6502 Assembly Language

by Ted Kosan

Part of The Professor And Pat series
(professorandpat.org)

Table of Contents

Assemblers

I was deep in thought when I heard a knock on the door of my shop.

"Professor, are you there?" A voice said. "Its Pat and I've come to learn about assemblers!"

"Come in, Pat!" I said.

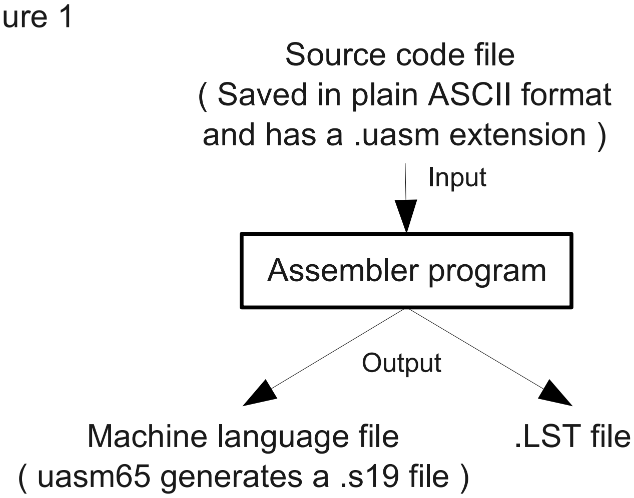
When Pat opened the door and entered, I smiled and said "have a seat next to the computer and boot it up."

While the computer was booting I said "So, you want to learn about assemblers?"

"Yes!" said Pat. "I couldn't stop thinking about machine language and assembly language since the last time we met and now I really want to know what an assembler does and how to use one."

I looked thoughtfully at Pat for a few moments then said "Okay, let me find a whiteboard and then we will discuss assemblers." Then I drew the following diagram while Pat watched. (see Fig. 1)

Figure 1



"An **assembler**," I said "is a program that takes a source code file that contains plain ASCII characters and converts it into a file that contains machine language. The type of application that is used to create a source code file is called a **text editor**. Text editors allow users to create documents that are similar to word processing documents, except the files are saved using only plain ASCII characters. For this reason, files that only contain plain ASCII characters are also called **text files**."

"Word processors can't be used to create source code files?" asked Pat.

"No," I replied "and the reason for this is because word processors need to save extra information in the files they create, including whether characters should be in bold or underlined, what font types the characters use, and what font sizes they use. Programs that take source code of any kind as input are not able to handle this extra information. These programs are only able to understand plain ASCII characters and, if a file that was created by a word processor was fed into them, the programs would produce errors."

"Can you show me what a text file looks like?" asked Pat.

"Yes." I replied. I then launched MathRider (http://mathrider.org), typed in the following text, and saved it in a file called '**abc123.txt**'.

```
ABC
123
Hello Pat!
```

(Note: I run the GNU/Linux operating system on my PC and so the **hexdump** command I use next will not work in Windows.)

I ran the **hexdump** command on the **abc123.txt** file and this is the output it produced:

```
$ hexdump -C abc123.txt
00000000  41 42 43 0d 0a 31 32 33  0d 0a 48 65 6c 6c 6f 20  |ABC..123..Hello |
00000010  50 61 74 21 0d 0a                                 |Pat!..|
```

"The hexdump command is similar to the umon65's Dump command," I said "except instead of dumping memory locations, it dumps the contents of files."

Pat studied the output for a few moments then said "Its output is arranged into 3 columns, just like the Dump command's output is! The first ASCII character in the file is a capital letter 'A' and hexdump displayed its value as 41 hex, just like the ASCII table showed. I see that 'B' is 42 hex, the numeral '1' is 31 hex, and 'Pat' is 50 hex, 61 hex, and 74 hex. I don't understand what the 0d 0a numerals are, though."

"Look at the source code again and also look for 0d hex and 0a hex in the ASCII table." I replied.

Pat did this then said "Oh, they represent a **carriage return** and a **line feed**! Is that what causes '123' to be placed on the line below 'ABC' and for 'Hello Pat!' to be placed below '123'?"

"Yes, Pat, this is exactly what the ASCII carriage return and line feed characters do!" I said. "On some operating systems (like Windows) both a carriage return and a line feed are used to drop down a line and move the cursor to the left side of the screen. On other operating systems, however, 0A hex is used by itself for both these operations and it is call a **newline** instead of a **line feed**. Another way to indicate a **carriage return followed by a line feed** is by saying or typing **CRLF**."

"I'm glad I know what hexadecimal and ASCII are now because they are helping me to understand how computers work!" said Pat.

I replied "You are discovering that the more knowledge that you possess, the easier it becomes to expand your knowledge. The hexadecimal numerals and ASCII characters are fundamental concepts that are used throughout the whole field of computing. A sound understanding of how they work is very useful for learning more advanced computing concepts."

After a few moments I said, "Lets get back to assemblers. When an assembler opens a file, the file must only contain plain ASCII characters and these ASCII characters must conform to the syntax that the assembler expects. The assembler will then convert this source code into machine language instructions that the target CPU can understand.

What we will do next is to type in the assembly language version of the machine language program we started with, assemble it, and then look at the machine language it generated."

"In the diagram," said Pat "I understand that the assembler is going to generate a file that contains machine language, but what is this other '.LST' file that it generates?"

"A .LST file," I replied "contains the original source code version of the program that was sent to the assembler, along with the machine language that each line of source code was converted into. The purpose of this file is to allow the programmer to see exactly how the source code was converted into machine language. We will look at a .LST file after we have assembled our first program."

The UASM65 Assembler, .S19 Files, and .LST files

I created a new file in MathRider called **u6502_programs.mrw**, typed the following assembly language source code into it, and then saved it. **(Note: This is a %uasm "fold" and folds are explained in the MathRider for Newbies book which can be found on the MathRider website.)**

```
%uasm65,description="Example 1"
      org 0200h

      lda #10d
      adc #5d
      sta 0208h
      brk

      end
%/uasm65
```

"The assembler we will be using is called **uasm65**," I said "and it stands for **Understandable Assembler for 6500 series CPUs**. The assembler is built into MathRider and it can be run by pressing **<shift><enter>** inside of a **%uasm65 fold** **(which must be placed into a file which has a .mrw extension)**.

The syntax that Example 1 contains is the syntax that the uasm65 assembler understands. **The empty space to the left of these commands is important too** and it can be created either with the **space bar** or with the **tab key**. Empty space like this is called **whitespace** and ASCII characters that produce whitespace when printed are called **whitespace characters**. The complete set of ASCII whitespace characters include the space, tab, newline, form feed, and carriage return characters."

Pat looked at the source code then said "I know that lda, adc, sta, and brk are 6502 instruction mnemonics, but what are **org** and **end**?"

"Those are called **pseudo ops** (which is short for pseudo operations) and another name for them is **assembler directives**. They are designed to look like instruction mnemonics, but instead of being instructions for a CPU, they are instructions which are meant for the assembler. Assembler directives allow a programmer to tell the assembler how to assemble the program.

For example, the **org** directive stands for **originate** and it tells the assembler what the beginning address of the code that follows it should be. In this case, the code will be placed into memory starting at address 0200 hex."

"Does the **end** directive tell the assembler where the end of the source code is?" asked Pat.

"Yes." I replied "There are 8 directives that uasm65 uses and we will be discussing them as we go. "

I then placed the cursor inside of the **%uasm65** fold and pressed <shift><enter> . Here is a copy of the %uasm65 fold and the output it generated:

```
%uasm65,description="Example 1"
    org 0200h

    lda #10d
    adc #5d
    sta 0208h
    brk

    end
%/uasm65

    %output ,preserve="false"
      *** List file ***

      0200              000001 |  org 0200h
                        000002 |
      0200 A9 0A        000003 |  lda #10d
      0202 69 05        000004 |  adc #5d
      0204 8D 08 02     000005 |  sta 0208h
      0207 00           000006 |  brk
                        000007 |
                        000008 |  end

      *** Executable code ***

      %s19,descrption="Execute this fold to send program to U6502 monitor."
        S00700000554153 47C8
        S10B0200A90A69058D0802003A
        S9030000FC
      %/s19
    %/output
```

I pointed at the output and said "The **.lst** file that was generated is present under the title which reads '*** **List file** ***' and the **s19** file is present in a **%s19** fold which is under the title '*** **Executable code** ***'.

Some assemblers generate machine language files which are not encoded in ASCII-based files like s19 files are and therefore they cannot be opened in a text editor. One reason the uasm65 assembler encodes its machine language in ASCII is so that it is easy for humans to read and another reason is so its code can be sent to a microcontroller easier."

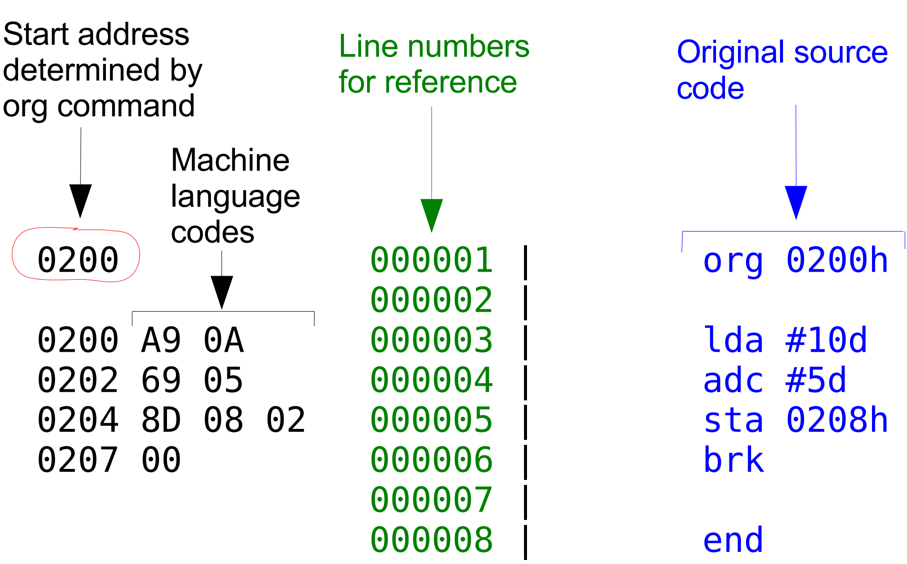
Pat studied the s19 code that was generated:

```
S00700005541534 7C8
S10B0200A90A69058D0802003A
S9030000FC
```

"It looks like machine language all right." said Pat "What does it all mean?"

"**S19** files consist of what are called **S records**," I said "and each line in an S19 file contains a separate S record. It will be easier to explain the contents of the **s19** file if we look at the **lst** file first." (see Fig. 2)

Figure 2



"The original source code is shown to the right along with the source code's line numbers." I said. "The machine language codes that each line of source code translate into are shown to the left. Notice that the **org** directive caused this program to be assembled starting at address 0200 hex.

Now, look at the machine language codes, which are A9 0A 69 05 8D 08 02 and 00. Can you see these numbers in the s19 file?"

Pat studied both files then said " I see them!"

"Where?" I asked.

"Right here!" said Pat "And I also found their starting address." Then Pat edited the s19 file and put spaces between the machine language codes so I could see them easier:

```
S00700000554153347C8
S10B 0200 A9 0A 69 05 8D 08 02 00 3A
S9030000FC
```

"Very good, Pat!" I said. "The purpose of the S19 file format is to allow assembled and compiled programs to be sent to small computer systems and microcontrollers. The emulator we have been using is also able to accept s19 files and our next step is to send this program to the emulator so that it can be executed. S19 files contain more detail than we have covered, but we will not discuss these details at this time."

