

6502 Assembly Language

by Ted Kosan

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

Copyright © 2008 by Ted Kosan

This work is licensed under the Creative Commons
Attribution-ShareAlike 3.0 License. To view a copy of
this license, visit <http://creativecommons.org/licenses/by-sa/3.0/>

The Professor And Pat series (professorandpat.org)

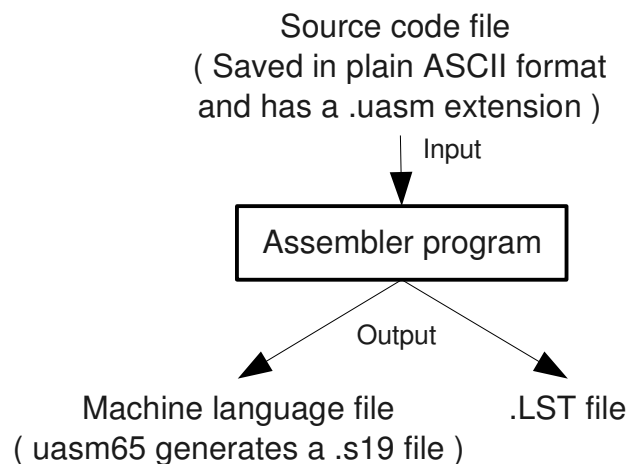
Table of Contents

Assemblers.....	3
The UASM65 Assembler, .S19 Files, and .LST files.....	6
Sending An S19 File To The Emulator.....	10
Models.....	14
Placing Models Into A Computer.....	16
Variables.....	18
The Status Register.....	22
How A Computer Makes Decisions.....	25
The JMP Instruction.....	26
Labels.....	27
Forward Branches And The Zero Flag.....	28
Negative Numbers And The Negative Flag.....	34
Backward Branches And Loops.....	38
The Carry Flag.....	44
Indexed Addressing Modes And Commenting Programs.....	48
Exercises.....	54
Appendix A - 6502 Instruction Set Reference (minus zero page addressing)	55

1 **Assemblers**

2 I was deep in thought when I heard a knock on the door of my shop.
3 "Professor, are you there?" A voice said. "Its Pat and I've come to learn
4 about assemblers!"
5 "Come in, Pat!" I said.
6 When Pat opened the door and entered, I smiled and said "have a seat next
7 to the computer and boot it up."
8 While the computer was booting I said "So, you want to learn about
9 assemblers?"
10 "Yes!" said Pat. "I couldn't stop thinking about machine language and
11 assembly language since the last time we met and now I really want to
12 know what an assembler does and how to use one."
13 I looked thoughtfully at Pat for a few moments then said "Okay, let me find
14 a whiteboard and then we will discuss assemblers." Then I drew the
15 following diagram while Pat watched. (see Fig. 1)

Figure 1



16 "An **assembler**," I said "is a program that takes a source code file that
17 contains plain ASCII characters and converts it into a file that contains
18 machine language. The type of application that is used to create a source

19 code file is called a **text editor**. Text editors allow users to create
20 documents that are similar to word processing documents, except the files
21 are saved using only plain ASCII characters. For this reason, files that only
22 contain plain ASCII characters are also called **text files**."

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

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

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

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

35 ABC
36 123
37 Hello Pat!

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

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

42 **\$ hexdump -C abc123.txt**
43 00000000 41 42 43 0d 0a 31 32 33 0d 0a 48 65 6c 6c 6f 20 |ABC...123..Hello |
44 00000010 50 61 74 21 0d 0a |Pat!...|

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

48 Pat studied the output for a few moments then said "Its output is arranged
49 into 3 columns, just like the Dump command's output is! The first ASCII
50 character in the file is a capital letter 'A' and hexdump displayed its value as
51 41 hex, just like the ASCII table showed. I see that 'B' is 42 hex, the

52 numeral '1' is 31 hex, and 'Pat' is 50 hex, 61 hex, and 74 hex. I don't
53 understand what the 0d 0a numerals are, though."

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

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

59 "Yes, Pat, this is exactly what the ASCII carriage return and line feed
60 characters do!" I said. "On some operating systems (like Windows) both a
61 carriage return and a line feed are used to drop down a line and move the
62 cursor to the left side of the screen. On other operating systems, however,
63 0A hex is used by itself for both these operations and it is call a **newline**
64 instead of a **line feed**. Another way to indicate a **carriage return**
65 **followed by a line feed** is by saying or typing **CRLF**."

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

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

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

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

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

84 "A .LST file," I replied "contains the original source code version of the

85 program that was sent to the assembler, along with the machine language
86 that each line of source code was converted into. The purpose of this file is
87 to allow the programmer to see exactly how the source code was converted
88 into machine language. We will look at a .LST file after we have assembled
89 our first program."

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

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

```
95 %uasm65,description="Example 1"  
96     org 0200h  
  
97     lda #10d  
98     adc #5d  
99     sta 0208h  
100     brk  
  
101     end  
102 %/uasm65
```

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

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

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

117 "Those are called **pseudo ops** (which is short for pseudo operations) and
118 another name for them is **assembler directives**. They are designed to look
119 like instruction mnemonics, but instead of being instructions for a CPU,

120 they are instructions which are meant for the assembler. Assembler
 121 directives allow a programmer to tell the assembler how to assemble the
 122 program.

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

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

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

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

```

134 1:%uasm65,description="Example 1"
135 2:    org 0200h
136 3:
137 4:    lda #10d
138 5:    adc #5d
139 6:    sta 0208h
140 7:    brk
141 8:
142 9:    end
143 10:%/uasm65
144 11:
145 12:    %output ,preserve="false"
146 13:    *** List file ***
147 14:
148 15:    0200          000001 | org 0200h
149 16:                000002 |
150 17:    0200 A9 0A    000003 | lda #10d
151 18:    0202 69 05    000004 | adc #5d
152 19:    0204 8D 08 02 000005 | sta 0208h
153 20:    0207 00          000006 | brk
154 21:                000007 |
155 22:                000008 | end
156 23:
157 24:    *** Executable code ***
158 25:
159 26:    %s19,descrption="Execute this fold to send program to U6502 monitor."
160 27:    S007000055415347C8

```

```
161 28:      S10B0200A90A69058D0802003A
162 29:      S9030000FC
163 30:      %/s19
164 31:      %/output
```

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

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

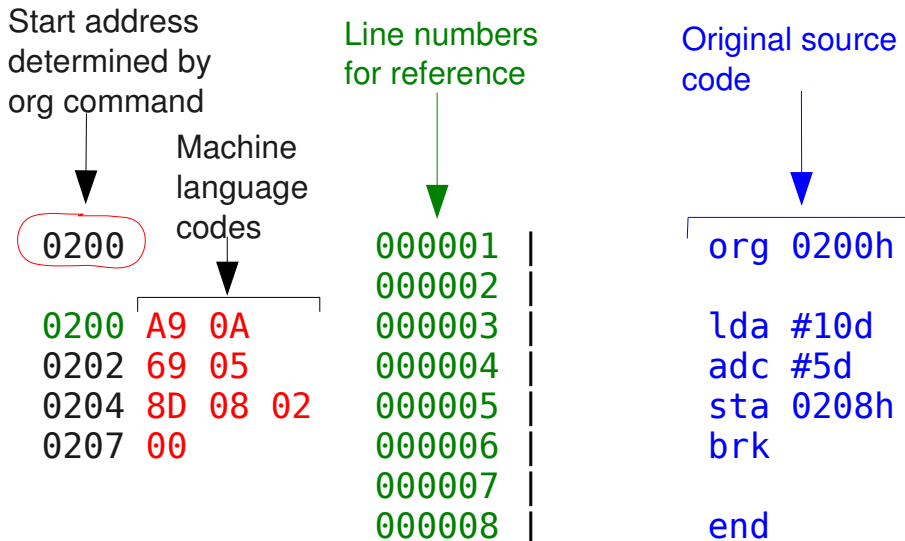
173 Pat studied the s19 code that was generated:

```
174 S007000055415347C8
175 S10B0200A90A69058D0802003A
176 S9030000FC
```

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

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

Figure 2



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

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

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

188 "Where?" I asked.

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

```
192 S007000055415347C8
193 S10B 0200 A9 0A 69 05 8D 08 02 00 3A
194 S9030000FC
```

195

196 "Very good, Pat!" I said. "The purpose of the S19 file format is to allow
 197 assembled and compiled programs to be sent to small computer systems
 198 and microcontrollers. The emulator we have been using is also able to
 199 accept s19 files and our next step is to send this program to the emulator so

200 that it can be executed. S19 files contain more detail than we have covered,
201 but we will not discuss these details at this time."

202 **Sending An S19 File To The Emulator**

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

205 ?

