6502 Machine Language

by Ted Kosan

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

Table of Contents

How The Dump Command Displays Memory Locations

"Now that you understand what binary numerals and hexadecimal numerals are Pat," I said "we can go back to the output from the Dump command and study it. By the way, the word **hex** is often used as a shorter version of hexadecimal and we will using both words from now on. The Dump command shows the contents of a computer's memory locations and its output is arranged in 3 columns." I recreated the Dump command's output 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 57 F3 4C   L..L..Lt.L..LW.L
E010  A7 F2 4C 03 F3 4C 41 EA - 4C 70 E0 A2 FF 9A D8 20   ..L..LA.Lp.....
E020  37 F3 4C 25 E0 A2 16 A0 - F4 20 57 F3 00 C9 02 D0   7.L%..... W.....
E030  13 29 F0 4A 4A 4A 18 69 - C0 8D 09 00 A9 00 8D 08   .).JJJ.i........
E040  00 6C 08 00 4C 47 E0 A0 - 00 8C 44 00 A2 64 C8 D0   .l..LG....D..d..
E050  FD CA D0 FA EE 44 00 AD - 44 00 4C 4C E0 20 69 E0   .....D..D.LL. i.
E060  20 DB E0 20 33 E1 4C 5D - E0 A2 66 A0 F4 20 57 F3    .. 3.L]..f.. W.
E070  A0 00 8C 5E 00 8C CD 00 - 8C 5F 00 8C 73 00 8C 87   ...^....._..s...
E080  00 A0 00 A9 00 99 A5 00 - C8 C0 27 D0 F8 A0 00 20   ..........'....
E090  97
```

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

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

"The same counting rules that we used with decimal numerals and binary numerals also apply to hexadecimal numerals." I said "In this case, when 1 is added to E00F hex, the 'F' roles around to 0 and 1 is added to the column to its left. The result is E010 hex and notice how this is the address of the first byte in the second row." I circled locations E00F hex and E010 hex in red and then drew a red line pointing from location E00F hex to location E010 hex. (again, see Fig. 1)

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

"A7 hex." said Pat.

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

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

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

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

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

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

American Standard Code For Information Interchange (ASCII)

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

"Yes," replied Pat "I remember".

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

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

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

"Yes," I replied "and the binary numerals that they are associated with are 00000000 through 01111111. I will draw a table on the whiteboard which shows most of the ASCII characters along with the decimal and hexadecimal numerals that they are associated with. We could have also included the binary numerals in this table but it is so easy to convert the hexadecimal numerals to binary that we will leave them off." I then created the table 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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

The Dump Command And The Enter Command

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

```
?

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

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

```
DUMP COMMAND

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

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

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

EXAMPLE:
D 1000 E0FF
D 1000
D
```

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

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

"Okay," said Pat "I understand."

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

```
-d 0200 024f

0200  00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00  ................
0210  00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00  ................
0220  00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00  ................
0230  00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00  ................
0240  00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00  ................
```

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

```
ENTER COMMAND

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

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

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

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

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

```
-e 0200 41

-d 0200 024f

0200  41 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00  A...............
0210  00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00  ................
0220  00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00  ................
0230  00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00  ................
0240  00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00  ................
```

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

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

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

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

```
-e 0201 42

-e 0202 43

-d 0200 024f

0200  41 42 43 00 00 00 00 00 - 00 00 00 00 00 00 00 00  ABC.............
0210  00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00  ................
0220  00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00  ................
0230  00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00  ................
0240  00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00  ................
```

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

The Register Command

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

