bits wide instead of eight, and only eight lines high instead of 25. Note: Plan on quite a bit of time to do this one. It is not easy. Also watch out for trying to do too much in one interrupt time. If you take too long you will not be ready to respond to the next interrupt.

```
        ..  PROGRAM 7.6 -- TV DIGITAL CLOCK
        ..
0000 90   REST: GHI 0     .. INITIALIZE R1, R2, R3
0001 B1         PHI 1
0002 B2         PHI 2
0003 B3         PHI 3
0004 F81B       LDI INTS  .. R1 = INTERRUPT PC
0006 A1         PLO 1
0007 F8FF       LDI #FF   .. R2 = STACK
0009 A2         PLO 2
000A F80F       LDI MAIN  .. R3 = MAIN PC
000C A3         PLO 3
000D 70         RET       .. X=0!
000E 23         #23       .. SET X=2, P=3
000F 69   MAIN: INP 1     .. TURN ON TV
0010 30AC       BR TEST   .. DO DIAGNOSTIC
0012    ..
0012    ..  DISPLAY REFRESH
0012    ..
0012 12   DONE: INC 2     .. ASSUME X=2!
0013 42         LDA 2     .. RESTORE DF
0014 F6         SHR
0015 42         LDA 2     .. RESTORE R7
0016 B7         PHI 7
0017 42         LDA 2
0018 A7         PLO 7
0019 42         LDA 2     .. NOW D
001A 70         RET       .. RESTORE X AND P
001B C4   INTS: NOP       .. EVEN OUT CYCLES
001C 22         DEC 2     .. PUSH STACK, TO
001D 78         SAV       .. SAVE X AND P (IN T)
001E 22         DEC 2
001F 73         STXD      .. SAVE D
0020 87         GLO 7     .. SAVE R7
0021 73         STXD
```

```
0022 97         GHI 7
0023 73         STXD
0024 7E         SHLC      .. SAVE DF
0025 73         STXD
0026 F8C8       LDI BUFF  .. SET UP R0
0028 3428       B1  *     .. WAIT FOR DISPLAY
002A A0   ROW:  PLO 0
002B            ...       .. DMA HERE
002B A0         PLO 0     .. RESET R0
002C B7         PHI 7
002D F80B       LDI 11    .. (RASTER COUNT - 3)/2
002F            ...
002F A7         PLO 7
0030 97         GHI 7     .. KEEP FIXING R0
0031 A0         PLO 0
0032            ...
0032 27   REPT: DEC 7     .. COUNTER RASTERS
0033 97         GHI 7
0034 A0         PLO 0
0035            ...
0035 A0         PLO 0
0036 87         GLO 7     .. TWO LINES PER LOOP
0037 3A32       BNZ REPT
0039            ...
0039 80         GLO 0     .. IF LAST TIME,
003A 3C2A       BN1 ROW
003C A0         PLO 0     .. JUST BLANK IT
003D            ...
003D 343C       B1 *-1    .. (3 LINES)
003F    ..
003F    ..  SECONDS CLOCK
003F    ..
003F 90         GHI 0
0040 B7         PHI 7
0041 F8C7       LDI FRCT  .. POINT TO FRAME COUNT
0043 A7         PLO 7     .. R7 IS AVAILABLE
0044 07         LDN 7
0045 FC01       ADI 1     .. BUMP COUNTER
0047 57         STR 7
0048 FF3D       SMI 61    .. MOD 61
004A 3B12       BNF DONE  .. NOT OVER
```

```
004C E7         SEX 7
004D 73         STXD      .. ROLL OVER
004E F0         LDX       .. TO SECONDS
004F FC03       ADI 3
0051 57         STR 7
0052 3B69       BNF UNIT  .. GO DISPLAY
0054 F8E2       LDI -30   .. ROLL OVER
0056 73         STXD
0057 F0         LDX       .. TO TENS
0058 FC03       ADI 3
005A 57         STR 7
005B FC0C       ADI 12    .. (OVERFLOW AT 60)
005D 3B62       BNF TENS
005F F8E2       LDI -30   .. ONE MINUTE!
0061 57         STR 7
0062            ...       .. COULD DO MINUTES, HOURS...
0062 F8C8 TENS: LDI BUFF  .. POINT TO LEFT DIGIT
0064 306B       BR UNIT+2
0066 F8C6       LDI SECS  .. (POINT TO COUNTER)
0068 A7         PLO 7
0069 F8CC UNIT: LDI BRIT  .. OR RIGHT DIGIT
006B A0         PLO 0
006C 47         LDA 7     .. POINT TO DIGITS
006D FCAC       ADI TABL  .. (TABLE OFFSET)
006F A7         PLO 7
0070 47   DOWN: LDA 7     .. GET DOTS
0071 52         STR 2     .. (SAVE)
0072 E2         SEX 2
0073 F0   HALF: LDX       .. CONVERT A DOT
0074 FE         SHL       .. FROM A BIT
0075 52         STR 2
0076 75         SDB       .. =00 IF DF=1, =FF IF DF=0
0077 50         STR 0     .. STORE INTO BUFFER
0078 10         INC 0
0079 80         GLO 0
007A FA03       ANI 3     .. DO THIS 4 TIMES
007C 3A73       BNZ HALF  .. (9*4 INSTRUCTIONS)
007E 10         INC 0
007F 10         INC 0
0080 10         INC 0
0081 10         INC 0
```

```
0082 F0         LDX       .. CHECK FOR SECOND 4 BITS
0083 3A73       BNZ HALF  .. ((36+6)*2)
0085 80         GLO 0     .. REPEAT IF THIS WAS LEFT
0086 FFF8       SMI BEND
0088 3B70       BNF DOWN  .. ((84+6)*3)
008A 3266       BZ UNIT-3 .. ((270+9)*2)
008C 3012       BR DONE   .. MAX TOTAL <600 INSTRUCTIONS
008E    ..
008E    ..  DOT TABLE FOR DIGITS
008E    ..
008E DAAADF     #DAAADF   .. 0
0091 D9DD8F     #D9DD8F   .. 1
0094 9EDB8F     #9EDB8F   .. 2
0097 9EDE9F     #9EDE9F   .. 3
009A EAA8EF     #EAA8EF   .. 4
009D 8B9E9F     #8B9E9F   .. 5
00A0 CB9ADF     #CB9ADF   .. 6
00A3 8EDBBF     #8EDBBF   .. 7
00A6 DADADF     #DADADF   .. 8
00A9 DACEDF     #DACEDF   .. 9
00AC    TABL=*
00AC    ..
00AC    ..  MINI DIAGNOSTIC
00AC    ..
00AC E7   TEST: SEX 7     .. SET UP R7
00AD A7         PLO 7     .. AS COUNTER
00AE 3BB1       BNF PLUS
00B0 17         INC 7
00B1 60   PLUS: IRX
00B2 7C01       ADCI 1    .. IN PARALLEL WITH D
00B4 3AB1       BNZ PLUS
00B6 E2         SEX 2     .. WHEN D=00,
00B7 87         GLO 7     .. COMPARE THEM:
00B8 22         DEC 2
00B9 52         STR 2
00BA 64         OUT 4     .. DISPLAY DIFFERENCE
00BB 32BF       BZ *+4    .. EQUAL CONTINUES,
00BD 3FBD       BN4 *     .. UNEQUAL WAITS
00BF 37BF       B4  *     .. WAIT FOR RELEASE
00C1 6C         INP 4     .. GET NEW COUNT
00C2 FE         SHL       .. SET DF
```

```
00C3 30AC       BR TEST   .. REPEAT
00C5    ..
00C5    ..  TIME COUNTERS AND DISPLAY BUFFER
00C5    ..
00C5 E2   STEN: #E2       .. MUST INITIALIZE
00C6 E2   SECS: #E2
00C7 00   FRCT: 00
00C8      BUFF=#C8        .. EMPTY BUFFER
00CC      BRIT=#CC
00F8      BEND=#F8
               END
```

