6502 Hacks

by Mark S. Ackerman

Computers and controllers using the 6502 CPU often demand efficient use of both processing time and memory space. With an address space of only 64K, data and code space efficiency can be critical. Moreover, though advancements in these 6502 systems have allowed the use of high-level languages, there will always be a need for fast subroutines and tight programs.

This article is a result of writing code for an Atari VCS 2600 game unit that had only 128 bytes of RAM and 8K of ROM. Because of the limited graphics hardware, all processing was in real time with cycles counted. Several of the hacks and tips presented here are also transferable to other microprocessors. Many of the tips are appropriate to any true real-time system—that is, any system where real-time is measured in very small fractions of a second. My 6502 experience has been helpful in finding ways to crunch memory requirements and timing on the Intel 8086 and 80286.

In short, this article is a collection of my favorite hacks for the 6502. A fair amount of 6502 code is included, so the next section gives a brief introduction to the 6502. If you've programmed the 6502 extensively, you've probably been forced to memorize the instruction set in hex, so perhaps you should skip ahead.

The 6502

The 6502 has only five registers. The instruction pointer (*IP*) points to the next instruction, as on most computers, and is not directly setable by the user. The accumulator (*A*) is the main register, and it is the only register with full arithmetic-logic unit (ALU) functionality. The two index registers, *X* and *Y*, are used for indexed addressing. The stack pointer (*S*) points to the top of the stack.

Mark S. Ackerman, 24 Chatham St., Cambridge, MA 02139. Mark is a computer researcher at the Massachusetts Institute of Technology, specializing in graphics environments.

There will always be a need for fast subroutines and tight programs.

The 6502 also has a nonregular instruction set. For example, the *S* register can be set only from the *X* register using the *TSX* (transfer *S* register to *X* register) or *TXS* (yep—transfer *X* to *S*). Moreover, if you want to transfer a value between the *X* and *Y* registers, you must transfer through the accumulator or through memory. For example, a typical sequence might be:

```
TXA     ;tfr X to accumulator
TAY     ;tfr accum to Y
```

The 6502 also has several flags. The important ones are

- the negative flag (or minus flag), which is set when loading or doing ALU functions
- the zero flag, which is set in similar situations
- the carry flag, which is set in arithmetic operations
- the overflow flag, which is also set in arithmetic operations

There are also flags for enabling interrupts and for decimal mode.

Basic Hacks

One of the simplest ways to save code space is at initialization time. The following code, for example, might be used to initialize a few variables:

```
LDA  #0       ;load accumulator
                            with 0
STA  FIRST    ;store accum in FIRST
LDA  #2
STA  SECOND
LDA  #1
STA  THIRD
LDA  #0
STA  FOURTH
LDA  #5
STA  FIFTH
```

This requires 20 bytes (2 per instruction) and a minimum of 25 cycles (2 per load immediate and 3 per store). The following would be cheaper:

```
LDX  #0       ;X register=0
STX  FIRST    ;store X in FIRST
STX  FOURTH
INX           ;inc X by 1 (X now=1)
STX  THIRD
INX           ;X now=2
STX  SECOND
LDX  #5       ;X now=5
STX  FIFTH
```

This takes two cycles and 4 bytes less; you save 1 byte each per *INX* instruction. An even cheaper result can be obtained by:

```
LDA  #5       ;accumulator=5
STA  FIFTH    ;store accum in FIFTH
LSR  A        ;shift right --
              ; (acc now=2)
STA  SECOND
LSR  A        ;acc now=1
STA  THIRD
LSR  A        ;acc now=0
STA  FIRST
STA  FOURTH
```

This last example uses the same number of cycles, 23, but costs only 15 bytes—a reduction of 25 percent over the first example. You might object that this is not "clean code." Without adequate documentation, it may be less than clear, but it does save bytes and cycles.

This example also demonstrates a reduction principle: general reduction algorithms, such as using the index registers to increment instead of loading the accumulator with immediate values, can produce significant savings. The best savings, however, require a sharp eye for special situations.