Sending An S19 File To The Emulator

I opened the U6502 emulator and had it display the help screen by sending it a question mark character:

```
?

Assemble       A start_address
Breakpoint     B (+,-,?) address
Dump           D [start_address [end_address]]
Enter          E address list
Fill           F start_address end_address list
Go             G [start_address]
Help           H or ?
Load           L
Move           M start_address end_address destination_address
Register       R [PC,AC,XR,YR,SP,SR]
Search         S start_address end_address list
Trace          T [start_address [value]]
Unassemble     U [start_address [end_address]]
```

"The command that tells the umon65 monitor to accept a s19 file is the **Load** command and this is what the manual says about it." I opened the umon65 manual in a text editor and located the section on the Load command:

```
LOAD COMMAND

SYNTAX:  L

DESCRIPTION:  The purpose of the Load command is to put the monitor into
a mode that will receive an ASCII-based S19 format file, convert it into
binary, and place it into memory as directed by the address information
in the S19 file.  After the Load command has been issued, the monitor will
enter load mode and wait until the file starts arriving through the serial
connection.  The file will be placed into memory one byte at a time as it
is received and the last byte of the S19 file will place the monitor back
into command mode.
```

"Before I load the program, I will check the area of memory near address 0200 hex to see what is there." I executed a Dump command and here is what it displayed:

```
-d 0200

0200  00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00  ................
```

"This area of memory has zeros in it and this will make it easier to see the program after it is loaded." I said. "**When a %s19 fold is executed by pressing <shift><enter> inside of it, the emulator is automatically placed into Load mode and the code inside of the fold is loaded into the emulator.**" This is what was displayed in the monitor after the %s19 fold was executed:

```
UMON65V1.15 - Understandable Monitor for the 6500 series microprocessors.


PgmCntr(PC)  Accum(AC)  XReg(XR)  YReg(YR)  StkPtr(SP)  NV-BDIZC(SR)
  E02C         00         16        00        FD       00000000
-L
S00700005541534 7C8
S10B0200A90A69058D0802003A
S9030000FC

Send S records when you are ready...

S0S1S9
S records successfully loaded (press <enter> if no cursor is shown).
```

```
-
```

"The monitor will display a message that says 'S records successfully loaded' after the file has been received." I said.

"Is the program in the emulator's memory now?" asked Pat.

"Yes it is and I will let you verify this." I replied.

Pat then executed a Dump command followed by an Unassemble command in order to verify that the program was successfully loaded:

```
-d 0200

0200  A9 0A 69 05 8D 08 02 00 - 00 00 00 00 00 00 00 00  ..i.............

-u 0200

0200  A9 0A     LDA #0Ah
0202  69 05     ADC #05h
0204  8D 08 02  STA 0208h
0207  00        BRK
0208  00        BRK
0209  00        BRK
020A  00        BRK
020B  00        BRK
020C  00        BRK
020D  00        BRK
020E  00        BRK
020F  00        BRK
0210  00        BRK
0211  00        BRK
0212  00        BRK
0213  00        BRK
0214  00        BRK
```

"It worked!" cried Pat. "The program was successfully loaded! Assembly language is definitely easier to work with than machine language is."

"Even though assembly language is just a little bit higher level than machine language is," I said "it is much easier to program in than machine language and fairly large and sophisticated programs can be written in it."

"Can you show me a fairly large program that is written in assembly language?" asked Pat. "I would like to see one."

"The **umon65** monitor program is written in assembly language," I replied "and its source code is included in the emulator's download archive file. The file is called **umon65uasm** and it is located in the **examples/u6502/** directory (or examples\u6502\ on Windows systems). The **manual** for the umon65 monitor is also in that directory."

Pat opened the **umon65.uasm** file in the text editor and looked at it. You should look at this program now too.

After a while Pat said "Wow, the monitor program is almost 4000 lines long!"

After studying the program for a while, though, Pat's excitement level drained away. Eventually Pat said "It certainly looks complicated and confusing. I don't think I'll ever be able to understand how it all works."

I looked at Pat and said "My grandfather came from Hungary and he told me that the Hungarians have the following saying: 'All beginnings are tough.' Over time, I have found this saying to be true and it has often given me the courage to push past difficult beginnings to reach the easier parts that lie beyond. If you continue to put forth the same level of effort you have exerted thus far towards learning these concepts, the day will come when you look at this monitor program and not one part of it will remain a mystery to you."

I paused to let these words sink in, then I continued. "Another great saying is 'What humans have done, humans can do.' What do you think this saying means?"

Pat thought about the saying for a while then said "I think it means that if somebody has already done something, this proves that the something can be done and that other people should be able to do it too."

"Very good, Pat." I said. "In life, you are going to encounter concepts that appear beyond your grasp and problems that seem beyond your ability to solve them. The message that this saying relays is that most things that humans have already done, even very difficult things, you can do to if you want it bad enough and are willing to work hard achieve it."

We sat quietly for a few moments then Pat looked at me and said "I really like learning about computers and I want to know everything there is to know about them. There are millions of computers in the world and so there must be a lot of people who understand them very well. If these people were able to figure out how computers work, then I can too!"

"That is the right attitude to have, Pat!" I said.

"Anyway," said Pat "now that I know I am learning how computers work from a genuine Martian, I am hoping that some of that Martian know-how will rub off on me!"

I gave Pat a questioning look.

"I didn't know you were Hungarian, Professor. Why didn't you tell me before?"

I smiled and said "There are a great many things that I have not told you yet, Pat, but each one is awaiting the right time and place to be passed along. You will just have to be patient."

Pat laughed and said "Okay professor, I'll be patient, but can you at least tell me what we will be learning next?"

"Every particle in the physical universe is constantly moving through space and time," I said "and while we have been discussing assemblers, the right time for me to tell you about variables has been quickly approaching." I looked down at my watch then said "And the time has arrived... right... now!"

Models

I looked at Pat and said "Before we discuss variables, we need to discuss the reason that computers were invented in the first place. In order to understand why computers were invented, one must first understand what a **model** is."

"Do you mean like a plastic model car?" Asked Pat.

"Yes," I replied "a scaled-down plastic model car is one example of a model."

"What does scaled-down mean?" asked Pat.

"When a scaled-down version of an object is made," I replied "it means that a smaller copy of the object is created, with each of the dimensions of all of its parts being shrunken by the same amount. For example, if a scaled-down car was 50 times smaller than a given full-size car, then all of the parts in the scaled-down car would be 50 times smaller than their analogous parts in the full-size car."

"I have never seen a model car that contained small working copies of all of the parts of a real car." Pat said.

"Why do you think that is?" I asked.

Pat thought about this question for a while then said "Because it would be very difficult to create small working copies of all of the parts in a real car. I suppose it could be done, but it would be very expensive."

"I agree, and this is why **models** are usually used to represent objects instead of either scaled or unscaled exact copies of the objects. A **model** is a simplified representation of an object that only copies some of its attributes. Examples of typical object attributes include weight, height, strength, and color.

The attributes that are selected for copying are chosen for a given purpose. The more attributes that are represented in the model, the more expensive the model is to make. Therefore, only those attributes that are absolutely needed to achieve a given purpose are usually represented in a model. The process of selecting a only some of an object's attributes when developing a model of it is called **abstraction**."

"I am not quite following you." said Pat.

I paused for a few moments then said "Suppose we wanted to build a garage that could hold 2 cars along with a workbench, a set of storage shelves, and a riding lawn mower. Assuming that the garage will have an adequate ceiling height, and that we do not want to build the garage any larger than it needs to be for our stated purpose, how could an adequate length and width be determined for the garage?"

Pat thought about this question for a while then said "I'm not sure."

"One strategy for determining the size of the garage," I said "is to build perhaps 10 garages of various sizes in a large field. When the garages are finished, take 2 cars to the field along with a workbench, a set of storage shelves, and a riding lawn mower. Then, place these items into each garage in turn to see which is the smallest one that these items will fit into without being too cramped. The test garages in the field can then be discarded and a garage which is the same size as the one that was chosen could be built at the desired location."

"Thats ridiculous!" cried Pat. "11 garages would need to be built using this strategy instead of just one. This would be very inefficient."

"Can you think of a way to solve the problem less expensively by using a model of the garage and models of the items that will be placed inside it?" I asked.

"I think I am beginning to see how to do this." replied Pat. "Since we only want to determine the dimensions of the garage's floor, we can make a scaled down model of just its floor, maybe using a piece of paper."

"Go on." I said.

"Each of the items that will be placed into the garage could also be represented by scaled-down pieces of paper. Then, the pieces of paper that represent the items can be placed on top of the the large piece of paper that represents the floor and these smaller pieces of paper can be moved around to see how they fit. If the items are too cramped, a larger piece of paper can be cut to represent the floor and, if the items have too much room, a smaller piece of paper for the floor can be cut.

When a good fit is found, the length and width of the piece of paper that represents the floor can be measured and then these measurements can be scaled up to the units used for the full-size garage. With this method, only a few pieces of paper are needed to solve the problem instead of 10 full-size garages that will later be discarded."

"Very good Pat!" I said. "And what makes these pieces of paper models of the full-size objects they represent and not exact scaled-down copies of them?"

Pat thought about this then replied "The only attributes of the full-sized objects that were copied to the pieces of paper were the object's length and width."

"What is the process called when only some of an object's attributes are placed into a model instead of all of them?" I asked.

"Abstraction!" replied Pat.

Placing Models Into A Computer

"Now that we have discussed what a model is Pat," I said "you may find it interesting to know that the reason one of the first modern programmable digital computer was invented was to model the paths of artillery projectiles."

"Really!?" asked Pat. "When was this computer invented and who invented it?"

"The computer was invented in the 1940s by John Mauchly and J. Presper Eckert," I replied "and it was called ENIAC. John Von Neumann later joined the team that built ENIAC to help them create a second computer called EDVAC."

"Back to Martians again!" cried Pat. "And if John Von Neumann is involved, I bet that the Von Neumann architecture can't be far behind!" said Pat.

I smiled and said "You are very perceptive!"

"So, ENIAC was used to model the paths of artillery projectiles?" asked Pat.

"Yes." I replied.

"I can see how paper can be used to model things," said Pat "but how can a computer be used to model things?"

"Do you remember earlier when I had you think of any idea and then I came up with a number that could be placed into a memory location to represent it?" I said.

"I remember," said Pat "I thought of the idea of a boat and the idea of a cat."

"The numbers that I came up with to represent the boat and the cat were really just patterns of bits in memory," I said "and these bit patterns were very simple models of each of these objects. Any attributes of any object can be represented by bit patterns . If the bit patterns are contained within a computer's memory, then the computer contains a model of the object."

Pat's mouth dropped open with surprise.

"Does this mean that instead of using paper to model the garage floor and the items, we could have used bit patterns to model them and then placed these bit patterns into a computer?" asked Pat.

"This is exactly what it means!" I replied. "The length and width values of the items could have been used to model them and the length and width values of the garage floor could have been used to model the garage'."

"But how can one keep track of all of these modeled values in a program?" asked Pat. "It seems that it would be very easy to become confused about which values belonged to which part of each model."

"It would be confusing if the programmer needed to keep track of every address where a value was stored" I replied "and this is why variables were invented."

Variables

"A **variable** allows a programmer to use a **letter** or a **name** instead of an **address** to refer to information that is being represented by memory locations." I said. "Almost all computer languages that are higher than machine language have the ability to use variables."

"Does this mean that assembly language has the ability to use variables?" asked Pat.

"Yes," I replied "and this is one of the reasons that assembly language is more powerful than machine language."

"Can you show me an example of a variable in assembly language?" asked Pat. "I want to see what one looks like."

"Yes," I replied "but first you need to tell me what you want the variable to model."

"How about modeling the garage floor we have been working with?" asked Pat.

"That is an excellent idea," I said. "but we will need 2 variables to model the floor, one to represent its length and one to represent its width."

I brought up an editor and typed in an assembly language program that had 2 variables in it. Then, I assembled the program and brought up the following .LST file that was generated into the text editor:

```
0200            000001 |        org 0200h
                000002 |
0200 AD 11 02   000003 |        lda garage_width
0203 69 01      000004 |        adc #1d
0205 8D 11 02   000005 |        sta garage_width
                000006 |
0208 AD 12 02   000007 |        lda garage_length
020B 69 01      000008 |        adc #1d
020D 8D 12 02   000009 |        sta garage_length
0210 00         000010 |        brk
                000011 |
0211 09         000012 |garage_width dbt 9d
0212 08         000013 |garage_length dbt 8d
                000014 |
                000015 |        end
```

While Pat studied the .LST file, I explained how the variables worked. "In this program, a variable called **garage_width** has been created to hold the width of the garage floor and another variable called **garage_length** has been created to hold its length. The **garage_width** variable has been set or **initialized** to **9** decimal and the address it has been bound to is 0211h. The **garage_length** variable has been initialized to **8** decimal and the address it has been bound to is 0212h. The measurement units that each of these variables are working with is meters. The **dbt** directive (which stands for **Define Byte**) is used to create byte-sized variables with this assembler."

