Assembly Lines

by Roger Wagner

Everyone's Guide to Assembly Language, Part 33

This month's discussion deals with a new version of our beloved 6502 microprocessor known as the 65C02. Although the chip has just been released within the last few months and has yet to find its way into the mainstream of computers, it seems likely that we'll be hearing more about this item in the upcoming year.

Before jumping right into its new functions, though, let's first get a little background information out of the way.

The 6502 was apparently first designed by Commodore Business Machines, and, as of the present, 70 percent of its use is by Apple, Atari, and Commodore. The current manufacturers of the 6502 are Rockwell International, MOS Technology, and Synertek. As sometimes happens with these things, though, some of the key persons involved with the 6502 went to work at a new company, Western Design Center. This company, then, is the original source of the new 65C02 chip. But the story doesn't end there. Western Design Center has sold the design to at least three independent manufacturers, Rockwell International, GTE, and NCR. These companies took the initial 65C02 design, corrected initial design errors, and added their own enhancements.

The picture at this point is that each of these three companies will be marketing its own version of the 65C02. The chips are more or less the same, but the Rockwell chip has the largest instruction set.

"Largest instruction set," you ask? Yes! The new 65C02 has had the old 6502 instruction set appended with a variety of new instructions. Because the Rockwell chip appears to be a superset of all the other chips, the bulk of this article will assume that it's the chip that's being used. At the end of the article we'll describe differences among the three chips.

The Rockwell chip has a total of twelve new instructions and two new addressing modes. In addition a number of addressing modes not previously available to an instruction (such as the immediate mode for the BIT instruction) are now available. There are a total of fifty-nine actual new op-codes. The meaning of all these numbers will become clear shortly.

New Addressing Modes. Since this is one of the smaller numbers, let's start here. You'll recall from many earlier discussions that each 6502 instruction has up to six addressing modes. That number is arrived at by counting some modes as mere variations of others and not including the value (relative addressing) associated with branch instructions (BEQ, BNE, BCC, BCS, and so on) as an addressing mode here. To refresh your memory, a list of modes and variations is provided in table 1 for the LDA (load accumulator) instruction.

Indirect Addressing. The first of the two new addressing modes is quite easy to explain because it is essentially another variation of an existing mode. The new mode is *indirect* addressing. This may sound very familiar because this is similar to the instructions used to access memory locations via a zero-page pointer. Usually, though, the Y register is set to zero or some other value, which is then added to the address indicated by the zero-page pointer to determine the address of interest.

This is fine for addressing a large table of data, but many times we are interested in only one byte of memory, and must then go through the obligatory LDY #$00 to properly condition the Y register. (See entries 5 and 6 in table 1.)

The new instruction allows us to ignore the contents of the Y register and gain access to the memory location directly. This conserves two bytes of code for each reference, since the Y register does not have to be loaded. If you want to scan a block of memory, such as for a table, this instruction can still be used if you are willing to INC or DEC the zero-page pointer accordingly.

This new addressing mode is available for the instructions listed in table 2.

Instruction and Common Syntax
ADC ($12)
AND ($12)
CMP ($12)
EOR ($12)
LDA ($12)
ORA ($12)
SBC ($12)
STA ($12)

Table 2. Instructions with indirect addressing.

Indexed Absolute Indirect. The second new addressing mode has a name that was obviously not designed with easy recall in mind. Fortunately, this too is a variation on an existing theme and as such should be easy to remember. In the past, we had indexed indirect addressing. We called this mode preindexed for clarity's sake. Item 5 in table 1 is an example. Preindexing means that the contents of the X register are added to the address of the zero-page reference *before* using the sum of those numbers to determine which zero-page pair to use. For example, the instruction LDA ($22,X), where the X register held the value 4, would actually use bytes $26,27 to get the final destination address.

This differs from indirect indexed, which we refer to as postindexing. In postindexing, the value of the Y register is added *after* the base address is determined. For example, in the instruction LDA ($22),Y, where the Y register holds the value 4 and $22,23 point to location $1000, the memory location accessed would be $1004.

