SubsySTEM

2

SUBSYSTEMS

Electronics Learning Module 2– Microprocessor Trainer

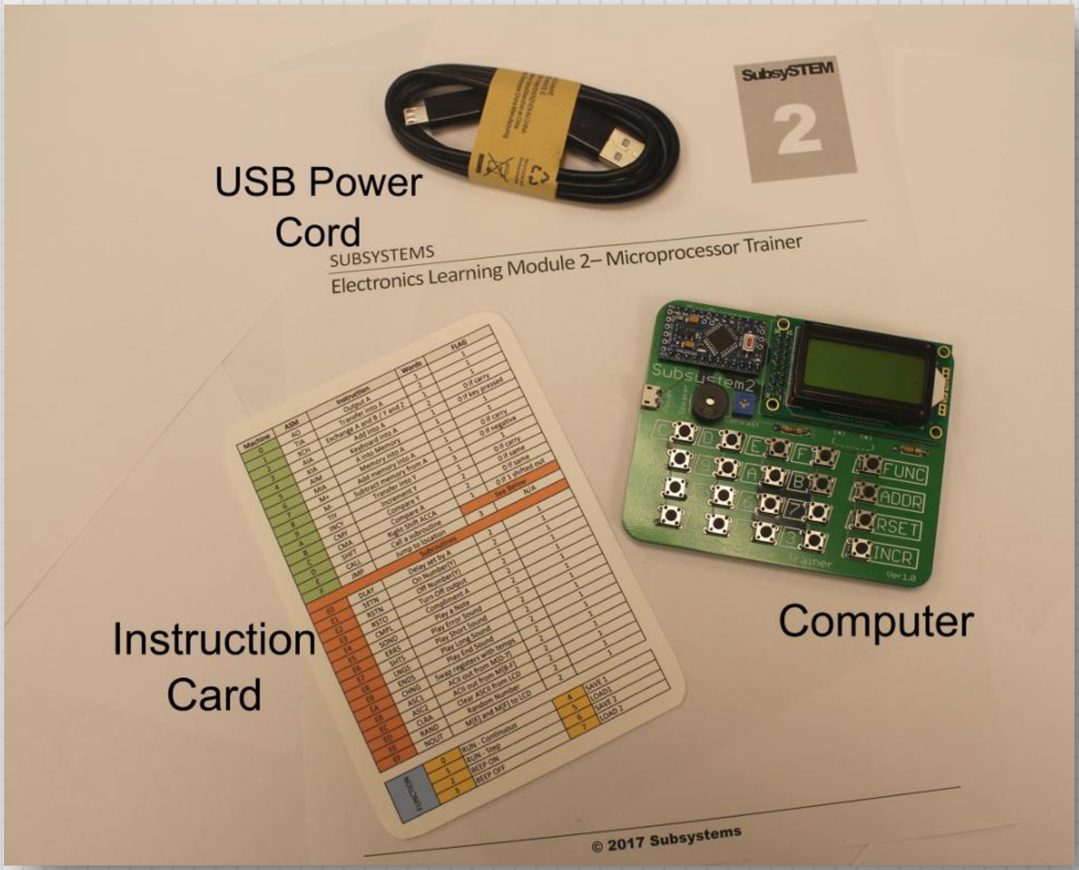


Table of Contents

Introduction

You can't go anywhere now without being near computers. This might be in part because you carry one around in your pocket. But computers have a pretty humble beginning. You may have heard that computers speak in "ones and zeros." But what does that mean? And why is it possibly important to know the inner workings of a computer. Modern computer architecture (the hardware) and programming (software) has been developed over time but stems from these humble beginnings. By understanding a very early implementation of a computer, it makes it easier to understand the operation and programming paradigms that exist today in these systems. It is also a fun way to learn computer history. Finally, at its core, computer programming is a method to logically instruct a computer on how to execute a desired sequence of calculations. By studying this, it helps our minds grow in their ability to logically assemble data into meaningful ideas.

Objectives

This SubsySTEM unit is designed to give you a basic understanding of computer structure and programming and offer some hands on experience with a simple, machine-language programmed 4-bit computer. Below is a list of the primary objectives of this unit:

After completion of this unit, the student will:

1. be able to explain the parts of a microprocessor to include the input, output, memory, arithmetic logic unit and the central processing unit.
2. understand the concept of computer memory including the distinctions between RAM, ROM, EPROM, Registers, and Accumulators.
3. understand how to read a flow chart and how it may be converted to a computer program.
4. understand the basics of the binary number system and be able to represent base 10 numbers in binary.
5. program a 4-bit computer with simple programs.
6. understand computer commands used in manipulating data in a computer.
7. understand the allocation and use of computer memory.
8. create new programs and enter them into the computer.

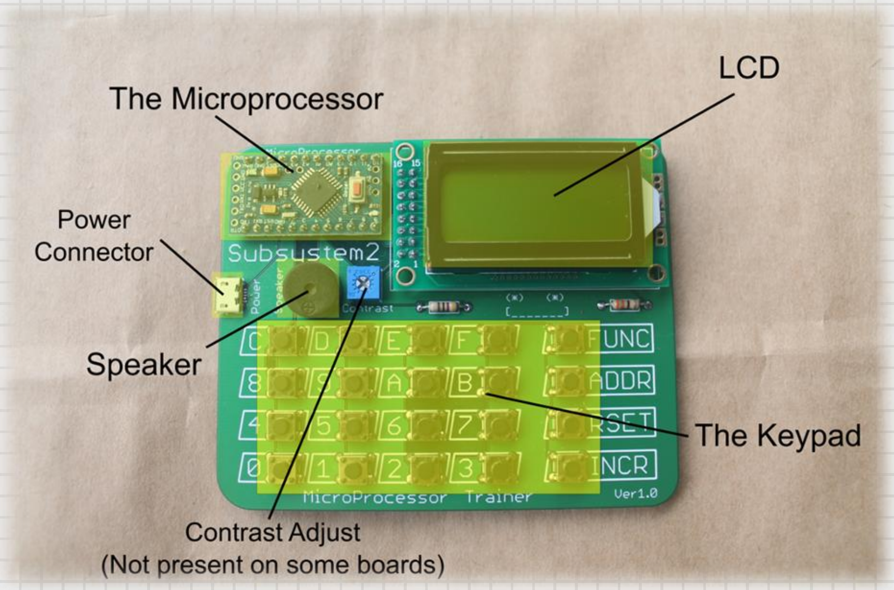
Programming computers can be an immensely challenging and rewarding endeavor. A single computer can be used for an unbelievable amount of different tasks. On the other hand, computer programs are just sets of on and off signals (ones and zeros), and if one of these bits of data is wrong, the computer usually doesn’t function properly. For that reason, they can drive people crazy and be the source of great frustration. During this study, I encourage you to experiment, learn, and have fun. But, if you find yourself getting very frustrated, take a break. Often, when you return to a task refreshed, you can see problems and solutions more easily.

Tools

To accomplish these objectives, you have a few tools.

First, you have this curriculum guide. It will walk you step by step through the entire curriculum that includes the text and the lab exercises. It will introduce you to the concepts of computers, number systems, programming, and moving ideas into code.

You also have a 4-bit microprocessor. This is a fully programmable computer (programmed in machine language) that has a keyboard input, and a screen and speaker for output. The following picture shows the layout of the main parts of the computer board and describes their functions.



The Microprocessor: This is the heart of the computer. It has all of the computing, processing, and input/output interfaces needed to run programs.

The LCD: This is a 2 line by 8 character display used to guide the user in programming the microprocessor and displaying information from running programs.

The Keypad: This is used to enter programs, execute functions, and take input during program execution.

Speaker: This generates tones and short preloaded sounds.

Power Connector: This is a micro USB connector that is used to power the board (This is only a power connection. No data is sent via the USB port).

Contrast Adjust: Using a small screwdriver you can adjust the contrast of the LCD. This is only included on boards with a variable contrast LCD display so it may or may not be present on your board.

Safety

Because the computer runs off of 5 volts from a USB power supply, there is no risk of electric shock. However, because of the exposed circuitry, there are a couple of safety precautions.

1) Do not sit the computer down on conductive material like metal. This has the potential to short out circuits and could damage the computer.
2) Some electrical components and the solder used to fuse them to the board may contain small amounts of lead. Wash your hands after handling.

Alright, with that introduction out of the way, let’s get started on learning about computers.

Section 1: Microprocessor Basics

To understand how a computer works on the most basic level, we first have to understand some vocabulary and definitions.

Internal Layout

Most computers are made up of the same basic components. Below is an explanation of the main parts.

Memory

The memory in a computer is normally made up of two parts:

Random Access Memory (RAM): This is memory used for the saving and retrieving of data. It can contain programs, numbers, words, or places to temporarily hold information needed by the computer. It is typically what is called **"volatile"** memory because the data in RAM is not saved when power is removed.

Read Only Memory (ROM): This is memory that can be read from at any time but can only be written to once. It is considered **"non-volatile"** because it retains its data even if power is removed. It is often used for storing programs in devices that need to have a master program available as soon as they are turned on. You will find these in vending machines, cell phones, home computers, and just about everything that is operated by computer.

It is interesting to note that ROM has changed as technology has changed. Early ROM memory chips were programmed with a high voltage (higher than normal operating voltage). Some had a transparent window on top which exposed the memory circuit to light. You could then erase this chip with a dose of high intensity ultra-violet light and program it again. These were referred to as Erasable Programmable Read Only Memory (EPROM). As technology continued to improve, we gained the ability to erase and program our ROM chips electronically. This meant we didn't need to remove the chip from the board and place it under light. We could just erase and reprogram it while it was still in the circuit. This process is called "In-System Programming" and is abbreviated ISP. These chips were referred to as "Electronically Erasable Programmable Read Only Memory" or EEPROM (often referred to as E-E-PROM, Double-E PROM, or E-squared PROM).

A very common type of non-volatile memory today is Flash Memory. You may have heard the term flash drive to describe a small USB drive (or thumb drive). Flash was developed from EEPROM but it has an advantage. EEPROM normally needs to erase the entire contents of memory and program the entire memory in one operation. Flash memory allows you to erase and write smaller sections (called blocks or pages). Some Flash will even let you read and write a single memory location. We have come a long way in the realm of computer memory.

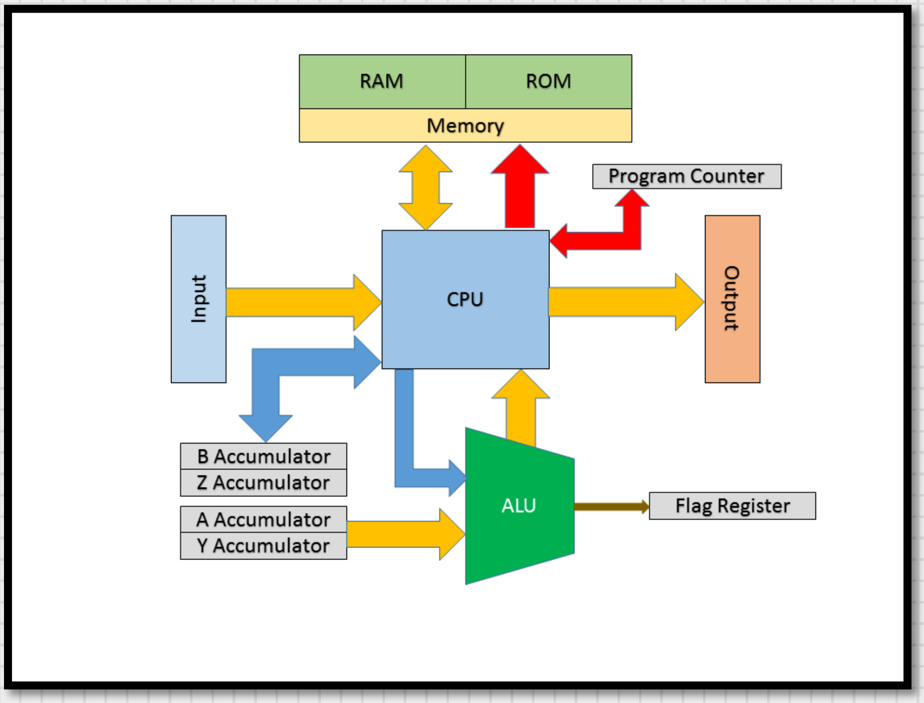
Your microprocessor trainer has both ROM and RAM. The ROM stores routines for operating the processor like reading the keyboard, running the LCD, and making sounds. You also have RAM. You will load this area of memory with your program and any data you may need.