"I see that the name **garage_width** and **garage_length** have been associated with the addresses 0211h and 0212h," said Pat "but why are these names called variables?"

"Look at the 3 assembly language instructions that have been placed into memory starting at address 0205h and tell me what you think they will do when they are executed." I replied.

Pat studied the instructions then said "The LDA instruction at address 0205h looks like it is copying the **9** that the variable **garage_width** refers to into register 'A' . The ADC instruction is adding **1** to the **9** and this should result in a **10** decimal being placed into the 'A' register. The STA instruction is then copying the **10** decimal which is in the 'A' register back into memory at the address that **garage_width** refers to.

Overall, it looks like the result of executing these 3 instructions is to increase the contents of the **garage_width** variable from **9** to **10**. I am only guessing, though, so I am not completely sure about this."

"How can you test your guess?" I asked.

"I suppose I could load this program into the emulator and trace through these 3 instructions to see what happens." replied Pat.

"That sounds like a good idea Pat." I said. "Load the program into the emulator and then execute a **d 0200 021f** command followed by a **u 0200** command then I will help you step through the program."

Pat loaded the program and executed the two commands. This is what was displayed on the screen:

```
-d 0200 021f

0200  AD 11 02 69 01 8D 11 02 - AD 12 02 69 01 8D 12 02  ...i.......i....
0210  00 09 08 00 00 00 00 00 - 00 00 00 00 00 00 00 00  ................

-u 0200

0200  AD 11 02  LDA 0211h
0203  69 01     ADC #01h
0205  8D 11 02  STA 0211h
0208  AD 12 02  LDA 0212h
020B  69 01     ADC #01h
020D  8D 12 02  STA 0212h
0210  00        BRK
0211  09 08     ORA #08h
0213  00        BRK
0214  00        BRK
```

I said "Look at the contents of memory locations 0211h and 0212h, Pat, and tell me what they contain."

Pat looked at the contents of these locations then replied "Memory location 0211h contains a **9** and memory location 0212h contains an **8**! These numbers are what we put into the **garage_width** and the **garage_length** variables!"

"That is right," I said "now I want you to look at address 0211h in the output from the Unassemble command and tell me what you see."

"The **9** and **8** are still in memory locations 0211h and 0212h," said Pat "but why is the ORA instruction there?"

"Think about it and see if you can figure it out." I replied.

Pat quietly looked at the screen for a while then said "Oh, I get it! The Unassemble command doesn't know that the **9** and the **8** are variables and so it interpreted them as an ORA instruction."

"Correct!" I said. "The Unassemble command can only interpret numbers in memory as assembly language instructions because this is the only **context** it knows. What do you think is providing the **context** for these two memory locations, Pat?"

"The **garage floor** that is being modeled by the **garage_width** and **garage_length** variables." replied Pat after a few moments of thought.

"Now Pat, you are going to see for yourself why variables are called variables." I said. "Execute a Register command and then trace the LDA instruction that is at address 0200h."

Pat did this and here is what was displayed:

```
-r

PgmCntr(PC)  Accum(AC)  XReg(XR)  YReg(YR)  StkPtr(SP)  NV-BDIZC(SR)
   102C         00         FC        00         FD       00010110

-t 0200

PgmCntr(PC)  Accum(AC)  XReg(XR)  YReg(YR)  StkPtr(SP)  NV-BDIZC(SR)
   0203         09         FC        00         FD       00010100

0203  69 01     ADC #01h
```

"Was the **9** from the **garage_width** variable loaded into the 'A' register?" I asked.

"Yes." replied Pat.

"Then execute another Trace command," I said "and verify that the ADC instruction increases the **9** by **1** then places the resulting **0A** hex into the 'A' register."

Pat executed the Trace command and verified that **0A** hex was placed into the 'A' register:

```
-t

PgmCntr(PC)  Accum(AC)  XReg(XR)  YReg(YR)  StkPtr(SP)  NV-BDIZC(SR)
   0205         0A         FC        00         FD       00010100

0205  8D 00 02  STA 0200h
```

"Dump address 0211h to verify that the **9** that we placed into the **garage_width** variable is still there." I said. Pat executed the Dump command and here was the result:

```
-d 0211

0211  09 08 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00  ................
```

"Finally," I said "execute the STA instruction with the Trace command then verify that the **garage_width** variable was changed from **9** to **0A** hex." Pat executed a Trace command followed by a Dump command and here was the result:

```
-t

PgmCntr(PC)  Accum(AC)  XReg(XR)  YReg(YR)  StkPtr(SP)  NV-BDIZC(SR)
   0208         0A         FC        00         FD       00010100

0208  AD 01 02  LDA 0201h

-d 0211

0211  0A 08 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00  ................
```

"The **garage_width** variable was changed from a **9** to a **0A** hex!" exclaimed Pat "My guess was right!"

"Yes, your guess was correct Pat," I said "and why are variables called variables?"

"Because the information they refer to can change!" replied Pat.

"Very good, Pat!" I said. "Variables need to change because the models that they are a part of need to change in order to be of maximum use.

Here are some final thoughts on variables. Their names need to consist of ASCII characters from 33 decimal through 122 decimal. The one exception to this is that variable names cannot contain a semi-colon with is an ASCII 59 decimal. **Variables also need to be placed up against the left side of the editor window with no spaces or tabs to the left of them**.

The Status Register

Pat studied the output from the trace command for a while then said "I think I understand what variables are now, and I understand what most of the registers do, but what does the SR register do?" Pat pointed to the part of the Trace command's output that contained the letters NV-BDIZC(SR).

"I was wondering when you would ask about those letters." I replied. "**SR** stands for **Status Register** and the bits in this register indicate the current state or status of the CPU. These bits are called status flags or **flags** for short and, as instructions are executed, certain instructions **set** or **clear** these flags. **Setting** a flag turns it into a **1** and **clearing** a flag turns it into a **0**. When the contents of the status register are displayed, the string of bits which are shown directly beneath the letters NV-BDIZC indicate the current state of each flag.

Perhaps the easiest flag to understand is the **zero flag** and therefore we will begin with it. The zero flag is represented by a capital letter Z and it is affected by about half of the 6502's instructions. When any of these instructions results in a 0 being calculated after it is executed, then the Z flag is **set**. If these instructions result in a nonzero value being calculated after execution, then the Z flag is **cleared**. The complete list of which instructions affect which flags is shown in the instruction set reference for the 6502."

I then brought up a web page that contained a 6502 instruction set reference and Pat looked at it. A 6502 instruction set reference can also be found in Appendix A in this document.

"One of the instructions that affects the Z flag is the DEX instruction. DEX stands for DEcrement X and it takes the contents of the X register and subtracts 1 from it. If the X register contained a 3, the DEX instruction would change it to a 2, and if it contained a 2, it would change it to a 1. In both cases, the Z flag would be set to 0 to indicate that the execution of the instruction did not result in a 0.

If we executed the DEX instruction one more time, however, the contents of the X register would go from 01 hex to 00 hex and the Z flag would be set to a 1 to indicate this. I will now enter a short program into the emulator that demonstrates what happens to the Z flag as the X register is decremented from 3 to 0 using the DEX instruction and you can trace it." I then entered the following short program into the emulator using the Assemble command and Pat traced through it:

```
0200  A2 03     LDX #03h
0202  CA        DEX
0203  CA        DEX
0204  CA        DEX
0205  00        BRK
```

```
-r

PgmCntr(PC)  Accum(AC)  XReg(XR)  YReg(YR)  StkPtr(SP)  NV-BDIZC(SR)
   102C         00         FC        00        FD       00010110

-t 0200

PgmCntr(PC)  Accum(AC)  XReg(XR)  YReg(YR)  StkPtr(SP)  NV-BDIZC(SR)
   0202         00         03        00        FD       00010100

0202  CA        DEX

-t

PgmCntr(PC)  Accum(AC)  XReg(XR)  YReg(YR)  StkPtr(SP)  NV-BDIZC(SR)
   0203         00         02        00        FD       00010100

0203  CA        DEX

-t

PgmCntr(PC)  Accum(AC)  XReg(XR)  YReg(YR)  StkPtr(SP)  NV-BDIZC(SR)
   0204         00         01        00        FD       00010100

0204  CA        DEX

-t

PgmCntr(PC)  Accum(AC)  XReg(XR)  YReg(YR)  StkPtr(SP)  NV-BDIZC(SR)
   0205         00         00        00        FD       00010110

0205  00        BRK
```

"Notice how the Z flag was set to 0 after the execution of each DEX instruction that resulted in a nonzero value," I said "but it was set to 1 as soon as the X register was decremented to 0."

"I see!" said Pat. "You know, those status register flags must have been changing all the time we have been tracing through programs in the emulator, but I never noticed it. Its funny how you can be looking at something, even for a long time, but not actually see it."

"Much of life is like that, Pat." I said. "Amazing and wonderful things lay spread before us in open sight, but we are blind to them for want of awareness. Some say that striving for awareness is one of the noblest goals that a person can pursue".

"The goal may be noble," said Pat "but it is definitely not easy to achieve! Anyway, I can see how the zero flag works now, but I don't understand what it is used for."

How A Computer Makes Decisions

"A CPU's status flags are very subtle but absolutely critical, Pat." I said. "Without its status flags, a CPU would be unable to make decisions, and a computer that can not make decisions is virtually useless."

"If computers can't actually think," said Pat "how can they make decisions?"

"The way that a CPU makes decisions," I replied "is by deciding to either execute a section of code or skip it and execute another section of code instead."

"How can a CPU skip a section of code?" asked Pat.

I replied "As we discussed earlier, a CPU determines where in memory to find the next instruction it is going to execute by looking at the contents of the Program Counter register. Normally, after the current instruction is finished executing, the Program Counter is set to the address of the instruction that immediately follows it in memory. However, if the Program Counter was not set to the address of the next instruction in memory, but rather to the address of an instruction in a different part of memory, then the code that was going to be run would be skipped."

"Can this be done?" asked Pat. "Can the Program Counter be set to a different address than that of the next instruction which would normally have been executed?"

"Yes." I said.

"How?" asked Pat.

"With the JMP instruction, the Branch instructions, and with a few other instructions." I replied. "I will show you some examples of how the JMP and the Branch instructions work and the first example will show how the JMP instruction can be used to skip over another instruction."

The JMP Instruction

I brought up the emulator, entered the following program using the Assemble command, and then had Pat trace through it:

```
0200  A9 01     LDA #01h
0202  4C 07 02  JMP 0207h
0205  A2 02     LDX #02h
0207  A0 03     LDY #03h
0209  EA        NOP
020A  00        BRK
...
```

"As you trace through this program Pat," I said "pay close attention to the value of the Program Counter. Tell me what happens to the Program Counter when the JMP instruction is executed."

```
-r

PgmCntr(PC)  Accum(AC)  XReg(XR)  YReg(YR)  StkPtr(SP)  NV-BDIZC(SR)
   102C         00         FC        00         FD      00010110

-t 0200

PgmCntr(PC)  Accum(AC)  XReg(XR)  YReg(YR)  StkPtr(SP)  NV-BDIZC(SR)
   0202         01         FC        00         FD      00010100

0202  4C 07 02  JMP 0207h

-t

PgmCntr(PC)  Accum(AC)  XReg(XR)  YReg(YR)  StkPtr(SP)  NV-BDIZC(SR)
   0207         01         FC        00         FD      00010100

0207  A0 03     LDY #03h

-t

PgmCntr(PC)  Accum(AC)  XReg(XR)  YReg(YR)  StkPtr(SP)  NV-BDIZC(SR)
```

```
   0209         01          FC         03         FD       00010100

0209  EA        NOP

-t

PgmCntr(PC)  Accum(AC)  XReg(XR)  YReg(YR)  StkPtr(SP)  NV-BDIZC(SR)
   020A         01          FC         03         FD       00010100

020A  00        BRK
```

"The Program Counter jumps from 0202h all the way to 0207h. When it did this, it skipped the LDX instruction." Pat said. "But how did you know that address 0207h was the address of the instruction that you wanted to jump to?"