You'll recall also that pre- and postindexing were limited in their use of the X and Y registers. Preindexing could use only the X register and postindexing only the Y. Before you get too excited in anticipating the possibilities of the new instruction, restrain yourself: This much has not changed.

What has changed is that preindexing is no longer limited to zero-page pointers. The new mode allows any two-byte value to be used. This

Addressing Mode	Common Syntax
1. Absolute	LDA $1234
Zero Page	LDA $12
2. Immediate	LDA #$12
3. Absolute,X	LDA $1234,X
Zero Page,X	LDA $12,X
4. Absolute,Y	LDA $1234,Y
5. (Indirect,X)	LDA ($12,X)
6. (Indirect),Y	LDA ($12),Y

Table 1. Addressing modes.

means that the X register can be added to the base address of a table of memory pointers that previously could only have been located on the zero page of memory.

Addressing Mode	Common Syntax
8. Indexed Absolute Indirect	LDA ($1234,X)

For example, suppose you had a command interpreter that accepted a command value between 0 and 2. With the 6502, such an interpreter could be used in conjunction with a JMP table located on page zero, constructed as in the following example:

```
JMP DATA TABLE:
20: 80 10
22: A0 10
24: C0 11

                 1   *****************
                 2   *SAMPLE COMMAND PROCESSOR*
                 3   *****************
                 4   *
                 5          OBJ   $1000
                 6   TABLE  EQU   $22
                 7   *
1000: 20 00 40   8   ENTRY  JSR   GETCMD     ; GET VAL FROM 0 - 2
1003: AA         9          TAX              ; PUT IN X REG
1004: 7C 22 00  10   GO     JMP   (TABLE,X)  ; EXECUTE PROPER ROUTINE
                11   *
                12   ...MORE CODE HERE...
1080: EA        50   CMD1   NOP              ; FIRST ROUTINE
                51   ...MORE CODE HERE...
10A0: EA       100   CMD2   NOP              ; SECOND ROUTINE
               101   ...MORE CODE HERE...
11C0: EA       150   CMD3   NOP              ; THIRD ROUTINE
               151   ...MORE CODE HERE...
```

This is a very fast and effective technique, but for a large set of command routines it can chew up valuable zero-page memory very fast. Wouldn't it be nice if we could put the table somewhere else? With the new addressing mode you can. The table could now be put, for example, at $1200, with line 6 modified accordingly. This would free up six bytes of valuable zero-page real estate.

Table 3 shows the instructions that can use this new mode.

STOR WARES™
•Super value •Storage for 5¼" disks •Hinged lid •Easy to open •Anti-magnetic •Easy to carry •Stackable •Durable •Convenient •Walnut or black pebble finish •Optional index/dividers just $2.95 per set.
75+ Disk Storage - $9.95 ea. plus $1.25 shipping and handling
150+ Disk Storage - $14.95 ea. plus $1.50 shipping and handling
Check, Cash, Master Card, VISA or Am. Exp. No COD's.
StorWares. We put a universe of data storage at your fingertips.
StorWares,™ Inc.
1849 East 65th Street
Cleveland, Ohio 44103
(216) 881-2424
Patent Pending

Instruction and Common Syntax	
ADC	($1234,X)
AND	($1234,X)
CMP	($1234,X)
EOR	($1234,X)
LDA	($1234,X)
ORA	($1234,X)
SBC	($1234,X)
STA	($1234,X)

Table 3. Instructions with indexed absolute indirect addressing.

New "Standard" Addressing Modes. There are a few instructions that have addressing modes that are new just to them. For example, two of the most exciting ones are INC and DEC.

Previously, any uses of INC and DEC were limited to memory locations. In addition (so to speak), using the X and Y registers was the only way to maintain a simple loop counter without using a dedicated memory location. The surprise here is that INC and DEC will now work on the accumulator. This is nice because you can now maintain a counter in the accumulator, or even do fudging of flag values as they are being handled in the accumulator.

