[Return to Main Page] Beyond 8-bit Unsigned Comparisons by Bruce Clark
[Up to Tutorials and Aids]

Table of Contents

---

1 INTRODUCTION

The 6502 has several options available for comparing numbers. Each option, naturally, has its pros and cons in terms of speed and size. There are also some infrequently used options that occasionally come in handy. Finally, and unfortunately, some misconceptions abound about signed (twos complement) numbers and the correct way to compare them. All of these topics will be covered below.

1.1 TYPES OF COMPARISONS

Generally, two numbers are compared to determine (a) if they are same, or (b) which one is smaller (or larger) than the other. For the sake of consistency, three terms are used from this point on:

- "Equality Comparison": a comparison of two numbers to determine whether they are the same. In this case it does not matter if the numbers are signed or unsigned, or even whether the bytes (or bits) being compared represent numbers.
- "Signed Comparison": a comparison of two signed numbers to determine which is smaller.
- "Unsigned Comparison": a comparison of two unsigned numbers to

determine which is smaller.

Note that since the valid BCD numbers are simply a subset of the unsigned numbers, an unsigned comparison can be used to compare two BCD numbers to determine which is smaller.

1.2 REVIEW OF TWOS COMPLEMENT (SIGNED) NUMBERS

There are 256 possible values for a byte; in hex, they are: $00 to $FF. The range of an 8-bit unsigned number is 0 ($00) to 255 ($FF). The range of a 16-bit unsigned number is 0 ($0000) to 65535 ($FFFF), and so on. They are called unsigned numbers because they are zero or greater, i.e. there is no (minus) sign. A signed number, on the other hand, can be negative or positive (or zero). The term "signed number" is used below to mean a twos complement number (although there are other ways of representing signed numbers). The range of an 8-bit signed number is -128 to 127. The values -128 through -1 are, in hex, $80 through $FF, respectively. The values 0 through 127 are, in hex, $00 through $7F, respectively. So the minimum value of a signed number is $80 and the maximum value of a signed number is $7F. The range of a 16-bit signed number is -32768 ($8000) to 32767 ($7FFF) ($8000 through $FFFF are the negative numbers), and so on. This may seem like a strange way of handling negative numbers, but this method has several useful properties.

First, 0 to 127 (the overlap of the ranges of 8-bit signed and unsigned numbers) is, in hex, $00 to $7F, regardless of whether the number is signed or unsigned.

Second, the most significant bit (bit 7 for an 8-bit number) is zero when the number is non-negative (0 to 127), and one when the number is negative. In fact, this is how the N (negative) flag of 6502 got its name. (Notice that the N flag, when affected by an instruction, reflects bit 7 of the result of that instruction.) One other note: in mathematics, zero is not a postive or a negative number, but in the computer world, things are less formal; the term "positive number" typically includes zero because (a) all of the other possible values of a signed number whose most significant bit is zero are positive numbers, and (b) all of the other possible values for an unsigned number are positive numbers.

Third, consider the following addition:

```
CLC
LDA #$FF
ADC #$01
```

The result (in the accumulator) is $00, and the carry is set. The addition, in unsigned numbers, is: 255 + 1 = 256 (remember, the carry is set). The addition, in signed numbers, is -1 + 1 = 0. In other words, adding (and subtracting) signed numbers is exactly the same as adding (and subtracting) unsigned numbers.

2 REVIEW OF COMPARE INSTRUCTIONS

There are three compare instructions on the 6502: CMP, CPX, and CPY. However, EOR and SBC can also be used for comparisions, and occasionally this is useful (for EOR more so than SBC).