INPUT/OUTPUT

Input/output is the generic term to indicate the components used to send information to the internals of the processor and to take results from the processor and make them available to external circuits. For our computer, we have a keyboard to input information into the computer and have a display and speaker to get information from the computer.

Central Processing Unit (CPU)

The CPU is the brains of the computer. It is responsible for fetching instructions from memory, keeping track of where we are in the program, controlling the flow of information to input and output systems, timing functions, and sending data to other sections to perform arithmetic operations. Let's look at a typical depiction of a CPU with its supporting components.



Below is a description of some of the key parts:

First, let's define a **register**. A register is a quickly accessed memory location that is available to the CPU. It is a working storage location for holding data or setting different functions within the CPU.

Program Counter (PC): The PC is a register that holds the memory address location of the next instruction to be executed. The program counter is used to keep track of where we are in our program. We will be able to use the JUMP command later to change the value of this register and pick up program execution from a new memory location.

Accumulator A, B, Y, and Z: An accumulator is a type of register. It is used as the main storage location that data is written to, manipulated, and sent to other parts of the computer. We use accumulators A and Y to do these functions and accumulators B and Z to act as temporary storage for A and Y.

Arithmetic Logic Unit (ALU): The ALU is a section of the CPU that performs math and certain logical functions on the A and Y registers. We will use the ALU frequently in our programming and it is one of the most powerful components of a computer.

ROM: This is the memory that has already been programmed into your computer. It contains the routines for operating the inputs and outputs and initializes the computer when you boot up. You do not have access to this memory.

RAM: This is a memory bank of 256 bytes of memory space for data. This is the memory you will write into when you program. The first 240 bytes of this storage is for your program. The last 16 bytes are a memory location that you can store data in for use in your program.

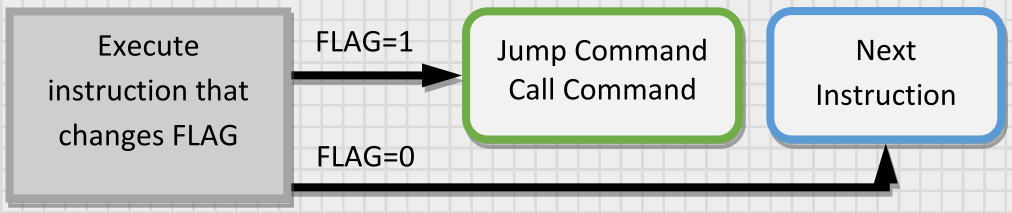
EEPROM: You also have 512 bytes of EEPROM. You can use this to save or load a program into memory. This memory will hold its contents even if power is removed. This allows you to save your work and continue it at a later time or to save a long program that you don’t want to have to type in again. There are two banks (bank 1 and bank 2) that you can use to store two different complete copies of system memory.

Flag Register: The flag register holds a single bit of information. For some of the program commands, there is a condition that results from performing the command that we want to test. For instance, we may want to compare a number with the contents of the A accumulator. If the number is equal, we may want to take some action and if the number is not equal, we may want to do a completely different action. When an operation wants to send this conditional information to our program, it will use the flag register. Commands will then use this flag value to determine where to continue program execution. When an instruction executes, the flag operates as follows:

FLAG = 1: The next instruction in memory is executed. (This is the default condition)

FLAG = 0: If the next instruction is a JMP or CAL instruction, that instruction is skipped.

The FLAG is then used to change the path of code execution depending on certain conditions. For example, the instruction INCY (increment ACCY) will add 1 to the Y accumulator each time it is executed. When ACCY gets to 15, the next INCY will overflow the ACCY because it can only hold numbers 0-15. ACCY will roll over to 0 and the FLAG will be set to 0. If this happens, the next instruction (if it is a JMP or CAL instruction) will be skipped. Below shows how this works in our code.



If the instruction that follows is not a JMP or CAL command, the next instruction is executed regardless of the FLAG value. Your instruction information card has a column that shows which instructions and what conditions are required to set the FLAG to zero. You will see this in action when we get to writing programs.

Address Bus: These are lines from the CPU to various components to convey the digital information about what address in memory we want to access.

We are close to powering up our computer and starting programming. We just need to cover a few more basics.

Binary Counting

We communicate with words, sounds, and gestures. As sophisticated as computers are, they really do just communicate in ones and zeros. Computers are made up of millions of transistors and these transistors are like switches. They have two possible conditions; on or off. By combining multiple switches together, we can build a number system that the computer can then use to operate.

The Bit

A **bit** is the smallest information that a computer can store. It is a single on/off state. So 1-bit can have two different possible states; 1 and 0. But let's look at a 2-bit system.

BIT 1	BIT 2	State Number
0	**0**	**0**
0	**1**	**1**
1	**0**	**2**
1	**1**	**3**

Since each bit can have two states, they can each be one or zero, so both together create 4 unique states (00, 01, 10, 11). So if we wanted to count to 3, we could use 2 bits to do it.

It is important to remember that counting systems start at zero so they always count up to one number less than the number of states. If you were asked to count in binary to 4, you would have to include one more bit to do it (you would need 3 bits total). Let's see how far we get with 3 bits. You can see something interesting from the table. We only added 1 bit but we have doubled the amount of states (we now have 8 unique states). In fact, every bit we add will double the total number of states. There is actually a mathematical relationship between the number of bits and the number of states:

BIT 1	**BIT2**	**BIT3**	**State Number**
0	**0**	**0**	**0**
0	**0**	**1**	**1**
0	**1**	**0**	**2**
0	**1**	**1**	**3**
1	**0**	**0**	**4**
1	**0**	**1**	**5**
1	**1**	**0**	**6**
1	**1**	**1**	**7**

$$Number\ of\ States = \ base^{Number\ of\ bits}$$

So for our 4-bit data binary computer,

$$Number\ of\ States = \ 2^4 = 16$$

So, we can store numbers from 0 to 15 on our 4-bit computer.

Converting from Binary

The way a binary number works is the same way our base 10 number system works. Look at the number 275 and how we represent it.

$$275 = 2x100 + 7x10 + 5x1$$

We have all known from our earliest lessons on counting that there is a one's column, a 10's column, a 100's column, etc. Because we use a decimal system, all of our place holders are multiples of 10. All other base systems work the same way. Binary is a base 2 system so all of the placeholders will be multiples of 2. Let's look at what these placeholders look like.

Digit	Digit 5	Digit 4	Digit 3	Digit 2	Digit 1
Multiplier	**16**	**8**	**4**	**2**	**1**

Notice, just like base 10, as we move to the higher digits, we multiply the previous number by the base (2). Because this is base 2, we can only have a 0 or 1 in each digit location. So what is the decimal value of the binary number 1101? Solve it like the way we broke down the base 10 number above.

$$1101 = 1x8 + 1x4 + 0x2 + 1x1 = 13$$

So the number **1101** in binary is actually **13** in decimal. We can use this same procedure for converting from base 10 to binary. What is the decimal number 11 converted into binary?

Here we can look at the placeholder values in binary and see how to fit our number into its allowed values. From the chart you can see that the fifth digit represents a value of 16. This is too high for our number (11) so let's try the next smallest digit. Digit 4 represents a value of 8. There is definitely an 8 in 11 so we will start our conversion at that point showing a 1 in that digit position. Since we have used up 8 of our initial 11, we are left with 3. The next digit represents a value of 4 which is bigger than our 3 so we will report no 4s. The next digit represents a value of 2. We definitely have a 2 in our 3 so we will report a 1 in this digit. Since we used 2 of our remaining 3, we only have 1 left. The last digit represents the 1's column so we can report the 1 we have left. The whole number is then constructed as follows:

$$\mathbf{11 = 1x8 + 0x4 + 1x2 + 1x1}$$

So if we pull off these 1s and 0s we get the binary number:

$$\mathbf{1011}$$

This is the same as saying this number is 1 eight, and 1 two, and 1 one. Add them up and you get to 11. This is one method for converting between binary and decimal. Today, it is pretty easy to go online and find numerous calculators and apps that will do these conversions for you. It is good to understand number systems when dealing with computers because it is often necessary to represent numbers in different forms depending on how you want to process data.

Hexadecimal

What I really want to discuss is base 16. This is what you will use to program your microprocessor. The number system that uses base 16 is called hexadecimal. Hexadecimal has 16 states, so based on our previous discussion we can see that a 2-digit hexadecimal number can have 16^2, or 256 states. That is the size of our RAM, so we can address any location in our memory with a 2-digit hexadecimal number. One interesting thing to note is that a base 16 number has 16 possible numbers that are in each digit. The problem with that is that we normally use base 10 for counting which only allows digits 0 through 9. We have no single digit that represents 10. We need a 1 and 0 to do it (a 2-digit number). To solve this problem, we assign the first 6 alphabet letters to represent numbers 10 through 15. This allows us to have a single character represent what is normally a 2-digit number. Below is a table showing these numbers.

Base 10 (Decimal)	Base 2 (Binary)	Base 16 (Hexadecimal)
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

It can be very strange to see letters in our numbers, but this is hexadecimal and the way our computer can represent base 16 numbers. Let’s look at a conversion to see how this works.

Convert 3A in hexadecimal to base 10 (decimal).

Like before, we know that the first digit is our 1’s place and the second digit is our 16’s place. Then we just construct the number from the digits displayed:

$$\mathbf{3A = 3 * 16 + A * 1 = 48 + 10 = 58}$$

If you were to express this number in base 16 you might say that it is 3 sixteens and 10 ones.

If you look at your keyboard on the microprocessor, you will see that, in addition to some command keys, you have keys labeled 0 through F. This is how you will enter hexadecimal numbers into the computer.

Ok, that is the last time I will say hexadecimal. People who use computers and program a lot will just refer to base 16 as hex. So that’s what we will do from here on out.

Your computer uses a 4-bit data bus and an 8-bit address bus. This means we can express the data with 1 hex digit and the address with 2 hex digits. The computer you normally use probably has a 64-bit data bus. What number can it count to?

$$\boldsymbol{Number\ of\ states = 2^{64} = 1.8447x10^{19}}$$

That is 18,447,000,000,000,000,000 states. Wow that is a big number!

Microprocessor Trainer

Let’s look closer at your new computer. Your microprocessor trainer has the following features:

Hex keypad Input: This is where you will enter data into your memory. This data will represent your program and other information need by the program.