206	Assemble	A start_address
207	Breakpoint	B (+,-,?) address
208	Dump	D [start_address [end_address]]
209	Enter	E address list
210	Fill	F start_address end_address list
211	Go	G [start_address]
212	Help	H or ?
213	Load	L
214	Move	M start_address end_address destination_address
215	Register	R [PC,AC,XR,YR,SP,SR]
216	Search	S start_address end_address list
217	Trace	T [start_address [value]]
218	Unassemble	U [start_address [end_address]]

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

223 LOAD COMMAND

224 SYNTAX: L

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

233 "Before I load the program, I will check the area of memory near address
234 0200 hex to see what is there." I executed a Dump command and here is
235 what it displayed:

```

236 -d 0200
237 0200 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
238 "This area of memory has zeros in it and this will make it easier to see the
239 program after it is loaded." I said. "When a %s19 fold is executed by
240 pressing <shift><enter> inside of it, the emulator is automatically
241 placed into Load mode and the code inside of the fold is loaded into
242 the emulator." This is what was displayed in the monitor after the %s19
243 fold was executed:

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

245 PgmCntr(PC)  Accum(AC)  XReg(XR)  YReg(YR)  StkPtr(SP)  NV-BDIZC(SR)
246   E02C          00        16        00        FD        00000000
247 -L
248 S007000055415347C8
249 S10B0200A90A69058D0802003A
250 S9030000FC

251 Send S records when you are ready...

252 S0S1S9
253 S records successfully loaded (press <enter> if no cursor is shown).
254 -

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

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

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

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

261 -d 0200
262 0200  A9 0A 69 05 8D 08 02 00 - 00 00 00 00 00 00 00 00 ..i.....

263 -u 0200

```

```
264 0200 A9 0A LDA #0Ah
265 0202 69 05 ADC #05h
266 0204 8D 08 02 STA 0208h
267 0207 00 BRK
268 0208 00 BRK
269 0209 00 BRK
270 020A 00 BRK
271 020B 00 BRK
272 020C 00 BRK
273 020D 00 BRK
274 020E 00 BRK
275 020F 00 BRK
276 0210 00 BRK
277 0211 00 BRK
278 0212 00 BRK
279 0213 00 BRK
280 0214 00 BRK
```

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

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

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

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

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

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

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

300 I looked at Pat and said "My grandfather came from Hungary and he told

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

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

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

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

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

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

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

328 I gave Pat a questioning look.

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

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

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

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

341 **Models**

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

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

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

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

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

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

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

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

361 "I agree, and this is why **models** are usually used to represent objects
362 instead of either scaled or unscaled exact copies of the objects. A **model** is
363 a simplified representation of an object that only copies some of its

364 attributes. Examples of typical object attributes include weight, height,
365 strength, and color.

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

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

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

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

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

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

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

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

396 "Go on." I said.

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

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

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

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

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

417 "Abstraction!" replied Pat.

418 **Placing Models Into A Computer**

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

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

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

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

431 I smiled and said "You are very perceptive!"

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

433 "Yes." I replied.

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

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

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

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

446 Pat's mouth dropped open with surprise.

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

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

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

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

458 invented."

459 Variables

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

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

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

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

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

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

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

476 I brought up an editor and typed in an assembly language program that had
477 2 variables in it. Then, I assembled the program and brought up the
478 following .LST file that was generated into the text editor:

479	0200	000001		org 0200h
480		000002		
481	0200 AD 11 02	000003		lda garage_width
482	0203 69 01	000004		adc #1d
483	0205 8D 11 02	000005		sta garage_width
484		000006		
485	0208 AD 12 02	000007		lda garage_length
486	020B 69 01	000008		adc #1d
487	020D 8D 12 02	000009		sta garage_length
488	0210 00	000010		brk
489		000011		
490	0211 09	000012		garage_width dbt 9d
491	0212 08	000013		garage_length dbt 8d
492		000014		

493 000015 | end

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

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

506 "Look at the 3 assembly language instructions that have been placed into
507 memory starting at address 0200h and tell me what you think they will do
508 when they are executed." I replied.

509 Pat studied the instructions then said "The LDA instruction at address
510 0200h looks like it is copying the **9** that the variable **garage_width** refers
511 to into register 'A' . The ADC instruction is adding **1** to the **9** and this
512 should result in a **10** decimal being placed into the 'A' register. The STA
513 instruction is then copying the **10** decimal which is in the 'A' register back
514 into memory at the address that **garage_width** refers to.

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

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

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

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

524 Pat loaded the program and executed the two commands. This is what was

525 displayed on the screen:

526 -d 0200 021f

527 0200 AD 11 02 69 01 8D 11 02 - AD 12 02 69 01 8D 12 02 ...i.....i....
528 0210 00 09 08 00 00 00 00 00 - 00 00 00 00 00 00 00 00

529 -u 0200

530 0200 AD 11 02 LDA 0211h
531 0203 69 01 ADC #01h
532 0205 8D 11 02 STA 0211h
533 0208 AD 12 02 LDA 0212h
534 020B 69 01 ADC #01h
535 020D 8D 12 02 STA 0212h
536 0210 00 BRK
537 0211 09 08 ORA #08h
538 0213 00 BRK
539 0214 00 BRK

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

542 Pat looked at the contents of these locations then replied "Memory location
543 0211h contains a **9** and memory location 0212h contains an **8**! These
544 numbers are what we put into the **garage_width** and the **garage_length**
545 variables!"

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

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

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

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

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

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

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

563 Pat did this and here is what was displayed:

564 -r

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

567 -t 0200

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

570 0203 69 01 ADC #01h

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

573 "Yes." replied Pat.

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

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

579 -t

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

582 0205 8D 11 02 STA 0211h

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

```

586 -d 0211
587 0211 09 08 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00 .....

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

592 -t

593 PgmCntr(PC)  Accum(AC)  XReg(XR)  YReg(YR)  StkPtr(SP)  NV-BDIZC(SR)
594    0208           0A          FC         00         FD       00010100

595 0208 AD 12 02 LDA 0212h

596 -d 0211
597 0211 0A 08 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00 .....

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

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

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

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

605 Here are some final thoughts on variables. Their names need to consist of
606 ASCII characters from 33 decimal through 122 decimal. The one exception
607 to this is that variable names cannot contain a semi-colon which is an ASCII
608 59 decimal. Variables also need to be placed up against the left side
609 of the editor window with no spaces or tabs to the left of them.

610 The Status Register

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

```

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

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

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

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

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

```
647 0200 A2 03    LDX #03h
648 0202 CA      DEX
649 0203 CA      DEX
650 0204 CA      DEX
```

```

651  0205  00          BRK

652  -r

653  PgmCntr(PC)  Accum(AC)  XReg(XR)  YReg(YR)  StkPtr(SP)  NV-BDIZC(SR)
654    102C      00        FC      00        FD      00010110

655  -t 0200

656  PgmCntr(PC)  Accum(AC)  XReg(XR)  YReg(YR)  StkPtr(SP)  NV-BDIZC(SR)
657    0202      00        03      00        FD      00010100

658  0202  CA          DEX

659  -t

660  PgmCntr(PC)  Accum(AC)  XReg(XR)  YReg(YR)  StkPtr(SP)  NV-BDIZC(SR)
661    0203      00        02      00        FD      00010100

662  0203  CA          DEX

663  -t

664  PgmCntr(PC)  Accum(AC)  XReg(XR)  YReg(YR)  StkPtr(SP)  NV-BDIZC(SR)
665    0204      00        01      00        FD      00010100

666  0204  CA          DEX

667  -t

668  PgmCntr(PC)  Accum(AC)  XReg(XR)  YReg(YR)  StkPtr(SP)  NV-BDIZC(SR)
669    0205      00        00      00        FD      00010110

670  0205  00          BRK

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

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

678  "Much of life is like that, Pat." I said. "Amazing and wonderful things lay
679  spread before us in open sight, but we are blind to them for want of

```


680 awareness. Some say that striving for awareness is one of the noblest goals
681 that a person can pursue".

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

685 **How A Computer Makes Decisions**

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

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

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

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

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

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

705 "Yes." I said.

706 "How?" asked Pat.

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

711 The JMP Instruction

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

```
714 0200 A9 01    LDA #01h
715 0202 4C 07 02 JMP 0207h
716 0205 A2 02    LDX #02h
717 0207 A0 03    LDY #03h
718 0209 EA      NOP
719 020A 00      BRK
720 ...
```

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

724 -r

725	PgmCntr(PC)	Accum(AC)	XReg(XR)	YReg(YR)	StkPtr(SP)	NV-BDIZC(SR)
726	102C	00	FC	00	FD	00010110

727 -t 0200

728	PgmCntr(PC)	Accum(AC)	XReg(XR)	YReg(YR)	StkPtr(SP)	NV-BDIZC(SR)
729	0202	01	FC	00	FD	00010100

```
730 0202 4C 07 02 JMP 0207h
```

731 -t

732	PgmCntr(PC)	Accum(AC)	XReg(XR)	YReg(YR)	StkPtr(SP)	NV-BDIZC(SR)
733	0207	01	FC	00	FD	00010100

```
734 0207 A0 03    LDY #03h
```

735 -t

736	PgmCntr(PC)	Accum(AC)	XReg(XR)	YReg(YR)	StkPtr(SP)	NV-BDIZC(SR)
737	0209	01	FC	03	FD	00010100

```
738 0209 EA      NOP
```

739 -t

740	PgmCntr(PC)	Accum(AC)	XReg(XR)	YReg(YR)	StkPtr(SP)	NV-BDIZC(SR)
741	020A	01	FC	03	FD	00010100

```
742 020A 00      BRK
```

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

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

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

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

756 **Labels**

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

766	0200	000001		org 0200h
767		000002		
768	0200 A9 01	000003		lda #01d
769	0202 4C 07 02	000004		jmp skip1
770		000005		
771	0205 A9 02	000006		lda #02d
772		000007		
773	0207 A9 03	000008	skip1	lda #03d
774	0209 4C 0E 02	000009		jmp skip2
775		000010		
776	020C A9 04	000011		lda #04d
777		000012		
778	020E 00	000013	skip2	brk
779		000014		
780		000015		end

781 000016 |

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

788 **Forward Branches And The Zero Flag**

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

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

797 BCC - Branch on Carry Clear.
798 BCS - Branch on Carry Set.

799 BEQ - Branch on result Equal.
800 BNE - Branch on result Not Equal.

801 BMI - Branch on result MInus.
802 BPL - Branch on result PPlus.

803 BVC - Branch on oVerflow Clear.
804 BVS - Branch on oVerflow Set.

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

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

810 "How are they related?" asked Pat.

