

6502 Machine 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/>

Table of Contents

How The Dump Command Displays Memory Locations.....	3
American Standard Code For Information Interchange (ASCII).....	4
The Dump Command And The Enter Command.....	8
The Register Command.....	11
A Simple Machine Language Program.....	13
Addressing Modes.....	14
The Unassemble Command.....	16
The Go Command.....	18
The Trace Command.....	20
Exercises.....	24

1 How The Dump Command Displays Memory Locations

2 "Now that you understand what binary numerals and hexadecimal numerals
 3 are Pat," I said "we can go back to the output from the Dump command and
 4 study it. By the way, the word **hex** is often used as a shorter version of
 5 hexadecimal and we will using both words from now on. The Dump
 6 command shows the contents of a computer's memory locations and its
 7 output is arranged in 3 columns." I recreated the Dump command's output
 8 on the whiteboard and labeled each column. (see Fig. 1)

Figure 1

Address of first byte in a row

16 bytes are contained in each row

ASCII interpretation of the 16 bytes in each row

	E000	E001	E002	...	E004	...	E006	E007	E008	E009	E00A	...	E00E	E00F	
E000	4C	1B	E0	4C	B5	F3	4C	74	-	F3	4C	97	F3	4C	L..L..Lt.L..LW.L
E010	A7	F2	4C	03	F3	4C	41	EA	-	4C	70	E0	A2	FF	..L..LA.Lp....
E020	37	F3	4C	25	E0	A2	16	A0	-	F4	20	57	F3	00	7.L%.... W....
E030	13	29	F0	4A	4A	4A	18	69	-	C0	8D	09	00	A9	.)...JJJ.i.....
E040	00	6C	08	00	4C	47	E0	A0	-	00	8C	44	00	A2	..L..LG....D..d..
E050	FD	CA	D0	FA	EE	44	00	AD	-	44	00	4C	4C	E0D..D.LL. i.
E060	20	DB	E0	20	33	E1	4C	5D	-	E0	A2	66	A0	F4	.. 3.L]..f.. W.
E070	A0	00	8C	5E	00	8C	CD	00	-	8C	5F	00	8C	73	...^....._...s...
E080	00	A0	00	A9	00	99	A5	00	-	C8	C0	27	D0	F8'.....
E090	97														

9 "Each row in the **center** column shows the contents of 16 consecutive
 10 memory locations," I said "and each row in the **left** column contains the
 11 address of the first byte in that row. For example, the byte **4C** hex is in
 12 memory location E000, **1B** hex is in location E001, and **E0** hex is in location
 13 E002. Toward the end of the top row, **F3** hex is in location E00E and **4C**
 14 hex is in location E00F. We will discuss the **right** column in a moment." As
 15 I said this I wrote the addresses for some of the bytes in the first row and
 16 drew arrows pointing from the addresses to the bytes they contained.

17 Pat studied the output for a while then asked "What address comes after
 18 E00F hex?"

19 "The same counting rules that we used with decimal numerals and binary
 20 numerals also apply to hexadecimal numerals." I said "In this case, when 1

21 is added to E00F hex, the 'F' rolls around to 0 and 1 is added to the column
22 to its left. The result is E010 hex and notice how this is the address of the
23 first byte in the second row." I circled locations E00F hex and E010 hex in
24 red and then drew a red line pointing from location E00F hex to location
25 E010 hex. (again, see Fig. 1)

26 Then I said "What numeral is in memory location E010 hex, Pat?"

27 "A7 hex." said Pat.

28 "Very good, now what are the contents of memory locations E04E hex, E076
29 hex, and E08C hex?" I asked.

30 Pat looked at the output again and replied "Memory location E04E hex
31 contains C8 hex, location E076 hex contains CD hex, and location E08C hex
32 contains F8 hex."

33 "Excellent!" I said "I think you now understand how the Dump command
34 displays memory locations."

35 "What's the little dash for that is in the middle of each row?" asked Pat.

36 "The dash," I replied "divides each row into 2 groups of 8 bytes. It is added
37 to the output to make it easier to find a given address in a row. For
38 example, if I wanted to know what the contents of location E028 hex was, I
39 would find the row which began with address E020 hex, then I would locate
40 the address that was to the immediate right of the dash."

41 "Okay," said Pat "That makes sense. I think I understand how the first two
42 columns work, but what is in the third column?"

43 **American Standard Code For Information Interchange (ASCII)**

44 "Do you remember our discussion about contextual meaning and how
45 numerals in a computer can be made to represent any idea one can think
46 of?" I asked.

47 "Yes," replied Pat "I remember".

48 "There is a specification called the **American Standard Code for**
49 **Information Interchange**, or **ASCII**, which associates all of the symbols

50 (or **characters**) on a keyboard with the numerals between 0 and 127 in
51 the decimal numeral system. Since 0 through 127 in the decimal numeral
52 system is equivalent to 0 through 7F in the hexadecimal numeral system,
53 the ASCII characters can also be thought of as being associated with these
54 hexadecimal numerals too."

55 "Does this mean that the ASCII characters are also associated with a range
56 of binary numerals?" asked Pat.

