So-Called Unused Opcodes#

from Apple Assembly Line 03/81

The 6502 has 104 so-called unused opcodes. The various charts and reference manuals I have checked either leave them blank or call them "unused", "no-operation", or "future expansion". The 6502 has been around since 1976; I think we have waited long enough to know there will be no "expansion". But are they really unused? Do they have any effect if we try to execute them? Are they really no-ops? If so, how many bytes does the processor assume for each one?

These questions had never bothered me until I was looking through some disassembled memory and thought I found evidence of someone USING the "unused". It turned out they were not, but my curiosity was aroused. Just for fun, I built a little test routine and tried out the $FF opcode. Lo and behold! The 6502 thinks it is a 3-byte instruction, and it changes the A-register and some status bits!

About 45 minutes later I pinned it down: FFxxyy performs exactly the same as the two instructions FExxyy and FDxxyy. It is just as though I had executed one and then the other. In other words, anywhere in a program I find:

```
INC VARIABLE,X
SBC VARIABLE,X
```

I can substitute:

```
.HS FF
.DA VARIABLE
```

You might wonder if I will ever find that sequence. I did try writing a program to demonstrate its use. It has the advantage of saving 3 bytes, and 4 clock cycles. (The SBC instruction is executed DURING the 7 cycles of the INC instruction!)

```
TEST LDX INDEX
     LDA #10       FOR COUNTER(X)=10 TO 39
     STA COUNTER,X
.1   LDA COUNTER,X GET COUNTER(X)
     JSR $FDDA     PRINT IT OUT (OR WHATEVER)
     LDA #39       LIMIT
     .HS FF        DO INC AND SBC
     .DA COUNTER   ON COUNTER,X
     BCS .1        NEXT
     RTS
```

Are there any more? Before I could rest my curiosity, I had spent at least ten more hours, and had figured out what all 104 "unused opcodes" really do!

The center-fold chart shows the fruit of my detective work. The shaded opcodes are the "unused" ones. I don't know if every 6502 behaves the same as mine or not. Mine appears to be made by Synertek, and has a date code of 7720 (20th week of 1977). It could be that later versions or chips from other sources (MOS Technology or Rockwell) are different. If you find yours to be different, please let me know!

Twelve of the opcodes, all in column "x2", hang up the 6502; the only way to get out is to hit RESET or turn off the machine.

There are 27 opcodes which appear to have no effect on any registers or on memory. These could be called "NOP", but some of them are considered by the 6502 to have 2 or 3 bytes. I have labeled

them "nop", "nop2", and "nop3" to distinguish how many bytes the 6502 thinks it is using. You could call nop2 "always skip one byte" and nop3 "always skip two bytes".

The action most of the rest perform can be deduced by looking at the other opcodes in the same row. For example, all of the xF column (except 8F and 9F) perform two instructions together: first the corresponding xE opcode, and then the corresponding xD opcode. In the same way, most of the opcodes in column x7 combine the x6 and x5 opcodes. The x3 column mirrors the x7 and xF columns, but with different addressing modes. And finally, the xB column mimics the other three columns, but with more exceptions. Most of the exceptions are in the 8x and 9x rows.

A few of the opcodes seem especially interesting and potentially useful. For example, A3xx performs three steps: first it loads xx into the X-register; then using this new value of X, it moves the byte addressed by (xx,X) into both the A- and X- registers. Another way of looking at this one is to say that whatever value xx has is doubled; then the two pagezero bytes at 2*xx and 2*xx+1 are used as the address for loading the A- and X-registers. You could use this for something, couldn't you?

There are five instructions which form the logical product of the A- and X-registers (without disturbing either register) and store the result in memory. If we call this new instruction "SAX", for "Store A&X", we have:

```
83   SAX (z,X)          8F   SAX a
87   SAX z              9F   SAX a,X
97   SAX z,Y
```

We get seven forms of the combination which shift a memory location using ASL, and then inclusive OR the results into A with an ORA instruction. If we call this new instruction ALO, we have:

```
03   ALO (z,X)          1B   ALO a,Y
13   ALO (z),Y          0F   ALO a
07   ALO z              1F   ALO a,X
17   ALO z,X
```

The same seven forms occur for the combinations ROL-AND, LSR-EOR, and ROR-ADC. Note that if you don't care what happens to the A-register, and the status register, these 28 instructions make two extra addressing modes available to the shift instructions: (z,X) and (z),Y.