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

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

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

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

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

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

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

843 First, I created the following program:

```
844 0200          000001 |      org 0200h
```

```

845          000002 |
846 0200 A9 02      000003 |      lda #02d
847 0202 C9 02      000004 |      cmp #02d
848 0204 F0 01      000005 |      beq Equal1
849          000006 |
850 0206          000007 |NotEqual1 *
851 0206 EA          000008 |      nop
852          000009 |
853 0207          000010 |Equal1 *
854 0207 EA          000011 |      nop
855 0208 A9 05      000012 |      lda #05d
856 020A C9 06      000013 |      cmp #06d
857 020C F0 02      000014 |      beq Equal2
858 020E EA          000015 |      nop
859          000016 |
860 020F          000017 |NotEqual2 *
861 020F EA          000018 |      nop
862          000019 |
863 0210          000020 |Equal2 *
864          000021 |
865 0210 00          000022 |      brk
866          000023 |      end
867          000024 |

```

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

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

876 "Okay." said Pat.

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

```

879 -u 0200

880 0200 A9 02      LDA #02h
881 0202 C9 02      CMP #02h
882 0204 F0 01      BEQ 0207h
883 0206 EA          NOP
884 0207 EA          NOP
885 0208 A9 05      LDA #05h

```

```

886 020A C9 06    CMP #06h
887 020C F0 02    BEQ 0210h
888 020E EA      NOP
889 020F EA      NOP
890 0210 00      BRK
891 ...

892 -t 0200

893 PgmCntr(PC) Accum(AC) XReg(XR) YReg(YR) StkPtr(SP) NV-BDIZC(SR)
894   0202      02      FC      00      FD      00010100

895 0202 C9 02    CMP #02h

896 -t

897 PgmCntr(PC) Accum(AC) XReg(XR) YReg(YR) StkPtr(SP) NV-BDIZC(SR)
898   0204      02      FC      00      FD      00010111

899 0204 F0 01    BEQ 0207h

900 -t

901 PgmCntr(PC) Accum(AC) XReg(XR) YReg(YR) StkPtr(SP) NV-BDIZC(SR)
902   0207      02      FC      00      FD      00010111

903 0207 EA      NOP

904 -t

905 PgmCntr(PC) Accum(AC) XReg(XR) YReg(YR) StkPtr(SP) NV-BDIZC(SR)
906   0208      02      FC      00      FD      00010111

907 0208 A9 05    LDA #05h

908 -t

909 PgmCntr(PC) Accum(AC) XReg(XR) YReg(YR) StkPtr(SP) NV-BDIZC(SR)
910   020A      05      FC      00      FD      00010101

911 020A C9 06    CMP #06h

912 -t

913 PgmCntr(PC) Accum(AC) XReg(XR) YReg(YR) StkPtr(SP) NV-BDIZC(SR)
914   020C      05      FC      00      FD      10010100

915 020C F0 02    BEQ 0210h

```

```

916  -t

917  PgmCntr(PC)  Accum(AC)  XReg(XR)  YReg(YR)  StkPtr(SP)  NV-BDIZC(SR)
918    020E      05         FC       00       FD       10010100

919  020E  EA      NOP

920  -t

921  PgmCntr(PC)  Accum(AC)  XReg(XR)  YReg(YR)  StkPtr(SP)  NV-BDIZC(SR)
922    020F      05         FC       00       FD       10010100

923  020F  EA      NOP

924  -t

925  PgmCntr(PC)  Accum(AC)  XReg(XR)  YReg(YR)  StkPtr(SP)  NV-BDIZC(SR)
926    0210      05         FC       00       FD       10010100

927  0210  00      BRK

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

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

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

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

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

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

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

943  "The more you work with it, the easier it will become." I replied. But now,

```


944 lets look at the program again to see how branch instruction know how far
945 ahead in memory to branch."

946 I then unassembled the program again:

947 -u 0200

948	0200	A9 02	LDA #02h
949	0202	C9 02	CMP #02h
950	0204	F0 01	BEQ 0207h
951	0206	EA	NOP
952	0207	EA	NOP
953	0208	A9 05	LDA #05h
954	020A	C9 06	CMP #06h
955	020C	F0 02	BEQ 0210h
956	020E	EA	NOP
957	020F	EA	NOP
958	0210	00	BRK

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

960 "Address 207 hex." replied Pat.

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

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

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

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

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

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

977 **Negative Numbers And The Negative Flag**

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

979 Pat thought about this for a few moments then said "2 to the 4th power is
980 16 so 16 patterns."

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

983 Pat said "Since the first of the 16 4-bit patterns needs to represent decimal
984 0, then there are only 15 patterns left to represent the decimal numerals 1
985 through 15. This means that the highest decimal numeral that 4 bits can
986 represent is 15."

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

Figure 2 Binary Decimal

990				"So far we have been working with positive
991	0000	-	0	numbers," I said "but how do you think bit
992	0001	-	1	patterns can be made to represent negative
993	0010	-	2	numbers?" I asked?
	0011	-	3	
994	0100	-	4	Pat studied the numbers on the whiteboard
995	0101	-	5	then said "I'm not sure."
	0110	-	6	
996	0111	-	7	"What do you think would happen," I asked "if
997	1000	-	8	we took the binary numeral 0000 and
998	1001	-	9	subtracted 1 from it?"
	1010	-	10	
999	1011	-	11	Pat thought about this for a while.
	1100	-	12	
1000	1101	-	13	"I'll give you a hint," I said "think back to the
1001	1110	-	14	odometer example we discussed earlier and
1002	1111	-	15	imagine what would happen if we added 1 to
1003				the bit pattern 1111."

1004 "Well," said Pat "all the 1's in the bit pattern
1005 1111 would roll around to 0's if you added 1 to it so I suppose that if 1 was
1006 subtracted from the bit pattern 0000, then all the 0's would roll backwards
1007 to 1111."

Figure 3

1008	Binary	Decimal	<p>"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)</p> <p>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?"</p> <p>"Pat studied the table further then said "Not really".</p>
1009	1000	- 8	
1010	1001	- 7	
1011	1010	- 6	
1012	1011	- 5	
1013	1100	- 4	
1014	1101	- 3	
1015	1110	- 2	
1016	1111	- 1	
1017	0000	0	
1018	0001	1	
1019	0010	2	
1020	0011	3	
1021	0100	4	
1022	0101	5	
1023	0110	6	
	0111	7	

1024 I then erased the leftmost bits in the patterns before and after 0000 and
 1025 redrew them with a red marker. "What do you notice now?" I asked.

Figure 4

1026	Binary	Decimal	<p>"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.</p> <p>"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."</p> <p>"How does the CPU know when a program is dealing with a signed number or with an unsigned number?" asked Pat.</p> <p>"The CPU does not really 'know' whether it is dealing with a signed number or an unsigned</p>
1027	1000	- 8	
1028	1001	- 7	
1029	1010	- 6	
1030	1011	- 5	
1031	1100	- 4	
1032	1101	- 3	
1033	1110	- 2	
1034	1111	- 1	
1035	0000	0	
1036	0001	1	
1037	0010	2	
1038	0011	3	
1039	0100	4	
	0101	5	
1040	0110	6	
1041	0111	7	

1042 number. It just executes the instructions it has been given. It is the
1043 programmer that decides which variables in the program contain signed
1044 numbers and which variables contain unsigned numbers. It is the object
1045 that the programmer is modeling with the program that is used to make this
1046 determination.

1047 "Since the CPU does not 'know' which values represent signed numbers and
1048 which values represent unsigned numbers, a flag in the status register
1049 (called the Negative flag) assumes that all the calculations that are being
1050 performed by the CPU are with signed numbers. If the value that is the
1051 result of a calculation has its leftmost bit set to a 1, then the Negative flag
1052 will also be set to a 1 to indicate the value is **negative** if it represents a
1053 signed number. If the leftmost bit is a 0, then the Negative flag will also be
1054 set to a 0 to indicate the value is **positive** if it represents a signed number."

1055 "Do you mean that the Negative flag has been indicating whether results
1056 have been negative or not the whole time we have been tracing programs?"
1057 asked Pat.

1058 I smiled and said "Yes."

1059 "I missed that too!" said Pat. "Can we enter in a short program into the
1060 emulator and trace through it so that I can see the Negative flag changing?"

1061 "Okay." I said. "If you look at the reference information for the LDA
1062 instruction you will see that every time it loads a number into the 'A'
1063 register, the Negative flag is set or cleared depending in whether or not the
1064 number was negative. I will enter a short program which contains 4 LDA
1065 instructions directly into the emulator. I will have 2 of these instructions
1066 load positive numbers and have 2 of them load negative numbers."

1067 I then entered the following program into the emulator using the Assemble
1068 command:

```
1069 0200 A9 05    LDA #05h
1070 0202 A9 80    LDA #80h
1071 0204 A9 27    LDA #27h
1072 0206 A9 C2    LDA #C2h
1073 0208 00      BRK
1074 ...
```

1075 "Which of these numbers are positive and which of them are negative Pat?"
1076 I asked.

1077 Pat looked at the numbers then picked up the whiteboard and wrote the
1078 following:

1079 0 5
1080 0000 0101

1081 8 0
1082 1000 0000

1083 2 7
1084 0010 0111

1085 c 2
1086 1100 0010

1087 "The 05 is positive," said Pat "the 80 hex is negative, the 27 hex is positive,
1088 and the c2 hex is negative. Am I right?"

1089 "Yes, you are right!" I replied. "Now trace through the program and see if
1090 the Negative flag agrees with you."

1091 Pat then traced through the program:

```
1092 -t 0200

1093 PgmCntr(PC)  Accum(AC)  XReg(XR)  YReg(YR)  StkPtr(SP)  NV-BDIZC(SR)
1094   0202             05             FC             00             FD             00010100

1095 0202  A9 80      LDA #80h

1096 -t

1097 PgmCntr(PC)  Accum(AC)  XReg(XR)  YReg(YR)  StkPtr(SP)  NV-BDIZC(SR)
1098   0204             80             FC             00             FD             10010100

1099 0204  A9 27      LDA #27h

1100 -t

1101 PgmCntr(PC)  Accum(AC)  XReg(XR)  YReg(YR)  StkPtr(SP)  NV-BDIZC(SR)
1102   0206             27             FC             00             FD             00010100

1103 0206  A9 C2      LDA #C2h

1104 -t
```

1105	PgmCntr(PC)	Accum(AC)	XReg(XR)	YReg(YR)	StkPtr(SP)	NV-BDIZC(SR)
1106	0208	C2	FC	00	FD	10010100

1107 0208 00 BRK

1108 "The Negative flag agreed with me!" said Pat.