57 "Yes," I replied "and the binary numerals that they are associated with are
58 00000000 through 01111111. I will draw a table on the whiteboard which
59 shows most of the ASCII characters along with the decimal and hexadecimal
60 numerals that they are associated with. We could have also included the
61 binary numerals in this table but it is so easy to convert the hexadecimal
62 numerals to binary that we will leave them off." I then created the table
63 while Pat watched. (see Table 1)

ASCII (American Standard Code for Information Interchange) Chart

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
10	0a	Linefeed/Newline	63	3F	?	96	60	`
13	0d	Carriage Return	64	40	@	97	61	a
32	20	Space	65	41	A	98	62	b
33	21	!	66	42	B	99	63	c
34	22	"	67	43	C	100	64	d
35	23	#	68	44	D	101	65	e
36	24	\$	69	45	E	102	66	f
37	25	%	70	46	F	103	67	g
38	26	&	71	47	G	104	68	h
39	27	'	72	48	H	105	69	i
40	28	(73	49	I	106	6A	j
41	29)	74	4A	J	107	6B	k
42	2A	*	75	4B	K	108	6C	l
43	2B	+	76	4C	L	109	6D	m
44	2C	,	77	4D	M	110	6E	n
45	2D	-	78	4E	N	111	6F	o
46	2E	.	79	4F	O	112	70	p
47	2F	/	80	50	P	113	71	q
48	30	0	81	51	Q	114	72	r
49	31	1	82	52	R	115	73	s
50	32	2	83	53	S	116	74	t
51	33	3	84	54	T	117	75	u
52	34	4	85	55	U	118	76	v
53	35	5	86	56	V	119	77	w
54	36	6	87	57	W	120	78	x
55	37	7	88	58	X	121	79	y
56	38	8	89	59	Y	122	7A	z
57	39	9	90	5A	Z	123	7B	{
58	3A	:	91	5B	[124	7C	
59	3B	;	92	5C	\	125	7D	}
60	3C	<	93	5D]	126	7E	~
61	3D	=	94	5E	^			
62	3E	>	95	5F	_			

Table 1

64 "Each row in the **right** column of the Dump command's output contains 16
 65 ASCII characters," I said "and each of these characters is matched with one
 66 of the 16 bytes in the **center** column. The leftmost byte is matched with the
 67 leftmost ASCII character, the next byte to the right is matched with the next
 68 ASCII character, and so on. The ASCII characters that are **periods** are
 69 either actual periods or they represent a number that is not matched with
 70 any of the ASCII characters."

71 "Why is the last ASCII character in the second row blank?" asked Pat.

72 "What byte is that blank associated with?" I asked.

73 Pat matched the blank character with the last byte in the second row then
74 said "the blank is matched with 20 hex."

75 "And what character is the 20 hex matched with in the ASCII table?" I
76 asked.

77 Pat located 20 hex in the ASCII table then said "A space! The blank
78 character is really a space!"

79 "Correct!" I said. "This is the ASCII character that is associated with the
80 space on a keyboard."

81 Pat looked at the 2 ASCII characters that were associated with the decimal
82 numerals 10 and 13 then said "What are the Newline and Carriage Return
83 characters?"

84 I replied "The ASCII characters that are associated with the decimal
85 numerals between 0 and 31 are called **control characters**. These
86 characters do not print symbols and instead they were created to control
87 the old mechanical teletypes that use to be used as input/output devices for
88 computers. These control characters are still used to control a text
89 interface to a computer and I have included the 2 most commonly used ones
90 in the table.

91 When a display receives a **Carriage Return**, it moves the cursor to the left
92 side of the display. When it receives a **Linefeed** character, it drops the
93 cursor down to the next line. Some computers use the **Linefeed** character
94 to move the cursor to the left side of the display, and also drop it down one
95 line, in one operation. When it is used in this manner it is called a **Newline**
96 character."

97 Pat looked at the column of ASCII characters in the memory dump for a
98 while then said "It does not seem like there are very many ASCII characters
99 in this area of memory. Why is that?"

100 "The reason could be that we are looking at an area of memory that has
101 garbage in it," I said "or these bytes may be associated with some other
102 context than the ASCII character context."

103 "Can we look at an area of memory that does have some ASCII characters in
104 it?" asked Pat.

105 "We can do something even better than that!" I said. "Lets put some
106 numerals in memory ourselves that represent ASCII characters and then
107 use the Dump command to look at them."

108 "Okay!" said Pat "How do we do that?"

109 The Dump Command And The Enter Command

110 "First, we must launch MathRider then open the U6502 emulator. What I
111 usually do after opening the U6502 emulator is to send a question mark
112 character to the UMON65 monitor that is running on it. If the monitor
113 responds with the help message, then we know that the emulator is running
114 correctly." I did these operations and the monitor displayed the following
115 help message. Feel free to work along with Pat and I on your own system
116 as we explore the monitor.

117 ?

