Self Modifying Code#

General Information

Author: various from Usenet
Assembler: generic
Published: Usenet
Download: from comp.sys.apple2.programmer

I've seen code that does this:

```
 LDA #<data1  ;get low address and modify code below
 sta load + 1
 lda ##data1    ;get high address and modify code below
 STA load+2
load    LDA $ABCD
;table of addresses
data1
data2
data3
```

---

what other self-modifying code schemes exist?

I was thinking about this last night: non modifying version:

```
do stuff
    lda wait                ;wait is a flag, 1 = make the program wait for
open apple to be pressed
    beq nowait
    jsr wait4OpenApple  ;wait = 1
nowait RTS
```

modifying version:

```
do stuff
modify jsr wait4openApple    ;this routine waits for the open apple
button to
                                                        ;be pressed, then just
RTS's
```

Include a flag, "wait" that is checked at the beginning of the program:

```
begin program
LDA WAIT
BNE yeswait (wait = 1, meaning code above is ok as it is)

;here, wait = 0, meaning we do NOT want to JSR the Wait4OpenApple code

lda #$60    ;rem OPcode for RTS
sta modify  ;this changes the JSR at label MODIFY to an RTS

yeswait  (rest of program)
```

thoughts, comments, or other self modifying code schemes?

Rich

---

aiiad...@gmail.com wrote:

> thoughts, comments, or other self modifying code schemes?

Self-modifying code can be difficult to debug. If you need it for speed in a critical section of code, by all means use it, but I wouldn't make a habit of it unless you're trying to be deliberately obscure.

I'm using it in some of the graphics code because STA absolute,Y is 1 cycle faster than STA (dp),Y. It takes time to do the LDA/STA to set up the address, but since I'm repeating the operation hundreds or thousands of times it's a net win.

If you use it for control of execution decisions, like stuffing an RTS to short-circuit a function, you're probably asking for trouble. At the very least, you have to ensure that everything is initialized on every execution, or you won't be able to run the code more than once without reloading it. The real trouble though is that you can't just look at the code and know what's going to happen, because what eventually gets executed is different. If all you're doing is changing the address of a STA, it's not so bad, because you know that it's storing a byte

- somewhere*. With the RTS example, what it does at any given point

depends heavily on what it did in the past.

>If you use it for control of execution decisions, like stuffing an RTS to short-circuit a function, you're probably asking for trouble. At the very least, you have to ensure that everything is initialized on every execution, or you won't be able to run the code more than once without reloading it.

the idea is to include/not include a JSR to a "wait for button press" routine for debugging.... and to use self modifying code for the fun of it.

I could add a command within the program:

```
lda key

cmp #enablewait
beq addwait

cmp #disablewait
beq removewait
<etc, rest of commands of my program>

addwait
lda #$XX    ;rem OPcode for JSR
sta modify  ;
sta modify2
sta modify3  (etc)
jmp end

removewait
lda #$60    ;rem OPcode for RTS
sta modify  ;
sta modify2
sta modify3  (etc)
jmp end

subroutine1
```

```
<do stuff>
modify jsr $XXXX
RTS

subroutine2
<do stuff>
modify2 jsr $XXXX
 RTS

subroutine1
modify3 jsr $XXXX
 RTS
```

I've used self-modifiying code to (1) preserve registers without using extra space, (2) in unrolling loops and (3) eliminate branches. I've also dynamically built inner loops as outlined by Andy. There's a lot of ways to use self-modifying code. Some of them might even be practical. :)

Lucas

Exmaple 1: This modifies an operand

```
      sty  restore+1
      jsr do_stuff
restore ldy \#00
```

Example 2: This modifies an operand

```
          lda  \#end
          sec
          sbc  num_loops
          sta  disp+1
          lda  \#00
disp     jmp  $0000
          sta  $FF
          .
          .
          sta  $01
          sta  $00
end   rts
```

Example 3: (if x = 0 save Y, if x = 1 do nothing). This modifies an opcode

```
            lda  opcode,x
            sta  patch
            tya
patch    sta  save
opcode  db  '$8D, $AD'  $8D = sta, $AD = lda
```

I like example 1... that is a lot faster than pulling registers off the stack.