Opcodes 4B and 6B might also be useful. You can do an AND-immediate followed by LSR or ROR on the A-register.

Opcodes 93, 9B, and 9E are really weird! It took a lot of head-scratching to figure out what they do.

```
93   Forms the logical product of the A-register
     and byte the at z+1 (which I call "hea")
     and stores it at (z),Y.

9B   Forms the logical product of the A- and X-
     registers, and stores the result in the S-
     register (stack pointer)!  Ouch!
     Then it takes up the third byte of the
     instruction (yy from 9B xx yy) and adds one
     to it (I call it "hea+1").  Then it forms
     the logical product of the new S-register
     and "hea+1" and stores the result at "a,Y".
     Whew!


9E   Forms the logical product of the X-register
     and "hea+1" and stores the result at "a,Y".
```

We get six forms of the new "LAX" instruction, which loads the same value into both the A- and X-registers:

```
     B3   LAX (z),Y            AB   LAX #v
     A7   LAX z                AF   LAX a
     B7   LAX z,Y              BF   LAX a,Y
```

I skipped over BB, because it is another extremely weird one. It forms the logical product of the byte at "a,Y" and S-register, and stores the result in the A-, X-, and S-registers. No wonder they didn't tell us about it!

Right under that one is the CB instruction. Well, good buddy (please excuse the CB talk!), it forms the logical product of the A- and X-registers, subtracts the immediate value (second byte of CB xx), and puts the result into the X-register.

The Cx and Dx rows provide us with seven forms that do a DEC on a memory byte, and then CMP the result with the A-register. Likewise, the Ex and Fx rows give us seven forms that perform INC followed by SBC.

It is a good thing to be aware that the so-called "unused" opcodes can be quite dangerous if they are accidentally executed. If your program goes momentarily wild and executes some data, chances are something somewhere will get strangely clobbered.

Since all of the above information was deduced by testing and observation, I cannot be certain that I am 100% correct. I may have overlooked or mis-interpreted some results, or even made a clerical error. Furthermore, as I said before, my 6502 may be different from yours. You can test your own, to see if it works like mine.