1109 "Yes it did." I replied. "Now we can look at how a branch instruction
1110 branches backwards in memory."

1111 Backward Branches And Loops

1112 "When I was young Pat," I said "I read a story about a man who had found a
1113 ring that would send him one minute backwards in time when he pressed it.
1114 The ring would not work again until the minute had passed again, so the
1115 furthest he could ever go back in time was just one minute. He eventually
1116 figured out how to use the ring to win money at gambling establishments
1117 and he did this until he was very rich. One day he decided to spend some of
1118 his money by taking a trip to a foreign country. While he was on the plane
1119 traveling high above the ocean, a meteor hit the plane and ripped a large
1120 hole in the fuselage. He was thrown through the hole and knocked
1121 unconscious. When he awoke, he found himself falling towards the ocean."

1122 "What did he do!? asked Pat.

1123 "What do you think he did?" I said.

1124 "He pressed the ring!" cried Pat "and put himself one minute back in time!"

1125 "Yes, he did," I said "but after he pressed the ring, he found that he was still
1126 falling over the ocean, jut higher up than he was before."

1127 "Oh no!" said Pat. "He couldn't press the ring again until a minute had
1128 passed so he was stuck repeating his fall towards the ocean over and over
1129 again! How awful!"

1130 "I agree," I said "and to this day I can still see the man being placed at the
1131 top of his fall and then falling, over and over again, in an infinite loop. What
1132 brought the story to mind was that when a computer uses a branch
1133 instruction or a jump instruction to move the Program Counter backwards
1134 in memory, it is similar to the man in the story falling in an infinite loop."

1135 "It is?" asked Pat. "How?"

1136 "When the Program Counter is set to an earlier part of memory, the
 1137 instructions that have already been executed are executed again. When the
 1138 branch or the jump instruction is encountered again, it acts like the man's
 1139 ring and sends the Program Counter back to the earlier set of instructions.
 1140 Sections of code that execute over and over like this are called **loops**.
 1141 Usually, there is some logic that is placed within a loop that will allow the
 1142 loop to eventually be exited. The word **logic** in this context means a group
 1143 of instructions that work together to accomplish a given purpose. If loop
 1144 exit logic does not exist, or if the logic was written incorrectly, the loop will
 1145 loop forever. Loops that do not contain exit logic are called **infinite loops**."

1146 "Can an infinite loop really run forever?" asked Pat.

1147 "Not really." I replied. "An infinite loop can be forced to exit by the
 1148 operating system, by pressing the computer's reset button, or by shutting
 1149 the computer off. Even if the computer were permitted to run continuously,
 1150 a part in it would eventually wear out which would cause it to crash.
 1151 Therefore, an infinite loop is really only infinite in theory."

1152 "Can you show me an infinite loop?" asked Pat. "I would like to see one."

1153 "Yes, an infinite loop is easy to create." I said "I will enter a short program
 1154 directly into the emulator that contains an infinite loop and then I will let
 1155 you trace through it. Pay close attention to the contents of the program
 1156 counter as you trace."

1157 I then entered the following program and let Pat trace it:

1158 -u 0200

```

1159 0200 A9 01    LDA #01h
1160 0202 A2 02    LDX #02h
1161 0204 4C 00 02  JMP 0200h
1162 0207 00      BRK
1163 ...

```

1164 -t 0200

1165	PgmCntr(PC)	Accum(AC)	XReg(XR)	YReg(YR)	StkPtr(SP)	NV-BDIZC(SR)
1166	0202	01	FC	00	FD	00010100
1167	0202 A2 02	LDX #02h				

1168 -t

1169	PgmCntr(PC)	Accum(AC)	XReg(XR)	YReg(YR)	StkPtr(SP)	NV-BDIZC(SR)
1170	0204	01	02	00	FD	00010100

1171 0204 4C 00 02 **JMP 0200h**
 1172 -t

1173	PgmCntr(PC)	Accum(AC)	XReg(XR)	YReg(YR)	StkPtr(SP)	NV-BDIZC(SR)
1174	0200	01	02	00	FD	00010100

1175 0200 A9 01 LDA #01h
 1176 -t

1177	PgmCntr(PC)	Accum(AC)	XReg(XR)	YReg(YR)	StkPtr(SP)	NV-BDIZC(SR)
1178	0202	01	02	00	FD	00010100

1179 0202 A2 02 LDX #02h
 1180 -t

1181	PgmCntr(PC)	Accum(AC)	XReg(XR)	YReg(YR)	StkPtr(SP)	NV-BDIZC(SR)
1182	0204	01	02	00	FD	00010100

1183 0204 4C 00 02 **JMP 0200h**
 1184 -t

1185	PgmCntr(PC)	Accum(AC)	XReg(XR)	YReg(YR)	StkPtr(SP)	NV-BDIZC(SR)
1186	0200	01	02	00	FD	00010100

1187 0200 A9 01 LDA #01h
 1188 -t

1189	PgmCntr(PC)	Accum(AC)	XReg(XR)	YReg(YR)	StkPtr(SP)	NV-BDIZC(SR)
1190	0202	01	02	00	FD	00010100

1191 0202 A2 02 LDX #02h
 1192 -t

1193	PgmCntr(PC)	Accum(AC)	XReg(XR)	YReg(YR)	StkPtr(SP)	NV-BDIZC(SR)
1194	0204	01	02	00	FD	00010100

1195 0204 4C 00 02 **JMP 0200h**

1196 -t

1197	PgmCntr(PC)	Accum(AC)	XReg(XR)	YReg(YR)	StkPtr(SP)	NV-BDIZC(SR)
1198	0200	01	02	00	FD	00010100

1199 0200 A9 01 LDA #01h

1200 "Wow, it does run in an infinite loop!" said Pat. "Can you now show me a
1201 loop that will run for a while and then exit?"

1202 "Yes, this is also easy to do." I said. "I will create a small program that will
1203 place the number 4 into the X register and then decrement the contents of
1204 the X register inside a loop until it reaches 0. When it reaches 0, the loop
1205 will exit. This time, pay close attention to the X register, the Program
1206 Counter, and the Zero flag."

1207 I then created the following program and had Pat trace through it:

1208 -u 0200

1209	0200	A2 04	LDX #04h
1210	0202	CA	DEX
1211	0203	D0 FD	BNE 0202h
1212	0205	00	BRK
1213	...		

1214 -t 0200

1215	PgmCntr(PC)	Accum(AC)	XReg(XR)	YReg(YR)	StkPtr(SP)	NV-BDI Z C(SR)
1216	0202	00	04	00	FD	000101 00

1217 0202 CA DEX

1218 -t

1219	PgmCntr(PC)	Accum(AC)	XReg(XR)	YReg(YR)	StkPtr(SP)	NV-BDI Z C(SR)
1220	0203	00	03	00	FD	000101 00

1221 0203 D0 FD **BNE 0202h**

1222 -t

1223	PgmCntr(PC)	Accum(AC)	XReg(XR)	YReg(YR)	StkPtr(SP)	NV-BDI Z C(SR)
1224	0202	00	03	00	FD	000101 00

1225 0202 CA DEX

1226 -t

1227	PgmCntr(PC)	Accum(AC)	XReg(XR)	YReg(YR)	StkPtr(SP)	NV-BDI Z C(SR)
1228	0203	00	02	00	FD	000101 00

1229 0203 D0 FD **BNE 0202h**

1230 -t

1231	PgmCntr(PC)	Accum(AC)	XReg(XR)	YReg(YR)	StkPtr(SP)	NV-BDI Z C(SR)
1232	0202	00	02	00	FD	000101 00

1233 0202 CA DEX

1234 -t

1235	PgmCntr(PC)	Accum(AC)	XReg(XR)	YReg(YR)	StkPtr(SP)	NV-BDI Z C(SR)
1236	0203	00	01	00	FD	000101 00

1237 0203 D0 FD **BNE 0202h**

1238 -t

1239	PgmCntr(PC)	Accum(AC)	XReg(XR)	YReg(YR)	StkPtr(SP)	NV-BDI Z C(SR)
1240	0202	00	01	00	FD	000101 00

1241 0202 CA DEX

1242 -t

1243	PgmCntr(PC)	Accum(AC)	XReg(XR)	YReg(YR)	StkPtr(SP)	NV-BDI Z C(SR)
1244	0203	00	00	00	FD	000101 10

1245 0203 D0 FD **BNE 0202h**

1246 -t

1247	PgmCntr(PC)	Accum(AC)	XReg(XR)	YReg(YR)	StkPtr(SP)	NV-BDI Z C(SR)
1248	0205	00	00	00	FD	000101 10

1249 0205 00 BRK

1250 "What did the program do?" I asked.

1251 "The loop kept looping until the X register was decremented to 0, then the
 1252 Zero flag was set and the BNE instruction fell through to the next
 1253 instruction instead of taking the branch." said Pat.

1254 "Correct." I said. "Now, look at the program again and tell me what the

1255 operand is for the BNE instruction."

1256 Pat looked at the program and then said "FD hex? That seems like too large
1257 of a number... wait, the BNE is branching **backwards** in memory so it must
1258 be a **negative** number!"

1259 "It is indeed a negative number, Pat." I said. "Can you determine what the
1260 number is in decimal?"