INCR Key: This key will increment the program counter in write mode to allow you to easily enter a command and then advance to the next memory location. It can also be used when the program is run to step through the lines in your program one at a time.

RSET Key: This will reset the microprocessor and return the program counter to 0.

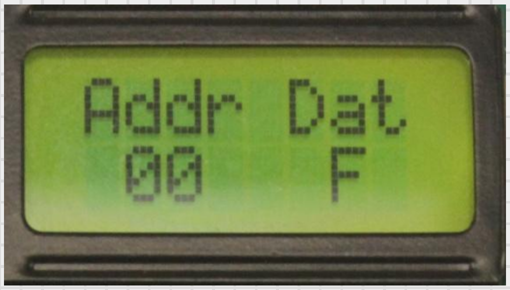
ADDR Key: This key allows you to key in a 2-digit hex address and will advance the program counter to that address and display the contents of the memory at that address. You can use this to jump to some specific location when you are programming or to access the upper memory locations to store data used during program execution. Just press this key and then follow it with the two-digit address (00-FF).

FUNC Key: This key gives you access to extended functions. The table below lists these functions along with a description.

Function	Name	Description
0	RUN	Runs your program
1	Step Run	Runs your program one command at a time
2	BEEP ON	Turns on a sound when a key is pressed
3	BEEP OFF	Turns the key sound off
4	SAVE1	Save the program to EEPROM memory bank 1
5	LOAD1	Load the program from EEPROM memory bank 1
6	SAVE2	Save the program to EEPROM memory bank 2
7	LOAD2	Load the program from EEPROM memory bank 2

Display: The microcontroller uses a 2 row by 8 column display to show the status of programming and as an output during run time. While programming, the display will show the current address and contents of that address. During run time, the top line will show the current program counter, any 4-bit data you direct the program to display, and any text you program. The bottom line will have 8 numbers that you can program to be on or off individually. These can be used to display binary numbers or counting sequences.

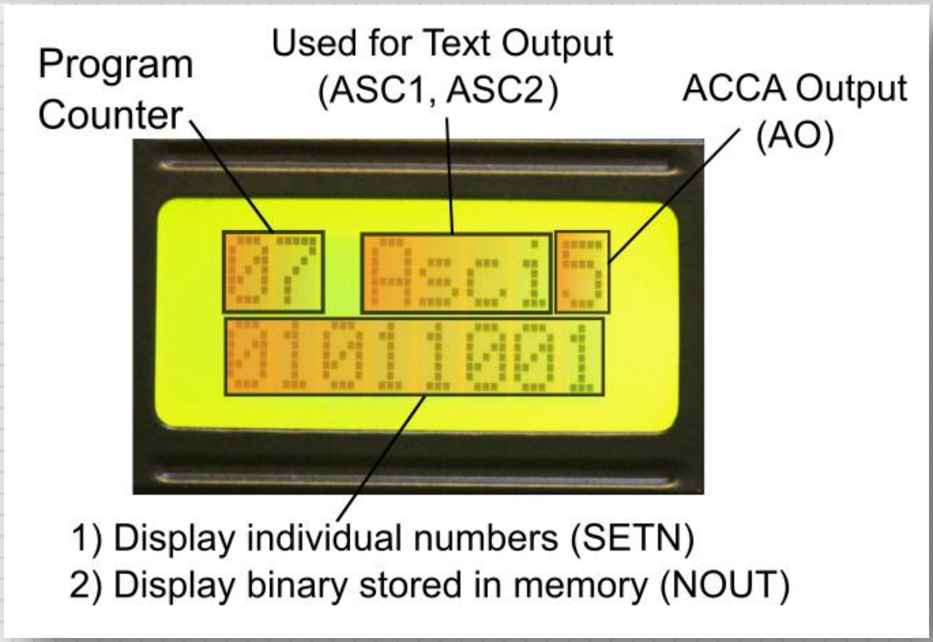
In **Program Mode** the display will look like the following:



“Addr” stands for Address. This displays the current value of the program counter which will point to the current memory location you are at. When you enter a number from the keypad and press INCR, that value will be stored in memory and the program counter will be incremented.

“Dat” stands for Data. This is the current value of the memory location shown under “Addr.” When you press a key, that value will show up under “Dat” but it will not be saved in memory until you press INCR.

In **Run Mode**, your display has certain areas that you can access with instructions.


Program Counter
Used for Text Output (ASC1, ASC2)
ACCA Output (AO)
07
Asc1
5
01011001
1) Display individual numbers (SETN)
2) Display binary stored in memory (NOUT)

The Program Counter is always displayed on the first two spaces on the first line. The other locations of output are shown above. Note that only output you program will be seen on the display. When a program is run, the display is initially cleared.

The items in parentheses indicate the instructions that write to that particular screen location. You will see more about them later.

Speaker: A small speaker to play tones, special sounds, and beeps when the keys are pressed if desired.

Power Connector: This is micro USB connection to power the microprocessor board. Use the supplied cable and connect to a computer USB port or a USB phone charger.

Microprocessor Reset: This is the small button on the microprocessor board itself. If your microprocessor ever locks up, you can use this to reboot your system. All of the program memory will be erased so you will lose anything you were working on.

Whew, finally done with the information you need to get going. Let’s get into some programming. We will introduce the commands in groups and use them to build more and more complex programs.

Let’s go!

Section

2

Section 2: Programming

Let’s power up the trainer. Plug one end of the included USB cable into the power jack on your trainer and plug the other end into the USB port of your computer or into a USB phone charger. Be careful not to force the connectors into place. They should connect easily and feel secure when inserted.

The display will light and you will see the trainer boot up into **Program Mode**. This is where you will enter computer commands into memory to create your program. We’ll jump into this by introducing the first couple of commands.

Using TIA, AIA, AO, JMP

We have discussed the concept of an accumulator. The most important accumulator in our computer is the A accumulator (I will refer to it as ACCA for accumulator A). ACCA is a 4-bit accumulator so it can hold a single number from 0 to 15. ACCA is attached to memory and the ALU so it is used constantly in our programs to manipulate our data.

Let’s look at the first commands:

TIA – (Transfer Into ACCA)

Description	Load a number into ACCA
Machine Code	1
Operation	n→ACCA
Flag	1

We will show this table every time we introduce an instruction. Here is what the table entries mean:

Description: Quick explanation of what the instruction does.

Machine Code: The hex number or numbers that represents this instruction

Operation: A quick shorthand description of what the instruction does. In this case, it takes a number (n) and loads it (→) into ACCA.

Flag: This will show you the condition of the flag after the operation is complete. In this case, the flag is 1, which means the next instruction will be run. If the flag has a condition that will set it to zero, it will be shown.

So that is the TIA command. It simply loads a number into ACCA.

AIA – (Add Into ACCA)

Description	Adds a number to ACCA and stores the result in ACCA
Machine Code	3
Operation	ACCA+n→ACCA
Flag	0 if carry (if the add creates a number greater than 15)

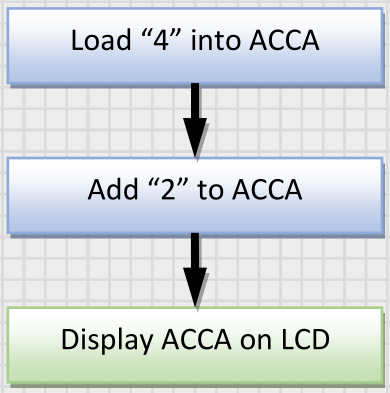
AO – (ACCA Output to LCD)

Description	Displays ACCA on the LCD during Run Mode
Machine Code	0
Operation	ACCA→LCD
Flag	1

JMP – (Jump to Memory Location)

Description	Loads an address into the Program Counter
Machine Code	F
Operation	nn→PC
Flag	1

With these four instructions, we can create our first program. Let's instruct the computer to add the numbers 4 and 2 and display the result on the LCD. Let's first generate a flow chart to accomplish this. This will help us generate our program.



From the flowchart, we can create our program. Each line lends itself to one of the instructions we have already introduced. The first block tells us we want to load the number 4 into ACCA. We need the TIA instruction. The second block adds a number to ACCA. That is the AIA instruction. The last one displays ACCA on the LCD. This is the AO instruction. This is a simple program, but it illustrates well the kind of thinking we need to do to write software. We want to add two numbers together. We can't just tell the computer to do this. We have to use the instructions to the computer to accomplish it. Let's convert these commands into actual code:

Program 1: Add Two Numbers

Memory Location	Command	Machine Code	Comment
00	TIA	1	Load 4 into ACCA
01		4	
02	AIA	3	Add 2 to the value in ACCA
03		2	
04	AO	0	Display ACCA on the LCD
05	JMP	F	End the program by just repeatedly looping here
06		0	
07		5	

So our entire program fits into 8 memory locations. Notice the blank Command entries for location 01, 03, 06, and 07. These are placeholders for the data we need to input. When we use the instruction TIA, we need to tell the computer what number to transfer into ACCA. The memory location directly after the command is where the computer knows to retrieve this number. This is true for the AIA instruction as well as many others. The "Words" column on your instruction card lists how many memory locations (words) the instruction takes. If you look at TIA, the card reports that it takes 2 words. That is one for the instruction, and one for the data needed by the instruction.

Let's enter this program into our trainer.

Start by hitting the **RSET** button. This stops operations, returns the Program Counter (PC) to 0 and places the trainer in Program Mode.

Now, enter the first command of our program. This is the number 1 which represents the command TIA. You will find all the commands to enter under the **Machine Code** column. Press the 1 key. Notice that a "1" now shows on the display under the "Dat" text. "Dat" stands for data and is the value of the memory location.

To enter this data into memory, press the **INCR** button (Increment). When you do this, the data will be stored in the current memory location, the PC will be incremented, and the display will show the updated address and data of the new address. The rest of the program will be entered in the same fashion.

If you make a mistake or need to go back, you can always press RSET and then INCR up to the address you need. This is also how you can verify your program one step at a time.

Now input the whole program: (press RSET first to start at 00)

1[INCR] 4 [INCR] 3 [INCR] 2 [INCR] 0 [INCR] F [INCR] 0 [INCR] 5 [INCR]

Remember to enter that last **INCR**. Just pressing a number doesn't change the memory location. You need to **INCR** to store the current data in the current location.

Let's run our program. First, press **RSET**. This is very important because the run command starts from the current position of the PC. This allows us to run different parts of our program, or even to run different programs stored in memory. Our program starts at 00, so press **RSET** to set the PC at 00. Now we can run the program.

Press **FUNC** and then 0. This runs the program in Continuous Mode. That means that one instruction after another will be run continuously.

What does the display show? You should see the number 6 on the right side of the display. That is the result of the addition in our program.

Let's learn another command.

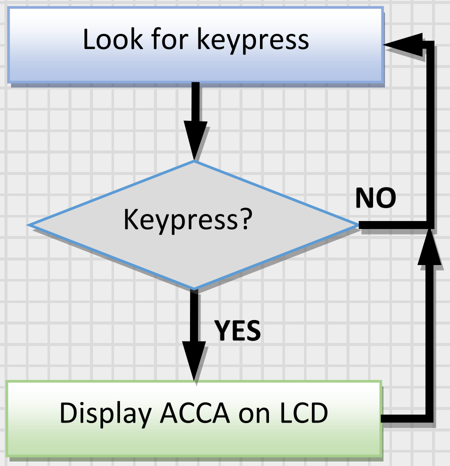
Using KIA

We need to have a way to enter information into our program during program operation. KIA allows us to do that. It is a command that scans the keypad and looks for a pressed key. If no key is pressed when the command is executed, the flag will stay set to 1 and the next command will be executed. If a key is pressed, the value will be transferred into ACCA and the flag will be set to zero. We can then use this flag to keep jumping back to the KIA command until a key is pressed. Let's see this in action.