Example 2 is a loop unrolling example, right? So you'll store 0 in addresses 0 thru num_loops without any branch logic -- but this extra speed is of course at the expense of additional memory for the code.

Example 3 seems like great obfuscation, but since you're clobbering the accumulator, couldn't you just do:

```
                txa
                bne no_save
                sty save
no_save ...
```

It's 12 bytes for your method (including the opcode table) and 6 bytes for mine. However, yours is a very cool way to throw someone WAAAAAY off the scent of what you're actually doing so would be a good in a copy-protection scheme. Unless I've lost the point of what it's also capable of doing (very likely :-)

>It's 12 bytes for your method (including the opcode table) and 6 bytes >for mine. However, yours is a very cool way to throw someone WAAAAAY >off the scent of what you're actually doing so would be a good in a >copy-protection scheme. Unless I've lost the point of what it's also >capable of doing (very likely :-)

Well, I tried just to present the skeleton of each idea. For a more 'real world' usage of something like Example 3, consider trying to look something up in a table where the index is x + y. The 'slow' way of doing this is

```
    stx  tmp
    tya
    clc
    adc  tmp
    tax
    lda  table,x
```

Using self-modifying code (assuming the table is page-aligned)

```
        sty  patch+1
patch lda table,x
```

Granted, this patches an operand instead of the opcode but it does let you index further than 255 bytes, e.g. x = 200, y = 200 will do the right thing.

Hmmm...let's try again. The one time I actually used Example 3, I chained several of these together. Here's code that implements the following bit of logic (assume x = 0 or 1, y = 0 or 1, paddle = 0 to 3, and action is at memory location $60XX)

```
if x = 0 then
  return
else
  if paddle+y > 1 then
     return action,y
  else
     return 7
end
```

```
         lda  opcode1,x   ; patch in RTS or LDA paddle
         sta  patch1
patch1 lda paddle
         sta patch2+1        ; patch in offset (opcode2 is page aligned)
patch2 lda opcode2,y
         sta patch3
patch3 lda action,y
         rts

org $6000
opcode2 db '$A9, $A9, $B9, $B9, $B9'
```

```
opcode1 db '$60, $AD'
action db '$5A, $A5'
```

The part around patch3 deserves a little more explanation. Since action = $6007, the four bytes at patch3 will either be

```
$B9 $07 $60 $60 = LDA $6007,y; RTS
```

or

```
$A9 $07 $60 $60 = LDA #$07; RTS; RTS.
```

So the correct behavior really depends on the location of the action table. I guess you could use something like that for copy-protection.

) It gets even more interesting when you patch the opcode to select

between things like LDA table,x and LDA table,y.

-Lucas

---

lscha...@d.umn.edu wrote:

```
> Exmaple 1: This modifies an operand
>            sty  restore+1
>            jsr do_stuff
> restore ldy \#00
```

"STY absolute" + "LDY \#imm" takes 4+3=7 cycles and 5 bytes.

```
  phy
  jsr do_stuff
  ply
```

takes 3+4=7 cycles and 2 bytes, and you won't burn in hell for all eternity.

```
> Example 2: This modifies an operand
>            lda  \#end
>            sec
>            sbc  num_loops
>            sta  disp+1
>            lda  \#00
> disp    jmp  $0000
>            sta  $FF
>            .
>            .
>            sta  $01
>            sta  $00
> end      rts
```

A "computed goto" is a reasonable use, though there is an indirect form of JMP that may work better. (It looks like JMP(addr,X) didn't exist until the 65c02 though.) The above doesn't actually work, if I understand your intention -- you have to multiply the value by 2 to line it up on a STA instruction, and you have to do a 16-bit add because "disp" crosses at least one page.

> Example 3: (if x = 0 save Y, if x = 1 do nothing). This modifies an > opcode

```
>             lda  opcode,x
```

```
>             sta  patch
>             tya
> patch   sta  save
> opcode db  '$8D, $AD'  $8D = sta, $AD = lda
```

This is the sort of thing that scares me, for the reasons mentioned earlier: it's hard to tell what's going to happen by reading the code.