1261 "Hmmm," said Pat "FD hex is equal to 11111101 in binary. Just a bit ago
1262 we created a table which showed 4-bit binary numerals and their positive
1263 and negative decimal equivalents. I am guessing that if we just extend this
1264 table to 8 bits and added a column for hex numerals, we can figure out what
1265 FD hex is equivalent to in decimal."

1266 "Go ahead and extend the table then." I said. Pat then modified the table.
1267 (see Fig. 5)

1268	Figure 5	Binary	Hex	Dec	"FD hex is equal to -3 1269 decimal!" said Pat.
		11111000	- F8	- -8	
1270		11111001	- F9	- -7	
1271		11111010	- FA	- -6	
1272		11111011	- FB	- -5	
1273		11111100	- FC	- -4	
1274		11111101	- FD	- -3	
1275		11111110	- FE	- -2	
1276		11111111	- FF	- -1	
1277		00000000	- 00	- 0	
1278		00000001	- 01	- 1	
1279		00000010	- 02	- 2	
1280		00000011	- 03	- 3	
		00000100	- 04	- 4	
1281		00000101	- 05	- 5	
1282		00000110	- 06	- 6	
1283		00000111	- 07	- 7	

1284 "What else can loops do?" asked Pat.

1285 "The ability to execute a group of instructions over and over again by
1286 looping," I replied "is one of the fundamental capabilities that give a
1287 computer its enormous power. In fact, machines of all types derive much of

1288 their power from the principle of **repeated cycling**.

1289 A simple example of this is a car tire. A tire would not be very useful if it
1290 could only be rolled through one revolution. This brings to mind the image
1291 of a person who just purchased a brand new car at a dealership. The
1292 papers have been signed, the whole family (including the dog) has just
1293 been loaded into the car, and they are ready to drive home. The person
1294 starts the car, puts it into drive, moves forward one full revolution of the
1295 tires, and stops. The person then jacks up the car, removes the tires,
1296 discards them, puts on a set of new ones, lowers the car, then drives
1297 forward one more revolution of the tires. This process is continued all the
1298 way home!"

1299 Pat burst out laughing and I did too!

1300 I then continued "Other examples of machines that make use of the
1301 repeated cycles principle include internal combustion engines, sewing
1302 machines, hammers, screws, drills, and pumps. Many more examples exist,
1303 but they are too numerous to list."

1304 "I hadn't thought about it before," said Pat "but you're right, lots of
1305 machines repeat their cycles. I also never would have guessed that
1306 computers repeat cycles too because, from the outside, it looks like they just
1307 sit there."

1308 "In a program," I said "loops are used for all kinds of purposes like adding
1309 series of numbers together, repeatedly checking to see if an event (like the
1310 pressing of a keyboard key) has occurred, moving graphics across a screen,
1311 searching files, generating sounds, and spell checking documents."

1312 "Can we create a program that uses a loop to do something useful?" asked
1313 Pat. "Maybe something simple like adding a series of numbers together."

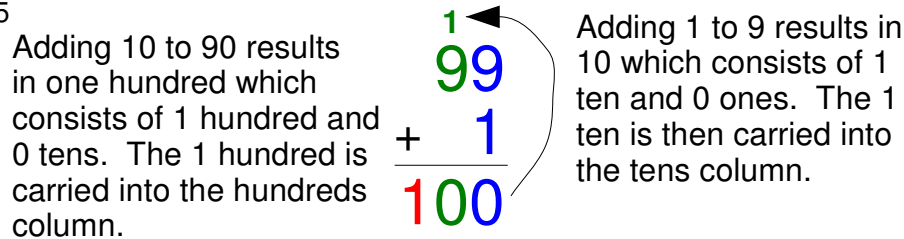
1314 "Yes, we can do this." I said. "But first we need to talk about the Carry flag,
1315 indexed addressing modes, and commenting programs.

1316 **The Carry Flag**

1317 "What I would like you to do now Pat," I said "is to add 1 to 99 decimal on
1318 the whiteboard and explain how carrying works when an addition in a given
1319 column results in a number that is too large to fit in that column."

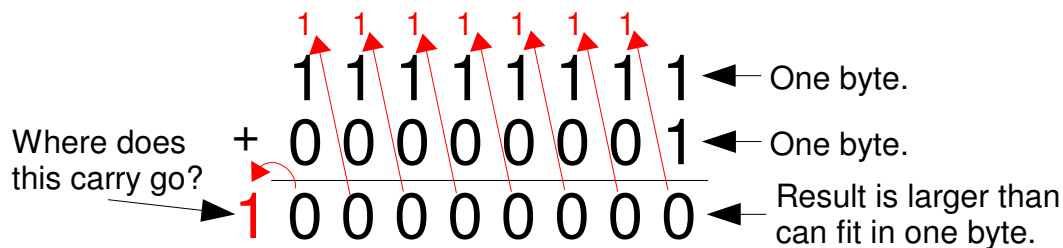
1320 Pat added 1 to 99 decimal on the whiteboard then said "Starting in the ones
 1321 column, 1 is added to 9 and the result is 1 ten and 0 ones. The 10 will not
 1322 fit into the one's column, so it is carried over to the tens column. The 90
 1323 that is in the tens column is then added to the 10 that was carried over
 1324 there and the result is 1 hundred and 0 tens. The 1 hundred is too large to
 1325 fit into the tens column, so it is carried over to the hundreds column." (see
 1326 Fig. 5)

Figure 5



1327 "Very good Pat." I said "Now I am going to do another addition on the
 1328 whiteboard except I will be adding 1 to 11111111 binary." (see Fig. 6)

Figure 6



1329 "1 + 1 binary equals 10 binary." I said. "Notice how the bits from each
 1330 addition in each column are carried over to the column to the left of it. Also
 1331 notice that the result is a 9 bit number, not an 8 bit number."

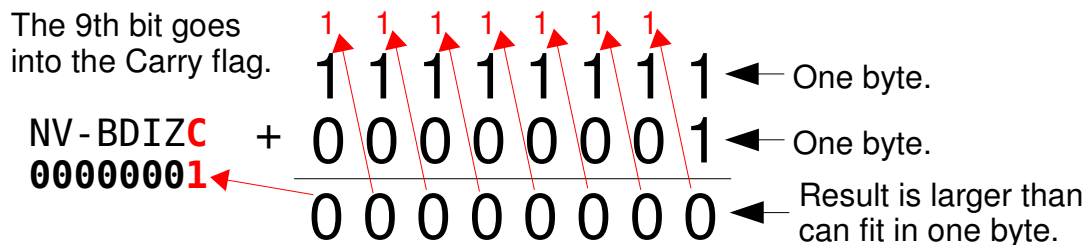
1332 "Uh oh," said Pat "we have a problem."

1333 "What is the problem?" I asked.

1334 "Our registers are only 8 bits wide so where is the 9th bit going?" replied
 1335 Pat.

1336 "You are very observant." I said. "Our registers are only 8 bits wide and so
 1337 are our memory locations. Even if our registers were wider, we would still
 1338 run into a problem like this eventually when we started using larger
 1339 numbers. This is the problem that the **Carry flag** has been designed to
 1340 solve and the way it does it is like this." I then added information about the
 1341 carry flag to the diagram on the whiteboard (see Fig. 7)

Figure 7



1342 Pat studied the diagram then said "But what happens to the bit after it has
 1343 been placed into the Carry flag?"

1344 "Have you ever wondered what the 'C' means in the ADC instruction's
 1345 name?" I asked.

1346 "Yes, I've wondered about it because it always seemed to me that this
 1347 instruction should have been called ADD instead of ADC." replied Pat.

1348 "The 'C' stands for Carry," I said "and what this means is that the ADC
 1349 instruction will add the value in the 'A' register with a value in memory **and**
 1350 **to this sum it will add the contents of the Carry flag**. Therefore, the
 1351 correct name of the ADC instruction is ADd with Carry."

1352 "Wait a minute!" said Pat. "If the ADC instruction always includes the value
 1353 of the Carry flag in its calculations, what happens if the Carry flag just
 1354 happens to be set to 1 when a calculation is performed? Wouldn't it result
 1355 in the answer being one more than it should be?"

1356 "Yes," I replied "and this is why a CLC or CLear Carry instruction is always
 1357 placed just before an ADC instruction unless a multi-byte addition is being
 1358 performed."

1359 "But we haven't been placing a CLC instruction before our ADC

1360 instructions," said Pat "so why have our answers have been coming out
1361 okay?"

1362 "The reason that our answers have been correct so far," I said "is because
1363 the emulator and the monitor have been programmed to launch with the
1364 Carry flag set to 0. I have not been placing a CLC instruction ahead of the
1365 ADC instructions we have been using because I was not ready yet to tell you
1366 about how the Status register's flags worked."

1367 "That was probably a good idea," said Pat "because I don't think I would
1368 have been able to understand what the flags did if you had told me about
1369 them earlier than you did. Now that I know about the Carry flag, though,
1370 can you show me how it is used to add together 2 bytes that have a result
1371 that is larger than 8 bits?"

1372 "Yes." I said "I will create a small program that performs the addition from
1373 the example on the whiteboard and you then can trace it."

1374 I created the following program:

1375	0200	000001		org 0200h
1376		000002		
1377	0200 FF	000003		number1 dbt 11111111b
1378	0201 01	000004		number2 dbt 00000001b
1379		000005		
1380	0205	000006		org 0205h
1381		000007		
1382	0205 AD 00 02	000008		lda number1
1383	0208 18	000009		clc
1384	0209 6D 01 02	000010		adc number2
1385		000011		
1386	020C 00	000012		brk
1387		000013		
1388		000014		end
1389		000015		

1390 And then Pat dumped it, unassembled it, and traced through it:

1391 -d 0200

1392 0200 FF 01 00 00 00 AD 00 02 - 18 6D 01 02 00 00 00 00m.....

1393 -u 0205

1394 0205 AD 00 02 LDA 0200h

1395 0208 18 CLC

```

1396 0209 6D 01 02  ADC 0201h
1397 020C 00      BRK
1398 ...