"I knew that 0207h was the address I needed to pass to the JMP instruction because the JMP instruction is 3 bytes long and the next instruction after the JMP instruction is 2 bytes long. The JMP instruction was placed in memory starting at 0202h and 0202h + 3 + 2 = 0207h."

"But what if you wanted to jump over a bunch of instructions?" asked Pat. "It would be tough to determine the lengths of all of these instructions, especially if you have not assembled them yet."

"You are right, Pat, and this is why assemblers allow a person to use something called **Labels** instead of addresses." I replied.

Labels

"**Labels** are names that can be used in the source code of an assembly language program to represent an address of an instruction. Labels, just like variables, are replaced with the addresses they represent during the assembly process. They make coding the program much easier for the programmer, however, because they remove the need for the programmer to keep track of the instruction's addresses. I will now create an assembly language program that uses labels and jump instructions so you can see how they work together." I then created and assembled the following program:

```
0200            000001 |        org 0200h
                000002 |
0200 A9 01      000003 |        lda #01d
0202 4C 07 02   000004 |        jmp skip1
                000005 |
0205 A9 02      000006 |        lda #02d
```

```
                  000007 |
0207 A9 03        000008 |skip1   lda #03d
0209 4C 0E 02     000009 |        jmp skip2
                  000010 |
020C A9 04        000011 |        lda #04d
                  000012 |
020E 00           000013 |skip2   brk
                  000014 |
                  000015 |        end
                  000016 |
```

"In this listing, you can see how the label **skip1** is bound to address 0207h and the label **skip2** is bound to address 020Eh. A programmer is free to place labels on any instruction they want to, but the characters in each each label's name must be taken from the same range of ASCII characters that variable names do. **Labels must also be placed against the left side of the editor windows with no spaces or tabs on their left sides**."

Forward Branches And The Zero Flag

"I understand now how JMP is able to skip over instructions," said Pat "but since it always jumps when it is executed, then it can't be used for making a decision, can it?"

"No Pat," I replied "the JMP instruction will always jump to another location in memory without exception so it can not be used to make a decision. The assembly language instructions that are designed to make decisions are the **branch** instructions." I then wrote all of the 6502's branch instructions on the whiteboard:

```
BCC - Branch on Carry Clear.
BCS - Branch on Carry Set.

BEQ - Branch on result EQual.
BNE - Branch on result Not Equal.

BMI - Branch on result MInus.
BPL - Branch on result PLus.

BVC - Branch on oVerflow Clear.
BVS - Branch on oVerflow Set.
```

"Hey!" cried Pat "Some of these instructions are related to flags in the Status Register."

"Actually, all of them are." I said. BCC and BCS are related to the **Carry** flag, BEQ and BNE are related to the **Zero** flag, BMI and BPL are related to the Negative flag, and BVC and BVS are related to the **oVerflow** flag."

"How are they related?" asked Pat.

"Each of these 4 flags determines whether or not the 2 instructions they are associated with will take the branch or not." I replied.

"I still don't quite understand." said Pat.

"I think an example will make it clear." I said. "Lets start with the two branch instructions which are associated with the Zero flag, which are BEQ and BNE. BEQ can be thought of in 2 ways. The first way means 'branch if the result equaled zero'. For example, if a BEQ instruction were placed directly beneath a DEX instruction, and the DEX instruction just decremented register X to zero, then the BEQ instruction would take the branch. If the DEX instruction resulted in register X containing a non-zero value, then the BEQ instruction would not branch and execution would continue with the instruction directly beneath BEQ.

The second way to think about the BEQ instruction is that it can be used to determine if 2 values are equal when used in cooperation with another instruction like CMP. The CMP instruction compares a value in the 'A' register with a value in memory by **internally subtracting** the value in memory from the value in the 'A' register. Internal subtraction means that the result is discarded and not placed into a register. If the result of the subtraction was 0 (meaning the values were equal) the Zero flag will be **set** and if the result was non-zero (meaning the values were not equal), the Zero flag will be **cleared**."

"Do the branch instructions usually need to work in cooperation with other instructions?" asked Pat.

"Yes they do." I replied. "Certain instructions set or clear flags in the Status register, and the branch instructions that look at the flags in question must be placed near the instructions that affect the flags. There is not much use in setting flags if nothing is going to look at them and conversely, there is not much use in looking at flags if nothing purposefully set or cleared them.

I will now create a small assembly language program that will compare 2 numbers and branch if they are equal or not branch if they are not equal. You can then load it into the emulator and trace through it to see what it does."

First, I created the following program:

```
0200            000001 |      org 0200h
                000002 |
0200 A9 02      000003 |      lda #02d
0202 C9 02      000004 |      cmp #02d
0204 F0 01      000005 |      beq Equal1
                000006 |
0206            000007 |NotEqual1 *
0206 EA         000008 |      nop
                000009 |
0207            000010 |Equal1 *
0207 EA         000011 |      nop
0208 A9 05      000012 |      lda #05d
020A C9 06      000013 |      cmp #06d
020C F0 02      000014 |      beq Equal2
020E EA         000015 |      nop
                000016 |
020F            000017 |NotEqual2 *
020F EA         000018 |      nop
                000019 |
0210            000020 |Equal2 *
                000021 |
0210 00         000022 |      brk
                000023 |      end
                000024 |
```

"Why are the labels on lines by themselves with asterisks instead on lines that have instructions?" asked Pat.

"This is an alternative way to put labels in a program." I replied "The asterisk is a symbol which means 'the address that the following instruction will be placed at'. This technique allows the label names to be long without pushing the instruction they are associated with too far to the right and out of line with the other instructions. It also allows code to be inserted immediately after the label easier."

"Okay." said Pat.

Pat then loaded the program into the emulator, unassembled it to make sure it was loaded correctly, and then traced through it:

```
-u 0200

0200  A9 02     LDA #02h
0202  C9 02     CMP #02h
```

```
0204  F0 01      BEQ 0207h
0206  EA         NOP
0207  EA         NOP
0208  A9 05      LDA #05h
020A  C9 06      CMP #06h
020C  F0 02      BEQ 0210h
020E  EA         NOP
020F  EA         NOP
0210  00         BRK
...


-t 0200

PgmCntr(PC)  Accum(AC)  XReg(XR)  YReg(YR)  StkPtr(SP)  NV-BDIZC(SR)
   0202         02         FC        00        FD       00010100

0202  C9 02      CMP #02h

-t

PgmCntr(PC)  Accum(AC)  XReg(XR)  YReg(YR)  StkPtr(SP)  NV-BDIZC(SR)
   0204         02         FC        00        FD       00010111

0204  F0 01      BEQ 0207h

-t

PgmCntr(PC)  Accum(AC)  XReg(XR)  YReg(YR)  StkPtr(SP)  NV-BDIZC(SR)
   0207         02         FC        00        FD       00010111

0207  EA         NOP

-t

PgmCntr(PC)  Accum(AC)  XReg(XR)  YReg(YR)  StkPtr(SP)  NV-BDIZC(SR)
   0208         02         FC        00        FD       00010111

0208  A9 05      LDA #05h

-t

PgmCntr(PC)  Accum(AC)  XReg(XR)  YReg(YR)  StkPtr(SP)  NV-BDIZC(SR)
   020A         05         FC        00        FD       00010101

020A  C9 06      CMP #06h

-t

PgmCntr(PC)  Accum(AC)  XReg(XR)  YReg(YR)  StkPtr(SP)  NV-BDIZC(SR)
   020C         05         FC        00        FD       10010100

020C  F0 02      BEQ 0210h

-t
```

```
PgmCntr(PC)  Accum(AC)  XReg(XR)  YReg(YR)  StkPtr(SP)  NV-BDIZC(SR)
   020E         05         FC        00         FD      10010100

020E  EA        NOP

-t

PgmCntr(PC)  Accum(AC)  XReg(XR)  YReg(YR)  StkPtr(SP)  NV-BDIZC(SR)
   020F         05         FC        00         FD      10010100

020F  EA        NOP

-t

PgmCntr(PC)  Accum(AC)  XReg(XR)  YReg(YR)  StkPtr(SP)  NV-BDIZC(SR)
   0210         05         FC        00         FD      10010100

0210  00        BRK
```

"The first BEQ instruction made the decision to branch and the second BEQ instruction made the decision not to branch!" said Pat.

"That is correct." I said. "Computers perform simple decisions using simple branch instructions like this and complex decisions are built up by having 2 or more branch instructions work together as a team."

"That's kind of hard to believe." said Pat.

"It is indeed hard to believe Pat," I said "yet it is true. It takes a while, but as you program more you will become comfortable with this concept."

"What about the BNE instruction?" asked Pat. "What does it do?"

"The BNE instruction is simply the opposite of the BEQ instruction," I said "and it will branch when a result is non-zero and not branch when it is zero. There are situations where BEQ is best to use and situations where BNE is best and you will learn how to decide when to use each over time."

"I will have to take your word for it Professor," said Pat "because this all still seems fuzzy to me."

"The more you work with it, the easier it will become." I replied. But now, lets look at the program again to see how branch instruction know how far ahead in memory to branch."
I then unassembled the program again:

```
-u 0200

0200  A9 02     LDA #02h
0202  C9 02     CMP #02h
0204  F0 01     BEQ 0207h
0206  EA        NOP
0207  EA        NOP
0208  A9 05     LDA #05h
020A  C9 06     CMP #06h
020C  F0 02     BEQ 0210h
020E  EA        NOP
020F  EA        NOP
0210  00        BRK
```

"What address is the first BEQ instruction set to branch to?" I asked.

"Address 207 hex." replied Pat.

"And what operand does the first BEQ instruction have?" I asked.

"01." Said Pat. "Hmmm, the address of the next instruction after the branch is 206 hex and address 207 hex is **1** memory location away from it.

The second BEQ instruction has an operand of **02** and it is branching to address 210 hex. The address of the next instruction after the second BEQ is 20E and address 210 is **2** locations away from it. Does this mean that a branch command's operand byte tells it how many locations to move ahead in memory from the address of the next instruction after it?"

"Yes, Pat, and that was very good reasoning on your part." I said.

"How about branching backwards in memory to previous instructions?" asked Pat "Can this be done too?"

"Yes, branches (and also jumps) can move the Program Counter to earlier instructions that are lower in memory too," I said "and in fact, a computer would be useless if it could not branch backwards in memory. Before we discuss branching backwards in memory, however, we must first talk about negative numbers."

Negative Numbers And The Negative Flag

"How many patterns can be formed by 4 bits, Pat?" I asked.

Pat thought about this for a few moments then said "2 to the 4th power is

16 so 16 patterns."

"If the bit pattern 0000 represents a decimal 0," I asked "what is the highest decimal numeral that 4 bits can represent?"

Pat said "Since the first of the 16 4-bit patterns needs to represent decimal 0, then there are only 15 patterns left to represent the decimal numerals 1 through 15. This means that the highest decimal numeral that 4 bits can represent is 15."

"Very good Pat," I said "now write the binary numerals 0000 through 1111 on the whiteboard and place their decimal numeral equivalents next to them." Pat then did this. (see Fig. 2)

Figure 2

Binary		Decimal
0000	-	0
0001	-	1
0010	-	2
0011	-	3
0100	-	4
0101	-	5
0110	-	6
0111	-	7
1000	-	8
1001	-	9
1010	-	10
1011	-	11
1100	-	12
1101	-	13
1110	-	14
1111	-	15

"So far we have been working with positive numbers," I said "but how do you think bit patterns can be made to represent negative numbers?" I asked?