2.1 THE CMP, CPX, AND CPY INSTRUCTIONS

The CMP, CPX, and CPY instructions are used for comparisons as their mnemonics suggest. The way they work is that they perform a subtraction. In fact,

```
CMP NUM
```

is very similar to:

```
SEC
SBC NUM
```

Both affect the N, Z, and C flags in exactly the same way. However, unlike SBC, (a) the CMP subtraction is not affected by the D (decimal) flag, (b) the accumulator is not affected by a CMP, and (c) the V flag is not affected by a CMP. A useful property of CMP is that it performs an equality comparison and an unsigned comparison. After a CMP, the Z flag contains the equality comparison result and the C flag contains the unsigned comparison result, specifically:

- If the Z flag is 0, then A <> NUM and BNE will branch
- If the Z flag is 1, then A = NUM and BEQ will branch
- If the C flag is 0, then A (unsigned) < NUM (unsigned) and BCC will branch
- If the C flag is 1, then A (unsigned) >= NUM (unsigned) and BCS will branch

In fact, many 6502 assemblers will allow BLT (Branch on Less Than) and BGE (Branch on Greater than or Equal) to be used as synonyms for BCC and BCS, respectively.

The N flag contains most significant bit of the of the subtraction result. This is only occasionally useful. However, it is NOT the signed comparison result, as is sometimes claimed, as the following examples illustrate:

After:

```
LDA #$01 ;  1 (signed),   1 (unsigned)
CMP #$FF ; -1 (signed), 255 (unsigned)
```

A = $01, C = 0, N = 0 (the subtraction result is $01 - $FF = $02), and Z = 0. The comparison results are:

- Equality comparison: false, since $01 <> $FF
- Signed comparison: 1 >= -1
- Unsigned comparison: 1 < 255

After:

```
LDA #$7F ;  127 (signed), 127 (unsigned)
```

```
CMP #$80 ; -128 (signed), 128 (unsigned)
```

A = $7F, C = 0, N = 1 (the subtraction result is $7F - $80 = $FF), and Z = 0. The comparison results are:

- Equality comparison: false, since $7F <> $80
- Signed comparison: 127 >= -128
- Unsigned comparison: 127 < 128

Notice that in both cases the signed comparison result is the same (the first number is greater than or equal to the second), but the N flag is different.

The CPX and CPY instructions are exactly like the CMP instruction, except that they use the X and Y registers, respectively, instead of the accumulator.

2.2 A TRICK SO SIMPLE THAT IT'S OFTEN OVERLOOKED

A surprisingly common sequence in 6502 code is:

```
LDA NUM1
CMP NUM2
BCC LABEL
BEQ LABEL
```

(or something similar) which branches to LABEL when NUM1 <= NUM2. (In this case NUM1 and NUM2 are unsigned numbers.) However, consider the following sequence:

```
LDA NUM2
CMP NUM1
BCS LABEL
```

which branches to LABEL when NUM2 >= NUM1, which is the same as NUM1 <= NUM2. Not only that, it's shorter and (in many cases) faster.

2.3 USING EOR FOR EQUALITY COMPARISONS

The EOR instruction can also be used for equality comparisions. Naturally, there are trade-offs. First, EOR, unlike CMP, affects the accumulator. Second, EOR is only available for an equality comparision to the accumulator and not the X or Y registers. Third, EOR does not affect the C flag. As it happens, not affecting the C flag is useful enough that EOR is often used for equality comparisons. After an EOR, the equality comparison result is in the Z flag, specifically:

- If the Z flag is 0, then A <> NUM and BNE will branch
- If the Z flag is 1, then A = NUM and BEQ will branch

This is exactly the same way CMP affects the Z flag.

2.4 USING EOR TO COMPARE ONLY SOME OF THE BITS IN A BYTE

An additional advantage EOR is that it is easy to compare only some of the bits in a byte. After the EOR, simply use the AND instruction to mask off the bits of interest, and the equality comparison result will be in Z, as usual. For

example, after:

```
LDA BYTE1
EOR BYTE2
AND #$AB  ; $AB = %10101011 (binary)
```

The Z flag will be 1 (BEQ branches) if bits 7, 5, 3, 1, and 0 of BYTE1 are the same as bits 7, 5, 3, 1, and 0 of BYTE2, and the Z flag will be 0 (BNE branches) if they are not.