```

```

1399 -t 0205

```

```

1400 PgmCntr(PC)  Accum(AC)  XReg(XR)  YReg(YR)  StkPtr(SP)  NV-BDIZC(SR)
1401   0208      FF          FC          00          FD      10010100

```

```

1402 0208 18      CLC

```

```

1403 -t

```

```

1404 PgmCntr(PC)  Accum(AC)  XReg(XR)  YReg(YR)  StkPtr(SP)  NV-BDIZC(SR)
1405   0209      FF          FC          00          FD      10010100

```

```

1406 0209 6D 01 02  ADC 0201h

```

```

1407 -t

```

```

1408 PgmCntr(PC)  Accum(AC)  XReg(XR)  YReg(YR)  StkPtr(SP)  NV-BDIZC(SR)
1409   020C      00          FC          00          FD      00010111

```

```

1410 020C 00      BRK

```

1411 "Notice that after the ADC instruction was executed," I said "it resulted in
 1412 00 being placed in the 'A' register and the Carry flag being set to 1. This
 1413 matches the calculation we made on the whiteboard." (again, see Fig. 7).

1414 Indexed Addressing Modes And Commenting Programs

1415 "Now that you know how the Carry flag works Pat," I said "we can create a
 1416 program that adds a series of numbers together in a loop. In order to do
 1417 this, however, we will need to use one of the indexed addressing modes."

1418 "What does an indexed addressing mode do?" asked Pat.

1419 I replied "An indexed addressing mode uses the contents of either the X
 1420 register or the Y register as an offset from some **base address** to determine
 1421 what is called the **effective address**."

1422 For example, with the **Absolute,X** addressing mode, the programmer
 1423 specifies an **absolute address** to use as the **base address** and then the
 1424 contents of the X register are added to this **base address** to determine the
 1425 **effective address** that will be accessed by the instruction."

1426 "I don't get it." said Pat, with a confused look.

1427 "Then I will create a program that shows how Absolute,X addressing works,
1428 trace through it, and then we will discuss it."

1429 I then created the following program and traced it:

```

1430 0200          000001 |      org 0200h
1431          000002 |
1432 0200 41      000003 |nums dbt 41h,42h,43h,44h,45h
1433 0201 42
1434 0202 43
1435 0203 44
1436 0204 45
1437 0205 46
1438          000004 |
1439 0210          000005 |      org 0210h
1440          000006 |
1441 0210 A2 02    000007 |      ldx #02d
1442 0212 BD 00 02 000008 |      lda nums,x
1443          000009 |
1444 0215 00      000010 |      brk
1445          000011 |
1446          000012 |      end
1447          000013 |

```

1448 -d 0200

1449 0200 41 42 43 44 45 46 00 00 - 00 00 00 00 00 00 00 00 ABCDEF.....

1450 -u 0210

```

1451 0210 A2 02    LDX #02h
1452 0212 BD 00 02 LDA 0200h,X
1453 0215 00      BRK
1454 ...

```

1455 -t 0210

1456	PgmCntr(PC)	Accum(AC)	XReg(XR)	YReg(YR)	StkPtr(SP)	NV-BDIZC(SR)
1457	0212	00	02	00	FD	00010100

1458 0212 BD 00 02 LDA 0200h,X

1459 -t

1460	PgmCntr(PC)	Accum(AC)	XReg(XR)	YReg(YR)	StkPtr(SP)	NV-BDIZC(SR)
------	-------------	-----------	----------	----------	------------	--------------

1461 0215 **43** 02 00 FD 00010100

1462 0215 00 BRK

1463 "The LDA instruction in this program uses the **Absolute,X** addressing mode
1464 to determine the memory location which it will copy the value from." I said
1465 "This memory location is called the **effective address**. The **base address**
1466 is **0200** hex and **02** has already been loaded into the X register. The
1467 **effective address** is calculated by adding the base address to the contents
1468 of the X register which, in this case, is 0200 hex + 02 which equals 0202
1469 hex."

1470 "What did I place into memory starting at location 0200h, Pat?" I asked.

1471 Pat looked at the program and said "You placed a variable there called
1472 **nums**, but instead of defining a single byte at address 0200 hex, you placed
1473 a series of 5 bytes in this area of memory with the first byte being located at
1474 address 0200 hex. I didn't know that the **dbt** directive could be used to
1475 place a series of bytes into memory, thats interesting."

1476 "When a group of values that are related to each other are placed into
1477 consecutive memory locations like this," I said "they are referred to as a
1478 **table**, an **array**, or a **list**. This array consists of 5 bytes and these bytes
1479 just happen to contain the first 5 capital ASCII letters.

1480 When the instruction **lda nums,x** was executed, it took the address of
1481 **nums** (which is 0200 hex) and added to it the contents of the X register
1482 (which is 02). It then used the resulting sum (0202 hex) to determine
1483 which memory location to copy the value from. What number is at address
1484 0202 hex, Pat?"

1485 Pat looked at the program and said "43 hex."

1486 "And what number was loaded into the 'A' register when it was traced?" I
1487 asked.

1488 "43 hex!" Pat replied. "The Absolute,X addressing mode worked!"

1489 "Yes it did," I replied "now I will create a program that determines the sum
1490 of an array of numbers."

1491 Here is the program I created:

```

1492      000001 |;The purpose of this program is to calculate the
1493      000002 |;sum of the array nums and then to place the
1494      000003 |;result into the variable sum.
1495      000004 |
1496      0200      000005 |          org 0200h
1497      000006 |
1498      000007 |;An array of 10 bytes.
1499      0200 01      000008 |nums  dbt 1d,2d,3d,4d,5d,6d,7d,8d,9d,10d
1500      0201 02
1501      0202 03
1502      0203 04
1503      0204 05
1504      0205 06
1505      0206 07
1506      0207 08
1507      0208 09
1508      0209 0A
1509      000009 |
1510      000010 |
1511      000011 |
1512      000012 |
1513      000013 |
1514      000014 |
1515      000015 |
1516      000016 |
1517      000017 |
1518      000018 |
1519      000019 |;Holds the sum of array at nums.
1520      020A 00      000020 |sum  dbt 0d
1521      000021 |
1522      0250      000022 |          org 0250h
1523      000023 |
1524      000024 |;Initialize the X register so that it offsets 0
1525      000025 |;positions into the array nums.
1526      0250 A2 00      000026 |          ldx #0d
1527      000027 |
1528      000028 |;Initialize register 'A' to 0.  This needs to
1529  be done
1530      000029 |;so that an old value in 'A' does not produce a
1531  wrong
1532      000030 |;sum during the first loop iteration.
1533      0252 A9 00      000031 |          lda #0d
1534      000032 |
1535      000033 |;This label is the top of the calculation loop.
1536      0254      000034 |AddMore  *
1537      000035 |
1538      000036 |;Clear the carry flag so that it does not cause
1539  a

```

```

1540      000037 |;wrong sum to be calculated by the ADC
1541 instruction.
1542      0254 18      000038 |      clc
1543      000039 |
1544      000040 |;Obtain a value from the array at offset X
1545 positions
1546      000041 |;into the array and add this value to the
1547 contents
1548      000042 |;of the 'A' register.
1549      0255 7D 00 02 000043 |      adc nums,x
1550      000044 |
1551      000045 |;Increment X to the next offset position.
1552      0258 E8      000046 |      inx
1553      000047 |
1554      000048 |;If X has been incremented to 10, fall through
1555 the
1556      000049 |;bottom of the loop.  If X is less than 10 then
1557 loop
1558      000050 |;back to AddMore and add another value from the
1559 array.
1560      0259 E0 0A      000051 |      cpx #10d
1561      025B D0 F7      000052 |      bne AddMore
1562      000053 |
1563      000054 |;After the loop has finished calculating the
1564 sum of
1565      000055 |;the array, store this sum into the variable
1566 called
1567      000056 |;'sum'.
1568      025D 8D 0A 02      000057 |      sta sum
1569      000058 |
1570      000059 |;Return program control back to the monitor.
1571      0260 00      000060 |      brk
1572      000061 |
1573      000062 |;The end command must have at least 1 blank line
1574      000063 |;underneath it.
1575      000064 |
1576      000065 |      end

```

1577 "What are all those lines that begin with semicolons for?" asked Pat

1578 "Those are called **comments**, I replied "and their purpose is to explain what
 1579 the various parts of a program do. The semicolon tells the assembler
 1580 to ignore everything after them on the line. Comment lines are
 1581 ignored by the assembler and none of their content makes it into the
 1582 program. Up to this point our programs have been small enough that they
 1583 did not need commenting, but from here on the programs will be more
 1584 sophisticated. If sophisticated programs are not commented, it is very
 1585 difficult to keep track of what they are doing."

```

1586 "I can believe that," said Pat "because I was even having trouble keeping
1587 track of what the smaller programs were doing."

1588 After Pat had finished studying the program and reading the comments it
1589 contained, I loaded it into the emulator and executed it with a Go command:

1590 -d 0200

1591 0200  01 02 03 04 05 06 07 08 - 09 0A 00 00 00 00 00 00 .....
1592
1593 -u 0250

1594 0250  A2 00      LDX #00h
1595 0252  A9 00      LDA #00h
1596 0254  18        CLC
1597 0255  7D 00 02   ADC 0200h,X
1598 0258  E8        INX
1599 0259  E0 0A      CPX #0Ah
1600 025B  D0 F7     BNE 0254h
1601 025D  8D 0A 02  STA 020Ah
1602 0260  00       BRK
1603 ...

1604 r

1605 PgmCntr(PC)  Accum(AC)  XReg(XR)  YReg(YR)  StkPtr(SP)  NV-BDIZC(SR)
1606    102C      00        FC        00        FD        00010110

1607 -g 0250

1608 PgmCntr(PC)  Accum(AC)  XReg(XR)  YReg(YR)  StkPtr(SP)  NV-BDIZC(SR)
1609    0260      37        0A        FF        FD        00010111

1610 -d 0200

1611 0200  01 02 03 04 05 06 07 08 - 09 0A 37 00 00 00 00 00 .....7.....

1612 "What values were in the 'A' register and in the variable 'sum' before the
1613 program was executed?" I asked.

1614 "0 and 0." replied Pat.

1615 "And what values were in the 'A' register and in the variable 'sum' after the
1616 program was executed?" I asked.

1617 "37 hex and 37 hex." replied Pat.