118 Assemble	A start_address
119 Breakpoint	B (+,-,?) address
120 Dump	D [start_address [end_address]]
121 Enter	E address list
122 Fill	F start_address end_address list
123 Go	G [start_address]
124 Help	H or ?
125 Load	L
126 Move	M start_address end_address destination_address
127 Register	R [PC,AC,XR,YR,SP,SR]
128 Search	S start_address end_address list
129 Trace	T [start_address [value]]
130 Unassemble	U [start_address [end_address]]

131 "The monitor program is called **umon65** and it has a manual that describes
132 what each command does. Lets see what it has to say about the Dump
133 command." I located the manual for umon65 and here is the information it
134 contained about the Dump command:

135 DUMP COMMAND

136 SYNTAX: D [START_ADDRESS [END_ADDRESS]] where START_ADDRESS and END_ADDRESS
137 are 4 digit hexadecimal numbers.

138 DESCRIPTION: The purpose of the dump command is to allow the user to dump
139 (print) the contents of the specified address locations. Each line of the
140 dump command's output consists of a starting dump address, the contents of

141 the 16 address locations beginning with the start address, and the ASCII
 142 conversion for each of the 16 dumped addresses. If no end address is
 143 specified then only 1 line is dumped starting at the start address. If no
 144 start address is specified then 1 line is dumped starting at the user's
 145 current Program Counter.

146 EXAMPLE:
 147 D 1000 E0FF
 148 D 1000
 149 D

150 "What are the brackets for after the command?" asked Pat.

151 "Information in brackets like [START_ADDRESS [END_ADDRESS]] indicate
 152 optional parameters that can be passed to a command." I replied. "In this
 153 case, passing a **start address** is optional and if a **start address** is passed,
 154 then including an **end address** is optional."

155 "Okay," said Pat "I understand."

156 "Now," I said "lets use the Dump command to look at a section of memory
 157 that does not have a context associated with it yet." I entered **d 0200 024f**
 158 and the emulator displayed the contents of these memory locations:

```
159 -d 0200 024f
160 0200  00 00 00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00 .....
161 0210  00 00 00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00 .....
162 0220  00 00 00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00 .....
163 0230  00 00 00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00 .....
164 0240  00 00 00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00 .....
```

165 "The command that allows the user to enter bytes into memory is called the
 166 **Enter** command," I said "and here is the information that the umon65
 167 manual contains on this command." I then located the Enter command's
 168 section in the manual and this is what it contained:

```
169 ENTER COMMAND

170 SYNTAX:  E ADDRESS LIST  where ADDRESS is a 4 digit hexadecimal number and
171 LIST is one 2 digit hexadecimal number or up to five 2 digit hexadecimal
172 numbers separated by commas.

173 DESCRIPTION:  The purpose of the Enter command is to allow the user to
174 enter one byte or a list of bytes directly into memory at a specified address.

175 EXAMPLE:
176 E 0200 F6
177 E 0200 23,6C,3A,D1
```

178 "Now Pat," I said "use the Enter command to place the hexadecimal
 179 numeral that represents an ASCII 'A' into memory location 0200 hex. Then
 180 use the Dump command to see if the number was indeed placed there."

181 Pat looked at the ASCII table on the whiteboard and determined that 41 hex
 182 represented a capital letter 'A'. Pat then typed **e 0200 41 <enter>**
 183 followed by **d 0200 024f <enter>** in the telnet window and the following
 184 information was displayed:

185 -e 0200 41

186 -d 0200 024f

```

187 0200  41 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  A.....
188 0210  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ..
189 0220  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ..
190 0230  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ..
191 0240  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ..

```

192 Pat looked at the 41 that was placed into location 0200 hex and then looked
 193 at the capital letter 'A' that was now present in the beginning of the first
 194 row of ASCII characters.

195 "There's the 41 hex and there's the capital letter 'A' that goes with it!" cried
 196 Pat. "This is fun!"

197 "Yes, this is fun!" I said "I was just as thrilled as you are when I placed a
 198 capital 'A' in memory for the first time and I still enjoy manually placing
 199 data into memory. Go ahead and place a capital letter 'B' into memory
 200 location 0201hex and a capital letter 'C' into memory location 0202 hex."

201 Pat typed the following to enter the two values into memory and to verify
 202 that the values were indeed present in the specified memory locations:

203 -e 0201 42

204 -e 0202 43

205 -d 0200 024f

```

206 0200  41 42 43 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ABC.....
207 0210  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ..
208 0220  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ..
209 0230  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ..
210 0240  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ..

```

211 "I think I understand how to view the contents of memory locations and how
212 to change the contents of memory locations." said Pat "Now I would like to
213 see how to view and change the contents of registers."

214 The Register Command

215 "The monitor's **Register** command is used to view and change the contents
216 of registers and here is what the umon65's manual says about it." I said. I
217 then brought up the Register command's section on the computer screen:

218 REGISTER COMMAND

219 SYNTAX: R [PC, AC, XR, YR, SP, SR]

220 DESCRIPTION: The purpose of the Register command is to dump (print) the
221 contents of all the microprocessor's user accessible registers or to modify
222 any of these registers individually. If R is entered without specifying a
223 register to be modified, then the contents of all the registers are shown.
224 If a specific register is given after the R command, then the current
225 contents of this register are shown and an opportunity is given to change the
226 contents of this register.

227 EXAMPLE:

228 R

229 R AC

230 "Send either an upper case or lower case 'R' to the monitor," I said "and lets
231 see what it displays."