Some future assemblers may require the "somewhat usual" (if not inconvenient) use of DEC A or INC A as they seem to prefer for LSR, ASL, and other operations on the accumulator.

The BIT instruction also allows some additional addressing modes that may prove useful. Previously, the BIT instruction supported only absolute addressing. That is to say that a directly referenced memory location was used as the value against which the accumulator was operated on.

Addressing Mode	Common Syntax
Absolute	BIT $1234
Zero Page	BIT $12

This is useful for testing a memory location for a given bit pattern, but not directly suitable for testing the bit pattern of the accumulator. For many operations, this means you have to rather artificially load some memory location with the value you wanted to compare to the accumulator.

The new 65C02 supports three new addressing modes for the BIT instruction:

1. Immediate	BIT #$12
2. Absolute,X	BIT $1234,X
3. Zero Page,X	BIT $12,X

At Last, the Real Scoop! Of course, the real question lurking in everyone's mind is: "But what are the new instructions?"

The great thing about the 65C02 is that not only are many of the old instructions enhanced, there are a number of absolutely terrific new instructions—twelve, to be exact.

The new instructions are shown in table 4.

BBR	Branch on bit reset (clear)
BBS	Branch on bit set
BRA	Branch always
PHX	Push X onto stack
PHY	Push Y onto stack
PLX	Pull X from stack
PLY	Pull Y from stack
RMB	Reset (clear) memory bit
SMB	Set memory bit
STZ	Store zero
TRB	Test and reset (clear) bit
TSB	Test and set bit

Table 4. New instructions in the 65C02.

So what exactly do these instructions do? Well, let's examine some of the easy ones first: PHX, PHY, PLX, and PLY.

Commands for pushing a byte onto the stack and pulling a byte off the stack exist for the accumulator but not for the X and Y registers in the 6502. One of the more common uses for the stack is to save all the registers prior to going into a routine so that everything can be restored just prior to exiting. Ordinarily, to save the A, X, and Y registers, we'd have to do something like this:

```
ENTRY     PHA     ; SAVE A
          TXA     ; PUT X IN A
```

```
            PHA        ; SAVE IT
            TYA        ; PUT Y IN A
            PHA        ; SAVE IT
WORK        NOP        ; YOUR PROGRAM HERE
DONE        PLA        ; GET Y
            TAY        ; PUT IT BACK
            PLA        ; GET X
            TAX        ; PUT IT BACK
            PLA        ; GET A
EXIT        RTS
```

The problem is complicated even further in programs like the character generator listed in the April issue. There we had to refer to the original value of the accumulator several times and this interfered with any simple way to push all the register data onto the stack.

With the new 65C02, this could all be resolved with the following:

```
ENTRY       PHX        ; SAVE X
            PHY        ; SAVE Y
            PHA        ; SAVE A, BUT LEAVE IT ON TOP
WORK        NOP        ; THE PROGRAM HERE
DONE        PLA        ; GET A
            PLY        ; GET Y
            PLX        ; GET X
EXIT        RTS
```

All four are one-byte commands, addressing only the indicated register.

BRA (branch always) is one of those instructions that will thrill writers of relocatable code. One of the techniques for writing code that is location-independent involves the use of a forced branch instruction, such as:

```
CLC                     ; CLEAR CARRY
BCC LABEL               ; ALWAYS
```

Unfortunately, this means we must force some flag of the status register, which may not be convenient at the time. In addition, the process takes up an extra byte on most occasions.

Branch always alleviates both these problems by always branching to the desired address, subject of course to the usual limitations of plus or minus 128 bytes as the maximum branching distance.

It is worth mentioning, in the interest of programming style, that many people indiscriminately use a JMP to go back to the top of a loop when a branch instruction would do the trick; this only adds one more limitation to the final code in the process. Hopefully, this new branch instruction will encourage people to make their code more location-independent. BRA, like the rest of the branch instructions on the 6502, uses only relative addressing.