2.5 A TRICK WHEN USING SBC FOR UNSIGNED COMPARISONS

Since CMP and SBC both subtract, SBC can also be used for unsigned comparisons, although such use is rare. This is because SBC affects the accumulator and CMP does not. There is one instance where SBC can occasionally be useful. In fact, this trick is more useful when combined with the method in Section 4.3 (comparison by subtraction) for extending unsigned comparisons beyond 8 bits than it is by itself (as it is presented here). The trick is to CLEAR the carry before the SBC instruction. For example, after:

```
CLC
SBC NUM
```

C still holds the unsigned comparison result, but in this case:

- If the C flag is 0, then A (unsigned) <= NUM (unsigned) and BCC will branch
- If the C flag is 1, then A (unsigned) > NUM (unsigned) and BCS will branch

This is an alternative to the trick in section 2.2. It is mainly useful when (a) the effect of previous instructions cleared the carry (as opposed to an explicit CLC instruction), and (b) it isn't a problem that A is affected.

As as aside, remember that the D flag affects the result of a SBC. So there is the question of what happens when the D flag is 1. Believe it or not, after:

```
CLD
SBC NUM
```

the C flag will be the same as after:

```
SED
SBC NUM
```

assuming that the accumulator and NUM are the same in both cases, even if the accumulator or NUM (or both) is not a valid BCD number! This is true of the 6502, the 65C02, and the 65C816. (This has been tested on a Rockwell 6502, a Synertek 6502, a GTE 65C02, and a GTE 65C816.) Of course, the effect on the accumulator is different in the two cases above, but the effect on the carry is the same.

3 EXTENDING THE EQUALITY COMPARISON BEYOND 8 BITS

There aren't any tricks when extending the equality comparison beyond 8 bits. The bytes are simply compared one a time. A few examples are in order.

Example 3.1: a 16-bit equality comparison (low byte in Y, high byte in A) which branches to LABEL if the numbers are not equal

```
CPY NUML  ; compare low bytes
BNE LABEL
CMP NUMH  ; compare high bytes
BNE LABEL
```

Example 3.2: a 16-bit equality comparison (again, low byte in Y, high byte in A) which branches to LABEL2 if the numbers are equal

```
        CPY NUML   ; compare low bytes
        BNE LABEL1
        CMP NUMH   ; compare high bytes
        BEQ LABEL2
LABEL1
```

Example 3.3: a 16-bit equality comparison (again, low byte in Y, high byte in A) which leaves the usual equality comparison result in the Z flag

```
        CPY NUML  ; compare low bytes
        BNE LABEL
        CMP NUMH  ; compare high bytes
LABEL
```

Example 3.4: a 24-bit equality comparison (low byte in Y, middle byte in X, high byte in A) which branches to LABEL if the numbers are not equal

```
CPY NUML  ; compare low bytes
BNE LABEL
CPX NUMM  ; compare middle bytes
BNE LABEL
CMP NUMH  ; compare high bytes
BNE LABEL
```

Example 3.5: a 24-bit equality comparison (again, low byte in Y, middle byte in X, high byte in A) which branches to LABEL2 if the numbers are equal

```
        CPY NUML   ; compare low bytes
        BNE LABEL1
        CPX NUML   ; compare middle bytes
        BNE LABEL1
        CMP NUMH   ; compare high bytes
        BEQ LABEL2
LABEL1
```

Note that in all five examples, (a) the bytes can be compared in any order (e.g. the high bytes could be compared first), and (b) an EOR could be used instead of CMP.

4 EXTENDING THE UNSIGNED COMPARISON BEYOND 8 BITS

There are several options for extending unsigned comparisons. As is usually the case, there are pros and cons to each option. There are trade-offs in terms of space and speed, so the best option depends on the situation.

4.1 COMPARING ONE BYTE AT A TIME

This is the most straightforward approach. It's similar to the approach for equality comparisons. The idea is to start by comparing the high bytes and work toward the low bytes. A few examples are in order.