Don't forget you can do tricks like this:

```
  [test something, branch to load+1]
          lda  #$00
load     bit  $03a9
          sta  somewhere
```

There's a "lda #$03" embedded in the BIT instruction. It's generally more sane to code it like this:

```
          lda  #$00
          dfb  $2c       ;BIT abs
load     lda  #$03
          sta  somewhere
```

It worries me a little that I'm doing most of this off the top of my head.

```
>> Example 1: This modifies an operand
>>             sty  restore+1
>>             jsr do_stuff
>> restore ldy #00
>"STY absolute" + "LDY \#imm" takes 4+3=7 cycles and 5 bytes.
>  phy
>  jsr do_stuff
>  ply
>takes 3+4=7 cycles and 2 bytes, and you won't burn in hell for all eternity.
```

Well, the place I actually use constructs like this is IIgs specific where I need to restore the stack pointer after some PEA slamming. My code really looks like this

```
      tsc
      sta patch+1
loop anop
      jmp mess_up_stack
patch lda #5A5A
      tcs
```

This lets me replace a save/restore with just a restore.

>A "computed goto" is a reasonable use, though there is an indirect form of JMP that may work better. (It looks like JMP(addr,X) didn't exist until the 65c02 though.) The above doesn't actually work, if I understand your intention -- you have to multiply the value by 2 to line it up on a STA instruction, and you have to do a 16-bit add because "disp" crosses at least one page.

Agreed. Also, for an unrolled loop the jmp (addr,x) instruction is not too useful since you need a jump table as big as the unrolled loop itself! And yes, I did miss some of the details.

```
>> Example 3: (if x = 0 save Y, if x = 1 do nothing).  This modifies an
>> opcode
>>             lda  opcode,x
>>             sta  patch
```

```
>>             tya
>> patch   sta  save
>> opcode db  '$8D, $AD'  $8D = sta, $AD = lda
>This is the sort of thing that scares me, for the reasons mentioned
```

Wait until you read my other post!

>earlier: it's hard to tell what's going to happen by reading the code. Don't forget you can do tricks like this:

```
>  [test something, branch to load+1]
>          lda  #$00
>load     bit  $03a9
>          sta  somewhere
```

>There's a "lda #$03" embedded in the BIT instruction. It's generally more sane to code it like this:

```
>          lda  #$00
>          dfb  $2c       ;BIT abs
>load     lda  #$03
>          sta  somewhere
```

I personally like the "jump into instruction" obfusication. I don't know exactly why -- there just something satisfying in writing code so synergistic that every byte is fullfilling multiple functions. ;)

I think that quite a few "self-modifying ticks" can be much more useful when you can wrap then in some good semantics, as you illustrated.

>It worries me a little that I'm doing most of this off the top of my head.

Nah, it just shows off your geek quotient.

-Lucas

If only Von Neumann knew what kind of insanity we'd get ourselves into with the whole stored program concept. Maybe he would have reworked things so that program code could only touch memory segments flagged as data-only. Then again, maybe Von Neumann would write enough self-modifying code to make someone like Dykstra want to take a flying leap off the UT clock tower. ;-D

BLuRry wrote: > If only Von Neumann knew what kind of insanity we'd get ourselves into with the whole stored program concept. Maybe he would have reworked things so that program code could only touch memory segments flagged as data-only. Then again, maybe Von Neumann would write enough self-modifying code to make someone like Dykstra want to take a flying leap off the UT clock tower. ;-D

Before the invention of the index register (or the "B-Box" as it was called at Manchester), modifying code was the *only* way to write useful loops. The possibility of programmatic code modification was not an unintended consequence of storing code and data in the same memory, but a primary motivation for doing so.

The very possibility of programs creating other programs was created, and with it, endless possibilities.

Today, with most instruction modification formalized and abstracted as indexing and indirection, we tend to regard code that re-writes code as a problem, not a solution. But where would we be without

compilers, optimizers, JIT code generators, dynamic optimizers, etc., all of which spring from the concept of code being able to be data, too.

So the *real* rule must be: "Don't modify code unless you really know what you're doing, and all of its implications." (Like playing with fire. ;-)

-michael