STZ (store zero) is used for zeroing out memory bytes without changing the contents of any of the registers.

Many times it is necessary to set a number of internal program registers to 0 before proceeding with the routine. This is especially needed in mathematical routines such as multiplication and division.

Ordinarily, this is done by loading the accumulator with 0 and then storing that value in the appropriate memory locations. This is easy to do when you have to load the A, X, or Y registers with 0 anyway. The problem is that on occasion the *only* reason one of the registers is loaded with 0 is because of the need to zero a memory location.

Store zero allows us to zero out any memory byte without altering current register contents. Not all of the addressing modes usually available to STA, STX, or STY instructions are available with STZ, though. Table 5 shows what modes are available.

Addressing Mode	Common Syntax
Absolute	STZ $1234
Zero Page	STZ $12
Absolute,X	STZ $1234,X
Zero Page,X	STZ $12,X

Table 5. STZ addressing modes.

SMB and RMB (set/reset memory bit) will allow you to set or clear a given bit of a byte in memory. Previously, this would have required three separate instructions to achieve the same result. For example:

```
LDA MEMORY              ; LOAD VALUE FROM MEMORY
AND #$7F                ; %0111 1111 IS PATTERN
                        ; NEEDED TO CLR BIT 7
STA MEMORY              ; PUT IT BACK
```

With the new instruction, we can accomplish the same thing with:

```
    RMB7 MEMORY              ; RESET (CLR) BIT 7 OF MEMORY
```

or set the bit again with:

```
    SMB7 MEMORY              ; SET BIT 7 OF MEMORY
```

Two interesting things to note here. The first is that for some reason they use the term *reset* instead of clear to indicate the zeroing of a given bit.

The second item is that we now have four-character instruction codes (mnemonics), the last character being the number of the bit being acted on. What problems this may cause in some assemblers remains to be seen, but this new species of instruction seems to have arrived.

These instructions are limited to zero-page addressing only.

BBS and BBR (branch on bit set/reset) are two new branch instructions that make it possible to test any bit of a zero-page location and then branch, depending on its condition. This instruction will be very useful for testing flags in programs that need to pack flag-type data into as few bytes as possible. By transferring I/O device registers to zero page, it is also possible to test bits in these registers directly for status-bit conditions.

These instructions are very similar in appearance and use to the bit set and reset instructions just discussed. They, too, use four-character mnemonics. The difference is, of course, that we are testing bit status, rather than changing it. These are three-byte instructions, the first byte being the op-code, the second being the byte to test, and the third a relative branch value. In assembly, these commands will actually require two labels!

One of the first applications is the testing of whether a number is odd or even. Previously, this had to be done with an LSR or ROR instruction, followed by a test of the carry flag, such as:

```
LDA MEMORY      ; LOAD A WITH VALUE
LSR             ; SHIFT BIT 0 INTO CARRY
BCS ODD         ; SET IF ODD
BCC EVEN        ; CLR IF EVEN
```

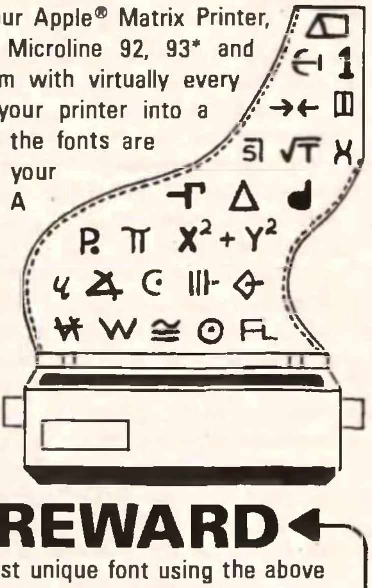
The equivalent can now be done without affecting the carry flag or

Just Released

Font DownLoader

Expand the capacity of your printer hundreds of times

Load custom fonts into your Apple® Matrix Printer, Prowriter® 8510A, OKI® Microline 92, 93* and Epson® FX* and use them with virtually every word processor to turn your printer into a **custom typesetter.** After the fonts are loaded, they will stay in your printer until it's turned off. A font editor is also provided to allow you to create your own graphics, text, foreign language letters, math and electronics symbols to load into your printer. **Available in 30 days*



Disk Software only $39.00

$100 REWARD

Submit the best or most unique font using the above software and we will make you $100 richer. Other prizes for the first 25 runners up.