Example 4.1.1: a 16-bit unsigned comparison which branches to LABEL2 if NUM1 < NUM2

```
        LDA NUM1H  ; compare high bytes
        CMP NUM2H
        BCC LABEL2 ; if NUM1H < NUM2H then NUM1 < NUM2
        BNE LABEL1 ; if NUM1H <> NUM2H then NUM1 > NUM2 (so NUM1 >= NUM2)
        LDA NUM1L  ; compare low bytes
        CMP NUM2L
        BCC LABEL2 ; if NUM1L < NUM2L then NUM1 < NUM2
LABEL1
```

Example 4.1.2: a 16-bit unsigned comparison which branches to LABEL2 if NUM1 >= NUM2

```
        LDA NUM1H  ; compare high bytes
        CMP NUM2H
        BCC LABEL1 ; if NUM1H < NUM2H then NUM1 < NUM2
        BNE LABEL2 ; if NUM1H <> NUM2H then NUM1 > NUM2 (so NUM1 >= NUM2)
        LDA NUM1L  ; compare low bytes
        CMP NUM2L
        BCS LABEL2 ; if NUM1L >= NUM2L then NUM1 >= NUM2
LABEL1
```

Example 4.1.3: a 24-bit unsigned comparison which branches to LABEL2 if NUM1 < NUM2

```
        LDA NUM1H  ; compare high bytes
        CMP NUM2H
        BCC LABEL2 ; if NUM1H < NUM2H then NUM1 < NUM2
        BNE LABEL1 ; if NUM1H <> NUM2H then NUM1 > NUM2 (so NUM1 >= NUM2)
        LDA NUM1M  ; compare middle bytes
        CMP NUM2M
        BCC LABEL2 ; if NUM1M < NUM2M then NUM1 < NUM2
        BNE LABEL1 ; if NUM1M <> NUM2M then NUM1 > NUM2 (so NUM1 >= NUM2)
        LDA NUM1L  ; compare low bytes
        CMP NUM2L
        BCC LABEL2 ; if NUM1L < NUM2L then NUM1 < NUM2
LABEL1
```

Example 4.1.4: a 24-bit unsigned comparison which branches to LABEL2 if NUM1 >= NUM2

```
        LDA NUM1H  ; compare high bytes
        CMP NUM2H
        BCC LABEL1 ; if NUM1H < NUM2H then NUM1 < NUM2
        BNE LABEL2 ; if NUM1H <> NUM2H then NUM1 > NUM2 (so NUM1 >= NUM2)
        LDA NUM1M  ; compare middle bytes
        CMP NUM2M
        BCC LABEL1 ; if NUM1M < NUM2M then NUM1 < NUM2
        BNE LABEL2 ; if NUM1M <> NUM2M then NUM1 > NUM2 (so NUM1 >= NUM2)
        LDA NUM1L  ; compare low bytes
        CMP NUM2L
        BCS LABEL2 ; if NUM1L >= NUM2L then NUM1 >= NUM2
LABEL1
```

There are times when comparing one byte at a time is the fastest way to perform an unsigned comparison. Consider an unsigned comparison of two

numbers NUM1 and NUM2. There is a BCC instruction right after NUM1H is compared to NUM2H. So if circumstances are such that NUM1H is usually less than NUM2H, the BCC will be taken and the comparison will finish almost immediately. In this case, the other methods for extending unsigned comparisons beyond 8 bits will not be as fast.

4.2 AN EQUALITY AND UNSIGNED COMPARISON

One of the useful things about using CMP is that it does an equality and a unsigned comparison. There is a method for doing comparisons beyond 8 bits that give an equality comparison result and an unsigned comparison result, without having to do the comparison twice. The idea is again to work from the high bytes to the low bytes. This idea is extended by noticing that if the high bytes are not equal, then no further bytes need to be compared. As it turns out, this approach will also provide an equality comparison result. A couple of examples are in order.

Example 4.2.1: a 16-bit comparison (low byte in Y, high byte in A) which leaves the usual equality comparison result in the Z flag, and the usual unsigned comparison result in the C flag

```
        CMP NUMH  ; compare high bytes
        BNE LABEL
        CPY NUML  ; compare low bytes
LABEL
```

Note that this example is similar to Example 3.3 (in Section 3), but unlike Example 3.3, the high byte is compared first, and the low byte is compared second.