Incidentally, if you use a loop during initialization, remember that the counter register contains −1 or 0 at the end of the loop. You can use this by-product for further savings:

```
   LDA   #$C0
   LDX   #7
LP STA   TABLE2,X  ;put acc at TABLE2
                   ; plus offset in X
   DEX             ;decrement X
   BPL   LP        ;loop while X is
                   ; positive (i.e., > 0)
   INX             ;at end of loop,
                   ; increment X
   STX   ZERO ;store X (=0) in ZERO
```

Zero-Page Savings
The 6502's first 256 bytes of memory, zero page, have a unique property. Reads from and writes to zero page, including indexed I/O using only the *X* register, save a cycle and a byte. Frequently used variables, or memory registers, should be kept in this portion of RAM.

It is critical when addressing this memory to use the *X* register. Using the *Y* register for indexed addressing such as this:

```
LDA   ZEROPAGE,Y
```

is actually an absolute addressed instruction—that is, the 6502 ignores whether the *ZEROPAGE* location is in zero page or not. The *Y*-indexed instruction uses an additional cycle for the fetch and a byte for the page address.

Using Left-Over Registers
Use all the registers. The index registers should be used for intermediate results. The following nonsense example assumes that the stack is at *$FF*:

```
     LDX   LOOP_COUNT
LOOP TXS         ;store loop ct,
                 ; freeing X
     LDA   WHATEVER
     LSR   A
     TAX         ;store acc/2
     LSR   A
     LSR   A
     AND   #07   ;get low 3 bits
     TAY         ;get offset of TABLE
     TXA         ;restore acc
     LDX   TABLE,Y    ;new index for X
     ORA   $80   ;OR $80 to low 3 bits
     STA   STORE,X    ;put at STORE plus
                      ; offset from TABLE+Y
     TSX         ;restore loop counter
     DEX         ;dec loop counter
     BPL   LOOP  ;loop if counter >=0
     TXS         ;restore stack ptr
```

Note that the *TXS* instruction can be used only if interrupts have been disabled. A temporary zero-page variable could have been used to replace the *TXS* and *TSX* at a cost of two cycles per loop execution and 1 byte.

Stack-Related Savings
It is often cheaper to place values onto the stack than to store them to temporary variables. A push (*PHA*) and pull (*PLA*) take seven cycles and 2 bytes. A store to zero-page memory with a following load takes six cycles and 4 bytes; a store to other memory takes eight cycles and 6 bytes.

When doing a substantial amount of I/O to temporary variables, it may make sense to actually reposition the stack pointer. This works only with page 1 variables.

Flags and the Bit Instruction
Careful use of bit flags can also save bytes and cycles substantially. The *BIT* instruction does a nondestructive test of a byte in memory. Bit 7 (the high-order bit) of the byte is placed in the negative flag, and bit 6 is placed in the overflow flag. No registers are affected.

For this reason, if RAM is limited, the two high-order bits of a byte are extremely valuable for Booleans. In fact, *bit7* is valuable because it also sets the negative flag upon loading:

```
   LDA   WORD   ;load acc with WORD
   BPL   NOT_SET    ;go if + (bit7=0)
IS_SET AND #$0F
   TAX
```

In addition, the carry flag, which is set or cleared in shift operations, can be used to store a flag value temporarily. In this case, bits 7 and 0 are the most valuable.

If RAM is not limited, then a single Boolean in *bit0* allows the use of the zero flag upon loading. The *BIT* instruction still cannot be used profitably unless the Boolean is in bit 7 or 6, however.