Appendix A -- Instruction Summary

MNEM	NAME	OPCODE	CYCLES
ADC	Add with Carry	74	2
ADCI b	Add with Carry Immediate	7C bb	2
ADD	Add	F4	2
ADI b	Add Immediate	FC bb	2
AND	Logical AND	F2	2
ANI b	AND Immediate	FA bb	2
B1 a	Branch on External Flag 1	34 aa	2
B2 a	Branch on External Flag 2	35 aa	2
B3 a	Branch on External Flag 3	36 aa	2
B4 a	Branch on External Flag 4	37 aa	2
BDF a	Branch if DF is 1	33 aa	2
BN1 a	Branch on Not External Flag 1	3C aa	2
BN2 a	Branch on Not External Flag 2	3D aa	2
BN3 a	Branch on Not External Flag 3	3E aa	2
BN4 a	Branch on Not External Flag 4	3F aa	2
BNF a	Branch if DF is 0	3B aa	2
BNQ a	Branch if Q is off	39 aa	2
BNZ a	Branch on Not Zero	3A aa	2
BQ a	Branch if Q is on	31 aa	2
BR a	Branch unconditionally	30 aa	2
BZ a	Branch on Zero	32 aa	2
DEC r	Decrement Register	2r	2
DIS	Return and Disable Interrupts	71	2
GHI r	Get High byte of Register	9r	2
GLO r	Get Low byte of Register	8r	2
IDL	Idle	00	2