Example 4.2.2: a 24-bit unsigned comparison (low byte in Y, middle byte in X, high byte in A) which leaves the usual equality comparison result in the Z flag, and the usual unsigned comparison result in the C flag

```
        CMP NUMH
        BNE LABEL
        CPX NUMM
        BNE LABEL
        CPY NUML
LABEL
```

4.3 COMPARISON BY SUBTRACTION

Remember that the CMP instruction performs a subtraction. So one way do a multi-byte unsigned comparison is simply to do a multi-byte subtraction. After:

```
SEC       ; NUM3 = NUM1 - NUM2
LDA NUM1L
SBC NUM2L
STA NUM3L
LDA NUM1H
SBC NUM2H
STA NUM3H
```

The C flag will contain the unsigned comparison result. But notice that the

subtraction result does not need to be stored, since the C flag is all that is of interest for an unsigned comparison, so the two STA instructions can be eliminated as follows:

```
SEC
LDA NUM1L
SBC NUM2L
LDA NUM1H
SBC NUM2H
```

Notice that after the first SBC, the accumulator is immediately overwritten. The only purpose of the first SBC is to prepare the carry flag for the second SBC. But remember, after:

```
CMP NUM
```

the C flag will be the same as after:

```
SEC
SBC NUM
```

so the SEC and the first SBC instructions can be replaced by a CMP instruction as follows:

```
LDA NUM1L
CMP NUM2L
LDA NUM1H
SBC NUM2H
```

Notice that the this method works from low byte to high byte. A typical 24-bit unsigned comparison is:

```
LDA NUM1L
CMP NUM2L
LDA NUM1M
SBC NUM2M
LDA NUM1H
SBC NUM2H
```

5 SIGNED COMPARISONS

As stated above, after a CMP instruction, the N flag is NOT the signed comparison result. A signed comparison works by performing a subtraction, but the signed comparison result is the exclusive-or (eor) of the N and V flags. Specifically, to compare the signed numbers NUM1 and NUM2, the subtraction NUM1-NUM2 is performed, and NUM1 < NUM2 when N eor V is 1, and NUM1 >= NUM2 when N eor V is 0. (A program that proves this is given in Appendix A.)

5.1 COMPARING TWO 8-BIT NUMBERS

Calulating N eor V has two difficulties. First, CMP does not affect the V flag. But SBC does, so the solution is simply to use SBC instead of CMP. The second difficulty is how to handle the exclusive-or of the N and V flags. One way is to handle the four possible cases (N=0 and V=0, N=0 and V=1, N=1 and V=0, and N=1 and V=1) with BMI, BPL, BVC, and BVS instructions. This works, but there is a faster and shorter way. Remember that SBC puts the subtraction result in the accumulator, and the N flag is bit 7 of the

subtraction result. But bit 7 of the accumulator is also bit 7 of the subtraction result. So when the V flag is 1, use EOR to invert bit 7 of the accumulator (which is the same as N flag). After the EOR instruction, the N flag will contain N eor V (the signed comparison result), as will bit 7 of the accumulator. Specifically, after:

```
        SEC       ; prepare carry for SBC
        SBC NUM   ; A-NUM
        BVC LABEL ; if V is 0, N eor V = N, otherwise N eor V = N eor 1
        EOR #$80  ; A = A eor $80, and N = N eor 1
LABEL
```

If the N flag is 1, then A (signed) < NUM (signed) and BMI will branch
If the N flag is 0, then A (signed) >= NUM (signed) and BPL will branch

One way to remember which is which is to remember that minus (BMI) is less than, and plus (BPL) is greater than or equal to.

Since BVC and EOR do not affect the carry, note that the usual unsigned comparison result is the C flag.

It also possible to put the signed comparison result in the C flag with a little extra effort.

```
        SEC
        SBC NUM
        BVS LABEL ; Note: BVS not BVC
        EOR #$80
LABEL ASL
```

Notice that BVS is used instead of BVC, so bit 7 of the accumulator will be 0 when A < NUM, and 1 when A >= NUM. The ASL is used to shift this bit into the carry, so that C = 0 (BCC branches) when A < NUM, and C = 1 (BCS branches) when A >= NUM, as is the case with unsigned comparisons.