KIA – (Keypad Into ACCA)

Description	Load a number from the keypad into ACCA
Machine Code	4
Operation	k→ACCA
Flag	0 if a key is pressed

We'll keep it simple again since this is just our second program. Let's create a program that will simply wait for us to press a key on the keypad and then display the result on the LCD. As before, we will start with a flow chart to help us collect our thought and to visually see the logic. We can then convert that flowchart to commands the computer can understand.



From the flowchart, we can create our program. The first block tells us we want to look at the keypad to see if a key was pressed. Since the KIA instruction has the ability to set the FLAG, we need to account for both possibilities we get from this command. The next box does this. It is a decision box. It is diamond shaped and gives us two possible routes to take. If a key is not pressed (denoted by the NO path), we send the program back to look again for a keypress. If a key is pressed (denoted by the YES path), we will display the contents of ACCA to the LCD and then jump to the start of the program. Let's convert these commands into actual code:

Program 2: Display Key Pressed

Memory Location	Command	Machine Code	Comment
00	KIA	4	Look for key press
01	JMP	F	Jump back to memory location 00 if no key is pressed
02		0	
03		0	
04	AO	0	If you get here, a key was pressed so output it to the display
05	JMP	F	Jump back to the beginning
06		0	
07		0	

Enter this program. Remember, start by pressing RSET to reset the computer and place the PC at 00. Enter by pressing the following:

4 [INCR] F [INCR] 0 [INCR] 0 [INCR] 0 [INCR] F [INCR] 0 [INCR] 0 [INCR]

Remember that last [INCR] to store the final instruction. Now run the program by pressing [RSET], [FUNC], then 0. You should now see any number you press on the keypad show up on the display.

Using XCH and DLAY

Let's take a look at two more instructions.

XCH – (Exchange ACCA/ACCB and ACCY/ACCZ)

Description	Switch the values in the A and B accumulators and the Y and Z accumulators
Machine Code	2
Operation	ACCA⇆ACCB and ACCY⇆ACCZ
Flag	1

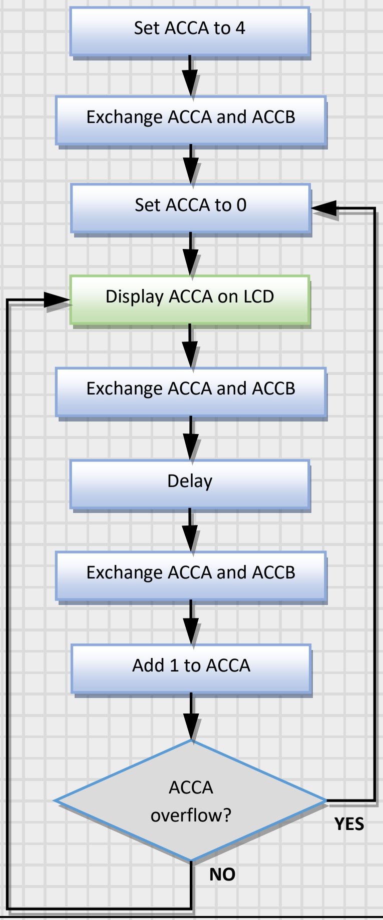
DLAY – (Delay)

Description	Delay program execution based on ACCA
Machine Code	E0
Operation	Delay (sec) = 0.1 * (ACCA + 1)
Flag	1

ACCA is a very busy accumulator. Many of the instructions use it as a reference for their operation. Because of this, it is handy to have a temporary place to hold the contents of ACCA while we need it to do other things with it. That is what the XCH instruction is for. It swaps the contents of the A and B accumulator. The sole function of the ACCB is to temporarily hold the contents of ACCA. XCH also swaps ACCY and ACCZ at the same time.

There are also sometimes in our program where we want to slow the program execution. The computer executes instructions very rapidly. If we want to display information to the user, we may need to wait to give the user a chance to read it.

Let's write a program to count from 0 to F and then recycle. Here is what the flow chart may look like:



In order to create a delay, we need a number in ACCA. But we also need ACCA to track the current count we are on. This is where ACCB will come in handy. We will load the delay value in ACCB and then just exchange ACCA and ACCB when we need to. After we put the value 4 into ACCA and exchange it into ACCB, we load ACCA with our starting number, 0. We then display the value in ACCA. Here is where we want our delay so the user can actually see this count before it changes. We exchange ACCA and ACCB to save the value in ACCA and to load the delay value into ACCA. We then call the delay routine and the exchange ACCA and ACCB again to restore ACCA to the current count. We add 1 to ACCA and look to see if there is an overflow (ACCA>15). If there is none, loop back and display the new number. If there is an overflow, loop back to where we set ACCA to 0 so we can start the count again from there. Let's convert this flow chart to a program.

Program 3: Count from 0 to F and Recycle

MEM	Cmd	MC	Comment
00	TIA	1	Load 4 into ACCA as our delay factor
01		4	
02	XCH	2	Switch ACCA and ACCB
03	TIA	1	Load 0 into ACCA so the count starts there
04		0	
05	AO	0	Display ACCA
06	XCH	2	Switch ACCA and ACCB so the delay value is in ACCA
07	DLAY	E	Call the Delay routine
08		0	
09	XCH	2	Switch back so ACCA has the current number
0A	AIA	3	Add 1 into ACCA to increment the count
0B		1	
0C	JMP	F	If there is no overflow, jump back to display new number
0D		0	
0E		5	
0F	JMP	F	If there is overflow, jump back to where we set ACCA to zero
10		0	
11		3	

You can now enter this program. Remember to RSET first.

[RSET] 1 [INCR] 4 [INCR] 2 [INCR] 1 [INCR] 0 [INCR] 0 [INCR] 2 [INCR] E [INCR] 0 [INCR] 2 [INCR] 3 [INCR] 1 [INCR] F [INCR] 0 [INCR] 5 [INCR] F [INCR] 0 [INCR] 3 [INCR]

(make sure to remember that last [INCR] to save the last instruction)

Now run the program by pressing: **[RSET] [FUNC] 0**

You should see the numbers 0 to F appear in the ACCA output segment of the display. When the count gets to F, you should see it recycle back to 0 and start the count again. If you don't see this, check your program. The easiest way to do this is to [RSET] the computer and [INCR] through the memory locations while comparing the data to the needed program values.

Using TIY, AIM, INCY, M+, M-

Let's take a look at a few more instructions.

TIY – (Transfer Into ACCY)

Description	Transfer a value into the Y accumulator
Machine Code	9
Operation	n→ACCY
Flag	1

AIM – (Transfer ACCA Into Memory)

Description	Transfer the value in ACCA to the memory location pointed to by ACCY
Machine Code	5
Operation	ACCA→M[ACCY]
Flag	1

The upper 16 memory locations (addresses F0 thru FF) are for your use to store data. We will use it to pre-store data for programs and as a holder for data we will need later. AIM will load whatever data is in ACCA into the memory location pointed to by ACCY. That means if you want to load data into memory location F3, you load 3 into ACCY and execute the AIM instruction. The instruction will load the contents of ACCA into memory location F3.

INCY – (Increment ACCY)

Description	Add 1 to ACCY. If ACCY >15, assign ACCY to 0
Machine Code	A
Operation	ACCY+1→ACCY
Flag	0 if carry

M+ – (Add Memory to ACCA)

Description	Add the data in memory location pointed to by ACCY into ACCA
Machine Code	7
Operation	ACCA + M[ACCY]→ACCA
Flag	0 if carry

M- – (Subtract Memory from ACCA)

Description	Subtract data in memory location pointed to by ACCY from ACCA
Machine Code	8
Operation	ACCA – M[ACCY]→ACCA
Flag	0 if negative (result is a negative number)

The M- instruction is used for performing subtraction operations. If the value at the memory location is less than or equal to ACCA, then the result of ACCA-M[ACCY] operation will be a positive number or zero. If M[ACCY] is bigger, then the result will be a negative number. Computer hardware and software have ways to handle negative numbers. For our computer, we are going to adopt a scheme where we subtract the resulting negative number from 16 and store it in ACCA. This is so the M+ and M- complement themselves. If

ACCA is 0 and we execute the M- instruction to subtract 1 from ACCA, this will result in an ACCA value of 15 (16-1). This is good, because if we then M+ and add 1 to ACCA, we get back to 0. Both of these will set the FLAG to zero so you can test for the carry (M+) and the negative number (M-).

Let's use these new instructions to learn some different concepts of memory manipulation. Our program will:

1. Transfer 5 to ACCA
2. Save ACCA to M[0] (memory location zero, or F0 in hex)
3. Add 3 to ACCA
4. Save ACCA to M[1] (memory location one, or F1 in hex)
5. Add M[0] to ACCA
6. Save ACCA to M[2] (memory location two… you get the point)
7. Subtract M[0] from ACCA
8. Print ACCA

At the end of running this program, the display should read 8 and the memory locations should have the following values:

M[0] = 5, M[1] = 8, M[2] = 13

Let's create this program and see if that is what we get.

Program 4: Manipulate Memory Locations

MEM	Cmd	MC	Comment
00	TIA	1	Load 5 into ACCA
01		5	
02	TIY	9	Set ACCY to point to M[0]
03		0	
04	AIM	5	Transfer ACCA to memory M[0]
05	AIA	3	Add 3 to ACCA
06		3	
07	INCY	A	Increment Y to point to the next location. We won't let ACCY overflow, so we don't need to handle that possibility here.

08	AIM	5	Transfer ACCA to M[1]
09	TIY	9	Have ACCY point to M[0]
0A		0	
0B	M+	7	Add M[0] to ACCA
0C	TIY	9	Set up ACCY to point to M[2]
0D		2	
0E	AIM	5	Transfer ACCA to memory M[2]
0F	TIY	9	Set up ACCY to point to M[0]
10		0	
11	M-	8	Subtract M[0] from ACCA
12	AO	0	Print ACCA to LCD
13	JMP	F	End the program by just looping here
14		1	
15		3	

Enter the program: (remember the increments after each entry)

1,5,9,0,5,3,3,A,5,9,0,7,9,2,5,9,0,8,0,F,1,3

When you run the program ([RSET], [FUNC], 0) you should see an 8 on the display. Now we want to check the memory locations. Press [RSET] to stop program execution. The user memory starts at F0, so press [ADDR] and then F and 0. The display will now show the address F0 and the value at that address which should be 5. Press [INCR] to see address F1. It should be 8. Press [INCR] one more time to show D (decimal 13) as the value for F2. By using the [ADDR] function we can view the value of any memory location (program and user memory). This will allow us to preposition data for use in our program as well as see the results of calculations. Let’s make a program that requires us to load data into memory before program execution. Let’s make some music!

Using MIA and SOND

Let’s take a look at a few more instructions.