More Advanced Hacking
There are two ways to depend on preexisting conditions. The first is to assume that the carry bit is either set or not set as needed. In the 6502, the carry bit signifies just that: a bit is being set to indicate the carry. To add two 16-bit numbers, then:

```
CLC
LDA   FIRST_VAR_FIRST_8
ADC   SECOND_VAR_FIRST_8
LDA   FIRST_VAR_SECOND_8
ADC   SECOND_VAR_SECOND_8
```

If you know the condition of the carry, such as in the following sequence of instructions:

```
LDA   FIRST
CMP   #$18    ;comp acc to $18
BCS   BRANCH1   ;go if acc >= $18
LDA   VAR1
ADC   VAR2
```

then a *CLC* can be omitted. Why? Because the branch was on a carry set condition, the only way into the addition would be if the carry were clear. In a similar manner, you can assume that there is no carry from a previous addition. For example, if VAR3 never exceeded 16 and VAR4 never exceeded 5, then the carry will never set. So the sequence:

```
CLC
LDA   VAR3
ADC   VAR4
STA   TEMP1
LDA   NEW1
ADC   NEW2
```

can be used. The *CLC* for the second addition can be omitted because the carry will not be set. This, however, reduces the robustness of the code.

6502 HACKS

(continued from page 25)

Removing clear carries or set carries—used for subtractions—can save many bytes. You may need to use some ingenuity. If the carry is set, for example:

```
LDA   TEMP
SEC   #$FF     ;subtract −1
```

you may want to add 1 to *TEMP* by subtracting −1.

Cheaper Branching

The second way to use preexisting conditions is with branching. If you know that a flag will be in a certain condition, the appropriate branch instruction can be used for an unconditional jump. This will save a byte but no cycles in the 6502; the unconditional *JMP* instruction takes 3 bytes whereas a conditional branch takes 2 bytes. Both take three cycles when the branch is made. (A conditional branch in which no branch is executed takes only two cycles.) For example, if the carry is set (perhaps from a *BIT* instruction as below or from a subtraction), then the code:

```
   BCC   JUMP_LOC
NEXT_LOC
```

forces an unconditional branch for the savings of a byte over a *JMP*.

Interestingly, the instruction sequence for a Boolean in *bit0*:

```
LDA   YOUR_FLAG
AND   #1                    ;get bit0
BNE   TRUE_SETTING          ;branch on 1
FALSE_SETTING
```

can be replaced by:

```
   LDA   YOUR_FLAG
   LSR   A         ;shift bit0 into carry
   BCS   TRUE_SETTING          ;go if set
FALSE_SETTING
```

at a savings of a byte. This is especially useful for testing several bit flags in a single byte. It also nicely sets the carry bit for unconditional branching for both branches of an *if . . . else* structure.

```
   LDA   YOUR_FLAG
   LSR   A
   BCS   TRUE_SETTING
FALSE_SETTING             ;carry is clear
   some code
   BCC   END_IF
TRUE_SETTING
   some code
END_IF
```

Table-Driven Code

You can make very large savings if you can replace code with preset data tables. Instead of attempting to compute divide-by-17s or sines, for example, it may be possible to have a table of the results for all expected values. For example, instead of computing *MOD7*, if the variable will never exceed 32, it will be far, far cheaper to have a table:

```
MOD7  DS 0,1,2,3,4,5,6
      DS 0,1,2,3 . . . etc.
```

Because many of these tables can be compressed or merged with other tables (as discussed later), the cost in bytes is reasonable. This method is certainly faster.

In a similar manner, decision tables, game-play paths, or timing decisions can often be decided prior to compilation rather than during execution. In games, it is often best to store the delta xs and delta ys instead of trying to compute sine wave patterns on the fly.

In many cases, you can use tables to speed up operations that are repeated often. If, for example, it is necessary to increment only the bottom nybble of a word, a normal addition cannot be used because the carry will ruin the top nybble. You could write:

```
LDA   WORD
AND   #$F0     ;get high nybble
STA   TEMP     ;store temporarily
LDA   WORD
AND   #$0F     ;get the low nybble
CLC           ;assume worst: clr carry
ADC   #1       ;add 1
AND   #$0F     ;watch for wrap
ORA   TEMP     ;OR in high nybble
STA   WORD     ;store back out
```

This costs 24 cycles and 19 bytes. If you had a table:

```
NEXTINC DS 1,1,1,1,1,1,1,1
        DS   1,1,1,1,1,1,1,−15
```

the cost could be reduced to 19 cycles and 13 bytes:

```
LDA   WORD
AND   #$0F     ;get current low nybble
TAY             ;index into NEXTINC
LDA   WORD
CLC             ;might not be needed
ADC   NEXTINC,Y          ;add, indexed
                ; by current value
STA   WORD
```

If this calculation were done in many locations in the program, the table would quickly become much cheaper than the calculation. Incidentally, the add instruction could be replaced with a subtract (or even a logical OR) instruction. Your choice of which instruction to use might depend on what table you had lying around!