Pat studied the numbers on the whiteboard then said "I'm not sure."

"What do you think would happen," I asked "if we took the binary numeral 0000 and subtracted 1 from it?"

Pat thought about this for a while.

"I'll give you a hint," I said "think back to the odometer example we discussed earlier and imagine what would happen if we added 1 to the bit pattern 1111."

"Well," said Pat "all the 1's in the bit pattern 1111 would roll around to 0's if you added 1 to it so I suppose that if 1 was subtracted from the bit pattern 0000, then all the 0's would roll backwards to 1111."

Figure 3

Binary		Decimal
1000	-	-8
1001	-	-7
1010	-	-6
1011	-	-5
1100	-	-4
1101	-	-3
1110	-	-2
1111	-	-1
0000	-	0
0001	-	1
0010	-	2
0011	-	3
0100	-	4
0101	-	5
0110	-	6
0111	-	7

"Very good Pat." I said. "Now, I am going to make a modified version of the bit pattern table you created by placing 0000 in the middle of the sequence instead of at the beginning. Then, instead of associating all positive decimal numerals with this sequence, I will associate the patterns after 0000 with positive decimal numerals and the patterns before it with negative decimal numerals." I then did this. (see Fig. 3)

After Pat had some time to study the new table I asked "Do you notice anything about the positive bit patterns and the negative bit patterns that can be used to tell them apart?"

"Pat studied the table further then said "Not really".

I then erased the leftmost bits in the patterns before and after 0000 and redrew them with a red marker. "What do you notice now?" I asked.

Figure 4

Binary		Decimal
1000	-	-8
1001	-	-7
1010	-	-6
1011	-	-5
1100	-	-4
1101	-	-3
1110	-	-2
1111	-	-1
0000	-	0
0001	-	1
0010	-	2
0011	-	3
0100	-	4
0101	-	5
0110	-	6
0111	-	7

"All the negative numbers have leftmost bits that are set to 1 and all of the positive numbers have leftmost bits that are set to 0!" said Pat.

"That is correct." I said. "When dealing with bit patterns of any size that represent signed numbers, the leftmost bit indicates whether a number is negative or not. A **1** in the leftmost bit position indicates that the number is negative and a **0** in the leftmost bit position indicates that it is positive."

"How does the CPU know when a program is dealing with a signed number or with an unsigned number?" asked Pat.

"The CPU does not really 'know' whether it is dealing with a signed number or an unsigned number. It just executes the instructions it has been given. It is the programmer that decides which variables in the program contain signed numbers and which variables contain unsigned numbers. It is the object that the programmer is modeling with the program that is used to make this determination.

"Since the CPU does not 'know' which values represent signed numbers and which values represent unsigned numbers, a flag in the status register (called the Negative flag) assumes that all the calculations that are being performed by the CPU are with signed numbers. If the value that is the result of a calculation has its leftmost bit set to a 1, then the Negative flag will also be set to a 1 to indicate the value is **negative** if it represents a signed number. If the leftmost bit is a 0, then the Negative flag will also be set to a 0 to indicate the value is **positive** if it represents a signed number."

"Do you mean that the Negative flag has been indicating whether results have been negative or not the whole time we have been tracing programs?" asked Pat.

I smiled and said "Yes."

"I missed that too!" said Pat. "Can we enter in a short program into the emulator and trace through it so that I can see the Negative flag changing?"

"Okay." I said. "If you look at the reference information for the LDA instruction you will see that every time it loads a number into the 'A' register, the Negative flag is set or cleared depending in whether or not the number was negative. I will enter a short program which contains 4 LDA instructions directly into the emulator. I will have 2 of these these instructions load positive numbers and have 2 of them load negative numbers."

I then entered the following program into the emulator using the Assemble command:

```
0200  A9 05     LDA #05h
0202  A9 80     LDA #80h
0204  A9 27     LDA #27h
0206  A9 C2     LDA #C2h
0208  00        BRK
...
```

"Which of these numbers are positive and which of them are negative Pat?"

I asked.

Pat looked at the numbers then picked up the whiteboard and wrote the following:

```
 0    5
0000 0101

 8    0
1000 0000

 2    7
0010 0111

 c    2
1100 0010
```

"The 05 is positive," said Pat "the 80 hex is negative, the 27 hex is positive, and the c2 hex is negative. Am I right?"

"Yes, you are right!" I replied. "Now trace through the program and see if the Negative flag agrees with you."

Pat then traced through the program:

```
-t 0200

PgmCntr(PC)  Accum(AC)  XReg(XR)  YReg(YR)  StkPtr(SP)  NV-BDIZC(SR)
   0202         05         FC        00         FD      00010100

0202  A9 80     LDA #80h

-t

PgmCntr(PC)  Accum(AC)  XReg(XR)  YReg(YR)  StkPtr(SP)  NV-BDIZC(SR)
   0204         80         FC        00         FD      10010100

0204  A9 27     LDA #27h

-t

PgmCntr(PC)  Accum(AC)  XReg(XR)  YReg(YR)  StkPtr(SP)  NV-BDIZC(SR)
   0206         27         FC        00         FD      00010100

0206  A9 C2     LDA #C2h

-t
```

```
PgmCntr(PC)  Accum(AC)  XReg(XR)  YReg(YR)  StkPtr(SP)  NV-BDIZC(SR)
   0208         C2         FC        00         FD       10010100

0208  00        BRK
```

"The Negative flag agreed with me!" said Pat.

"Yes it did." I replied. "Now we can look at how a branch instruction branches backwards in memory."

Backward Branches And Loops

"When I was young Pat," I said "I read a story about a man who had found a ring that would send him one minute backwards in time when he pressed it. The ring would not work again until the minute had passed again, so the furtherest he could ever go back in time was just one minute. He eventually figured out how to use the ring to win money at gambling establishments and he did this until he was very rich. One day he decided to spend some of his money by taking a trip to a foreign country. While he was on the plane traveling high above the ocean, a meteor hit the plane and ripped a large hole in the fuselage. He was thrown through the hole and knocked unconscious. When he awoke, he found himself falling towards the ocean."

"What did he do!? asked Pat.

"What do you think he did?" I said.

"He pressed the ring!" cried Pat "and put himself one minute back in time!"

"Yes, he did," I said "but after he pressed the ring, he found that he was still falling over the ocean, jut higher up than he was before."

"Oh no!" said Pat.   "He couldn't press the ring again until a minute had passed so he was stuck repeating his fall towards the ocean over and over again! How awful!"

"I agree," I said "and to this day I can still see the man being placed at the top of his fall and then falling, over and over again, in an infinite loop. What brought the story to mind was that when a computer uses a branch instruction or a jump instruction to move the Program Counter backwards in memory, it is similar to the man in the story falling in an infinite loop."

"It is?" asked Pat. "How?"

"When the Program Counter is set to an earlier part of memory, the instructions that have already been executed are executed again. When the branch or the jump instruction is encountered again, it acts like the man's ring and sends the Program Counter back to the earlier set of instructions. Sections of code that execute over and over like this are called **loops**. Usually, there is some logic that is placed within a loop that will allow the loop to eventually be exited. The word **logic** in this context means a group of instructions that work together to accomplish a given purpose. If loop exit logic does not exist, or if the logic was written incorrectly, the loop will loop forever. Loops that do not contain exit logic are called **infinite loops**."

"Can an infinite loop really run forever?" asked Pat.

"Not really." I replied. "An infinite loop can be forced to exit by the operating system, by pressing the computer's reset button, or by shutting the computer off. Even if the computer were permitted to run continuously, a part in it would eventually wear out which would cause it to crash. Therefore, an infinite loop is really only infinite in theory."

"Can you show me an infinite loop?" asked Pat. "I would like to see one."

"Yes, an infinite loop is easy to create." I said "I will enter a short program directly into the emulator that contains an infinite loop and then I will let you trace through it. Pay close attention to the contents of the program counter as you trace."

I then entered the following program and let Pat trace it.:

```
-u 0200

0200  A9 01     LDA #01h
0202  A2 02     LDX #02h
0204  4C 00 02  JMP 0200h
0207  00        BRK
...

-t 0200

PgmCntr(PC)  Accum(AC)  XReg(XR)  YReg(YR)  StkPtr(SP)  NV-BDIZC(SR)
   0202         01         FC        00        FD       00010100

0202  A2 02     LDX #02h

-t
```

```
PgmCntr(PC)   Accum(AC)   XReg(XR)   YReg(YR)   StkPtr(SP)   NV-BDIZC(SR)
   0204          01          02         00         FD        00010100

0204  4C 00 02  JMP 0200h
-t

PgmCntr(PC)   Accum(AC)   XReg(XR)   YReg(YR)   StkPtr(SP)   NV-BDIZC(SR)
   0200          01          02         00         FD        00010100

0200  A9 01     LDA #01h
-t

PgmCntr(PC)   Accum(AC)   XReg(XR)   YReg(YR)   StkPtr(SP)   NV-BDIZC(SR)
   0202          01          02         00         FD        00010100

0202  A2 02     LDX #02h
-t

PgmCntr(PC)   Accum(AC)   XReg(XR)   YReg(YR)   StkPtr(SP)   NV-BDIZC(SR)
   0204          01          02         00         FD        00010100

0204  4C 00 02  JMP 0200h
-t

PgmCntr(PC)   Accum(AC)   XReg(XR)   YReg(YR)   StkPtr(SP)   NV-BDIZC(SR)
   0200          01          02         00         FD        00010100

0200  A9 01     LDA #01h
-t

PgmCntr(PC)   Accum(AC)   XReg(XR)   YReg(YR)   StkPtr(SP)   NV-BDIZC(SR)
   0202          01          02         00         FD        00010100

0202  A2 02     LDX #02h
-t

PgmCntr(PC)   Accum(AC)   XReg(XR)   YReg(YR)   StkPtr(SP)   NV-BDIZC(SR)
   0204          01          02         00         FD        00010100

0204  4C 00 02  JMP 0200h
-t

PgmCntr(PC)   Accum(AC)   XReg(XR)   YReg(YR)   StkPtr(SP)   NV-BDIZC(SR)
   0200          01          02         00         FD        00010100

0200  A9 01     LDA #01h
```

"Wow, it does run in an infinite loop!" said Pat. "Can you now show me a loop that will run for a while and then exit?"

"Yes, this is also easy to do." I said. "I will create a small program that will place the number 4 into the X register and then decrement the contents of the X register inside a loop until it reaches 0. When it reaches 0, the loop will exit. This time, pay close attention to the X register, the Program Counter, and the Zero flag."

I then created the following program and had Pat trace through it:

```
-u 0200

0200  A2 04     LDX #04h
0202  CA        DEX
0203  D0 FD     BNE 0202h
0205  00        BRK
...

-t 0200

PgmCntr(PC)  Accum(AC)  XReg(XR)  YReg(YR)  StkPtr(SP)  NV-BDIZC(SR)
   0202         00         04        00        FD       00010100

0202  CA        DEX

-t

PgmCntr(PC)  Accum(AC)  XReg(XR)  YReg(YR)  StkPtr(SP)  NV-BDIZC(SR)
   0203         00         03        00        FD       00010100

0203  D0 FD     BNE 0202h

-t

PgmCntr(PC)  Accum(AC)  XReg(XR)  YReg(YR)  StkPtr(SP)  NV-BDIZC(SR)
   0202         00         03        00        FD       00010100

0202  CA        DEX

-t

PgmCntr(PC)  Accum(AC)  XReg(XR)  YReg(YR)  StkPtr(SP)  NV-BDIZC(SR)
   0203         00         02        00        FD       00010100

0203  D0 FD     BNE 0202h

-t

PgmCntr(PC)  Accum(AC)  XReg(XR)  YReg(YR)  StkPtr(SP)  NV-BDIZC(SR)
```

```
   0202         00          02          00          FD        00010100

0202  CA        DEX

-t

PgmCntr(PC)  Accum(AC)  XReg(XR)  YReg(YR)  StkPtr(SP)  NV-BDIZC(SR)
   0202         00          01          00          FD        00010100

0203  D0 FD     BNE 0202h

-t

PgmCntr(PC)  Accum(AC)  XReg(XR)  YReg(YR)  StkPtr(SP)  NV-BDIZC(SR)
   0202         00          01          00          FD        00010100

0202  CA        DEX

-t

PgmCntr(PC)  Accum(AC)  XReg(XR)  YReg(YR)  StkPtr(SP)  NV-BDIZC(SR)
   0203         00          00          00          FD        00010110

0203  D0 FD     BNE 0202h

-t

PgmCntr(PC)  Accum(AC)  XReg(XR)  YReg(YR)  StkPtr(SP)  NV-BDIZC(SR)
   0205         00          00          00          FD        00010110

0205  00        BRK
```

"What did the program do?" I asked.