MIA – (Memory Into ACCA)

Description	Transfer the value in the memory location pointed to by ACCY to ACCA
Machine Code	6
Operation	M[ACCY]→ACCA
Flag	1

SOND – (Sound Out)

Description	Play a note based on the value in ACCA
Machine Code	E5
Operation	SOUND[ACCA]→SPKR
Flag	1

Let’s create a program that takes pre-loaded notes in memory and plays them on the speaker.

Program 5: Play Music

MEM	Cmd	MC	Comment
00	TIY	9	Set ACCY to point at the first memory location
01		0	
02	MIA	6	Transfer the value of memory to ACCA
03	SOND	E	Play the note for the value of ACCA
04		5	
05	INCY	A	Increment to the next memory location
06	JMP	F	More notes to play, jump back to play them
07		0	
08		2	
09	JMP	F	If INCY causes a carry, you are out of memory. Jump in a loop to stop program
0A		0	
0B		9	

Enter the above program using the usual method entering and incrementing the values in the "MC" column but don't run it yet. Our program is going to cycle through the 16 user memory locations and play the notes associated with each value. That means we want to load notes into memory. Do this by pressing [ADDR] then F then 0. You are now pointed at the first memory location. Just like entering program data, user data is stored the same way. Have fun randomly entering values into memory by selecting a number on the keypad and then pressing [INCR] until you finish at address FF and it resets you back to the beginning of program memory (so stop and don't write over your program!).

Now that you have entered your song, run the program (press [RSET] [FUNC] 0). You should hear your song play and stop at the end. If you want to continuously repeat, you could change the JMP address at 0B to 2. This would jump back to the start instead of just cycle on itself.

Change up the memory and try different songs. This exercise has showed that the computer sees its program instructions and user memory as the same thing. It is up to the user to keep track of what is what when you program on the machine language level. When you use high level languages, you can instruct the computer to do a lot of this memory management for you. For instance, you may create a “variable” called “Value” and then assign it various numerical values. The computer would then assign a memory location and use that location anytime your program assigns a value to or reads a value from your variable “Value.”

We also talk about ACCY “pointing to” memory locations. This is another very common high level language use. A **pointer** holds the memory address of a variable. If I ask my program to retrieve the value in a variable I use the variable name. If I want my program to give me the memory location of this data, I ask for the pointer to that variable. We often need to do this because an operation requires the address and not the data. In our computer, ACCY points to our memory location F0-FF. M+, M-, AIM, and MIA all use this pointer to locate which memory location to put or pull data to/from.

The bottom row of the display is used to create an 8-bit output for you to use in your programs. It can be used to turn on and off individual bits or display a number from memory. Let’s check this out.

Using SETN, RSTN, and CMY

SETN – (Set Number on LCD)

Description	Turns on the lower LCD line number held in ACCY
Machine Code	E1
Operation	ACCY→LCD_NUMBER
Flag	1

RSTN – (Reset Number on LCD)

Description	Turns off the lower LCD line number held in ACCY
Machine Code	E2
Operation	ACCY→CLEAR[LCD_NUMBER]
Flag	1

CMY – (Compare ACCY)

Description	Compares a number to ACCY
Machine Code	B
Operation	n-ACCY→FLAG
Flag	0 if number equals ACCY

Let’s write a program that turns on the lower numbers one at a time and then repeats when it gets to the end.

Program 6: Count Using the LCD Number Output

MEM	Cmd	MC	Comment
00	TIA	1	Load 4 into ACCA to be used by TIMER
01		5	
02	TIY	9	Set ACCY to Set the first number
03		0	
04	SETN	E	Turn on the current number
05		1	
06	DLAY	E	Wait a bit
07		0	
08	RSTN	E	Turn off current number
09		2	
0A	INCY	A	Increment ACCY to point to M[1]
0B	CMY	B	Check to see if we are at the max number
0C		8	
0D	JMP	F	Not at max, jump back to display next number
0E		0	
0F		4	
10	JMP	F	At max, jump to reset ACCY to 0 and start over
11		0	
12		2	

Enter and run this program. You should see the numbers appear on the lower part of the LCD display. You will see that the numbers start at the left and increase as you move right. These are good for displaying moving displays (like this program) or for fun games (roulette is coming later). Another option you have for controlling this bottom line of the LCD is with the instruction NOUT. NOUT will display the binary representation of the memory location FF on the first four numbers and the binary value of memory location FE on the second four as shown below:

Binary number 2				Binary number 1			
8	4	2	1	8	4	2	1
M[FE]				M[FF]			

Here, the numbers start on the right (least significant digit) and get bigger as you move to the left (most significant digit). Let's do another program and see how this works. Let's make a clock.

Using NOUT and CMA

NOUT – (Number Out)

Description	Displays the numbers in memory location FE and FF on the lower LCD as two 4-bit binary numbers
Machine Code	EF
Operation	M[FF]M[FE]→LCD_D2/LCD_D1
Flag	1

CMA – (Compare ACCA)

Description	Compares a number to ACCA
Machine Code	C
Operation	n-ACCY→FLAG
Flag	0 if number equals ACCA

Remembering back to our discussion of binary numbers we showed how our hexadecimal numbers can all be displayed with a 4-bit binary number. So 1 is 0001, 2 is 0010, 3 is 0011, and so on. We have 8 place holders on the lower LCD line. This space can be used to display two 4-bit numbers. When you use the NOUT command, the value in memory location FE will be sent to the first 4-bit number (located on the right of the display, and the value in memory location FF will be sent to the second 4-bit number (the left most 4 bits). We will create a timer that will start counting minutes and seconds. We will use the NOUT command to show the seconds as a two binary numbers on the lower display, and the number of elapsed minutes on the LCD using the AO instruction. We'll store the minutes in M[D]. Let's roll!

Program 7: A Simple Clock

MEM	Cmd	MC	Comment
00	TIA	1	Load 4 into ACCA to be used by TIMER
01		5	
02	XCH	2	Free up ACCA for use
03	TIA	1	Transfer 0 to ACCA to initialize data
04		0	
05	TIY	9	Point to M[D]
06		D	
07	AIM	5	Load 0 into M[D]
08	TIY	9	Point to M[E]
09		E	
0A	AIM	5	Load 0 into M[E]
0B	TIY	9	Point to M[F]
0C		F	
0D	AIM	5	Load 0 into M[F]
0E	TIY	9	Point to M[D]
0F		D	
10	MIA	6	Transfer minutes into ACCA
11	AO	0	Display minutes
12	NOUT	E	Display seconds
13		F	
14	XCH	2	Recall the Delay
15	DLAY	E	Insert Delay
16		0	
17	XCH	2	Recall the ACCA value
18	TIY	9	Point to M[F]
19		F	
1A	MIA	6	Transfer M[F] to ACCA
1B	AIA	3	Add 1 to the seconds

1C		1	
1D	JMP	F	There won’t be an overflow so just jump through
1E		2	
1F		0	
20	CMA	C	See if ACCA > 9
21		A	
22	JMP	F	ACCA not > 9 cycle back to display new time
23		0	
24		D	
25	TIA	1	Seconds are >9, let ACCA = 0 and then we will add one to M[E]
26		0	
27	AIM	5	Right digit of seconds = 0
28	TIY	9	Point to M[E]
29		E	
2A	MIA	6	Transfer left digit of seconds into ACCA
2B	AIA	3	Add 1 to the left seconds digit
2C		1	
2D	JMP	F	We are about to check if this number is 6 and then reset it so we won’t overflow so just fall through to next instruction
2E		3	
2F		0	
30	CMA	C	Compare left minute digit to 6
31		6	
32	JMP	F	If it is not 6, recycle back to show the new time value
33		0	
34		D	
35	TIA	1	When we hit 60 seconds, we reset the seconds and add one to the minutes
36		0	
37	AIM	5	
38	TIY	9	Point to M[D]
39		D	
3A	MIA	6	Load M[D] into ACCA

3B	AIA	3	Increment minutes by 1
3C		1	
3D	JMP	F	If there is an overflow, the count will automatically roll to zero so just return to display for either result
3E		0	
3F		D	
40	JMP	F	Jump back to start next cycle
41		0	
42		D	

Load and run this program. You will see the bottom 8 LCD locations display seconds, with the right 4 locations showing the binary value of the least significant digit of seconds, and the left 4 locations showing the binary value of most significant digit of seconds. The ACCA output will show the elapsed minutes. If you let the counter run, when the binary value gets to 0101 1001 (binary for 59), the counter will roll over to 0000 0000 on seconds and add one to the minute display. You just programmed a timer. Knowing how the program used the memory to track minutes and seconds, how might you run this program with an initial value for the timer (not start at zero)?

If you remember, we stored the seconds in M[E] and M[F], and the minutes in M[D]. We could change the program to not initialize these locations to 0 and before running the program, load the desired values into these memory locations and the timer would start from these settings.

Let's look at a quick program to demonstrate some of the pre-built sounds you can use in your programs.

Using ERRS, SHTS, LNGS and ENDS

ERRS – (Error Sound)

Description	Play Error sound
Machine Code	E6
Operation	ERR→SPKR
Flag	1

SHTS – (Short Sound)

Description	Play a short sound
Machine Code	E7
Operation	SHT→SPKR
Flag	1

LNGS – (Long Sound)

Description	Play a long sound
Machine Code	E8
Operation	LNG→SPKR
Flag	1

ENDS – (End Sound)

Description	Play End sound
Machine Code	E9
Operation	END→SPKR
Flag	1

These instructions give you immediate access to sounds that you can use in your programs without having to use the SOND instruction and a lot of code. The SHTS and LNGS instructions just cause the speaker to emit a short or long sound. You can use this to indicate different operations in your program or use them in games. ERRS is a negative sounding set of tones. You can use this to indicate some error (like a bad guess in a game). ENDS is a little positive sounding set of tones to indicate a right choice or the end of a program. Use these as you like in your programs. We will write a quick program that just plays these sounds when particular keys are pressed. This will give you a chance to hear them all and to get more practice in compare and jump operations.

Program 8: Play Special Sounds

MEM	Cmd	MC	Comment
00	KIA	4	Look for a key input
01	JMP	F	If there is no input, jump back to the start to look again
02		0	
03		0	
04	CMA	C	Is 0 pressed?
05		0	
06	JMP	F	0 wasn't pressed so jump to next check
07		0	
08		E	
09	SHTS	E	0 was pressed so play SHTS
0A		7	
0B	JMP	F	Jump back to the start
0C		0	
0D		0	
0E	CMA	C	Is 1 pressed?
0F		1	
10	JMP	F	1 wasn't pressed so jump to next check
11		1	
12		8	
13	LNGS	E	Play LNGS
14		8	
15	JMP	F	Jump back to start
16		0	

17		0	
18	CMA	C	Is 2 pressed?
19		2	
1A	JMP	F	2 wasn't pressed so jump to next check
1B		2	
1C		2	
1D	ENDS	E	Play ENDS
1E		9	
1F	JMP	F	Jump back to start
20		0	
21		0	
22	ERRS	E	Default to playing ERRS
23		6	
24	JMP	F	Jump back to start
25		0	
26		0	

Load and run this program. Pressing 0 will sound the Short Sound (SHTS), 1 will sound the Long Sound (LNGS), 2 will sound the End Sound (ENDS), and any other key will sound the Error Sound (ERRS). Play with the program to get to know the tones and then use them in your programs.

Let's take a short break from programming and look at how modern programming still uses the concepts you see in our machine language programming on our 4-bit computer.

Programming Concepts

First, let's define and explain different levels of programming.