```

1618 "What is 37 hex in decimal?" I asked.

1619 Pat picked up the calculator that was on the table, pressed some of its
1620 buttons then said "55."

1621 "Finally," I asked "what is the sum of 1+2+3+4+5+6+7+8+9+10?"

1622 Pat calculated the sum on the calculator then said "55! It worked! But now
1623 I want to trace through the program so I can see it work step-by-step."

1624 Pat then did this and so should you.

1625 **Exercises**

1626 1) The source code for the umon65 monitor is in the the
1627 mathrider/examples/u6502 directory in the download file that contained the
1628 emulator. Open this file and study it.

1629 2) Write an **assembly language** program that adds the numbers 1,2,3,4,5,
1630 and 6 together and places the sum into location 0275h. Have the program
1631 start at 0200h in memory. Assemble the program, load it into the emulator,
1632 run it, and verify that it works correctly.

1633 **Appendix A - 6502 Instruction Set Reference (minus zero page**
 1634 **addressing)**

1635 **Registers:**

1636	PC	program counter	(16 bit)
1637	AC	accumulator	(8 bit)
1638	X	X register	(8 bit)
1639	Y	Y register	(8 bit)
1640	SR	status register [NV-BDIZC]	(8 bit)
1641	SP	stack pointer	(8 bit)

1642

1643 **Status Register (SR) Flags (bit 7 to bit 0):**

1644	N	Negative
1645	V	Overflow
1646	-	ignored
1647	B	Break
1648	D	Decimal (use BCD for arithmetics)
1649	I	Interrupt (IRQ disable)
1650	Z	Zero
1651	C	Carry

1652 **Processor Stack:**

1653 Top down, 0x0100 - 0x01FF

1654

1655 **Words:**

1656 16 bit words in lowbyte-highbyte representation (Little-Endian).

1657 **Addressing Modes:**

1658	#	Immediate / OPC #\$BB / Operand is byte (BB).
1659	A	Accumulator / OPC A / Operand is AC.
1660	abs	Absolute / OPC \$HHLL / Operand is address \$HHLL.
1661	abs,X	Absolute,X-indexed / OPC \$HHLL,X / Operand is address incremented by X
1662		with carry.
1663	abs,Y	Absolute,Y-indexed / OPC \$HHLL,Y / Operand is address incremented by Y
1664		with carry.
1665	impl	Implied / OPC / Operand implied.
1666	ind	Indirect / OPC (\$HHLL) / Operand is effective address, effective
1667		address is value of address.
1668	X,ind	X-indexed,indirect / OPC (\$BB,X) / Operand is effective zeropage
1669		address, effective address is byte (BB) incremented by X without
1670		carry.
1671	ind,Y	Indirect,Y-indexed / OPC (\$LL),Y / Operand is effective address
1672		incremented by Y with carry, effective address is word at zeropage
1673		address.
1674	rel	Relative / OPC \$BB / Branch target is PC + offset (BB), bit 7

1675 signifies negative offset.

1676 **Instructions:**

1677 **Legend to Flags:**

1678 + modified
 1679 - not modified
 1680 1 set
 1681 0 cleared
 1682 M6 memory bit 6
 1683 M7 memory bit 7

1684 **ADC Add Memory to Accumulator with Carry**

1685 A + M + C -> A, C N Z C I D V
 1686 + + + - - +

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

1695 **AND AND Memory with Accumulator**

1696 A AND M -> A N Z C I D V
 1697 + + - - - -

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

1706 **ASL Shift Left One Bit (Memory or Accumulator)**

1707 C <- [76543210] <- 0 N Z C I D V
 1708 + + + - - -

1709	addressing	assembler	opc	bytes
1710	-----			
1711	accumulator	ASL A	0A	1
1712	absolute	ASL oper	0E	3
1713	absolute,X	ASL oper,X	1E	3

1714 **BCC Branch on Carry Clear**

1715	branch on C = 0	N	Z	C	I	D	V
1716		-	-	-	-	-	-

1717	addressing	assembler	opc	bytes
1718	-----			
1719	relative	BCC oper	90	2

1720 **BCS Branch on Carry Set**

1721	branch on C = 1	N	Z	C	I	D	V
1722		-	-	-	-	-	-

1723	addressing	assembler	opc	bytes
1724	-----			
1725	relative	BCS oper	B0	2

1726 **BEQ Branch on Result Zero**

1727	branch on Z = 1	N	Z	C	I	D	V
1728		-	-	-	-	-	-

1729	addressing	assembler	opc	bytes
1730	-----			
1731	relative	BEQ oper	F0	2

1732 **BIT Test Bits in Memory with Accumulator**

1733 bits 7 and 6 of operand are transferred to bit 7 and 6 of SR (N,V);
 1734 the zeroflag is set to the result of operand AND accumulator.

1735	A AND M, M7 -> N, M6 -> V	N	Z	C	I	D	V
1736		M7	+	-	-	-	M6

1737	addressing	assembler	opc	bytes
1738	-----			
1739	absolute	BIT oper	2C	3

1740 **BMI Branch on Result Minus**

1741	branch on N = 1	N	Z	C	I	D	V
1742		-	-	-	-	-	-

1743	addressing	assembler	opc	bytes
1744	-----			
1745	relative	BMI oper	30	2

1746 **BNE Branch on Result not Zero**

1747	branch on Z = 0	N	Z	C	I	D	V
1748		-	-	-	-	-	-

1749	addressing	assembler	opc	bytes
1750	-----			
1751	relative	BNE oper	D0	2

1752 **BPL Branch on Result Plus**

1753	branch on N = 0	N	Z	C	I	D	V
1754		-	-	-	-	-	-

1755	addressing	assembler	opc	bytes
1756	-----			
1757	relative	BPL oper	10	2

1758 **BRK Force Break**

1759	interrupt,	N	Z	C	I	D	V
1760	push PC+2, push SR	-	-	-	1	-	-

1761	addressing	assembler	opc	bytes
1762	-----			
1763	implied	BRK	00	1

1764 **BVC Branch on Overflow Clear**

1765	branch on V = 0	N	Z	C	I	D	V
1766		-	-	-	-	-	-

1767	addressing	assembler	opc	bytes
1768	-----			
1769	relative	BVC oper	50	2

1770 **BVS Branch on Overflow Set**

1771	branch on V = 1	N	Z	C	I	D	V
1772		-	-	-	-	-	-

1773	addressing	assembler	opc	bytes
1774	-----			
1775	relative	BVC oper	70	2

1776 **CLC Clear Carry Flag**

1777	0 -> C	N	Z	C	I	D	V
1778		-	-	0	-	-	-

1779	addressing	assembler	opc	bytes
1780	-----			
1781	implied	CLC	18	1

1782 **CLD Clear Decimal Mode**

1783	0 -> D	N	Z	C	I	D	V
1784		-	-	-	-	0	-

1785	addressing	assembler	opc	bytes
1786	-----			
1787	implied	CLD	D8	1

1788 **CLI Clear Interrupt Disable Bit**

1789	0 -> I	N	Z	C	I	D	V
1790		-	-	-	0	-	-

1791	addressing	assembler	opc	bytes
1792	-----			
1793	implied	CLI	58	1

1794 **CLV Clear Overflow Flag**

1795	0 -> V	N	Z	C	I	D	V
1796		-	-	-	-	-	0

1797	addressing	assembler	opc	bytes
1798	-----			
1799	implied	CLV	B8	1

1800 CMP Compare Memory with Accumulator

1801	A - M	N Z C I D V
1802		+ + + - - -

1803	addressing	assembler	opc	bytes
1804	-----			
1805	immediate	CMP #oper	C9	2
1806	absolute	CMP oper	CD	3
1807	absolute,X	CMP oper,X	DD	3
1808	absolute,Y	CMP oper,Y	D9	3
1809	(indirect,X)	CMP (oper,X)	C1	2
1810	(indirect),Y	CMP (oper),Y	D1	2

1811 CPX Compare Memory and Index X

1812	X - M	N Z C I D V
1813		+ + + - - -

1814	addressing	assembler	opc	bytes
1815	-----			
1816	immediate	CPX #oper	E0	2
1817	absolute	CPX oper	EC	3

1818 CPY Compare Memory and Index Y

1819	Y - M	N Z C I D V
1820		+ + + - - -

1821	addressing	assembler	opc	bytes
1822	-----			
1823	immediate	CPY #oper	C0	2
1824	absolute	CPY oper	CC	3

1825 DEC Decrement Memory by One

1826	M - 1 -> M	N Z C I D V
1827		+ + - - - -

1828	addressing	assembler	opc	bytes
1829	-----			
1830	absolute	DEC oper	CE	3
1831	absolute,X	DEC oper,X	DE	3

1832 DEX Decrement Index X by One

1833	X - 1 -> X	N Z C I D V
1834		+ + - - - -

1835	addressing	assembler	opc	bytes
1836	-----			
1837	implied	DEC	CA	1

1838 **DEY Decrement Index Y by One**

1839	Y - 1 -> Y	N Z C I D V
1840		+ + - - - -

1841	addressing	assembler	opc	bytes
1842	-----			
1843	implied	DEC	88	1

1844 **EOR Exclusive-OR Memory with Accumulator**

1845	A EOR M -> A	N Z C I D V
1846		+ + - - - -

1847	addressing	assembler	opc	bytes
1848	-----			
1849	immediate	EOR #oper	49	2
1850	absolute	EOR oper	4D	3
1851	absolute,X	EOR oper,X	5D	3
1852	absolute,Y	EOR oper,Y	59	3
1853	(indirect,X)	EOR (oper,X)	41	2
1854	(indirect),Y	EOR (oper),Y	51	2