```
REGISTER COMMAND

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

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

EXAMPLE:
R
R AC
```

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

Pat did this and the following information was printed:

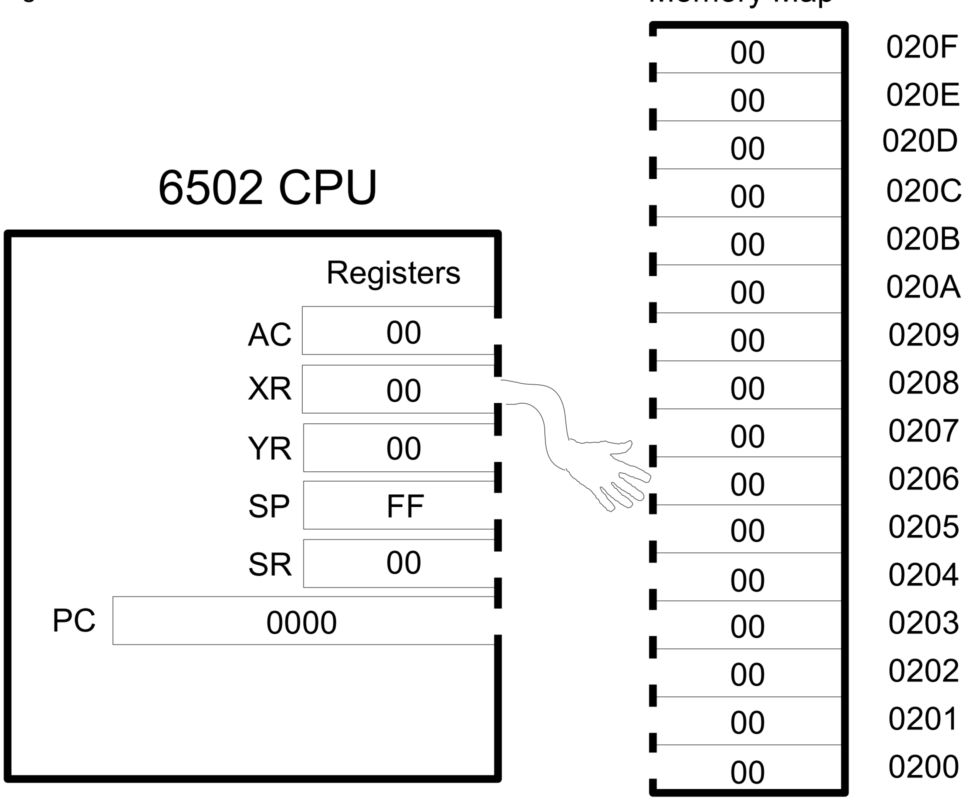
```
r

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

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

In addition to the 'A' register, the 6502 contains 2 **index** registers called register **X (XR in the listing)** and register **Y (YR in the listing)**. The index registers can be used for a number of purposes. One purpose they serve is to allow the CPU to access a desired memory location by holding a value which indicates how many locations ahead in memory from a **'base'** address this memory location is. These two index registers can also be used as counters and to temporarily hold values.

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

Figure 2



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

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

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

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

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

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

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

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

```
r

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

-r ac

00
 :41

-r

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

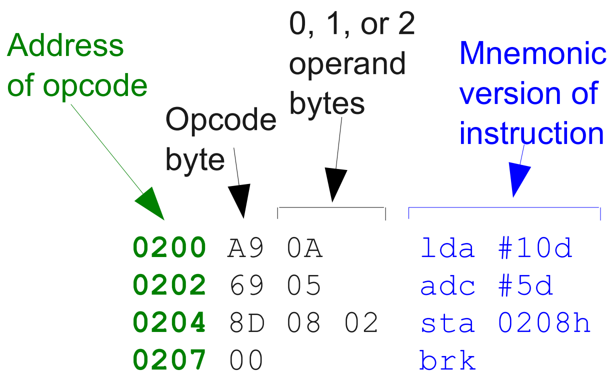
"It worked!" cried Pat.

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

A Simple Machine Language Program

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

Figure 3



Addressing Modes

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

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

LDA (LoaD A register) Instruction

MODE	SYNTAX	OPCODE	BYTES
Immediate	LDA #41h	A9	2
Absolute	LDA 02A0h	AD	3
Absolute,X	LDA 02A0h,X	BD	3
Absolute,Y	LDA 02A0h,Y	B9	3
Indirect,X	LDA (20h,X)	A1	2
Indirect,Y	LDA (20h),Y	B1	2

"What does syntax mean?" asked Pat.

"Syntax refers to the rules that determine how to properly type an instruction." I replied. "If a programmer does not follow the language's syntax rules when typing in a program, the software that transforms the source code into machine language will become confused and then issue what is called a syntax error. For example, typing LMA instead of LDA would be considered a syntax error."

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

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

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

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

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

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

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

"That is okay, Pat." I replied "It is normal for machine language and assembly language to be confusing when it is first being learned. All these addressing modes will become clearer to you when you start using them. For now, though, I suggest that you use the Enter command to place the machine language instructions for this simple program into memory."

Here is what Pat typed while entering the program:

```
-e 0200 a9

-e 0201 0a

-e 0202 69

-e 0203 05

-e 0204 8d

-e 0205 08

-e 0206 02

-e 0207 00

-d 0200

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

The Unassemble Command

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

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

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

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

```
UNASSEMBLE COMMAND

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

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

EXAMPLE:
U
U 1000
U 1000 10FF
```

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

```
-u 0200

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

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

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

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

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

"We will cover that next". I replied.

The Go Command

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

```
GO COMMAND

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

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

EXAMPLE:
G
G 1000
```

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

```
-d 0200

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

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

```
-r

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

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

```
-g 0200

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

-
```

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

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

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

"15 decimal." replied Pat.

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

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

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

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

"When I get home," said Pat "the first thing I am going to do is to find my scientific calculator and see if it can do numeral conversions!"

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

"What's that?" asked Pat.

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

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

```
-d 0200

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

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

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

The Trace Command

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

```
TRACE COMMAND

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

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

```
EXAMPLE:
T
T 1000
T 1000 0C
```

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

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

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

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