Machine Language

Machine language is a set of instructions that are executed directly by a computer's CPU. It is the lowest level of computer programming in that it requires the direct manipulation of data and memory. You are programming in machine language when you enter instructions in your computer. When you enter 1 into a memory location or your 4-bit computer, you are telling it to transfer the number in the following memory location into ACCA. The computer understands this because 1 is the instruction hard-wired into the CPU to accomplish this.

However, if you talk to a seasoned computer programmer and tell him you are executing instruction 1, he would have no idea what that accomplishes. Programming on the machine level creates a code that is hard to read, hard to troubleshoot, and complicated to produce. However, it does put the user in full control of all aspects of memory usage and program flow. But because of the disadvantages, it is seldom used for programming. But, ultimately whatever method you use to write the program, the program will have to be converted to machine language because it is the only language our machine speaks.

Assembly Language

This is a low level, symbolic code used to make writing programs easier for a user. We have already seen this in a simple form. The machine language instruction 1 stands for Transfer into ACCA. We have used a shorthand expression for this called TIA. By this time, you are probably use to the instruction and by seeing the term TIA know what it means to the computer. By having these representative symbols, it is easier to read code, troubleshoot problems, and explain coding to others. Many programmers will still write some of their programs in Assembly Language because it has the advantage of being easier to read and understand, but also has the advantage of machine language in that it gives the user full control of the manipulation of data in the computer. Ultimately, the assembly code must be made into machine code for use on the computer. This is the job of an **Assembler**. An assembler is a program that takes the code written in assembly language and converts it to machine code so the computer can run it.

High Level Languages

High Level Languages are programming languages like Java, C++, Visual Basic and others that are written in more like human language than machine language. These have the advantage of being easy to read, write, troubleshoot, and share code segments than the other languages. They reduce many code-intensive operations to a single, simple command. Like all programming, high level code must be translated into machine language to run on a computer. This is the job of the **Compiler**. A compiler is a program that takes the high level code (often called source code) and converts it to machine code (often called object code). One of the major advantages of high level languages is that they can be programs written independent of the type of computer that they will eventually run on. C++ is a highly structured language. I can write a program in C++ and then use a compiler to allow it to be run on a Personal Computer. I can then take the same C++ code, run it through a different compiler and run it on a cell phone. In each case, you just need the right compiler to transform your code for the specific platform you want to run it on.

Computer Language Example

Let's look at a really simple program to see the differences in these languages. Let's write a program to put the number 5 in a memory location, and then output the value of 4 added to that memory to a display. We'll start with the high level language. We'll use a language called **Basic**.

```
A=5
Print A+4
```

Yep, that is it; two lines of code. The first line (A=5) establishes a memory location and gives it a name of 'A' so it can be referenced by that name in our program. It also assigns that memory location the value 5. The next line adds 4 to the value at A and prints it to a screen. You can see how the code is very readable and can be understood quickly. Let's look at what it would look like in assembly language.

```
TIY    0
TIA    5
AIM
TIA    4
M+
JMP    0B
AO
```

This is the kind of programming you have been doing. You can see that this program accomplishes the same thing, but is a little longer and a little harder to understand. We can see the first line (**TIY 0**) loads ACCY with 0 so we can point to a memory location. We then load ACCA with 5 (**TIA 5**) and then place that (**AIM**) in M[0]. We then transfer 4 into ACCA (**TIA 4**) and then add (**M+**) the number in M[0] to it. We add the Jump command (**JMP 0B**) in case of an overflow condition. Finally, we output this value to the display (**AO**). Because the instructions are represented with symbols, we can still follow what is going on, but it is more difficult than the Basic program. Finally, let's look at this in machine code. You have done this when you converted the assembly symbols into machine instructions.

```
9 0 1 5 5 1 4 7 F 0 B 0
```

This is the program that the computer will actually run. If you handed this to a programmer, she wouldn't know what to make of it. You can see, the closer we get to the language the computer understands, the less we can easily read it. Let's continue exploring our computer.

Using RAND, ASC1, and ASC2

RAND – (Random Number)

Description	Loads a random number from 0 to F into ACCA
Machine Code	EE
Operation	RAND→ACCA
Flag	1

RAND loads a random number into ACCA. We can use it for games, lighting displays, all sorts of things we want to program. We'll use this in a guessing game.

ASC1 – (ASCII1 out to LCD)

Description	Displays ASCII characters in M[0]-M[7] to LCD
Machine Code	EB
Operation	ASCII(M[0]-M[7])→LCD
Flag	1

ASC2 – (ASCII2 out to LCD)

Description	Displays ASCII characters in M[8]-M[F] to LCD
Machine Code	EC
Operation	ASCII(M[8]-M[F])→LCD
Flag	1

Ever wonder how we decided to take 1's and 0's and have computers interface with humans. We use a translator to convert numbers to letter so computers can display words to use. The most widely used standard for this translation is what is known as **American Standard Code for Information Interchange** or **ASCII**. This is a table of alphanumeric and other symbols with an associated number for each. We use these numbers to tell the computer what letters we want. An ASCII table can be found in Appendix A. What ASC1 and ASC2 do is take 4 sets of hex numbers stored in memory and present them as their ASCII characters on the LCD. This allows us to have simple 4 letter words in our program. By using memory for storing these words, we save a lot of programming time by having to load each memory location individually, but we have to remember to set these memory locations with the correct values before we run the program. To complicate things further, we need to use one memory location for holding the secret number in our guessing game while we use the ACCA to get your guess so we will need to keep this in mind. Let's look at how to use our resources to accomplish this.

We need to display the words "Low" and "High" during the game. Since "Low" has three letters, we can use memory location zero (M[0]) to hold the secret number, making sure that we save this value and set the memory location to display a space when we need to show the text "Low." For the correct guess, we can load the text "Yes!" into memory and display it. Finally, we will load the ASCII values of the text into memory before we run the program to prevent having to do this during program run time which would significantly lengthen the code. Let's get programming. This is a long program so take your time entering it.

Program 9: Guessing Game

MEM	Cmd	MC	Comment
00	RAND	E	Put secret number into ACCA
01		E	
02	TIY	9	Have ACCY point to M[0]
03		0	
04	AIM	5	Load our secret number into M[0]
05	KIA	4	Wait for the user to enter a guess
06	JMP	F	
07		0	
08		5	
09	M-	8	Subtract the secret number from the guess
0A	JMP	F	If there is no overflow, the guess was bigger than or equal to the secret number. Jump to code to handle each case.
0B		1	
0C		9	

0D	MIA	6	If you are here, guess was too low. Transfer the secret number into ACCA and then XCH so it is kept safe in ACCB.
0E	XCH	2	
0F	TIA	1	Now we need to load a 2 into M[0] so it will complete the ASCII code for a space (20) so we can display "Low" on the LCD.
10		2	
11	AIM	5	Do the transfer
12	ASC1	E	Print "Low" to the LCD
13		B	
14	XCH	2	Now we need to put the secret number back. XCH to get it back into ACCA and store it in M[0].
15	AIM	5	
16	JMP	F	Jump back to get another number from the user.
17		0	
18		5	
19	CMA	C	You are here because the guess was either equal to or higher than the secret number. Let's check for the equal first. That would mean that the M- command resulted in a zero left in ACCA.
1A		0	
1B	JMP	F	It is not zero, so jump to display "High."
1C		4	
1D		5	
1E	TIY	9	If you are here, the user guessed the secret number and the game is over. There is just one more thing to do. We need to load "Yes!" into memory so we can display this text with our ASC2 command. This whole block will just alternately load the ASCII code into ACCA and then into memory. It will then move to the next memory location and repeat until all the text is ready. Then we call ASC2 to display it on the LCD.
1F		8	
20	TIA	1	
21		5	
22	AIM	5	
23	INCY	A	
24	TIA	1	
25		9	
26	AIM	5	
27	INCY	A	
28	TIA	1	
29		6	

2A	AIM	5	
2B	INCY	A	
2C	TIA	1	
2D		5	
2E	AIM	5	
2F	INCY	A	
30	TIA	1	
31		7	
32	AIM	5	
33	INCY	A	
34	TIA	1	
35		3	
36	AIM	5	
37	INCY	A	
38	TIA	1	
39		2	
3A	AIM	5	
3B	AIM	5	
3C	INCY	A	
3D	TIA	1	
3E		1	
3F	AIM	5	
40	ASC2	E	Whew. That was a lot of code to load 8 numbers in memory but it is done. Display the result.
41		C	
42	JMP	F	The game is over so just entering an endless loop and wait for the user to press reset.
43		4	
44		2	
45	ASC2	E	If you are here, the guess was too high. Display the "High" text to the user.
46		C	
47	JMP	F	Jump back to get another guess from the user.
48		0	

49		5	

Wow, that was a lot of code! After you enter this, I recommend that you verify the code by hitting RST and then INCR through the program to check the memory locations match the ones in the table above. This is always a good thing to do before you run a program. It is a painful lesson to take a lot of time to enter a program, only to run it and have it lock up. Sometimes the only way to recover is to reset and lose all of the work you have done. It is good to save frequently while programming.

We have one more thing to do before we play our game. We have to load the text "Low" and "High" into memory. Go to address F0 and enter the following:

Memory Location	Value	ASCII Character
F0	2	"space"
F1	0	
F2	4	L
F3	C	
F4	6	o
F5	F	
F6	7	w
F7	7	
F8	4	H
F9	8	
FA	6	i
FB	9	
FC	6	g
FD	7	
FE	6	h
FF	8	

Now would be a great time to save this program. Save it into one of the memory locations by entering FUNC 4 for memory bank 1 or FUNC 6 for memory bank 2.

Now, run the program. The program will wait for you to enter a guess. When you do, it will tell you whether you are too low or high, or will display “Yes!” when you are correct. When you are done with the game, the memory locations M[7]-M[F] do not have the correct text in them (they have “Yes!” instead of “High”). If you want to play again, you will need to reload those memory locations with the correct text. Now aren’t you glad you saved the program? When you save, not only is the program data saved but the user memory is saved too. Just reload (FUNC 5 or 7) the program and run it again.

Using SHFT AND CLRA

SHFT – (Shift ACCA)

Description	Shift the binary bits stored in ACCA one place to the Right
Machine Code	D
Operation	ACCA/2→ACCA
Flag	0 if a 1 is shifted out

This SHFT function operates on the binary value stored in ACCA. It shifts all of the ones and zeros to the right one place. So the value A, which is 1010 in binary, when shifted will be 0101. We move all of the digits one place to the right and insert a zero in the left most digit. If the digit that is shifted out to the right is a 0, the Flag remains at 1. If the digit shifted out is a 1, the Flag is set to 0. There are a couple of things going on here. If you look at our original number A (1010) and the final number after the shift 5 (0101), you can see that the shifting operation has divided the original number by two. This is the same thing we do when we move the decimal point of a number. Moving the decimal point to the left and right in base 10 causes us to multiply or divide by 10. When we do this in binary, we multiply or divide by 2. So we could use this instruction to divide a number by two.

The other thing to note is our Flag. If the number shifted out is a 0, the Flag is 1 and if the number shifted out is a 1, the Flag is set to zero. This right most digit is our one's place, so if we shift out a 1, the original number must have been odd. If we shift out a 0, the original number must have been even. So the flag will indicate if the number is odd or even. We will use this in our program.

CLRA – (Clear ASCII)

Description	Clear the LCD of any ASCII out text
Machine Code	ED
Operation	clear→LCD ASCII
Flag	1

Let's use these instructions in a program that will display whether a key we press is odd or even.