232 Pat did this and the following information was printed:

233 r

234	PgmCntr (PC)	Accum (AC)	XReg (XR)	YReg (YR)	StkPtr (SP)	NV-BDIZC (SR)
235	102C	00	FC	00	FD	00010110

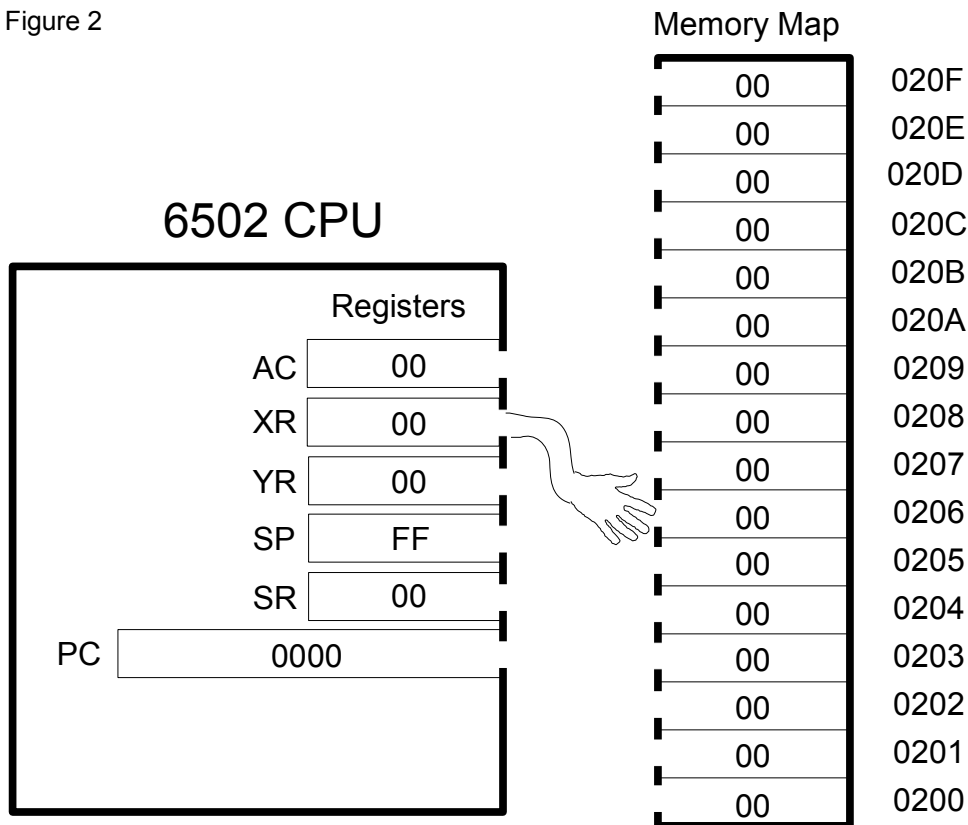
236 "We have already discussed the **Program Counter** register and the '**A**'
237 register but there four registers in this listing we have not discussed yet and
238 one register we have discussed which is missing. The register which is
239 missing is the **Instruction Register** and the reason for this is because it is
240 not directly accessible by the programmer.

241 In addition to the 'A' register, the 6502 contains 2 **index** registers called
242 register **X (XR in the listing)** and register **Y (YR in the listing)**. The
243 index registers can be used for a number of purposes. One purpose they
244 serve is to allow the CPU to access a desired memory location by holding a
245 value which indicates how many locations ahead in memory from a '**base**'

246 address this memory location is. These two index registers can also be used
 247 as counters and to temporarily hold values.

248 The **Stack Pointer register (SP in the listing)** and the **Status Register**
 249 **(SR in the listing)** are special registers which we will discuss later. I will
 250 now draw a version of the CPU and memory diagram we used earlier which
 251 contains the new registers and which also has its numerals expressed in
 252 hexadecimal." I then drew the diagram on the whiteboard. (see Fig. 2)

Figure 2



253 "Notice that all of the registers and memory locations are 8 bits wide except
 254 for the program counter, which is 16 bits wide." I said.

255 "Why is the program counter 16 bits wide?" asked Pat.

256 "How many patterns can be formed by 16 bits?" I asked.

257 Pat picked up the calculator, entered 2^{16} then said "65536."

258 "This means that the program counter is able to point to a maximum of
 259 65536 memory locations." I said. "The lowest address it can point to is 0
 260 and the highest address it can point to is 65535 decimal or FFFF hex."

261 "That makes sense," said Pat "because if the program counter was only 8
 262 bits wide, it would only be able to point to 2^8 or 256 memory locations.
 263 This wouldn't be very many memory locations to have in a computer."

264 Pat then said "How can we change the number that is in the 'A' register?"

265 "First enter a Register command to see what value is currently in the 'A'
 266 register, then enter **r ac** and when the colon ':' is displayed enter the byte
 267 you want to have placed into the 'A' register." Pat then did as I suggested:

268 r

269	PgmCntr (PC)	Accum (AC)	XReg (XR)	YReg (YR)	StkPtr (SP)	NV-BDIZC (SR)
270	E02C	00	FC	00	FD	00010110

271 -r ac

272 00
 273 :41

274 -r

275	PgmCntr (PC)	Accum (AC)	XReg (XR)	YReg (YR)	StkPtr (SP)	NV-BDIZC (SR)
276	E02C	41	FC	00	FD	00010110

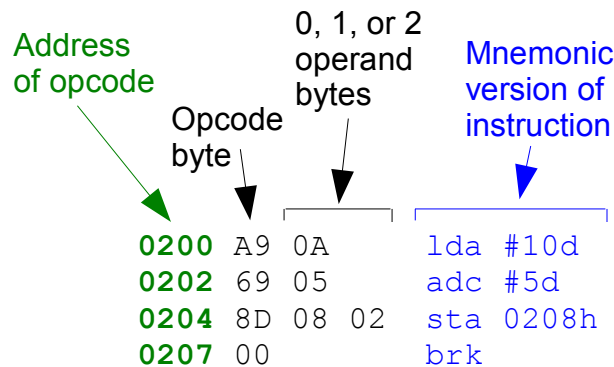
277 "It worked!" cried Pat.