"The loop kept looping until the X register was decremented to 0, then the Zero flag was set and the BNE instruction fell through to the next instruction instead of taking the branch." said Pat.

"Correct." I said. "Now, look at the program again and tell me what the operand is for the BNE instruction."

Pat looked at the program and then said "FD hex? That seems like too large of a number... wait, the BNE is branching **backwards** in memory so it must be a **negative** number!"

"It is indeed a negative number, Pat." I said. "Can you determine what the number is in decimal?"

"Hmmm," said Pat "FD hex is equal to 11111101 in binary. Just a bit ago we created a table which showed 4-bit binary numerals and their positive and negative decimal equivalents. I am guessing that if we just extend this table to 8 bits and added a column for hex numerals, we can figure out what FD hex is equivalent to in decimal."

"Go ahead and extend the table then." I said. Pat then modified the table. (see Fig. 5)

Figure 5

Binary		Hex		Dec
11111000	-	F8	-	-8
11111001	-	F9	-	-7
11111010	-	FA	-	-6
11111011	-	FB	-	-5
11111100	-	FC	-	-4
11111101	-	FD	-	-3
11111110	-	FE	-	-2
11111111	-	FF	-	-1
00000000	-	00	-	0
00000001	-	01	-	1
00000010	-	02	-	2
00000011	-	03	-	3
00000100	-	04	-	4
00000101	-	05	-	5
00000110	-	06	-	6
00000111	-	07	-	7

"FD hex is equal to -3 decimal!" said Pat.

"Look at the program again and tell me how many locations backwards in memory the address is that the BNE is branching to from the address of the instruction that is underneath it."

Pat counted the addresses then said "3 memory locations, that's cool!"

"I agree," I said "the way loops work is strange, simple, and exciting!"

"What else can loops do?" asked Pat.

"The ability to execute a group of instructions over and over again by looping," I replied "is one of the fundamental capabilities that give a computer its enormous power. In fact, machines of all types derive much of their power from the principle of **repeated cycling**.

A simple example of this is a car tire. A tire would not be very useful if it could only be rolled through one revolution. This brings to mind the image of a person who just purchased a brand new car at a dealership. The papers have been signed, the whole family (including the dog) has just been loaded into the car, and they are ready to drive home. The person starts the car, puts it into drive, moves forward one full revolution of the tires, and stops. The person then jacks up the car, removes the tires, discards them, puts on a set of new ones, lowers the car, then drives forward one more revolution of the tires. This process is continued all the way home!"

Pat burst out laughing and I did too!

I then continued "Other examples of machines that make use of the repeated cycles principle include internal combustion engines, sewing machines, hammers, screws, drills, and pumps. Many more examples exist, but they are too numerous to list."

"I hadn't thought about it before," said Pat "but you're right, lots of machines repeat their cycles. I also never would have guessed that computers repeat cycles too because, from the outside, it looks like they just sit there."

"In a program," I said "loops are used for all kinds of purposes like adding series of numbers together, repeatedly checking to see if an event (like the pressing of a keyboard key) has occurred, moving graphics across a screen, searching files, generating sounds, and spell checking documents."

"Can we create a program that uses a loop to do something useful?" asked Pat. "Maybe something simple like adding a series of numbers together."

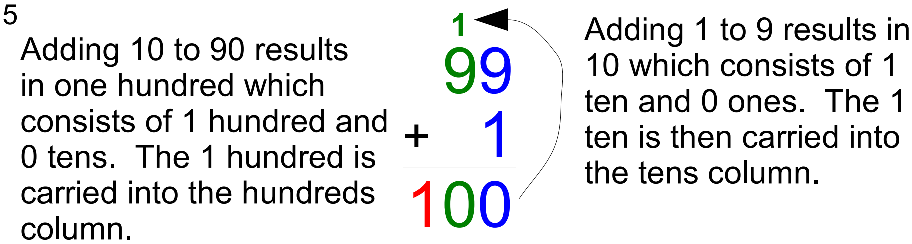
"Yes, we can do this." I said. "But first we need to talk about the Carry flag, indexed addressing modes, and commenting programs.

The Carry Flag

"What I would like you to do now Pat," I said "is to add 1 to 99 decimal on the whiteboard and explain how carrying works when an addition in a given column results in a number that is too large to fit in that column."

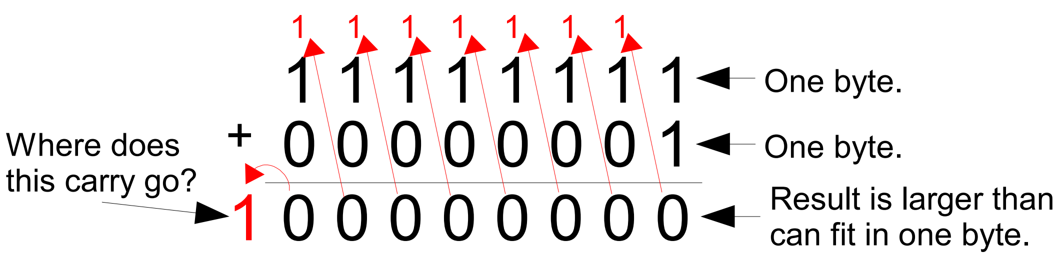
Pat added 1 to 99 decimal on the whiteboard then said "Starting in the ones column, 1 is added to 9 and the result is 1 ten and 0 ones. The 10 will not fit into the one's column, so it is carried over to the tens column. The 90 that is in the tens column is then added to the 10 that was carried over there and the result is 1 hundred and 0 tens. The 1 hundred is too large to fit into the tens column, so it is carried over to the hundreds column." (see Fig. 5)

Figure 5



"Very good Pat." I said "Now I am going to do another addition on the whiteboard except I will be adding 1 to 11111111 binary." (see Fig. 6)

Figure 6



"1 + 1 binary equals 10 binary." I said. "Notice how the bits from each addition in each column are carried over to the column to the left of it. Also notice that the result is a 9 bit number, not an 8 bit number."

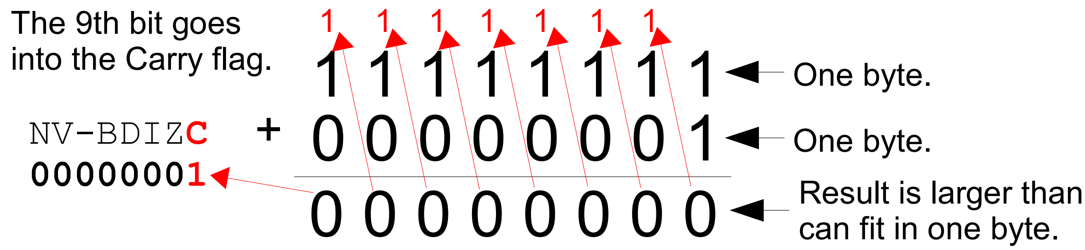
"Uh Oh," said Pat "we have a problem."

"What is the problem?" I asked.

"Our registers are only 8 bits wide so where is the 9th bit going?" replied Pat.

"You are very observant." I said. "Our registers are only 8 bits wide and so are our memory locations. Even if our registers were wider, we would still run into a problem like this eventually when we started using larger numbers. This is the problem that the **Carry flag** has been designed to solve and the way it does it is like this." I then added information about the carry flag to the diagram on the whiteboard (see Fig. 7)

Figure 7



Pat studied the diagram then said "But what happens to the bit after it has been placed into the Carry flag?"

"Have you ever wondered what the 'C' means in the ADC instruction's name?" I asked.

"Yes, I've wondered about it because it always seemed to me that this instruction should have been called ADD instead of ADC." replied Pat.

"The 'C' stands for Carry," I said "and what this means is that the ADC instruction will add the value in the 'A' register with a value in memory **and to this sum it will add the contents of the Carry flag**. Therefore, the correct name of the ADC instruction is ADd with Carry."

"Wait a minute!" said Pat. "If the ADC instruction always includes the value of the Carry flag in its calculations, what happens if the Carry flag just happens to be set to 1 when a calculation is performed? Wouldn't it result in the answer being one more than it should be?"

"Yes," I replied "and this is why a CLC or CLear Carry instruction is always placed just before an ADC instruction unless a multi-byte addition is being performed."

"But we haven't been placing a CLC instruction before our ADC instructions," said Pat "so why have our answers have been coming out okay?"

"The reason that our answers have been correct so far," I said "is because the emulator and the monitor have been programmed to launch with the Carry flag set to 0. I have not been placing a CLC instruction ahead of the ADC instructions we have been using because I was not ready yet to tell you about how the Status register's flags worked."

"That was probably a good idea," said Pat "because I don't think I would have been able to understand what the flags did if you had told me about them earlier than you did. Now that I know about the Carry flag, though, can you show me how it is used to add together 2 bytes that have a result that is larger than 8 bits?"

"Yes." I said "I will create a small program that performs the addition from the example on the whiteboard and you then can trace it."

I created the following program:

```
0200             000001 |      org 0200h
                 000002 |
0200 FF          000003 |number1 dbt 11111111b
0201 01          000004 |number2 dbt 00000001b
                 000005 |
0205             000006 |      org 0205h
                 000007 |
0205 AD 00 02    000008 |      lda number1
0208 18          000009 |      clc
0209 6D 01 02    000010 |      adc number2
                 000011 |
020C 00          000012 |      brk
                 000013 |
                 000014 |      end
                 000015 |
```

And then Pat dumped it, unassembled it, and traced through it:

```
-d 0200

0200  FF 01 00 00 00 AD 00 02 - 18 6D 01 02 00 00 00 00  .........m......

-u 0205

0205  AD 00 02  LDA 0200h
0208  18        CLC
0209  6D 01 02  ADC 0201h
020C  00        BRK
...

-t 0205

PgmCntr(PC)  Accum(AC)  XReg(XR)  YReg(YR)  StkPtr(SP)  NV-BDIZC(SR)
   0208         FF         FC        00         FD       10010100

0208  18        CLC

-t
```

```
PgmCntr(PC)  Accum(AC)  XReg(XR)  YReg(YR)  StkPtr(SP)  NV-BDIZC(SR)
   0209         FF         FC        00         FD       10010100

0209  6D 01 02  ADC 0201h

-t

PgmCntr(PC)  Accum(AC)  XReg(XR)  YReg(YR)  StkPtr(SP)  NV-BDIZC(SR)
   020C         00         FC        00         FD       00010111

020C  00        BRK
```

"Notice that after the ADC instruction was executed," I said "it resulted in 00 being placed in the 'A' register and the Carry flag being set to 1. This matches the calculation we made on the whiteboard." (again, see Fig. 7).

Indexed Addressing Modes And Commenting Programs

"Now that you know how the Carry flag works Pat," I said "we can create a program that adds a series of numbers together in a loop. In order to do this, however, we will need to use one of the indexed addressing modes."

"What does an indexed addressing mode do?" asked Pat.

I replied "An indexed addressing mode uses the contents of either the X register or the Y register as an offset from some **base address** to determine what is called the **effective address**.

For example, with the **Absolute,X** addressing mode, the programmer specifies an **absolute address** to use as the **base address** and then the contents of the X register are added to this **base address** to determine the **effective address** that will be accessed by the instruction."

"I don't get it." said Pat, with a confused look.

"Then I will create a program that shows how Absolute,X addressing works, trace through it, and then we will discuss it."