Program 10: Odd or Even

MEM	Cmd	MC	Comment
00	KIA	4	Wait for user input
01	JMP	F	
02		0	
03		0	
04	SHFT	D	Shift ACCA Right
05	JMP	F	The Flag is 0 so the number is even. Jump to display "Even"
06		0	
07		D	
08	ASC1	E	Display "Odd"
09		B	
0A	JMP	F	Jump to our subroutine that handles the delay and screen clear
0B		0	
0C		F	
0D	ASC2	E	Display "Even"
0E		C	

0F	TIA	1	Enter a value for the DLAY routine
10		5	
11	DLAY	E	Wait a bit
12		0	
13	CLRA	E	Clear the LCD
14		D	
15	JMP	F	Go back and get another number
16		0	
17		0	

Let’s load our text into memory as before.

Memory Location	Value	ASCII Character
F0	4	O
F1	F	
F2	6	d
F3	4	
F4	6	d
F5	4	
F6	2	“space”
F7	0	
F8	4	E
F9	5	
FA	7	v
FB	6	
FC	6	e
FD	5	
FE	6	n
FF	E	

Now when you run this program, you will see the computer waiting for input. When you press a key, the output will show either “Odd” or “Even” for a short time and then the screen will clear.

Using CMPL and RSTO

CMPL – (Compliment ACCA)

Description	Take the compliment of ACCA
Machine Code	E4
Operation	F-ACCA→ACCA
Flag	1

RSTO – (Reset Output)

Description	Reset the LCD Output
Machine Code	E3
Operation	clear→AO
Flag	1

The compliment of a binary number is when you make all of the 1’s zeros and all of the 0’s ones. So 0110 becomes 1001. This gives the same result as if you subtract ACCA from 15.

RSTO is a reset to erase the data printed by AO.

Let’s use these in a quick program to display the compliment of a user input for a short time.

Program 11: The Compliment of an Input

MEM	Cmd	MC	Comment
00	KIA	4	Wait for user input
01	JMP	F	
02		0	
03		0	

04	CMPL	E	Compliment ACCA
05		4	
06	AO	0	Display ACCA
07	TIA	1	Delay a bit
08		5	
09	DLAY	E	
0A		0	
0B	RSTO	E	Clear the Display
0C		3	
0D	JMP	F	Go back to get another user input
0E		0	
0F		0	

Load and run this program. When you press a key, the computer displays the compliment of the number for a short time and then clears the display.

Using CHNG

CHNG – (Exchange Accumulators)

Description	Swap all accumulators with temporary accumulators
Machine Code	EA
Operation	A⇆A’ B⇆B’ Y⇆Y’ Z⇆Z’
Flag	1

If you want to see this instruction in action, look at Sample Program 5 (Roulette). We use this when we need for memory manipulation intensive programs. Executing this instruction will swap all accumulators (ACCA, ACCB, ACCY, and ACCZ) with temporary alternate memory locations. This allows you to have two complete sets of accumulators so you can do operations on memory, and then shift to the alternate set to free the accumulators to do keyboard and display operations, and then shift back to your memory operations. This way you don’t need to store a lot of data between shifts.

That is all of the instructions.

Well done. You now have everything you need to do your own programming with your computer.

Now, let’s look at an example of how to take an idea and turn it into a program.

Section

3

Section 3: Writing Programs

It is one thing to enter a program that someone else wrote. It is another to actually generate programs yourself. But, the real fun of computers is their versatility. If there is something you need the computer to do, you can program it to do that. Computer programming is a huge and complex topic. There are many techniques and strategies for coming up with reliable and reusable code. We are going to just touch the surface of this topic.

Let’s start at the beginning.

The Concept

The first thing you need is an idea. Let’s look at the following scenario: You want to play a game that you found in your closet, but you cannot find the die that is required to roll in the game to generate a random number from 1 to 6. That is when you remember that your computer can generate random numbers. You get the idea that you could write a program to act as the die in your game. You envision the computer waiting for you to press a key, and then the computer displaying a random number from 1 to 6, and then waiting for another key press. Congratulations, you have your concept.

Now you need to start taking the idea from human language to computer language.

The Outline

Now start to convert your concept into a structured outline. This is a simple way to start moving from the unstructured creative thinking to more logical steps to accomplish the concept.

Let’s break the concept down into major functional blocks:

1. Have the computer select a random number.
2. Ensure that number is from 1 to 6.
3. Wait for the user to hit a key.
4. When the user hits a key, display the number.
5. Jump back to start again.

This looks really simple, but it is what we need to collect our thoughts into logical steps.

Resource Allocation

Now that we have a layout, we can start to determine what resources we will need in the computer. Let’s formalize this structure by laying out requirements and constraints:

1. We need a place to hold the random number.
2. We cannot use ACCA because it will receive input from the key press and overwrite the number.
3. We need some part of the display for the output.

These points are starting to solidify the decisions that have to be made in our programming. For instance, I can use ACCB to temporarily hold the number using XCH while I wait for the key press from KIA. Alternately, I could decide to use a memory location. Either is straight forward, but you can see, each uses a different resource in the computer. If part of our concept included printing text to the screen, we would probably opt for the XCH option with ACCB so we could leave the memory free to output text with ASC1 and ASC2.

Also, using a memory location will require the use of ACCY to point to the desired location.

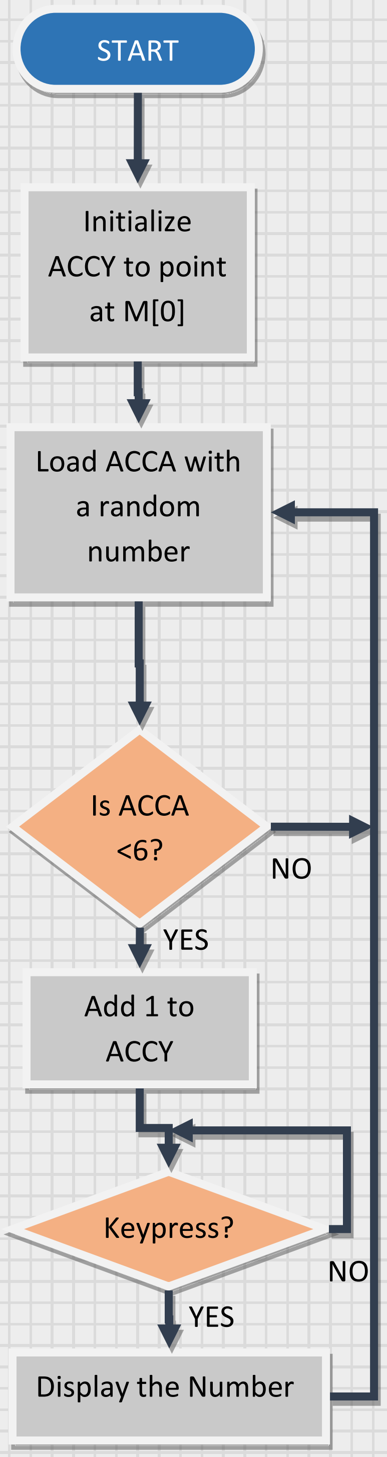
So let's make those decisions and solidify our resource usage.

1. We will use M[0], and by necessity ACCY as the pointer to hold the secret number.
2. We will output using AO to the LCD display

Alright, with our resources determined, we can start to build our program.

Flow Chart

Let's now shift to a time honored way of organizing our resources and concept into a logical sequence of events. Flow charts are great for this. A flow chart is just a graphical representation of the flow of logic and instructions in a computer program. It will be what we use to actually produce our code. Let's look at how it might be laid out.



Flowcharts often begin with a Terminator (a block that starts or stops a process). Let's use one that says "START" to begin our design.

This is a Process block. It is used for things that need to be done. We will use this block to setup our initial conditions. Let's make sure ACCY is pointing to the memory location we want to use to store the number.

Here is where we start the main part of our program. We load a random number into ACCA. This number will be from 0 to 15, so we need to do some work to get it in the range we want.

Now we come to a Decision block. The Decision block tests a condition and then sends us off in the direction of the answer. We need a number with 6 possibilities. That means our number should be from 0 to 5. If it is too big, go back and get another.

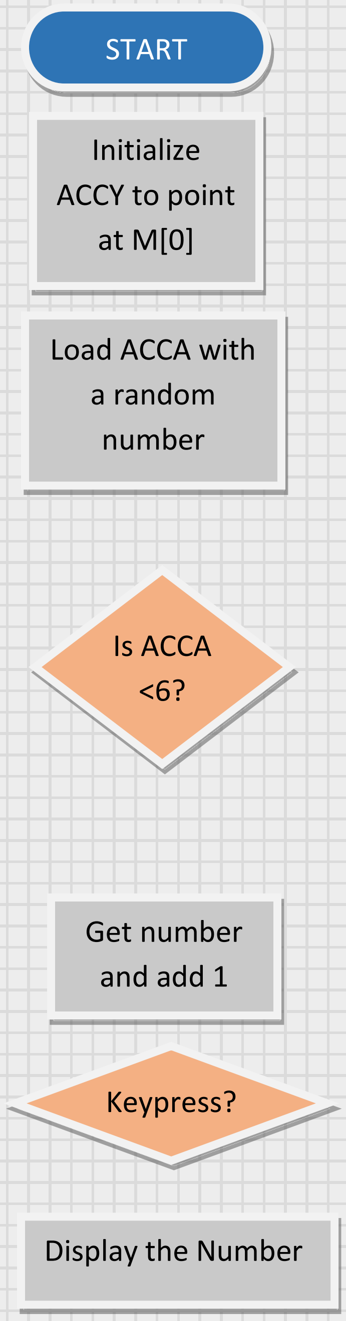
If we got here, it is because our number is from 0 to 5. Let's add 1 to get it into the range we need.

Now that we have a good number, let's look for the keypress. If we don't get it, circle back until we do.

That should be it. Display the number and head back in our flowchart to where we get another number.

Now that we have our logic all laid out, let's turn this flowchart into code.

Assemble the Program



We will now convert the blocks in our flowchart to Assembly language.

TIY 0
This gives ACCY the value it needs.

RAND
This loads the random number.

Store this number and then check if it is too big.
AIM
AIA A
FLAG=1 **JMP ahead**
FLAG=0 **JMP back** to RAND
Here, we are adding 10 to ACCA and looking for overflow. If it overflows, the number was too big so we go back to get another. If not jump to the next command. We don't know the address of **ahead** and **back** yet. We'll fill that in later.

The number is good. Get the original back.
MIA
AIA 1

KIA
JMP to KIA

AO
JMP back to RAND

So the program is just about done. We just need to assign addresses to the instructions and see where we need to direct the jumps (JMPs).

Compile the Program

Now we will compile the program. This is where we take our Assembly language and convert it to pure Machine language.

Program 12: Computer Dice

First, copy all of the instructions from above in order into the Command section reserving extra spaces for 2 word instructions and Jumps.

MEM	Command	Machine Code
00	TIY	
01		
02	RAND	
03		
04	AIM	
05	AIA	
06		
07	JMP	
08		
09		
0A	JMP	
0B		
0C		
0D	MIA	
0E	AIA	
0F		
10	KIA	
11	JMP	
12		
13		
14	MIA	
15	AO	
16	JMP	
17		
18		

Now go back and fill in the Machine language instructions and data from the assembled flowchart.