INC r	Increment Register	1r	2
INP p	Input to memory and D (for p = 9 to F)	6p	2
IRX	Increment R(X)	60	2
LBDF aa	Long Branch if DF is 1	C3 aaaa	3
LBNF aa	Long Branch if DF is 0	CB aaaa	3
LBNQ aa	Long Branch if Q is off	C9 aaaa	3
LBNZ aa	Long Branch if Not Zero	CA aaaa	3
LBQ aa	Long Branch if Q is on	C1 aaaa	3
LBR aa	Long Branch unconditionally	C0 aaaa	3
LBZ aa	Long Branch if Zero	C2 aaaa	3
LDA r	Load D and Advance	4r	2
LDI b	Load D Immediate	F8 bb	2
LDN r	Load D via N (for r = 1 to F)	0r	2
LDX	Load D via R(X)	F0	2
LDXA	Load D via R(X) and Advance	72	2
LSDF	Long Skip if DF is 1	CF	3
LSIE	Long Skip if Interrupts Enabled	CC	3
LSKP	Long Skip	C8	3
LSNF	Long Skip if DF is 0	C7	3
LSNQ	Long Skip if Q is off	C5	3
LSNZ	Long Skip if Not Zero	C6	3
LSQ	Long Skip if Q is on	CD	3
LSZ	Long Skip if Zero	CE	3
MARK	Save X and P in T	79	2
NOP	No Operation	C4	3
OR	Logical OR	F1	2
ORI b	OR Immediate	F9 bb	2
OUT p	Output from memory (for p = 1 to 7)	6p	2
PHI r	Put D in High byte of register	Br	2
PLO r	Put D in Low byte of register	Ar	2
REQ	Reset Q	7A	2
RET	Return	70	2
SAV	Save T	78	2
SD	Subtract D from memory	F5	2
SDB	Subtract D from memory with Borrow	75	2
SDBI b	Subtract D with Borrow, Immediate	7D bb	2
SDI b	Subtract D from memory Immediate byte	FD bb	2
SEP r	Set P	Dr	2
SEQ	Set Q	7B	2
SEX r	Set X	Er	2
SHL	Shift D Left	FE	2

SHLC	Shift D Left with Carry	7E	2
SHR	Shift D Right	F6	2
SHRC	Shift D Right with Carry	76	2
SKP	Skip one byte	38	2
SM	Subtract Memory from D	F7	2
SMB	Subtract Memory from D with Borrow	77	2
SMBI b	Subtract Memory with Borrow, Immediate	7F bb	2
SMI b	Subtract Memory from D, Immediate	FF bb	2
STR r	Store D into memory	5r	2
STXD	Store D via R(X) and Decrement	73	2
XOR	Exclusive OR	F3	2
XRI b	Exclusive OR, Immediate	FB bb	2

Appendix B -- Binary and Hexadecimal Numbers

People think in decimal numbers, using the ten digits 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. It turns out that computers are easier to make if they use binary numbers, which have only two digits: 0 and 1.

In decimal, you count 1, 2, ... 8, 9, 10, 11, 12, ... and so on. As soon as you have used the last available decimal digit in the units digit position, you go back to zero in that digit and increase the next digit to its left by one. Binary works the same way, except that we run out of digits much sooner: In binary, you count 1, 10, 11, 100, 101, 110, 111, 1000, and so on.

Any number we happen to think of can be written in binary. All that is necessary is to count in binary at the same time as you count in ordinary decimal, and when you reach the number you were thinking of in decimal, the binary number you reached at the same time is the same number in binary. This is a little slow (something like counting on your fingers). It works, but we would like to do it faster, so let's look for another analogy with decimal numbers.

When you talk about ninety-nine dollars, you do not mean that you are counting the dollars from one to ninety-nine. Rather, you think of writing a "9" which means "nine ten-dollar bills" and another "9" next to it which means "nine one-dollar bills". And for numbers over a hundred, you write a single digit which counts the hundreds. There is a peculiar similarity to the way we write one, ten, a hundred, etc: (1, 10, 100...). Notice that it is a "1" followed by some number of zeros. We can do the same in binary: 1, 10, 100... Only now, instead of being powers of ten, they represent powers of two. In decimal you multiply ten times itself four times to get ten thousand. In binary you multiply two times itself four times to get sixteen, but it is written the same way: 10000.