Dealer and Distributor Inquiries Invited

Designed by RAK-Ware

micro ware

1342 B Rt. 23, Butler, NJ 07405
201-838-9027

the accumulator:

```
BBR0 MEMORY, EVEN      ; BRANCH IF BIT 7 = 0 = EVEN
BBS0 MEMORY, ODD       ; BRANCH IF BIT 7 = 1 = ODD
```

This could be useful also in creating drivers for the new Apple IIe eighty-column extended memory board since this card uses one bank of memory or the other for the text screen, depending on whether the screen column position is odd or even.

TSB and **TRB** (test and set/reset bit) are the most complex of the new instructions. These instructions are rather like a combination of the **BIT** and **AND/ORA** instructions of the 6502.

The instructions seem primarily designed for controlling I/O devices but may have other interesting applications as things develop.

The action of these two instructions is to use a mask stored in the accumulator to condition a memory location. The mask in the accumulator is unaltered, but the Z flag of the status register is conditioned based on the memory contents prior to the operation.

For example, to set both bits 0 and 7 of a memory location, we could use the following set of instructions:

```
LDA #$81        ;%1000 0001 = MASK PATTERN
TSB MEM1        ; SET BITS 0,7 OF MEMORY
BNE PRSET       ; ONE OF THESE WAS 'ON' ALREADY
BEQ PRCLR       ; NEITHER OF THESE WAS 'ON' ALREADY
```

This would clear the bits:

```
LDA #$81        ; %1000 0001 = MASK PATTERN
TRB MEM2        ; CLR BIT 0,7 OF MEMORY
BNE PRSET       ; ONE OF THESE WAS 'ON' ALREADY
BEQ PRCLR       ; NEITHER OF THESE WAS 'ON' ALREADY
```

These instructions use only absolute and zero-page addressing.

Other Differences. There are a number of other differences in the chips, most notably the power consumption. The power use of the 65C02 is one-tenth that of the 6502, so the chip runs considerably cooler. The lower-power requirement opens new possibilities for portable computers and terminals.

One point of interest is that the old 6502 indirect jump problem has been fixed. If you're not aware of it, the 6502 has a well-documented problem with indirect jumps that use a pair of bytes that straddle a page boundary.

For example, consider these three instructions:

Instruction	Pointers Wanted	Pointers Used
JMP ($36)	$36,37	$36,37
JMP ($380)	$380,381	$380,381
JMP ($3FF)	$3FF,400	$3FF,300

Notice that, in the third instance, the pointers used are *not* those anticipated. This is because the high byte of the pointer address is not properly incremented by the standard 6502.

This problem has been fixed on the 65C02. The only possible problem here is "clever" protection schemes that use this bug to throw off people trying to decode the protection method. Otherwise, this should not present any problems to existing software.

Are there any problems to be anticipated? In theory, no. The new 65C02 is compatible pin for pin with the old one, and also upwardly compatible in terms of software. Software for the Apple, PET, Atari, or other 6502-based microcomputers *should* work without problems with the new chip. Are there any exceptions? Unfortunately, yes.

The first big problem concerns internal microprocessor timing on the Apple II and II Plus computers. The Apple II and II Plus do not handle the microprocessor clock cycles in the same way the IIe does. On the surface, the 65C02 should directly replace the 6502; however, because the 65C02 is a faster chip, data is not available for as long, and bits can get lost. What this means for now is that the 65C02 can be used only on the Apple IIe and Apple III machines. None of the manufacturers at this time produce a chip that works on the Apple II or II Plus. It can be expected, though, that revisions will be made in the near future that will allow the 65C02 to be implemented in the older machines.