I then created the following program and traced it:

```
0200             000001 |      org 0200h
                 000002 |
0200 41          000003 |nums  dbt 41h,42h,43h,44h,45h
0201 42
0202 43
```

```
0203 44
0204 45
0205 46
                    000004 |
0210                000005 |      org 0210h
                    000006 |
0210 A2 02          000007 |      ldx #02d
0212 BD 00 02       000008 |      lda nums,x
                    000009 |
0215 00             000010 |      brk
                    000011 |
                    000012 |      end
                    000013 |
```

```
-d 0200

0200  41 42 43 44 45 46 00 00 - 00 00 00 00 00 00 00 00  ABCDEF..........


-u 0210

0210  A2 02     LDX #02h
0212  BD 00 02  LDA 0200h,X
0215  00        BRK
...

-t 0210

PgmCntr(PC)  Accum(AC)  XReg(XR)  YReg(YR)  StkPtr(SP)  NV-BDIZC(SR)
   0212         00         02        00         FD       00010100

0212  BD 00 02  LDA 0200h,X

-t

PgmCntr(PC)  Accum(AC)  XReg(XR)  YReg(YR)  StkPtr(SP)  NV-BDIZC(SR)
   0215         43         02        00         FD       00010100

0215  00        BRK
```

"The LDA instruction in this program uses the **Absolute,X** addressing mode to determine the memory location which it will copy the value from." I said "This memory location is called the **effective address**. The **base address** is **0200** hex and **02** has already been loaded into the X register. The **effective address** is calculated by adding the base address to the contents of the X register which, in this case, is 0200 hex + 02 which equals 0202 hex."

"What did I place into memory starting at location 0200h, Pat?" I asked.

Pat looked at the program and said "You placed a variable there called

nums, but instead of defining a single byte at address 0200 hex, you placed a series of 5 bytes in this area of memory with the first byte being located at address 0200 hex. I didn't know that the **dbt** directive could be used to place a series of bytes into memory, thats interesting."

"When a group of values that are related to each other are placed into consecutive memory locations like this," I said "they are referred to as a **table**, an **array**, **or a list**. This array consists of 5 bytes and these bytes just happen to contain the first 5 capital ASCII letters.

When the instruction **lda nums,x** was executed, it took the address of **nums** (which is 0200 hex) and added to it the contents of the X register (which is 02). It then used the resulting sum (0202 hex) to determine which memory location to copy the value from. What number is at address 0202 hex, Pat?"

Pat looked at the program and said "43 hex."

"And what number was loaded into the 'A' register when it was traced?" I asked.

"43 hex!" Pat replied. "The Absolute,X addressing mode worked!"

"Yes it did," I replied "now I will create a program that determines the sum of an array of numbers."

Here is the program I created:

```
                000001 |;The purpose of this program is to calculate the
                000002 |;sum of the array nums and then to place the
                000003 |;result into the variable sum.
                000004 |
0200            000005 |     org 0200h
                000006 |
                000007 |;An array of 10 bytes.
0200 01         000008 |nums  dbt 1d,2d,3d,4d,5d,6d,7d,8d,9d,10d
0201 02
0202 03
0203 04
0204 05
0205 06
0206 07
0207 08
0208 09
0209 0A
```

```
                  000009 |
                  000010 |;Holds the sum of array at nums.
020A 00           000011 |sum   dbt 0d
                  000012 |
0250              000013 |      org 0250h
                  000014 |
                  000015 |;Initialize the X register so that it offsets 0
                  000016 |;positions into the array nums.
0250 A2 00        000017 |      ldx #0d
                  000018 |
                  000019 |;Initialize register 'A' to 0.  This needs to be done
                  000020 |;so that an old value in 'A' does not produce a wrong
                  000021 |;sum during the first loop iteration.
0252 A9 00        000022 |      lda #0d
                  000023 |
                  000024 |;Clear the carry flag so that it does not cause a
                  000025 |;wrong sum to be calculated by the ADC instruction.
0254 18           000026 |      clc
                  000027 |
                  000028 |;This label is the top of the calculation loop.
0255              000029 |AddMore  *
                  000030 |
                  000031 |;Obtain a value from the array at offset X positions
                  000032 |;into the array and add this value to the contents
                  000033 |;of the 'A' register.
0255 7D 00 02     000034 |      adc nums,x
                  000035 |
                  000036 |;Increment X to the next offset position.
0258 E8           000037 |      inx
                  000038 |
                  000039 |;If X has been incremented to 10, fall through the
                  000040 |;bottom of the loop.  If X is less than 10 then loop
                  000041 |;back to AddMore and add another value from the array.
0259 E0 0A        000042 |      cpx #10d
025B D0 F8        000043 |      bne AddMore
                  000044 |
                  000045 |;After the loop has finished calculating the sum of
                  000046 |;the array, store this sum into the variable called
                  000047 |;'sum'.
025D 8D 0A 02     000048 |      sta sum
                  000049 |
                  000050 |;Return program control back to the monitor.
0260 00           000051 |      brk
                  000052 |
                  000053 |;The end command must have at least 1 blank line
                  000054 |;underneath it.
                  000055 |
                  000056 |      end
                  000057 |
```

"What are all those lines that begin with semicolons for?" asked Pat

"Those are called **comments**, I replied "and their purpose is to explain what the various parts of a program do. The semicolon tells the assembler to ignore everything after them on the line. Comment lines are ignored by the assembler and none of their content makes it into the program. Up to this point our programs have been small enough that they did not need commenting, but from here on the programs will be more sophisticated. If sophisticated programs are not commented, it is very difficult to keep track of what they are doing."

"I can believe that," said Pat "because I was even having trouble keeping track of what the smaller programs were doing."

After Pat had finished studying the program and reading the comments it contained, I loaded it into the emulator and executed it with a Go command:

```
-d 0200

0200  01 02 03 04 05 06 07 08 - 09 0A 00 00 00 00 00 00   ................

-u 0250

0250  A2 00     LDX #00h
0252  A9 00     LDA #00h
0254  18        CLC
0255  7D 00 02  ADC 0200h,X
0258  E8        INX
0259  E0 0A     CPX #0Ah
025B  D0 F8     BNE 0255h
025D  8D 0A 02  STA 020Ah
0260  00        BRK
...

r

PgmCntr(PC)  Accum(AC)  XReg(XR)  YReg(YR)  StkPtr(SP)  NV-BDIZC(SR)
   102C         00         FC        00        FD       00010110

-g 0250

PgmCntr(PC)  Accum(AC)  XReg(XR)  YReg(YR)  StkPtr(SP)  NV-BDIZC(SR)
   0260         37         0A        FF        FD       00010111

-d 0200

0200  01 02 03 04 05 06 07 08 - 09 0A 37 00 00 00 00 00   ..........7.....
```

"What values were in the 'A' register and in the variable 'sum' before the program was executed?" I asked.

"0 and 0." replied Pat.

"And what values were in the 'A' register and in the variable 'sum' after the program was executed?" I asked.

"37 hex and 37 hex." replied Pat.

"What is 37 hex in decimal?" I asked.

Pat picked up the calculator that was on the table, pressed some of its buttons then said "55."

"Finally," I asked "what is the sum of 1+2+3+4+5+6+7+8+9+10?"

Pat calculated the sum on the calculator then said "55! It worked! But now I want to trace through the program so I can see it work step-by-step."

Pat then did this and so should you.

Exercises

1) The source code for the umon65 monitor is in the umon65 directory in the download file that contained the emulator. Open this file and study it.

2) Write an assembly language program that adds the numbers 1,2,3,4,5, and 6 together and places the sum into location 0280h. Assemble the program, load it into the emulator, run it, and verify that it works correctly.

Appendix A - 6502 Instruction Set Reference (minus zero page addressing)

Registers:

```
PC    ....  program counter              (16 bit)
AC    ....  accumulator                  (8 bit)
X     ....  X register                   (8 bit)
Y     ....  Y register                   (8 bit)
SR    ....  status register [NV-BDIZC]   (8 bit)
SP    ....  stack pointer                (8 bit)
```

Status Register (SR) Flags (bit 7 to bit 0):

```
N     ....  Negative
V     ....  Overflow
-     ....  ignored
B     ....  Break
D     ....  Decimal (use BCD for arithmetics)
I     ....  Interrupt (IRQ disable)
Z     ....  Zero
C     ....  Carry
```

Processor Stack:

Top down, 0x0100 - 0x01FF

Words:

16 bit words in lowbyte-highbyte representation (Little-Endian).

Addressing Modes:

```
#       Immediate / OPC #$BB / Operand is byte (BB).
A       Accumulator / OPC A / Operand is AC.
abs     Absolute / OPC $HHLL / Operand is address $HHLL.
abs,X   Absolute,X-indexed / OPC $HHLL,X / Operand is address incremented by X
        with carry.
abs,Y   Absolute,Y-indexed / OPC $HHLL,Y / Operand is address incremented by Y
        with carry.
impl    Implied / OPC / Operand implied.
ind     Indirect / OPC ($HHLL) / Operand is effective address, effective
        address is value of address.
X,ind   X-indexed,indirect / OPC ($BB,X) / Operand is effective zeropage
        address, effective address is byte (BB) incremented by X without
        carry.
ind,Y   Indirect,Y-indexed / OPC ($LL),Y / Operand is effective address
        incremented by Y with carry, effective address is word at zeropage
        address.
rel     Relative / OPC $BB / Branch target is PC + offset (BB), bit 7
        signifies negative offset.
```

Instructions:

Legend to Flags:
```
+ .... modified
- .... not modified
1 .... set
0 .... cleared
M6 .... memory bit 6
M7 .... memory bit 7
```

ADC Add Memory to Accumulator with Carry

```
    A + M + C -> A, C          N Z C I D V
                               + + + - - +

    addressing    assembler    opc  bytes
    ------------------------------------
    immediate     ADC #oper     69    2
    absolute      ADC oper      6D    3
    absolute,X    ADC oper,X    7D    3
    absolute,Y    ADC oper,Y    79    3
    (indirect,X)  ADC (oper,X)  61    2
    (indirect),Y  ADC (oper),Y  71    2
```

AND AND Memory with Accumulator

```
    A AND M -> A               N Z C I D V
                               + + - - - -

    addressing    assembler    opc  bytes
    ------------------------------------
    immediate     AND #oper     29    2
    absolute      AND oper      2D    3
    absolute,X    AND oper,X    3D    3
    absolute,Y    AND oper,Y    39    3
    (indirect,X)  AND (oper,X)  21    2
    (indirect),Y  AND (oper),Y  31    2
```

ASL Shift Left One Bit (Memory or Accumulator)

```
    C <- [76543210] <- 0       N Z C I D V
                               + + + - - -

    addressing    assembler    opc  bytes
    ------------------------------------
    accumulator   ASL A         0A    1
    absolute      ASL oper      0E    3
    absolute,X    ASL oper,X    1E    3
```

BCC Branch on Carry Clear

```
    branch on C = 0            N Z C I D V
                               - - - - - -
```

```
     addressing    assembler    opc  bytes
     --------------------------------------
     relative      BCC oper      90    2
```

BCS Branch on Carry Set

```
     branch on C = 1           N Z C I D V
                               - - - - - -

     addressing    assembler    opc  bytes
     --------------------------------------
     relative      BCS oper      B0    2
```

BEQ Branch on Result Zero

```
     branch on Z = 1           N Z C I D V
                               - - - - - -

     addressing    assembler    opc  bytes
     --------------------------------------
     relative      BEQ oper      F0    2
```

BIT Test Bits in Memory with Accumulator

```
     bits 7 and 6 of operand are transfered to bit 7 and 6 of SR (N,V);
     the zeroflag is set to the result of operand AND accumulator.

     A AND M, M7 -> N, M6 -> V   N Z C I D V
                                M7 + - - - M6

     addressing    assembler    opc  bytes
     --------------------------------------
     absolute      BIT oper      2C    3
```

BMI Branch on Result Minus

```
     branch on N = 1           N Z C I D V
                               - - - - - -

     addressing    assembler    opc  bytes
     --------------------------------------
     relative      BMI oper      30    2
```