278 "Yes, it did!" I replied. "Our next step is to have you enter your first
 279 machine language program."

280 A Simple Machine Language Program

281 "Earlier, we created a small machine language program that added 10
 282 decimal to 5 decimal then placed the sum into a memory location. I will
 283 write this program on the whiteboard and include the assembly language
 284 version of each instruction along with the address in memory which each
 285 instruction starts at. Instead of starting a 0, this program will start at 0200
 286 hex." (see Fig. 3)

Figure 3



287 Addressing Modes

288 "In this program, the opcode for the LDA instruction is A9 hex and it is in
 289 memory location 0200 hex. Earlier, we had used 169 decimal to represent
 290 this instruction because we had not discussed hexadecimal yet. Another
 291 topic we did not discuss earlier is that some assembly language instructions
 292 are able to access memory locations in different ways. These different ways
 293 are called **addressing modes** and I will draw a table on the whiteboard
 294 which shows most of the addressing modes that the LDA instruction can
 295 use.

296 This table will contain the name of the addressing mode, an example of
 297 what its assembly language syntax looks like, the opcode that is associated
 298 with the addressing mode, and the number of bytes that each version of the
 299 instruction takes." I then created the following table:

300 LDA (Load A register) Instruction

301	MODE	SYNTAX	OPCODE	BYTES
302	Immediate	LDA #41h	A9	2
303	Absolute	LDA 02A0h	AD	3
304	Absolute,X	LDA 02A0h,X	BD	3
305	Absolute,Y	LDA 02A0h,Y	B9	3
306	Indirect,X	LDA (20h,X)	A1	2
307	Indirect,Y	LDA (20h),Y	B1	2

308 "What does syntax mean?" asked Pat.

309 "Syntax refers to the rules that determine how to properly type an
 310 instruction." I replied. "If a programmer does not follow the language's
 311 syntax rules when typing in a program, the software that transforms the
 312 source code into machine language will become confused and then issue

313 what is called a syntax error. For example, typing LMA instead of LDA
314 would be considered a syntax error."

315 "Okay." said Pat. "Now, can you tell me more about these addressing
316 modes?"

317 "The **immediate** addressing mode indicates that the data the instruction is
318 going to work on has been placed **immediately** after the **opcode** in
319 memory. I said. "In our program, the number 10 decimal will be copied
320 into the 'A' register when the instruction is executed, therefore I placed a
321 0A hex into memory location 0201 hex. This is a 2 byte instruction because
322 it consists of an opcode and 1 byte of immediate data.

323 In the assembly language we are using, a pound sign '#' is placed before a
324 piece of data to indicate that immediate addressing mode should be used to
325 access it. Also notice that when a '**d**' is placed at the end of a numeral, it
326 indicates that it is a **decimal** numeral. An '**h**' at the end of a numeral
327 indicates that it is a **hexadecimal** numeral and a '**b**' at the end of a numeral
328 indicates that it is a **binary** numeral.

329 **Absolute** addressing mode allows the programmer to specify the address of
330 the memory location that an instruction will either copy a byte from or copy
331 a byte to. The absolute mode example in the syntax column indicates that
332 the LDA instruction will copy the numeral that is in location 020A hex into
333 the 'A' register when it is executed.

334 The **Absolute,X** and **Absolute,Y** addressing modes are similar to the
335 normal **Absolute** addressing mode except that the contents of either the 'X'
336 register or the 'Y' register are added to the absolute address that the
337 programmer specified in order to determine which location in memory will
338 have a byte copied from it or copied into it.

339 The **Indirect,X** and **Indirect,Y** addressing modes are somewhat advanced
340 and therefore I will wait until you have gained some programming
341 experience before explaining them."

342 "The Immediate and Absolute addressing modes make sense to me," said
343 Pat "but I am a bit fuzzy on how the Absolute,X and Absolute,Y addressing
344 modes work."

345 "That is okay, Pat." I replied "It is normal for machine language and
346 assembly language to be confusing when it is first being learned. All these

347 addressing modes will become clearer to you when you start using them.
348 For now, though, I suggest that you use the Enter command to place the
349 machine language instructions for this simple program into memory."

350 Here is what Pat typed while entering the program:

```
351 -e 0200 a9
352 -e 0201 0a
353 -e 0202 69
354 -e 0203 05
355 -e 0204 8d
356 -e 0205 08
357 -e 0206 02
358 -e 0207 00
359 -d 0200
360 0200 A9 0A 69 05 8D 08 02 00 - 00 00 00 00 00 00 00 00 ..i.....
```

361 The Unassemble Command

362 Pat looked at the output from the Dump command then said "There's my
363 first machine language program! Looking at just the machine language
364 version of a program's instructions, without having their assembly language
365 equivalent available, is difficult though."