And if the whole exercise seems academic to you, you can at least enjoy the first legible and complete hexadecimal opcode chart for the 6502.

```
    x0      x1          x2      x3

0x  BRK     ORA (z,X)   hang    ASL (z,X)
                                ORA (z,X)

1x  BPL r   ORA (z),Y   hang    ASL (z),Y
                                ORA (z),Y

2x  JSR a   AND (z,X)   hang    ROL (z,X)
                                AND (z,X)

3x  BMI r   AND (z),Y   hang    ROL (z),Y
                                AND (z),Y

4x  RTI     EOR (z,X)   hang    LSR (z,X)
                                EOR (z,X)

5x  BVC r   EOR (z),Y   hang    LSR (z),Y
                                EOR (z),Y

6x  RTS     ADC (z,X)   hang    ROR (z,X)
                                ADC (z,X)

7x  BVS r   ADC (z),Y   hang    ROR (z),Y
                                ADC (z),Y

8x  nop2    STA (z,X)   nop2    A&X
```

```
                                --> (z,X)

9x  BCC r   STA (z),Y  hang    A&hea
                                --> (z),Y


Ax  LDY #v  LDA (z,X)  LDX #v  LDX #v
                               LDA (z,X)
                               LDX (z,X)

Bx  BCS r   LDA (z),Y  hang    LDA (z),Y
                               LDX (z),Y

Cx  CPY #v  CMP (z,X)  nop2    DEC (z,X)
                               CMP (z,X)

Dx  BNE r   CMP (z),Y  hang    DEC (z),Y
                               CMP (z),Y

Ex  CPX #v  SBC (z,X)  nop2    INC (z,X)
                               SBC (z,X)

Fx  BEQ r   SBC (z),Y  hang    INC (z),Y
                               SBC (z),Y



x4       x5       x6       x7

nop2     ORA z    ASL z    ASL z
                           ORA z

nop2     ORA z,X  ASL z,X  ASL z,X
                           ORA z,X

BIT z    AND z    ROL z    ROL z
                           AND z

nop2     AND z,X  ROL z,X  ROL z,X
                           AND z,X

nop2     EOR z    LSR z    LSR z
                           EOR z

nop2     EOR z,X  LSR z,X  LSR z,X
                           EOR z,X

nop2     ADC z    ROR z    ROR z
                           ADC z

nop2     ADC z,X  ROR z,X  ROR z,X
                           ADC z,X

STY z    STA z    STX z    A&X
                           --> z

STY z,X  STA z,X  STX z,Y  A&X
                           --> z,Y
```

```
LDY z    LDA z    LDX z    LDX z
                           LDA z

LDY z,X  LDA z,X  LDX z,Y  LDX z,Y
                           LDA z,Y

CPY z    CMP z    DEC z    DEC z
                           CMP z

nop2     CMP z,X  DEC z,X  DEC z,X
                           CMP z,X

CPX z    SBC z    INC z    INC z
                           SBC z

nop2     SBC z,X  INC z,X  INC z,X
                           SBC z,X


x8   x9       xA   xB

PHP  ORA #v   ASL  AND #v


CLC  ORA a,Y  nop  ASL a,Y
                   ORA a,Y

PLP  AND #v   ROL  AND #v


SEC  AND a,Y  nop  ROL a,Y
                   AND a,Y

PHA  EOR #v   LSR  AND #v
                   LSR

CLI  EOR a,Y  nop  LSR a,Y
                   EOR a,Y

PLA  ADC #v   ROR  AND #v
                   ROR

SEI  ADC a,Y  nop  ROR a,Y
                   ADC a,Y

DEY  nop2     TXA  #v&X
                   --> A

TYA  STA a,Y  TXS  A&X-->S
                   S&hea+1
                   --> a,Y

TAY  LDA #v   TAX  LDA #v
                   TAX


CLV  LDA a,Y  TSX  a,Y & S
```

```
                    -->AXS

INY  CMP #v   DEX  A&X-#v
                    --> X

CLD  CMP a,Y  nop  DEC a,Y
                    CMP a,Y

INX  SBC #v   NOP  SBC #v


SED  SBC a,Y  nop  INC a,Y
                    SBC a,Y


xC       xD       xE       xF

nop3     ORA a    ASL a    ASL a
                           ORA a

nop3     ORA a,X  ASL a,X  ASL a,X
                           ORA a,X

BIT a    AND a    ROL a    ROL a
                           AND a

nop3     AND a,X  ROL a,X  ROL a,X
                           AND a,X

JMP a    EOR a    LSR a    LSR a
                           EOR a

nop3     EOR a,X  LSR a,X  LSR a,X
                           EOR a,X

JMP (a)  ADC a    ROR a    ROR a
                           ADC a

nop3     ADC a,X  ROR a,X  ROR a,X
                           ADC a,X

STY a    STA a    STX a    A&X
                           --> a

nop3     STA a,X  X&hea+1  A&X
                  --> a,Y  --> a,X


LDY a    LDA a    LDX a    LDX a
                           LDA a


LDY a,X  LDA a,X  LDX a,Y  LDX a,Y
                           LDA a,Y

CPY a    CMP a    DEC a    DEC a
                           CMP a

nop3     CMP a,X  DEC a,X  DEC a,X
```

```
                            CMP a,X

CPX a    SBC a    INC a     INC a
                            SBC a

nop3     SBC a,X  INC a,X   INC a,X
                            SBC a,X

!lm11
!rm76
A     A-register (Accumulator)
S     S-register (Stack Pointer)
X     X-register
Y     Y-register

a     2-byte absolute address
r     1-byte relative address
v     1-byte immediate value
z     1-byte pagezero address

hea   high-byte of effective address
      93:  the byte at z+1
      9B:  3rd byte of instruction
      9E:  3rd byte of instruction

&     and-function (logical product)

hang  computer hangs up, only way to
      regain control is to hit RESET

nop   1-byte instruction, no operation
nop2  2-byte instruction, no operation
nop3  3-byte instruction, no operation

-->   "result is stored in"
```