Note that the trick described in Section 2.5 also works with signed comparisons. When the carry is cleared before the subtraction, as with:

```
        CLC       ; Note: CLC, not SEC
        SBC NUM
        BVC LABEL
        EOR #$80
LABEL
```

N will contain the signed comparison result, but in this case:

- If the N flag is 1, then A (signed) <= NUM (signed) and BMI will branch
- If the N flag is 0, then A (signed) > NUM (signed) and BPL will branch

It should be noted that the Z flag does NOT contain the equality comparison result. For example:

```
        SEC
        LDA #$7F
        SBC #$FF
        BVC LABEL
        EOR #$80
```

```
    LABEL           ; The Z flag is 1 but $7F <> $FF !
```

Since the purpose of the EOR is to invert bit 7 of the accumulator, it is possible to use any value between $80 and $FF, inclusive, with EOR. The Z flag will be 0 in the example above if the EOR #$80 is replaced with EOR #$FF, but EOR #$FF has its own counterexample, specifically:

```
          SEC
          LDA #$7F
          SBC #$80
          BVC LABEL
          EOR #$FF
    LABEL           ; The Z flag is 1 but $7F <> $80 !
```

As as aside, remember that the D flag affects the result of a SBC. So again the question of what happens when the D flag is 1 arises. Believe it or not, as was the case with the C flag, after:

```
    CLD
    SBC NUM
```

the V flag will be the same as after:

```
    SED
    SBC NUM
```

Assuming that the accumulator and NUM are the same in both cases, even if the accululator or NUM (or both) is not a valid BCD number! This is true of the 6502, the 65C02, and the 65C816. (This has been tested on a Rockwell 6502, a Synertek 6502, a GTE 65C02, and a GTE 65C816.) Again, the effect on the accumulator is different in the two cases above, but the effect on the the V flag is the same.

Since the EOR #$80 uses the result in the accumulator, the signed comparison won't work properly in decimal mode. For example:

```
        SED
        SEC
        LDA #$80
        SBC #$10
        BVC LABEL ; $80 - $10 = $70, V = 1 since -128 - 16 = -144
        EOR #$80  ; $70 eor $80 = $F0, so N = 1
  LABEL
```

returns N = 1, which is correct since $80 (-128) < $10 (16), but:

```
        SED
        SEC
        LDA #$10
        SBC #$80
        BVC LABEL ; $10 - $80 = $30, V = 1 since 16 - (-128) = 144
        EOR #$80  ; $30 eor $80 = $B0, so N = 1
  LABEL
```

also returns N = 1, which is not correct.

5.2 A CAVEAT: THE SO PIN

The 6502 and the 65C02 have a seldom-used pin named SO (DIP pin 38), which stands for Set Overflow. As its name suggests, the SO pin allows the

hardware to set the V (overflow) flag, without using software instructions that affect the V flag. Since the signed comparison makes use the V flag, the SO pin is something to be aware of. However, the SO pin is usually not a problem for several reasons. First, the SO pin is seldom used, as noted above. Second, when the SO pin is used, it will usually be for a specific purpose, so WHEN it sets the V flag will be known (i.e. normally, it's not used to go around setting the V flag at random). Third, the BVC instruction, which tests the V flag, immediately follows the SBC instruction, which affects the V flag, so setting the V flag elsewhere won't change anything. Of course, all of this depends what exactly is connected to the SO pin. When troubleshooting signed comparisons, if the V flag is getting set when it shouldn't, it may be good idea to look at what is connected to the SO pin.