366 "It is difficult," I agreed "but fortunately the monitor has a command that
367 will display a machine language program's instructions along with the
368 assembly language version of these instructions."

369 "There is?!" asked Pat. "Which command is it?"

370 "It is called **Unassemble** and here is what the umon65's manual has to say
371 about it." I then brought up the section of the manual that contained
372 information about the Unassemble command:

```
373 UNASSEMBLE COMMAND

374 SYNTAX: U [START_ADDRESS [END_ADDRESS]] where START_ADDRESS and
375 END_ADDRESS are 4 digit hexadecimal numbers.
```


376 DESCRIPTION: The purpose of the Unassemble command is to convert machine
377 language instructions present in memory into their assembly language
378 equivalents and display them. If the U command is given with no starting
379 address, then approximately 1 screen full of instructions will be unassembled
380 starting at the current user's Program Counter. If a start address is given,
381 then approximately 1 screen full of instructions will be unassembled starting
382 at this start address. If an end address is specified, than all instructions
383 between the start address and the end address will be unassembled.

384 EXAMPLE:
385 U
386 U 1000
387 U 1000 10FF

388 "Enter the command **u 0200** Pat and see what happens." I said and this is
389 what the Unassemble command displayed:

390 -u 0200

391	0200	A9 0A	LDA #0Ah
392	0202	69 05	ADC #05h
393	0204	8D 08 02	STA 0208h
394	0207	00	BRK
395	0208	00	BRK
396	0209	00	BRK
397	020A	00	BRK
398	020B	00	BRK
399	020C	00	BRK
400	020D	00	BRK
401	020E	00	BRK
402	020F	00	BRK
403	0210	00	BRK
404	0211	00	BRK
405	0212	00	BRK
406	0213	00	BRK
407	0214	00	BRK

408 "That's amazing!" said Pat. "This programming stuff just keeps getting
409 more and more interesting!"

410 Pat studied the output from the Unassemble command for a while then said
411 "I know what the LDA, ADC, and STA instructions do, but what does the
412 BRK command do?"

413 "The BRK command generates what is called a **software interrupt** and an
414 interrupt causes the Program Counter to be set to a predefined address in
415 memory where a special program called an **interrupt handler** has been
416 placed. We will discuss interrupts later but for now you can think of the
417 BRK command as the instruction that ends a program. When a BRK

418 instruction is executed, control will be transfered from the program back to
419 the monitor."

420 "How do we run a program?" asked Pat.

421 "We will cover that next". I replied.

422 **The Go Command**

423 "The command that is used to execute a program with the monitor is the Go
424 command," I said "and here is the section in the umon65 manual that talks
425 about it." I then located the section on the Go command in the manual:

426 GO COMMAND

427 SYNTAX: G [START_ADDRESS] where START_ADDRESS is a 4 digit hexadecimal
428 number.

429 DESCRIPTION: The purpose of the Go command is to allow the user to start
430 execution of a program that was placed into memory. Execution will begin at
431 START_ADDRESS or if a start address is not given then execution will begin at
432 the user's current Program Counter.

433 EXAMPLE:

434 G

435 G 1000

436 "Before running the program with the Go command," I said "the first thing
437 you should do is to dump the 16 bytes starting at 0200 hex so that we can
438 verify that the program is indeed in memory and also that location 0208 hex
439 contains 00. The reason for making sure that 0208 hex has 00 in it is so we
440 can watch it change when the program places the sum it calculates into it."
441 Pat dumped the memory starting at address 0200 hex and we both verified
442 that the program was there and that 0208 hex had 00 in it:

443 -d 0200

444 0200 A9 0A 69 05 8D 08 02 00 - 00 00 00 00 00 00 00 00 ..i.....

445 "The next thing you should do is to look at the contents of the 'A' register
446 before running the program so you can see what it is changed to after the
447 program is finished." Pat did this too:

448 -r

449 PgmCntr (PC)	Accum (AC)	XReg (XR)	YReg (YR)	StkPtr (SP)	NV-BDIZC (SR)
450 E000	00	FE	FF	FD	00010100

451 "Now, execute the program by entering **g 0200**." I said. Pat ran the
 452 program and this was the result:

453 -g 0200

454	PgmCntr (PC)	Accum (AC)	XReg (XR)	YReg (YR)	StkPtr (SP)	NV-BDIZC (SR)
455	0207	0F	FE	FF	FD	00010100

456 -

457 "When you ran the program," I said "each of the instructions was executed
 458 one after the other and when the BRK command was executed, control was
 459 transfered back to the monitor and the monitor displayed the contents of
 460 the registers. The monitor then displayed the dash prompt '-' indicating
 461 that it was ready to accept commands again."

462 Pat compared the contents of the 'A' register before the program was
 463 executed and after it was executed. After a while Pat asked "why does the
 464 'A' register have a 0F hex in it?"

465 "When 10 decimal is added to 5 decimal, what is the sum?" I asked.

466 "15 decimal." replied Pat.

467 "And what is the hexadecimal equivalent of 15 decimal?" I asked.

468 Pat looked at the binary/decimal/hexadecimal table we had created earlier,
 469 located the row that contained 15 decimal then said "0F is the hex
 470 equivalent of 15 decimal! I understand now, the monitor can only display
 471 hexadecimal numerals and, if a person wants to know what a value is in
 472 decimal, they have to do the conversion themselves."