Unrolling Loops

One large trade-off between time and space is in unrolling loops. The loop:

```
     LDA  #1     ;outside the loop
     LDX  #4
LOOP STA LOC,X
     DEX
     BPL  LOOP
```

can be changed to:

```
LDA  #1
STA  LOC
STA  LOC+1
STA  LOC+2
STA  LOC+3
STA  LOC+4
```

This is more costly in terms of bytes (12 bytes vs. 9 bytes), but it is far faster (17 cycles vs. 48 cycles). (The loop overhead takes 4*(2+3) + 1*(2+2).) It is often surprising how much time can be saved by unrolling simple loops.

It is also possible to combine loops, even of different sizes, saving the costly loop overhead:

```
   LDA   PICKUP+7
    ;move PICKUP's contents to TABLE
   STA   TABLE+7
   LDA   PICKUP+6
   STA   TABLE+6
   LDX   #5
   LDY   #$80
LOOP2 LDA PICKUP,X       ;continue the
                        ; move with loop
   STA   TABLE,X
```

(continued from page 26)

```
    STY   TAB2,X          ;set TAB2 to $80
    DEX
    BPL   LOOP2
```

Chaining Subroutines
One of the simplest ways to save bytes is to use subroutines for common code. This requires the time cost of the *JSR* (six cycles) and the *RTS* (six cycles), however.

One way to save in subroutines is to create multiple-entry subroutines. That is, if two subroutines share a common ending, do not put that common ending in another subroutine. Instead, create a single subroutine. In a "pure" multiple-entry subroutine, you fall through all sections of code until the return statement. You can also jump around the noncommon code:

```
FIRST_ENTRY
    first section of code
    JMP   END_SUB         ;also try BCC
          ; or the like
SECOND_ENTRY
    second section of code
END_SUB
    final processing
    RTS
```

The calling routines can call either *FIRST_ENTRY* or *SECOND_ENTRY* with their different processing. Both subroutine sections will exit from the same *RTS*, however.

Finding the Extra Microprocessor Cycle
Sometimes, in coding for real-time processing, you may need to kill an extra cycle or two. The 6502 has a two-cycle *NOP* (no operation) instruction. What about a single cycle? The 6502 has no single cycle *NOP*.

Of course, sometimes a single cycle isn't needed—only an odd number of cycles. You can get seven cycles by a combination of push and pull stack operations; five cycles can be bought by doing a *NOP* and a load from zero-page memory.

Single cycles can be gained only through other operations. One such operation is the absolute addressed load using an index register. Normally this instruction (*LDA ADDRESS,X* or *LDA ADDRESS,Y*) takes four cycles. If there is a page boundary crossing (say that *ADDRESS* is at *C0F0* and the *X* register is 18), however, then the instruction takes five cycles. To gain the extra cycle, the table can be placed to force a page boundary crossing.

Occasionally you can use hardware memory mapping to the same effect. In the Atari VCS, for example, page 1 memory and page 0 memory were mapped together. Therefore, if *ZEROADD* were at *0065*, it could also be found at *0165*. A zero-page fetch costs three cycles, but the same fetch from page 1 memory costs four cycles.

It is also possible to branch to the next instruction depending on a flag. This kills either two cycles (for a nonexecuted branch) or three cycles (for a branch taken). This is occasionally useful for synchronizing *if . . . then . . . else* code.