If the SO pin is interfering with signed comparisons, there is an another way to do signed comparisons. Remember that an 8-bit signed number ranges from -128 ($80) to 127 ($7F), and that -128 to -1 are represented by $80 to $FF and 0 to 127 are represented by $00 to $7F. If the most significant bit is inverted, then -128 ($80) to -1 ($FF) becomes $00 to $7F, and 0 ($00) to 127 ($7F) becomes $80 to $FF. So to do a signed comparison, invert the most significant bit of EACH number first, THEN do an unsigned comparison. For example, after:

```
LDA NUM2
EOR #$80 ; invert the most significant bit of NUM1
STA TEMP ; store the result so it can be used by a CMP instruction
LDA NUM1
EOR #$80 ; invert the most significant bit of NUM1
CMP TEMP
```

If the C flag is 0, then NUM1 (signed) < NUM2 (signed) and BCC will branch
If the C flag is 1, then NUM1 (signed) >= NUM2 (signed) and BCS will branch

Note that, unlike in Section 5.1, the usual equality comparison result will be in the Z flag, but the unsigned comparison result is not available.

6 EXTENDING THE SIGNED COMPARISON BEYOND 8 BITS

Since the signed comparison in Section 5.1 does not give a equality comparison result, using a method similar to the one in section 4.3 will usually be the most convenient way to extend the signed comparsion beyond 8 bits.

Example 6.1: a 16-bit signed comparison which leaves the usual signed comparison result in the N flag

```
        LDA NUM1L ; NUM1-NUM2
        CMP NUM2L
        LDA NUM1H
        SBC NUM2H
        BVC LABEL ; N eor V
        EOR #$80
LABEL
```

Notice that the V flag isn't needed until after the high bytes have been

subtracted, so the low bytes can be subtracted with a CMP instruction and an SEC instruction isn't needed.

Example 6.2: a 24-bit signed comparison which leaves the usual signed comparison result in the N flag

```
        LDA NUM1L ; NUM1-NUM2
        CMP NUM2L
        LDA NUM1M
        SBC NUM2M
        LDA NUM1H
        SBC NUM2H
        BVC LABEL ; N eor V
        EOR #$80
LABEL
```

It is possible to take an approach similar to the ones in Sections 4.1 and 4.2 which compared one byte a time. This may be occasionally useful. However, some additional effort is necessary to obtain the equality comparison result (remember that both of those approaches made use of the equality comparison result). One way to do this is shown in the following example.

Example 6.3: a 16-bit signed comparison that branches to LABEL4 if NUM1 < NUM2 (similar to Example 4.1.1 in Section 4.1)

```
        SEC
        LDA NUM1H  ; compare high bytes
        SBC NUM2H
        BVC LABEL1 ; the equality comparison is in the Z flag here
        EOR #$80   ; the Z flag is affected here
LABEL1  BMI LABEL4 ; if NUM1H < NUM2H then NUM1 < NUM2
        BVC LABEL2 ; the Z flag was affected only if V is 1
        EOR #$80   ; restore the Z flag to the value it had after SBC NUM2H
LABEL2  BNE LABEL3 ; if NUM1H <> NUM2H then NUM1 > NUM2 (so NUM1 >= NUM2)
        LDA NUM1L  ; compare low bytes
        SBC NUM2L
        BCC LABEL4 ; if NUM1L < NUM2L then NUM1 < NUM2
LABEL3
```

APPENDIX A: A PROGRAM THAT DEMONSTRATES THAT THE SIGNED COMPARISON WORKS

The signed comparison routine was presented above without any proof that it is correct. The following program demonstrates that the signed comparison works as described. This program below was not written for maximum speed (it takes about 3 seconds to complete at 1 MHz) or minimal space; it was written to be easy to understand and easy to modify or extend (so that other claims above can be tested if desired). Also, it was written to provide a template for writing other test/proof-style programs.

The approach is simple: compare two (8-bit) numbers, N1 and N2, testing all 256 possible values of N1 and N2. There are four cases:

1. N1 positive ($00 to $7F), N2 positive ($00 to $7F)
2. N1 positive ($00 to $7F), N2 negative ($80 to $FF)
3. N1 negative ($80 to $FF), N2 positive ($00 to $7F)

4. N1 negative ($80 to $FF), N2 negative ($80 to $FF)

For case 1, remeber that $00 to $7F = 0 to 127 for both signed and unsigned numbers. So the signed comparison result is simply checked against the unsigned comparison result.

For case 2, N1 will always be greater than N2, so the signed comparison result is checked.

For case 3, N1 will always be less than N2, so the signed comparison result is checked.