473 "Correct, Pat." I said. "The nice thing is that most scientific calculators are
 474 able to convert between decimal, binary, and hexadecimal very easily. The
 475 Windows operating system also has a calculator application that can do
 476 these conversions if it is placed into scientific mode."

477 "I didn't know that!" cried Pat. Then Pat picked up the calculator we had
 478 been using and looked at it with new eyes. I explained how to use the
 479 calculator to convert between the numeral systems and Pat was very
 480 excited about this.

481 "When I get home," said Pat "the first thing I am going to do is to find my

482 scientific calculator and see if it can do numeral conversions!"

483 I smiled at this then said "We need to check one last thing with this
484 program, Pat." I said.

485 "What's that?" asked Pat.

486 "We need to see if the sum that was calculated was placed into location
487 0208 hex." I replied.

488 "That's right!" said Pat. Then Pat dumped the area of memory that
489 contained location 0208 hex and we both looked at it:

490 -d 0200

491 0200 A9 0A 69 05 8D 08 02 00 - **0F** 00 00 00 00 00 00 00 ..i.....

492 "There's the sum 0F hex, sitting in location 0208 hex just like it should be!"
493 said Pat. "It would be nice, though, to be able to run a program 1
494 instruction at a time like we did when we were stepping through this
495 program on the whiteboard."

496 "The monitor is able to run 1 instruction at a time Pat," I said "and what we
497 will do is to create a program that adds 3 numbers together and then step
498 through it to see how it works."

499 **The Trace Command**

500 "The monitor command that allows a program to be executed 1 instruction
501 at a time is called the **Trace** and here is its section in the umon65 manual."
502 I brought up this section on the monitor and we both read it:

503 TRACE COMMAND

504 SYNTAX: T [START_ADDRESS [STEPS]] where START_ADDRESS is a 4 digit
505 hexadecimal number and STEPS is a 2 digit hexadecimal number.

506 DESCRIPTION: The purpose of the Trace command is to allow the user to
507 execute a program in memory 1 instruction at a time and dump the contents of
508 all the registers after each instruction is executed. Entering the T
509 command without a start address will execute 1 instruction starting at the
510 user's current Program Counter. If a start address is given without any
511 steps, then 1 instruction is executed at the start address. If a number of
512 steps is given with a start address, then the number of instructions
513 indicated by the steps count will be executed starting at the start address.
514 Once the first instruction has been Traced, simply typing T then <enter> will
515 execute the next instruction in memory.

```

516 EXAMPLE:
517 T
518 T 1000
519 T 1000 0C

```

520 "The Trace command is very powerful because it allows one to see exactly
 521 what happens in the registers and in memory after each instruction is
 522 executed. Now, I would like you to enter the following program into
 523 memory using the Enter command." I then wrote this program on the
 524 whiteboard and Pat entered into memory:

```

525 0200 A9 0A 69 05 69 02 8D 0A 02 00

```

526 "Lets see what this machine language program looks like in assembly
 527 language Pat." I said. "Do you remember how to do this?"

528 "Yes," replied Pat "the Unassemble command is used to view the assembly
 529 language equivalent of a machine language program." Pat then entered
 530 the following command:

```

531 -u 0200

```

```

532 0200 A9 0A      LDA #0Ah
533 0202 69 05      ADC #05h
534 0204 69 02      ADC #02h
535 0206 8D 0A 02    STA 020Ah
536 0209 00          BRK
537 020A 00          BRK
538 020B 00          BRK

```

539 "What does this program do Pat?" I asked?

540 Pat studied the program then said "It looks like the program is adding 10
 541 decimal and 5 together and then adding 2 to their sum. The final sum is
 542 then being copied into memory location 020A hex."

543 "Very good, Pat." I said. "Now, the first step I would like you to perform is
 544 to dump the area of memory which contains location 020A hex so that we
 545 can see what it contains before we copy a number into it." Pat used the
 546 Dump command to do this as follows:

```

547 -d 0200

```

```

548 0200 A9 0A 69 05 69 02 8D 0A - 02 00 00 00 00 00 00 00 ..i.i.....

```

549 "The next step is to view the registers so that we can see what they contain

550 before the program changes them." I said and then Pat executed a Register
551 command:

552 -r

553	PgmCntr (PC)	Accum (AC)	XReg (XR)	YReg (YR)	StkPtr (SP)	NV-BDIZC (SR)
554	E02C	00	FC	00	FD	00010110

555 "Now," I said "enter a **t 0200** command and this will execute the LDA #0A
556 instruction which is at memory location 0200 hex." Pat entered the Trace
557 command and here was the result:

558 -t 0200

559	PgmCntr (PC)	Accum (AC)	XReg (XR)	YReg (YR)	StkPtr (SP)	NV-BDIZC (SR)
560	0202	0A	FC	00	FD	00010100

561 0202 69 05 ADC #05h

562 "Notice that the 'A' register now has the number 0A hex in it because the
563 LDA instruction copied it there." I said. "Also notice that the Trace
564 command unassembled the next instruction to be executed so that we could
565 see what will happen during the next trace."

566 "This is exciting!" said Pat. "Can I trace the next instruction?"

567 "In a moment," I replied "but first tell me what you think the result of
568 executing the next instruction will be."