```
-u 0200
```

```
0200  A9 0A     LDA #0Ah
0202  69 05     ADC #05h
0204  69 02     ADC #02h
0206  8D 0A 02  STA 020Ah
0209  00        BRK
020A  00        BRK
020B  00        BRK
```

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

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

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

```
-d 0200
```

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

"The next step is to view the registers so that we can see what they contain before the program changes them." I said and then Pat executed a Register command:

```
-r
```

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

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

```
-t 0200
```

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

```
0202  69 05     ADC #05h
```

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

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

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

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

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

```
-t
```

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

```
0204  69 02     ADC #02h
```

"I was right!" cried Pat.

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

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

```
-t

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

0206  8D 0A 02  STA 020Ah
```

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

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

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

0209  00        BRK
-d 0200

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

After we verified that the number 11 hex was copied from the 'A' register into memory location 020A hex I said "This was your first experience with programming a computer in machine language. Understanding how a computer works at the machine language level will help you to write better programs when you start learning how to program in higher level languages. The next time we meet, we will be taking one step towards those higher level languages by learning how to program in assembly language."

"I can't wait!" said Pat.

Exercises

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

2) Write a machine language program that adds the numbers 1,2,3,4,5, and 6 together and places the sum into location 0275h. Begin the program at 0200h. Run the program and verify that it works correctly (if you would like to save your program, unassemble it, highlight the unassembled program with the mouse, and then type <ctrl>c to copy it. It can then be pasted into a MathRider file or another application such as a text editor.)

Hint for this exercise:

My recommendation for how to complete exercise 2 is to use the program given in Figure 3 as a starting point. Here is the program:

```
0200 A9 0A      lda #10d
0202 69 05      adc #5d
0204 8D 08 02   sta 0208h
0207 00         brk
```

And here is how to use the monitor's Enter command to place this program into the emulator's memory one byte at a time:

```
-e 0200 a9

-e 0201 0a

-e 0202 69

-e 0203 05

-e 0204 8d
```

```
-e 0205 08

-e 0206 02

-e 0207 00
```

You can verify that the program's instruction numbers have been entered correctly using the Dump command:

```
-d 0200

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

But it is more useful to use the Unassemble command to look at these numbers because it also gives the mnemonic form of each instruction:

```
0200  A9 0A     LDA #0Ah
0202  69 05     ADC #05h
0204  8D 08 02  STA 0208h
0207  00        BRK
```

The exercise wants us to calculate 1 + 2 + 3 + 4 + 5 + 6 and a good place to start is to add 1 + 2. In order to do this the first instruction in the example program needs to load 1 into the 'A' register instead of 10. The opcode and operand for **lda #10d** are **A9** and 0A**.** The 0A is the hex equivalent for 10 decimal so if we change this 0A into a **01**, then the first instruction will load **01** into the 'A' register instead of 10.

Before we can change the 0A into **01**, we need to know what address it is at. If the A9 is at address 0200, then the 0A must be at address 0201. The Enter command can now be used to change the number:

```
-e 0201 01

-u 0201

0200  A9 01     LDA #01h
0202  69 05     ADC #05h
0204  8D 08 02  STA 0208h
0207  00        BRK
```

This is a step in the right direction. The next thing we need to do is to add 2 to the 'A' register instead of adding 5 to it. This can be done by changing the operand byte in the **ADC #05h** instruction from a **05** to a **02** using the Enter command:

```
-e 0203 02

-u 0200

0200  A9 01     LDA #01h
0202  69 02     ADC #02h
0204  8D 08 02  STA 0208h
```

To finish this program, more ADC instruction will need to be added to it using the Enter command. This will overwrite the STA instruction that is currently at the bottom of the program but that is okay because another STA instruction can be placed at the end of the program when all of the ADC instructions have been added.

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

```
;(a)
0200 A2 00      ldx #0h
0202 E8         inx
0203 E8         inx
0204 E8         inx
0205 E8         inx
0206 E8         inx
0207 00         brk

;(b)
0200 A0 00      ldy #0h
0202 C8         iny
0203 C8         iny
0204 C8         iny
0205 C8         iny
0206 C8         iny
0207 00         brk

;(c)
0200 A2 04      ldx #04h
0202 CA         dex
0203 CA         dex
0204 CA         dex
0205 CA         dex
```

```
0206 CA         dex
0207 CA         dex
0208 CA         dex
0209 00         brk


;(d)
0200 A9 00      lda #0h
0202 8D 20 02   sta 0220h
0205 EE 20 02   inc 0220h
0208 EE 20 02   inc 0220h
020B EE 20 02   inc 0220h
020E 00         brk


;(e)
0200 A2 08      ldx #8d
0202 CA         dex
0203 D0 FD      bne 0202h
0205 00         brk
```

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

```
debug
-?
-d 0000 ffff
```

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