For case 4, note a useful property of signed numbers: in the range $80 to $FF, the smallest number is $80 (-128) and the largest number is $FF (-1). For unsigned numbers, in the range $80 to $FF, the smallest number is also $80 (128) and the largest number is also $FF (255), so the signed comparison result can be verified by simply checking it against the unsigned comparison result, like case 1. In other words, the signed numbers $80 (-128) and $81 (-127) do not represent the same numbers that the unsigned numbers $80 (128) and $81 (129) do, but $80 < $81 regardless of whether $80 and $81 are signed or unsigned numbers.

One slightly sneaky part of the program below is the use of TSX and TXS. First, a 1 is stored in ERROR until all tests pass. Then TSX is used to save the stack pointer during TEST. Then, if test fails in a subroutine, a TXS followed by an RTS will return not to the loops in TEST, but to whatever called TEST, and ERROR will contain 1, and N1 and N2 will contain the values they had when the test failed. This comes in handy for debugging.

```
; Test the signed compare routine
;
; Returns with ERROR = 0 if the test passes, ERROR = 1 if the test fails
;
; Three (additional) memory locations are used: ERROR, N1, and N2
; These may be located anywhere convenient in RAM
;
TEST    CLD         ; Clear decimal mode for test
        LDA #1
        STA ERROR   ; Store 1 in ERROR until test passes
        TSX         ; Save stack pointer so subroutines can exit with ERROR = 1
;
; Test N1 positive, N2 positive
;
        LDA #$00    ; 0
        STA N1
PP1     LDA #$00    ; 0
        STA N2
PP2     JSR SUCMP   ; Verify that the signed and unsigned comparison agree
        INC N2
        BPL PP2
        INC N1
        BPL PP1
;
; Test N1 positive, N2 negative
;
        LDA #$00    ; 0
        STA N1
```

```
PN1     LDA #$80   ; -128
        STA N2
PN2     JSR SCMP   ; Signed comparison
        BMI TEST1  ; if N1 (positive) < N2 (negative) exit with ERROR = 1
        INC N2
        BMI PN2
        INC N1
        BPL PN1
;
; Test N1 negative, N2 positive
;
        LDA #$80   ; -128
        STA N1
NP1     LDA #$00   ; 0
        STA N2
NP2     JSR SCMP   ; Signed comparison
        BPL TEST1  ; if N1 (negative) >= N2 (positive) exit with ERROR = 1
        INC N2
        BPL NP2
        INC N1
        BMI NP1
;
; Test N1 negative, N2 negative
;
        LDA #$80   ; -128
        STA N1
NN1     LDA #$80   ; -128
        STA N2
NN2     JSR SUCMP  ; Verify that the signed and unsigned comparisons agree
        INC N2
        BMI NN2
        INC N1
        BMI NN1

        LDA #0
        STA ERROR  ; All tests pass, so store 0 in ERROR
TEST1   RTS

; Signed comparison
;
; Returns with:
;   N=0 (BPL branches) if N1 >= N2 (signed)
;   N=1 (BMI branches) if N1 < N2 (signed)
;
; The unsigned comparison result is returned in the C flag (for free)
;
SCMP    SEC
        LDA N1     ; Compare N1 and N2
        SBC N2
        BVC SCMP1  ; Branch if V = 0
        EOR #$80   ; Invert Accumulator bit 7 (which also inverts the N flag)
SCMP1   RTS

; Test the signed and unsigned comparisons to confirm that they agree
;
SUCMP   JSR SCMP   ; Signed (and unsigned) comparison
        BCC SUCMP2 ; Branch if N1 < N2 (unsigned)
        BPL SUCMP1 ; N1 >= N2 (unsigned), branch if N1 >= N2 (signed)
        TSX        ; reset stack and exit with ERROR = 1
SUCMP1  RTS
SUCMP2  BMI SUCMP3 ; N1 < N2 (unsigned), branch if N1 < N2 (signed)
        TSX        ; reset stack and exit with ERROR = 1
SUCMP3  RTS
```

Last Updated April 3, 2004.