Savings by Stepping Back
Perhaps the greatest savings can be had by proper planning. Putting two flags in bits 7 and 6 or in bits 0 and 1 makes more sense than putting them in bits 3 and 5. Often it is necessary to make simple changes in the midst of programming in order to crunch code. This type of planning comes with experience.

Stepping back from the actual programming always helps. For example, you may have converted one data structure to another through easily understandable transformations, perhaps with intermediate data structures. If you are used to high-level languages in which this is encouraged, your assembler code will reflect it. Unfortunately, this type of elegance often turns out to be costly. The type of elegance that will benefit you in crunching will be the elegance of simple algorithms—almost always single-pass algorithms—that do not require special cases. Special cases cost.

Another type of planning that is often helpful is determining when program actions will occur. It may not be necessary to have all program functionality present at the same time. In a game, where time is critical, the x,y positions, for example, do not need to be updated every 1/60 second be-

6502 HACKS
(continued from page 29)

cause the human eye does not demand that. In business software, where space is more critical, you can overlay code.

Is It Really Necessary?
In addition, there is always a time to ask yourself, is that feature really necessary?

In a time-sharing scheme I was working on, for example, it was necessary to determine three-way time sharing. The ideal would have been to have a scheduling sequence such as:

```
1 2 3   1 2 3   1 2 3   1 2 3
```

and so on, with each of the three actions (placing a graphics sprite on the display) getting one-third of the time. This order was needed rarely, however, the usual case being either two-way time sharing or no time sharing. Divides-by-3 are extremely expensive on the 6502, so I came up with this alternative:

```
1 2 3 1   1 2 3 2   1 2 3 3   1 2 3 x
```

where *x* was a skipped slice. It turned out that the result was not significantly distinguishable, which saved many bytes and much time. (This could be implemented by *AND*ing a timer with *$07* and using a lookup from a table.) This approach transformed a difficult computation into a divide-by-4 problem.

All of this is not to say that design can be separated from coding. The inspired moment of coding often finds the necessary time and bytes when all the planning weeks (or even months) ago failed. It takes patience and skill to notice that *TABLE2* can be created by taking the *TABLE1* entry, exclusive *OR*-ing *$7F*, and adding 5. It takes the same patience and skill to rip out a section of code and rearrange it to get the same overt behavior.

Killer Hacks
When all else fails, there are always a few tricks left. The following hacks are not for novices, though. These methods squeeze the final cycles and bytes from your program. They make debugging your code nearly impossible, and you might as well forget about maintaining the code later.

Please don your safety goggles.

Chaining Branches
One ugly way to reduce the number of bytes of code is to chain branches. As mentioned earlier, the 6502 uses 2 bytes for a relative branch instruction and 3 bytes for an absolute jump instruction. Unfortunately, the relative byte instruction can address a space of only 127 bytes forward or backward. Therefore, the unconditional *JMP* instruction is often used, even for implementing *if . . . then . . . else* structures. It is possible to convert these *JMP*s to conditional branches, saving bytes.

If a condition is known, such as the carry bit being set or the overflow flag being clear, it is possible to branch to another branch. By chaining branches, it is possible to have conditional branching of more than 127 bytes distance. This is recommended only when it is important to conserve space at the cost of execution time—two branches have to be executed—and maintainability.

Self-Modifying Code
Self-modifying code can be used

profitably in critically real-time routines when literally every cycle counts. Instead of loading a loop counter or some such from a zero-page variable, you can just change the *LDX* immediate instruction on the fly. This saves a cycle.

In addition, instead of performing a load or store indirect indexed, which uses 2 bytes of zero-page RAM:

```
LDA   (INDIRECT),Y
```

you can modify the destination address on the fly and perform an indexed absolute load or store to save a cycle:

```
LDA   ADDRESS,Y
```

This, of course, will have you barred permanently from any MIS employment for the rest of your life. Actually, I could not use this on the VCS because of the lack of RAM, so I am not familiar with its difficulties. I have been told, however, that with adequate documentation, this sort of treachery can be maintained.