So we can think of a binary number as some number of ones, plus some number of twos, plus some number of fours, plus some number of eights, and so on. This is just like decimal, where a number is some number of ones plus some number of tens plus some number of hundreds, etc. In decimal you can have any number of each from zero to nine (because those are the digits you have to write it), but in binary you can have only zero or one of each.

Let us look at that ninety-nine again. We could say that it is the same as one 64 plus one 32 plus no 16 plus no 8 plus no 4 plus one 2 plus one 1. We can write this as:

```
        decimal    binary
1 x 64 =     64 = 1000000
1 x 32 =     32 =  100000
0 x 16 =      0 =       0
0 x  8 =      0 =       0
0 x  4 =      0 =       0
1 x  2 =      2 =      10
1 x  1 =      1 =       1
             99 = 1100011
```

Notice how many more digits it takes to write "1100011" than "99"! We are all much too lazy to do all that writing, so we bunch the binary digits into groups of four, and give them special names, which curiously happen to match the decimal numbers to some degree. Of course there are sixteen combinations of four binary digits, so we run out of decimal digits to name them with and so use the letters A, B, C, D, E, and F for the last six. We call this the **hexadecimal** number system, because there are sixteen possible digits (instead of ten or two). Everything else works the same way. Of course each digit represents some number of a power of sixteen (such as one, sixteen, two-hundred-fifty-six, etc.).

The computer uses binary inside. We use decimal outside. So we often use hexadecimal to communicate with the computer as a "bridge" between the two systems.

As far as the computer is concerned, hexadecimal is only a way of writing binary. Hexadecimal is not "only a way of writing decimal." You have to do some arithmetic to convert it. For small numbers, use the following table (which you'll soon memorize):

Decimal	Hex	Binary
0	0	0
1	1	1
2	2	10

3	3	11
4	4	100
5	5	101
6	6	110
7	7	111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

To convert hexadecimal to decimal, you multiply each successive hex digit by its respective power of sixteen, then add the products. To convert from decimal to hexadecimal, you divide the decimal number by the largest possible power of sixteen; the quotient is the first hex digit; dividing the remainder by successively smaller powers of sixteen will give you the rest of the digits. These are the powers of sixteen you will need for this book:

Decimal	Hexadecimal
65536	10000
4096	1000
256	100
16	10
1	1

EXERCISES

1. Convert 1234 from decimal to hex.

```
1234 + 256 = 4 remainder 46
  46 +  16 = 2 remainder 14
  14 +   1 = E (14) remainder 0
             ^-- 42E is equivalent hex number
```

2. Convert 66666 from decimal to hex.

```
66666 + 65536 = 1 remainder 1130
 1130 +  4096 = 0 remainder 1130
 1130 +   256 = 4 remainder  106
  106 +    16 = 6 remainder   10
   10 +     1 = A remainder    0
                 ^-- 1046A is equivalent hex number
```

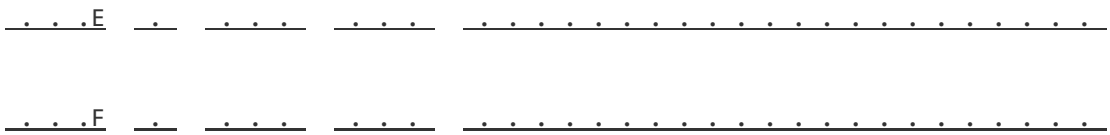
3. Convert 1234 from hexadecimal to decimal.

```
1234 = 1x4096 + 2x256 + 3x16 + 4x1
     =  4096  +  512  +  48  +  4
     =  4660 in decimal
```

Coding Form

Program Name_____ Date_____ Page____ of____

ADDRESS	CODE	LABEL	MNEMONIC	OPERAND	COMMENTS
. . .0	.	. . . .	. . . .	. . . . .	
. . .0	.	. . . .	. . . .	. . . . .	
. . .1	.	. . . .	. . . .	. . . . .	
. . .2	.	. . . .	. . . .	. . . . .	
. . .3	.	. . . .	. . . .	. . . . .	
. . .4	.	. . . .	. . . .	. . . . .	
. . .5	.	. . . .	. . . .	. . . . .	
. . .6	.	. . . .	. . . .	. . . . .	
. . .7	.	. . . .	. . . .	. . . . .	
. . .8	.	. . . .	. . . .	. . . . .	
. . .9	.	. . . .	. . . .	. . . . .	
. . .A	.	. . . .	. . . .	. . . . .	
. . .B	.	. . . .	. . . .	. . . . .	
. . .C	.	. . . .	. . . .	. . . . .	
. . .D	.	. . . .	. . . .	. . . . .	


E
F