There is also a possibility of problems with some existing software. A small percentage of software may be using undocumented bugs or "features" of the old 6502 chip, and these might not function as anticipated.

For example, a reasonable question might be, "Where did all the new op-codes come from? After all, wasn't the chip full?" To answer this, consider how the instructions we use now are structured. The 6502 operates by scanning memory and performing specific operations based on the values that it finds in each memory location. You would then expect a total of 256 possible op-codes. As it happens, all 256 possible values are not used. It is this group of unused op-codes that allows for the new instructions and also creates the possibility of a small percentage of difficulties with existing programs.

Although rarely documented, the previously "unused" values will cause certain things to happen, much the same way that a legal value would. For instance, the code $FF on a 6502 is labeled as an alternate NOP. This is one of the codes that have been converted to a new function in the 65C02, namely **BBS7** (branch on bit 7 set).

There are other unused codes, though, that have combination effects, usually of little use, such as loading the accumulator and decrementing a register at the same time. Their main application is similar to the indirect jump problem: creating code that cannot be casually interpreted. If these instructions have been used in existing software, problems could arise with the 65C02.

With such difficulties, then, why bother to substitute the new 65C02 into an existing Apple? The answers are varied.

First of all, the 65C02 is likely to appear in upcoming releases of existing computers (in a new release of the IIe, perhaps?), and as such you can experiment now with the newest version of this versatile device.

Second, there will likely be specific applications where the advantages of the chip will make it worth supplying with the software, since the disadvantages are practically nonexistent for the Apple IIe and Apple III. Code rewritten to take advantage of the new instructions can be expected to be 10 to 15 percent smaller and run proportionally faster. In certain applications, even greater improvements could be expected.

At this writing, the Rockwell chip seems to have the largest set of instructions of the three varieties available. The GTE and NCR chips lack the bit manipulation instructions but are otherwise identical.

As to assemblers supporting the instructions, the current version of *Merlin* supports all the new op-codes in both the assembly and *Sourceror* portions of the product. S-C Software is offering a 65C02 cross assembler to registered owners of the *S-C Assembler* at a reduced rate. Hayden will be offering an update to *ORCA* to support the GTE version of the chip. Any assembler that supports macro capabilities should be able to be used immediately by defining the proper hex codes.

If readers have any unique problems or questions about the 65C02, send them to *Softalk*; if possible, the answers will be published in a subsequent issue.

This installment marks the last in this series. I want to thank the many readers of this column over the last several years for their enthusiastic support and valuable suggestions. I have always believed that the human element to this industry, and in fact any endeavor, is the truly rewarding part. I would also like to thank *Softalk* for giving me the opportunity to share the excitement of programming with its readers, and also thank Brian Britt for his help in researching this month's article.

For better or worse, though, you're not likely to be completely rid of me. There are rumors of other columns and projects, and I look forward to being a small part of the *Softalk* family for some years to come.

It was nearly three years ago that Roger Wagner's Assembly Lines began appearing in Softalk; *the magazine was only one month old. In that first year, Wagner's column elicited more mail from* Softalk*'s readers than any other feature, and properly so: It was the first time assembly language had been explained from step one. In fact, in his first column, Wagner didn't even introduce a command.*

With this issue, Roger Wagner's Assembly Lines ends. The first year's columns plus appendixes and revisions have been available for some time in Assembly Lines: The Book. *Volume 2, covering the rest of the columns, will be released shortly by Softalk Books.*

Roger Wagner will not fade away. He's planning occasional feature articles for Softalk *and he's promised to remain available to answer questions from* Softalk *readers.*

Next month, a new assembly language tutorial will begin, back at step one. The new column will be written by Jock Root, a frequent guest reviewer in Softalk *and the author of "IInd Grade Chats: A Custom Menu Generator" in May.*