Use of the NMI Interrupt

Another nasty trick to save bytes is the use of the break (perform interrupt) instruction. The *JSR* (call subroutine) instruction requires 3 bytes per call, whereas the *BRK* instruction requires only a single byte. This method, however, requires that you not be expecting nonmaskable interrupts.

To use this method, you must set the *NMI* vector to the address of the most frequently used subroutine. The *BRK* instruction can then be used to call the routine. *BRK*, however, not only places the return address onto the stack but also pushes the flag byte onto the stack. If you do not return information in the flags, you can return from the interrupt with an *RTI*. If you need the flags that were set in the subroutine, however, return with:

```
PLA   ;pop caller's flags
RTS   ;return normally
```

Unfortunately, the *BRK* instruction takes seven cycles instead of the *JSR*'s six. If a *PLA* is needed, that will also add four cycles. After you set the *NMI* vector (at a cost of 2 bytes), however, each call will save 2 bytes.

Overlapping Code and Data

You can squeeze data table space in two ways. The first sounds more difficult to maintain but actually turns out to be easier in practice. This is to find the appropriate data table values in your code space. For example, you might need the following flag table:

```
TABLE  $80,00
```

If you will be testing only *bit7*, however, the following table would also do:

```
TABLE  $A9,00
```

This just happens to be a *LDA #0* (load immediate of 0) instruction. Finding the appropriate code in your program:

```
TABLE LDA #0
```

also eliminates the 2 bytes from your data space. Although you might want to comment this table extensively to remind yourself of what you did, the only real problem you will have is to find the table again. Note that this technique generally works only with small tables.

The second method is much more effective in finding bytes. If you have four data tables:

6502 HACKS
(continued from page 31)

```
TABLE1 DS  0,1,2,3,0,0,0
       DS  $80,3,2,1,0
TABLE2 DS  3,2,1,0
TABLE3 DS  0,0,0
TABLE4 DS  1,0,6
```

they can be profitably reorganized as:

```
TABLE1 DS  0,1,2,3
TABLE3 DS  0,0,0
       DS  $80
TABLE2 DS  3,2
TABLE4 DS  1,0,6
```

Confused? *TABLE3* and *TABLE2* are now completely contained in *TABLE1*. *TABLE1* also extends throughout most (but not all) of *TABLE4*. Note that this has saved 9 bytes from the original 22 bytes.

Often the program can be altered slightly to increase this type of savings. Using the nybble-increment example from the section on table-driven code, the choice of instruction used might also depend on what tables are available for merging. It is also possible to find the table backward within another table: your indexing must then proceed backward also, decrementing instead of incrementing.

Similarly huge savings are almost always possible. If you have a bug and one of the tables must be changed, however, it will be extremely difficult to separate the original data tables without adequate documentation. I have always used this technique last.

Code as Data

There are particular instances—especially when there is absolutely no room left—when code can substitute completely for data tables. This is especially effective for simulating random movements, such as for a self-play mode (called the attract mode in games). Code for the 6502 tends to be somewhat heavy toward having *bit7* set, but otherwise it can create effective random tables. It is often necessary to try many code sections, however, for the desired effect in the software action.

The Endless Trade-Off

The crunching process revolves around the standard trade-off of time vs. space. Even a simple change, such as removing redundant code, requires this trade-off. Subroutines have extra overhead and slow processing down. In-line code, as with macros, can greatly speed up critical processing but at the cost of an enormous code space.

Many of the hacks described here require this trade-off. Most are tricks that sacrifice one for the other. The other hacks all have the added cost of reduced maintainability or increased programmer effort.

When is it worth using these hacks? If you're writing Pascal code on the Macintosh or C code on the PC, they cannot help much in removing 30K from your 200K program. But if you're in a situation in which you need a fast interrupt routine, these techniques can help on any machine. They can also be useful if you need to reduce the code space for a desk accessory or a similar routine or if you're just the sort of person who gets excited by realizing that a flag can be reused.

I'd like to thank the folks at General Computer for all their help.

DDJ

Vote for your favorite feature/article.
Circle Reader Service **No. 3.**