1855 **INC Increment Memory by One**

1856	M + 1 -> M	N Z C I D V
1857		+ + - - - -

1858	addressing	assembler	opc	bytes
1859	-----			
1860	absolute	INC oper	EE	3
1861	absolute,X	INC oper,X	FE	3

1862 **INX Increment Index X by One**

1863	X + 1 -> X	N Z C I D V
1864		+ + - - - -

1865	addressing	assembler	opc	bytes
------	------------	-----------	-----	-------

```

1866 -----
1867 implied      INX          E8      1

```

1868 **INY Increment Index Y by One**

```

1869      Y + 1 -> Y          N Z C I D V
1870                      + + - - - -

```

```

1871      addressing  assembler  opc  bytes
1872      -----
1873      implied      INY          C8      1

```

1874 **JMP Jump to New Location**

```

1875      (PC+1) -> PCL          N Z C I D V
1876      (PC+2) -> PCH          - - - - -

```

```

1877      addressing  assembler  opc  bytes
1878      -----
1879      absolute      JMP oper      4C      3
1880      indirect      JMP (oper)    6C      3

```

1881 **JSR Jump to New Location Saving Return Address**

```

1882      push (PC+2),          N Z C I D V
1883      (PC+1) -> PCL          - - - - -
1884      (PC+2) -> PCH

```

```

1885      addressing  assembler  opc  bytes
1886      -----
1887      absolute      JSR oper      20      3

```

1888 **LDA Load Accumulator with Memory**

```

1889      M -> A              N Z C I D V
1890                      + + - - - -

```

```

1891      addressing  assembler  opc  bytes
1892      -----
1893      immediate      LDA #oper      A9      2
1894      absolute      LDA oper      AD      3
1895      absolute,X      LDA oper,X     BD      3
1896      absolute,Y      LDA oper,Y     B9      3
1897      (indirect,X)    LDA (oper,X)   A1      2
1898      (indirect),Y    LDA (oper),Y   B1      2

```

1899 **LDX Load Index X with Memory**

1900	M -> X		N	Z	C	I	D	V
1901			+	+	-	-	-	-

1902	addressing	assembler	opc	bytes
1903	-----	-----	-----	-----
1904	immediate	LDX #oper	A2	2
1905	absolute	LDX oper	AE	3
1906	absolute,Y	LDX oper,Y	BE	3

1907 **LDY Load Index Y with Memory**

1908	M -> Y		N	Z	C	I	D	V
1909			+	+	-	-	-	-

1910	addressing	assembler	opc	bytes
1911	-----	-----	-----	-----
1912	immediate	LDY #oper	A0	2
1913	absolute	LDY oper	AC	3
1914	absolute,X	LDY oper,X	BC	3

1915 **LSR Shift One Bit Right (Memory or Accumulator)**

1916	0 -> [76543210] -> C		N	Z	C	I	D	V
1917			-	+	+	-	-	-

1918	addressing	assembler	opc	bytes
1919	-----	-----	-----	-----
1920	accumulator	LSR A	4A	1
1921	absolute	LSR oper	4E	3
1922	absolute,X	LSR oper,X	5E	3

1923 **NOP No Operation**

1924	---		N	Z	C	I	D	V
1925			-	-	-	-	-	-

1926	addressing	assembler	opc	bytes
1927	-----	-----	-----	-----
1928	implied	NOP	EA	1

1929 **ORA OR Memory with Accumulator**

1930	A OR M -> A		N	Z	C	I	D	V
------	-------------	--	---	---	---	---	---	---

1931			+ + - - - -	
1932	addressing	assembler	opc	bytes
1933	-----			
1934	immediate	ORA #oper	09	2
1935	absolute	ORA oper	0D	3
1936	absolute,X	ORA oper,X	1D	3
1937	absolute,Y	ORA oper,Y	19	3
1938	(indirect,X)	ORA (oper,X)	01	2
1939	(indirect),Y	ORA (oper),Y	11	2

1940 PHA Push Accumulator on Stack

1941	push A		N Z C I D V	
1942			- - - - -	
1943	addressing	assembler	opc	bytes
1944	-----			
1945	implied	PHA	48	1

1946 PHP Push Processor Status on Stack

1947	push SR		N Z C I D V	
1948			- - - - -	
1949	addressing	assembler	opc	bytes
1950	-----			
1951	implied	PHP	08	1

1952 PLA Pull Accumulator from Stack

1953	pull A		N Z C I D V	
1954			- - - - -	
1955	addressing	assembler	opc	bytes
1956	-----			
1957	implied	PLA	68	1

1958 PLP Pull Processor Status from Stack

1959	pull SR		N Z C I D V	
1960			from stack	
1961	addressing	assembler	opc	bytes
1962	-----			
1963	implied	PLP	28	1
				4

1964 ROL Rotate One Bit Left (Memory or Accumulator)

1965	C <- [76543210] <- C	N	Z	C	I	D	V
1966		+	+	+	-	-	-

1967	addressing	assembler	opc	bytes
1968	-----	-----	-----	-----
1969	accumulator	ROL A	2A	1
1970	absolute	ROL oper	2E	3
1971	absolute,X	ROL oper,X	3E	3

1972 ROR Rotate One Bit Right (Memory or Accumulator)

1973	C -> [76543210] -> C	N	Z	C	I	D	V
1974		+	+	+	-	-	-

1975	addressing	assembler	opc	bytes
1976	-----	-----	-----	-----
1977	accumulator	ROR A	6A	1
1978	absolute	ROR oper	6E	3
1979	absolute,X	ROR oper,X	7E	3

1980 RTI Return from Interrupt

1981	pull SR, pull PC	N	Z	C	I	D	V
1982							from stack

1983	addressing	assembler	opc	bytes
1984	-----	-----	-----	-----
1985	implied	RTI	40	1

1986 RTS Return from Subroutine

1987	pull PC, PC+1 -> PC	N	Z	C	I	D	V
1988		-	-	-	-	-	-

1989	addressing	assembler	opc	bytes
1990	-----	-----	-----	-----
1991	implied	RTS	60	1

1992 SBC Subtract Memory from Accumulator with Borrow

1993	A - M - C -> A	N	Z	C	I	D	V
1994		+	+	+	-	-	+

1995	addressing	assembler	opc	bytes
1996	-----			
1997	immediate	SBC #oper	E9	2
1998	absolute	SBC oper	ED	3
1999	absolute,X	SBC oper,X	FD	3
2000	absolute,Y	SBC oper,Y	F9	3
2001	(indirect,X)	SBC (oper,X)	E1	2
2002	(indirect),Y	SBC (oper),Y	F1	2

2003 SEC Set Carry Flag

2004	1 -> C		N	Z	C	I	D	V
2005			-	-	1	-	-	-

2006	addressing	assembler	opc	bytes
2007	-----			
2008	implied	SEC	38	1

2009 SED Set Decimal Flag

2010	1 -> D		N	Z	C	I	D	V
2011			-	-	-	-	1	-

2012	addressing	assembler	opc	bytes
2013	-----			
2014	implied	SED	F8	1

2015 SEI Set Interrupt Disable Status

2016	1 -> I		N	Z	C	I	D	V
2017			-	-	-	1	-	-

2018	addressing	assembler	opc	bytes
2019	-----			
2020	implied	SEI	78	1

2021 STA Store Accumulator in Memory

2022	A -> M		N	Z	C	I	D	V
2023			-	-	-	-	-	-

2024	addressing	assembler	opc	bytes
2025	-----			
2026	absolute	STA oper	8D	3
2027	absolute,X	STA oper,X	9D	3

2028	absolute,Y	STA oper,Y	99	3
2029	(indirect,X)	STA (oper,X)	81	2
2030	(indirect),Y	STA (oper),Y	91	2

2031 STX Store Index X in Memory

2032	X -> M		N Z C I D V
2033			- - - - -

2034	addressing	assembler	opc	bytes
2035	-----			
2036	absolute	STX oper	8E	3

2037 STY Store Index Y in Memory

2038	Y -> M		N Z C I D V
2039			- - - - -

2040	addressing	assembler	opc	bytes
2041	-----			
2042	absolute	STY oper	8C	3

2043 TAX Transfer Accumulator to Index X

2044	A -> X		N Z C I D V
2045			+ + - - -

2046	addressing	assembler	opc	bytes
2047	-----			
2048	implied	TAX	AA	1

2049 TAY Transfer Accumulator to Index Y

2050	A -> Y		N Z C I D V
2051			+ + - - -

2052	addressing	assembler	opc	bytes
2053	-----			
2054	implied	TAY	A8	1

2055 TSX Transfer Stack Pointer to Index X

2056	SP -> X		N Z C I D V
2057			+ + - - -

2058	addressing	assembler	opc	bytes
2059	-----			
2060	implied	TSX	BA	1

2061 **TXA Transfer Index X to Accumulator**

2062	X -> A		N	Z	C	I	D	V
2063			+	+	-	-	-	-

2064	addressing	assembler	opc	bytes
2065	-----			
2066	implied	TXA	8A	1

2067 **TXS Transfer Index X to Stack Register**

2068	X -> SP		N	Z	C	I	D	V
2069			+	+	-	-	-	-

2070	addressing	assembler	opc	bytes
2071	-----			
2072	implied	TXS	9A	1

2073 **TYA Transfer Index Y to Accumulator**

2074	Y -> A		N	Z	C	I	D	V
2075			+	+	-	-	-	-

2076	addressing	assembler	opc	bytes
2077	-----			
2078	implied	TYA	98	1