MEM	Command	Machine Code
00	TIY	9
01		0
02	RAND	E
03		E
04	AIM	5
05	AIA	3
06		A
07	JMP	F
08		
09		
0A	JMP	F
0B		
0C		
0D	MIA	6
0E	AIA	3
0F		1
10	KIA	4
11	JMP	F
12		
13		
14	MIA	6
15	AO	0
16	JMP	F
17		
18		

The last thing to do is add the addresses of the JMPs. Use the descriptions in the assembled code of where to jump to and then find that instruction in the memory map and add it after the JMP. For instance, the JMP at memory location 11 was labeled **JMP to KIA**. We can see that KIA is at memory location 10. So enter a 1 and then a 0 into the memory location following the JMP command. Do this for all of the JMPs.

The final program should look like below.

MEM	Command	Machine Code
00	TIY	9
01		0
02	RAND	E

03		E
04	AIM	5
05	AIA	3
06		A
07	JMP	F
08		0
09		D
0A	JMP	F
0B		0
0C		2
0D	MIA	6
0E	AIA	3
0F		1
10	KIA	4
11	JMP	F
12		1
13		0
14	MIA	6
15	AO	0
16	JMP	F
17		0
18		2

That is it. When you run this program, the dice will “roll” as you hold down a key and stop on a random roll when you release it.

That is the process from idea to software. At least, it is one way of doing it. For a very simple program, you may not need to go through all of this. But as your programs become more complex, you will find that keeping your prototype code clean and well organized will help greatly in getting to the finish line.

I encourage you to go back and change the resource decision to use ACCB for the number instead of M[0] and see what the advantages are.

Appendix B has a blank programming sheet that you can print out to help you writing programs.

2 Last Tips

Want to know how to extend your saved memory space? You can load multiple programs into each of your 2 memory storage locations. Just distribute the programs throughout the entire memory space (00-EF). Then, use the ADDR function to move the program counter to the location of a particular program. Then just run. This way, you may have one program loaded from 00-1F in memory, another loaded from 20-4F, and another from 50-EF. Same the whole memory space with FUNC 4 or 6. Then, when you load that memory, ADDR to the start of the individual programs and FUNC 0 to run it. Run always starts execution from the current program counter. That is why we have always RSET before running a program.

Lastly, a big help in troubleshooting code is if you can walk through the program instruction by instruction. Our computer will let you do that. Instead of FUNC 0, use FUNC 1 and the program will step each time the INCR key is pressed (the screen will be initially blank until you INCR for the first instruction). The only think we need to clarify is the AO command. This command looks for user input but falls through if there is none. This is hard to simulate in the Run-step function since we want to test both cases; when a key is pressed and when it is not. To do this, when you get to an AO instruction in your code, if you hit INCR, the code will progress as though no key was pressed. When you want to have keyboard input, and when you are at the AO instruction, press the desired key and then INCR. This will enter your selection into ACCA and set the FLAG to 0 so you can then jump out of the AO loop.

Conclusion

By now, you should have a solid grasp of the fundamentals of computer structure, memory usage, instruction sets, accumulators, and memory addressing. You should now feel like you can start writing your own programs. Start with simple concepts and build on them. Use the examples presented to modify them to your liking. Most of all, have fun as you continue to explore the amazing world of the inner workings of computers.

Sample Programs

Here are some additional programs for you to use. Try to use the description of the program to write your own. Then see if you used the same techniques that the sample used. There are often a lot of ways to program the same concept. Appendix B has some templates that you can print to make generating the code easier.

Sample 1: Rock/Paper/Scissors

MEM	Command	ML		MEM	Command	ML
00	TIY	9		**1D**	CMA	C
01		0		**1E**		0
02	RAND	E		**1F**	JMP	F
03		E		**20**		2
04	AIM	5		**21**		5
05	AIA	3		**22**	JMP	F
06		E		**23**		3
07	JMP	F		**24**		8
08		0		**25**	CMA	C
09		D		**26**		1
0A	JMP	F		**27**	JMP	F
0B		0		**28**		3
0C		2		**29**		1
0D	KIA	4		**2A**	TIA	1
0E	JMP	F		**2B**		1
0F		0		**2C**	ENDS	E
10		D		**2D**		9
11	M-	8		**2E**	JMP	F
12	JMP	F		**2F**		3
13		1		**30**		C
14		D		**31**	TIA	1
15	CMA	C		**32**		2
16		E		**33**	ERRS	E
17	JMP	F		**34**		6
18		3		**35**	JMP	F
19		1		**36**		3
1A	JMP	F		**37**		C
1B		2		**38**	TIA	1
1C		A		**39**		0

3A	SHTS	E				
3B		7				
3C	AO	0				
3D	TIA	1				
3E		6				
3F	DLAY	E				
40		0				
41	RSTO	E				
42		3				
43	JMP	F				
44		0				
45		2				

When you select 0, 1, or 2, the program will evaluate your choice compared to the computers, and display the results as 0 = Tie, 1 = user won, 2 = computer won.

This program actually goes through the steps of choosing inputs and comparing them to determine the correct outcome. If you think about it though, if all we are doing is playing this game with the computer, we could really just write a program to randomly generate one of the potential outcomes of the game and then just display it.

To enhance this game, you might want to display the user's choice and computers choice using NOUT after the match. Experiment with modification to the program.

Sample 2: Electronic Piano

This is a simple program but is ripe for modification. Press a key and the associated note will play. You could use this program to write a song, and then go back to Program 5 and enter your song in memory and have the computer play it.

MEM	Command	ML
00	KIA	4
01	JMP	F
02		0
03		0
04	SOND	E
05		5
06	JMP	F
07		0
08		0

Sample 3: Centipede

This program lights the bottom row of numbers from left to right and then clears them the same way. It simulates a centipede running across the LCD.

MEM	Command	ML		MEM	Command	ML
00	TIA	1		**0F**		0
01		1		**10**	RSTN	E
02	TIY	9		**11**		2
03		0		**12**	DLAY	E
04	SETN	E		**13**		0
05		1		**14**	INCY	A
06	DLAY	E		**15**	CMY	B
07		0		**16**		8
08	INCY	A		**17**	JMP	F
09	CMY	B		**18**		1
0A		8		**19**		0
0B	JMP	F		**1A**	JMP	F
0C		0		**1B**		0
0D		4		**1C**		2
0E	TIY	9				

Sample 4: Binary to Hex Practice

This program displays a random binary number on the right 4 digits of the lower display. It then waits for you to enter the number on the keyboard. You get the ENDS if right and the ERRS in wrong. The left 4 digits light up but are not used. The default memory value is F, which is 1111 in binary. If you want, load 0 into M[E] prior to running the program to make the left digits all 0. If you like this better, modify the program to do it automatically.

MEM	Command	ML		MEM	Command	ML
00	TIY	9		**0E**	JMP	F
01		F		**0F**		1
02	RAND	E		**10**		6
03		E		**11**	ENDS	E
04	AIM	5		**12**		9
05	NOUT	E		**13**	JMP	F
06		F		**14**		0
07	KIA	4		**15**		2
08	JMP	F		**16**	ERRS	E
09		0		**17**		6
0A		7		**18**	JMP	F
0B	M-	8		**19**		0
0C	CMA	C		**1A**		2
0D		0				

Sample 5: Roulette

Write a program that simulates the game Roulette. Have the numbers on the bottom of the LCD cycle and slow, and then stop on a random number.

MEM	Command	ML		MEM	Command	ML
00	KIA	4		**20**		1
01	JMP	F		**21**	XCH	2
02		0		**22**	AIA	3
03		0		**23**		1
04	TIY	9		**24**	XCH	2
05		0		**25**	AIA	3
06	RAND	E		**26**		1
07		E		**27**	CMA	C
08	SHFT	D		**28**		3
09	AIM	5		**29**	JMP	F
0A	TIA	1		**2A**		0
0B		0		**2B**		F
0C	XCH	2		**2C**	CHNG	E
0D	TIA	1		**2D**		A
0E		1		**2E**	TIY	9
0F	TIY	9		**2F**		0
10		0		**30**	CHNG	E
11	SETN	E		**31**		A
12		1		**32**	TIY	9
13	SHTS	E		**33**		0
14		7		**34**	TIA	1
15	XCH	2		**35**		0
16	DLAY	E		**36**	CHNG	E
17		0		**37**		A
18	XCH	2		**38**	SETN	E
19	RSTN	E		**39**		1
1A		2		**3A**	SHTS	E
1B	INCY	A		**3B**		7
1C	CMY	B		**3C**	CHNG	E
1D		8		**3D**		A
1E	JMP	F		**3E**	XCH	2
1F		1		**3F**	DLAY	E

MEM	Command	ML
40		0
41	XCH	2
42	M-	8
43	CMA	C
44		0
45	JMP	F
46		4
47		B
48	JMP	F
49		5
4A		6
4B	M+	7
4C	AIA	3
4D		1
4E	CHNG	E
4F		A
50	RSTN	E
51		2
52	INCY	A
53	JMP	F
54		3
55		8
56	ENDS	E
57		9
58	KIA	4
59	JMP	F
5A		5
5B		8
5C	CHNG	E
5D		A
5E	RSTN	E
5F		2
60	CHNG	E
61		A
62	JMP	F
63		0
64		4

APPENDIX A: ASCII TABLE OF CHARACTERS

HEX	ASCII	HEX	ASCII	HEX	ASCII	HEX	ASCII
00	NOT USED	20	Space	40	@	60	`
01		21	!	41	A	61	a
02		22	“	42	B	62	b
03		23	#	43	C	63	c
04		24	$	44	D	64	d
05		25	%	45	E	65	e
06		26	&	46	F	66	f
07		27	‘	47	G	67	g
08		28	(	48	H	68	h
09		29	)	49	I	69	i
0A		2A	*	4A	J	6A	j
0B		2B	+	4B	K	6B	k
0C		2C	,	4C	L	6C	l
0D		2D	-	4D	M	6D	m
0E		2E	.	4E	N	6E	n
0F		2F	/	4F	O	6F	o
10		30	0	50	P	70	p
11		31	1	51	Q	71	q
12		32	2	52	R	72	r
13		33	3	53	S	73	s
14		34	4	54	T	74	t
15		35	5	55	U	75	u
16		36	6	56	V	76	v
17		37	7	57	W	77	w
18		38	8	58	X	78	x
19		39	9	59	Y	79	y
1A		3A	:	5A	Z	7A	z
1B		3B	;	5B	[	7B	{
1C		3C	<	5C	¥	7C	|
1D		3D	=	5D	]	7D	}
1E		3E	>	5E	^	7E	→
1F		3F	?	5F	_	7F	←

APPENDIX B: Program Sheet

MEM	ASM	ML		MEM	ASM	ML		MEM	ASM	ML
00				20				40		
01				21				41		
02				22				42		
03				23				43		
04				24				44		
05				25				45		
06				26				46		
07				27				47		
08				28				48		
09				29				49		
0A				2A				4A		
0B				2B				4B		
0C				2C				4C		
0D				2D				4D		
0E				2E				4E		
0F				2F				4F		
10				30				50		
11				31				51		
12				32				52		
13				33				53		
14				34				54		
15				35				55		
16				36				56		
17				37				57		
18				38				58		
19				39				59		
1A				3A				5A		
1B				3B				5B		
1C				3C				5C		
1D				3D				5D		
1E				3E				5E		
1F				3F				5F		