569 Pat studied the output from the Trace command for a while then said "The
570 ADC instruction is going to add 5 to the contents of the 'A' register and then
571 place the result back into the 'A' register. The 'A' register currently holds
572 0A hex which is 10 decimal and 10 plus 5 is 15 decimal. 15 decimal is 0F
573 hex so after we trace this instruction the number 0F hex should be present
574 in register 'A'.

575 "Okay, perform another trace and lets see if you are right." I said. Pat
576 traced the next instruction and this was the result:

577 -t

578	PgmCntr (PC)	Accum (AC)	XReg (XR)	YReg (YR)	StkPtr (SP)	NV-BDIZC (SR)
579	0204	0F	FC	00	FD	00010100

580 0204 69 02 ADC #02h

581 "I was right!" cried Pat.

582 "Yes, you were right," I said "now what will be the result of executing the
583 next instruction?

584 "Well," said Pat "2 is going to be added to the 0F hex that is currently in the
585 'A' register and the result will be placed back into the 'A' register. 0F hex
586 plus 1 is 10 hex and 10 hex plus 1 is 11 hex so the number 11 hex should be
587 present in the 'A' register after the next instruction is executed." Pat then
588 entered another Trace command and this was the result:

589 -t

PgmCntr (PC)	Accum (AC)	XReg (XR)	YReg (YR)	StkPtr (SP)	NV-BDIZC (SR)
0206	11	FC	00	FD	00010100

592 0206 8D 0A 02 STA 020Ah

593 Pat smiled then said "I was right again! So far machine language is not as
594 hard as I thought it was going to be, but it is very different than I thought it
595 would be. I would have never imagined that this was how a computer really
596 worked at its lowest levels."

597 "Sometimes I still find it hard to believe that all of the complex things
598 computers are able to do are made possible by a small set of very simple
599 machine language instructions." I said. "Lets finish this program by tracing
600 the last instruction which will copy the final sum from the 'A' register to
601 memory location 020A hex. We will not need to trace the BRK instruction
602 because the Trace command already brings us back into the monitor when
603 it is done executing." Pat then traced the last instruction and this was the
604 result:

PgmCntr (PC)	Accum (AC)	XReg (XR)	YReg (YR)	StkPtr (SP)	NV-BDIZC (SR)
0209	11	FC	00	FD	00010100

607 0209 00 BRK

608 -d 0200

609 0200 A9 0A 69 05 69 02 8D 0A - 02 00 11 00 00 00 00 00 ..i.i.....
610 -

611 After we verified that the number 11 hex was copied from the 'A' register
612 into memory location 020A hex I said "This was your first experience with
613 programming a computer in machine language. Understanding how a
614 computer works at the machine language level will help you to write better
615 programs when you start learning how to program in higher level

616 languages. The next time we meet, we will be taking one step towards
617 those higher level languages by learning how to program in assembly
618 language."

619 "I can't wait!" said Pat.

620 Exercises

621 1) The umon65 monitor program is written in 6502 assembly language and
622 it is located in memory starting at address E000h. Use the Unassemble
623 command to look at the beginning part of this program. Note: You can type
624 'u E000' to begin the unassemble process and then just type a 'u' with no
625 parameters to unassemble further sections of the program.

626 2) Write a machine language program that adds the numbers 1,2,3,4,5, and
627 6 together and places the sum into location 0280h. Begin the program at
628 0200h. Run the program and verify that it works correctly (if you would
629 like to save your program, unassemble it, highlight the unassembled
630 program with the mouse, and then type <ctrl>c to copy it. It can then be
631 pasted into another application such as a text editor.)

632 3) Enter in the following short machine language programs into the
633 emulator one at a time and execute each one using the Trace command.
634 Look closely at what happens to the register or memory location that is
635 being worked with after each trace is executed. Also, pay close attention to
636 what happens to the program counter.

```
637  
638 ; (a)  
639 0200 A2 00      ldx #0h  
640 0202 E8         inx  
641 0203 E8         inx  
642 0204 E8         inx  
643 0205 E8         inx  
644 0206 E8         inx  
645 0207 00        brk  
646  
647 ; (b)  
648 0200 A0 00      ldy #0h  
649 0202 C8         iny  
650 0203 C8         iny  
651 0204 C8         iny  
652 0205 C8         iny  
653 0206 C8         iny  
654 0207 00        brk  
655  
656 ; (c)  
657 0200 A2 04      ldx #04h  
658 0202 CA         dex
```



```
659 0203 CA      dex
660 0204 CA      dex
661 0205 CA      dex
662 0206 CA      dex
663 0207 CA      dex
664 0208 CA      dex
665 0209 00      brk
666
667
668 ; (d)
669 0200 A9 00      lda #0h
670 0202 8D 20 02    sta 0220h
671 0205 EE 20 02    inc 0220h
672 0208 EE 20 02    inc 0220h
673 020B EE 20 02    inc 0220h
674 020E 00      brk
675
676
677 ; (e)
678 0200 A2 08      ldx #8d
679 0202 CA      dex
680 0203 D0 FD      bne 0202h
681 0205 00      brk
682
```

683 4) Launch a DOS shell window and type the following at the command prompt:

```
684 debug
685 -?
686 -d 0000 ffff
```

687 (On Windows machines, select Start -> Run and then type "command" or "cmd" in
688 the Run window's text box in order to launch the shell.)