BNE Branch on Result not Zero

```
     branch on Z = 0           N Z C I D V
                               - - - - - -

     addressing    assembler    opc  bytes
     --------------------------------------
```

```
1731      relative      BNE oper      D0    2
```

1732 **BPL Branch on Result Plus**

```
1733      branch on N = 0           N Z C I D V
1734                                - - - - - -

1735      addressing    assembler    opc  bytes
1736      ------------------------------------
1737      relative      BPL oper      10    2
```

1738 **BRK Force Break**

```
1739      interrupt,                N Z C I D V
1740      push PC+2, push SR        - - - 1 - -

1741      addressing    assembler    opc  bytes
1742      ------------------------------------
1743      implied       BRK           00    1
```

1744 **BVC Branch on Overflow Clear**

```
1745      branch on V = 0           N Z C I D V
1746                                - - - - - -

1747      addressing    assembler    opc  bytes
1748      ------------------------------------
1749      relative      BVC oper      50    2
```

1750 **BVS Branch on Overflow Set**

```
1751      branch on V = 1           N Z C I D V
1752                                - - - - - -

1753      addressing    assembler    opc  bytes
1754      ------------------------------------
1755      relative      BVC oper      70    2
```

1756 **CLC Clear Carry Flag**

```
1757      0 -> C                    N Z C I D V
1758                                - - 0 - - -

1759      addressing    assembler    opc  bytes
1760      ------------------------------------
1761      implied       CLC           18    1
```

1762 **CLD Clear Decimal Mode**

```
     0 -> D                   N Z C I D V
                              - - - - 0 -

     addressing    assembler    opc  bytes
     --------------------------------------
     implied       CLD           D8    1
```

CLI Clear Interrupt Disable Bit

```
     0 -> I                   N Z C I D V
                              - - - 0 - -

     addressing    assembler    opc  bytes
     --------------------------------------
     implied       CLI           58    1
```

CLV Clear Overflow Flag

```
     0 -> V                   N Z C I D V
                              - - - - - 0

     addressing    assembler    opc  bytes
     --------------------------------------
     implied       CLV           B8    1
```

CMP Compare Memory with Accumulator

```
     A - M                    N Z C I D V
                              + + + - - -

     addressing    assembler    opc  bytes
     --------------------------------------
     immediate     CMP #oper     C9    2
     absolute      CMP oper      CD    3
     absolute,X    CMP oper,X    DD    3
     absolute,Y    CMP oper,Y    D9    3
     (indirect,X)  CMP (oper,X)  C1    2
     (indirect),Y  CMP (oper),Y  D1    2
```

CPX Compare Memory and Index X

```
     X - M                    N Z C I D V
                              + + + - - -

     addressing    assembler    opc  bytes
     --------------------------------------
     immediate     CPX #oper     E0    2
     absolute      CPX oper      EC    3
```

CPY Compare Memory and Index Y

```
     Y - M                        N Z C I D V
                                  + + + - - -

     addressing    assembler    opc  bytes
     --------------------------------------
     immediate     CPY #oper     C0    2
     absolute      CPY oper      CC    3
```

DEC Decrement Memory by One

```
     M - 1 -> M                   N Z C I D V
                                  + + - - - -

     addressing    assembler    opc  bytes
     --------------------------------------
     absolute      DEC oper      CE    3
     absolute,X    DEC oper,X    DE    3
```

DEX Decrement Index X by One

```
     X - 1 -> X                   N Z C I D V
                                  + + - - - -

     addressing    assembler    opc  bytes
     --------------------------------------
     implied       DEC           CA    1
```

DEY Decrement Index Y by One

```
     Y - 1 -> Y                   N Z C I D V
                                  + + - - - -

     addressing    assembler    opc  bytes
     --------------------------------------
     implied       DEC           88    1
```

EOR Exclusive-OR Memory with Accumulator

```
     A EOR M -> A                 N Z C I D V
                                  + + - - - -

     addressing    assembler    opc  bytes
     --------------------------------------
     immediate     EOR #oper     49    2
     absolute      EOR oper      4D    3
     absolute,X    EOR oper,X    5D    3
     absolute,Y    EOR oper,Y    59    3
     (indirect,X)  EOR (oper,X)  41    2
     (indirect),Y  EOR (oper),Y  51    2
```

INC Increment Memory by One

```
M + 1 -> M                   N Z C I D V
                             + + - - - -
```

addressing	assembler	opc	bytes
absolute	INC oper	EE	3
absolute,X	INC oper,X	FE	3

INX Increment Index X by One

```
X + 1 -> X                   N Z C I D V
                             + + - - - -
```

addressing	assembler	opc	bytes
implied	INX	E8	1

INY Increment Index Y by One

```
Y + 1 -> Y                   N Z C I D V
                             + + - - - -
```

addressing	assembler	opc	bytes
implied	INY	C8	1

JMP Jump to New Location

```
(PC+1) -> PCL                N Z C I D V
(PC+2) -> PCH                - - - - - -
```

addressing	assembler	opc	bytes
absolute	JMP oper	4C	3
indirect	JMP (oper)	6C	3

JSR Jump to New Location Saving Return Address

```
push (PC+2),                 N Z C I D V
(PC+1) -> PCL                - - - - - -
(PC+2) -> PCH
```

addressing	assembler	opc	bytes
absolute	JSR oper	20	3

LDA Load Accumulator with Memory

```
M -> A                       N Z C I D V
                             + + - - - -
```

```
addressing    assembler    opc  bytes
--------------------------------------
immediate     LDA #oper     A9    2
absolute      LDA oper      AD    3
absolute,X    LDA oper,X    BD    3
absolute,Y    LDA oper,Y    B9    3
(indirect,X)  LDA (oper,X)  A1    2
(indirect),Y  LDA (oper),Y  B1    2
```

LDX Load Index X with Memory

```
M -> X                     N Z C I D V
                           + + - - - -

addressing    assembler    opc  bytes
--------------------------------------
immediate     LDX #oper     A2    2
absolute      LDX oper      AE    3
absolute,Y    LDX oper,Y    BE    3
```

LDY Load Index Y with Memory

```
M -> Y                     N Z C I D V
                           + + - - - -

addressing    assembler    opc  bytes
--------------------------------------
immediate     LDY #oper     A0    2
absolute      LDY oper      AC    3
absolute,X    LDY oper,X    BC    3
```

LSR Shift One Bit Right (Memory or Accumulator)

```
0 -> [76543210] -> C       N Z C I D V
                           - + + - - -

addressing    assembler    opc  bytes
--------------------------------------
accumulator   LSR A         4A    1
absolute      LSR oper      4E    3
absolute,X    LSR oper,X    5E    3
```

NOP No Operation

```
---                        N Z C I D V
                           - - - - - -

addressing    assembler    opc  bytes
--------------------------------------
implied       NOP           EA    1
```

ORA OR Memory with Accumulator

```
    A OR M -> A                 N Z C I D V
                                + + - - - -

    addressing    assembler    opc  bytes
    --------------------------------------
    immediate     ORA #oper     09    2
    absolute      ORA oper      0D    3
    absolute,X    ORA oper,X    1D    3
    absolute,Y    ORA oper,Y    19    3
    (indirect,X)  ORA (oper,X)  01    2
    (indirect),Y  ORA (oper),Y  11    2
```

PHA Push Accumulator on Stack

```
    push A                      N Z C I D V
                                - - - - - -

    addressing    assembler    opc  bytes
    --------------------------------------
    implied       PHA           48    1
```

PHP Push Processor Status on Stack

```
    push SR                     N Z C I D V
                                - - - - - -

    addressing    assembler    opc  bytes
    --------------------------------------
    implied       PHP           08    1
```

PLA Pull Accumulator from Stack

```
    pull A                      N Z C I D V
                                - - - - - -

    addressing    assembler    opc  bytes
    --------------------------------------
    implied       PLA           68    1
```

PLP Pull Processor Status from Stack

```
    pull SR                     N Z C I D V
                                from stack

    addressing    assembler    opc  bytes
    --------------------------------------
    implied       PHP           28    1     4
```

ROL Rotate One Bit Left (Memory or Accumulator)

```
C <- [76543210] <- C      N Z C I D V
                          + + + - - -

addressing    assembler    opc  bytes
--------------------------------------
accumulator   ROL A         2A    1
absolute      ROL oper      2E    3
absolute,X    ROL oper,X    3E    3
```

ROR Rotate One Bit Right (Memory or Accumulator)

```
C -> [76543210] -> C      N Z C I D V
                          + + + - - -

addressing    assembler    opc  bytes
--------------------------------------
accumulator   ROR A         6A    1
absolute      ROR oper      6E    3
absolute,X    ROR oper,X    7E    3
```

RTI Return from Interrupt

```
pull SR, pull PC          N Z C I D V
                          from stack

addressing    assembler    opc  bytes
--------------------------------------
implied       RTI           40    1
```

RTS Return from Subroutine

```
pull PC, PC+1 -> PC       N Z C I D V
                          - - - - - -

addressing    assembler    opc  bytes
--------------------------------------
implied       RTS           60    1
```

SBC Subtract Memory from Accumulator with Borrow

```
A - M - C -> A            N Z C I D V
                          + + + - - +

addressing    assembler    opc  bytes
--------------------------------------
immediate     SBC #oper     E9    2
absolute      SBC oper      ED    3
absolute,X    SBC oper,X    FD    3
absolute,Y    SBC oper,Y    F9    3
(indirect,X)  SBC (oper,X)  E1    2
```

```
     (indirect),Y  SBC (oper),Y  F1    2
```

SEC Set Carry Flag

```
     1 -> C                    N Z C I D V
                               - - 1 - - -

     addressing    assembler    opc  bytes
     ------------------------------------
     implied       SEC           38    1
```

SED Set Decimal Flag

```
     1 -> D                    N Z C I D V
                               - - - - 1 -

     addressing    assembler    opc  bytes
     ------------------------------------
     implied       SED           F8    1
```

SEI Set Interrupt Disable Status

```
     1 -> I                    N Z C I D V
                               - - - 1 - -

     addressing    assembler    opc  bytes
     ------------------------------------
     implied       SEI           78    1
```

STA Store Accumulator in Memory

```
     A -> M                    N Z C I D V
                               - - - - - -

     addressing    assembler    opc  bytes
     ------------------------------------
     absolute      STA oper      8D    3
     absolute,X    STA oper,X    9D    3
     absolute,Y    STA oper,Y    99    3
     (indirect,X)  STA (oper,X)  81    2
     (indirect),Y  STA (oper),Y  91    2
```

STX Store Index X in Memory

```
     X -> M                    N Z C I D V
                               - - - - - -

     addressing    assembler    opc  bytes
     ------------------------------------
     absolute      STX oper      8E    3
```

STY Sore Index Y in Memory

```
     Y -> M                     N Z C I D V
                                - - - - - -

     addressing    assembler    opc  bytes
     ------------------------------------
     absolute      STY oper      8C    3
```

TAX Transfer Accumulator to Index X

```
     A -> X                     N Z C I D V
                                + + - - - -

     addressing    assembler    opc  bytes
     ------------------------------------
     implied       TAX           AA    1
```

TAY Transfer Accumulator to Index Y

```
     A -> Y                     N Z C I D V
                                + + - - - -

     addressing    assembler    opc  bytes
     ------------------------------------
     implied       TAY           A8    1
```

TSX Transfer Stack Pointer to Index X

```
     SP -> X                    N Z C I D V
                                + + - - - -

     addressing    assembler    opc  bytes
     ------------------------------------
     implied       TSX           BA    1
```

TXA Transfer Index X to Accumulator

```
     X -> A                     N Z C I D V
                                + + - - - -

     addressing    assembler    opc  bytes
     ------------------------------------
     implied       TXA           8A    1
```

TXS Transfer Index X to Stack Register

```
     X -> SP                    N Z C I D V
                                + + - - - -
```

```
addressing    assembler    opc  bytes
-------------------------------------
implied       TXS           9A    1
```

TYA Transfer Index Y to Accumulator

```
Y -> A                     N Z C I D V
                           + + - - - -

addressing    assembler    opc  bytes
-------------------------------------
implied       TYA           98    1
```