DIY Calculator: BCD Instructions

Introduction

The purpose of this paper is to document some new Binary Coded Decimal (BCD) instructions and opcodes that have been added to the DIY Calculator's CPU.

A Brief Overview of BCD

BCD is introduced in detail on the "More Cool Stuff" page of the DIY Calculator's website at **www.DIYCalculator.com**. This overview is intended only to "set the scene" for the following discussions.

The idea underlying BCD is that a group of four bits (binary digits) – known as a nybble or nibble – are used to represent the decimal digits '0' through '9' (Figure 1).

Decimal	BCD	
0	0 0 0 0	
1	0 0 0 1	
2	0 0 1 0	
3	0 0 1 1	
4	0 1 0 0	
5	0 1 0 1	
6	0 1 1 0	
7	0 1 1 1	
8	1 0 0 0	
9	1 0 0 1	
-	1 0 1 0	Unused
-	1 0 1 1	Unused
-	1 1 0 0	Unused
-	1 1 0 1	Unused
-	1 1 1 0	Unused
-	1 1 1 1	Unused

Figure 1. Decimal to BCD mapping

As we see, six of the binary combinations are unused, which makes this format somewhat wasteful of memory, but there are also advantages to BCD as discussed on the main website. If only a single BCD digit is stored in an 8-bit byte, the most significant nybble (four bits) would contain 0s. Such a format, which would be referred to as "unpacked BCD," would be incredibly wasteful of memory. Thus, one would almost invariably use a "packed BCD" format, in which both nybbles in a byte are used to store BCD digits (Figure 2).

Figure 2. Two BCD digits packed into an 8-bit byte

 Copyright © 2005 Clive "Max" Maxfield and Alvin Brown. All rights reserved.

For the purposes of these discussions, we shall consider only integers (other representations such as floating-point are discussed in more detail on the main website). If we initially consider only a single byte containing two BCD digits, then in the case of “unsigned BCD” (which is used to represent only positive integers), these two digits can be used to represent values in the range 00 to 99, which directly equates to 0 to +99 in decimal (Figure 3).

BCD	Decimal
0 0	+0
0 1	+1
0 2	+2
:	:
0 9	+9
1 0	+10
1 1	+11
1 2	+12
:	:
9 7	+97
9 8	+98
9 9	+99

Figure 3. Unsigned BCD

If we wish to represent both positive and negative BCD values, there are several techniques open to us. One is to use a *sign-magnitude* format, in which case the first digit us used to represent the sign (0 = positive and 9 = negative), while the remaining digit is used to represent a magnitude. In this case, we can only represent values in the range –9 to +9 (Figure 4). (As for any sign-magnitude format, we have to deal with the fact that we have both negative and positive representations of zero.)

BCD	Decimal
0 0	+0
0 1	+1
0 2	+2
:	:
0 7	+7
0 8	+8
0 9	+9
9 0	–0
9 1	–1
9 2	–2
:	:
9 7	–7
9 8	–8
9 9	–9

Figure 4. Sign-magnitude BCD

Alternatively, if we use a true tens complement formal, then the most significant digit represents both a sign and a quantity, in which case we can represent values in the range –50 to +49 (Figure 5). As we see, BCD values 00 through 49 represent 0 through +49 in decimal; a BCD value of 50 equates to –50 in decimal; a BCD value of 51 equates to –50 + 1 = –49 in decimal; a BCD value of 52 equates to –50 + 2 = –48 in decimal; and so on up to a BCD value of 99, which equates to –50 + 49 = –1 in decimal.

 Copyright © 2005 Clive “Max” Maxfield and Alvin Brown. All rights reserved.

BCD	Decimal
0 0	+0
0 1	+1
0 2	+2
:	:
4 7	+47
4 8	+48
4 9	+49
5 0	–50
5 1	–49
5 2	–48
:	:
9 7	–3
9 8	–2
9 9	–1

Figure 5. Tens complement BCD

Similarly, if we were to work with 2-byte values which can contain four BCD digits, then in the case of unsigned BCD, we can represent 0000 to 9999, which equates 0 to +9999 in decimal; in the case of sign-magnitude BCD, we can represent values in the range –999 to +999 in decimal; and in the case of tens complement BCD, we can represent values in the range –5,000 to +4,999 in decimal.

It does take a little effort to wrap one's brain around this tens complement concept, but it essentially works in exactly the same manner as twos complement in binary. Don't worry; you'll soon get the hang of it.

Introducing the New BCD Instructions

In order to facilitate experiments with binary coded decimal, we've augmented the DIY calculator with four new instructions, each of which support three addressing modes (immediate, absolute, and indexed). The opcodes associated with these instructions are as shown in Figure 6.

Mnemonic	Addressing Modes			Comment
	imm	abs	abs-x	
DADD	$48	$49	$4A	BCD Add without carry
DADDC	$68	$69	$6A	BCD Add with carry
DSUB	$88	$89	$8A	BCD Subtract without carry
DSUBC	$B8	$B9	$BA	BCD Subtract with carry

Figure 6. New BCD instructions and associated opcodes

These "decimal" `DADD`, `DADDC`, `DSUB`, and `DSUBC` instructions correspond to the existing binary `ADD`, `ADDC`, `SUB`, and `SUBC` instructions, except that they work with true tens complement BCD values. For example:

$27 + $15 = $3C in binary using the `ADD` instruction
$27 + $15 = $42 in BCD using the `DADD` instruction

 Copyright © 2005 Clive "Max" Maxfield and Alvin Brown. All rights reserved.

One way to visualize the way in which these instructions work is to assume that the CPU has a nybble-carry flag that is used to store the carry-out from operations on the least-significant nybbles of the BCD values. Keeping this in mind, we may consider the new instructions to work as follows:

The DADD and DADDC Instructions

When the `DADD` instruction requests a new byte to be added to the existing contents of the accumulator:

1) The two least-significant (LS) nybbles are added together with a carry-in of 0 and the result is stored in the LS nybble of the accumulator. If the result of this addition is greater than 9, then we subtract 10 from this result and set the nybble-carry flag to 1; otherwise we leave the result as-is and clear the nybble-carry flag to 0.
2) The two most-significant (MS) nybbles are added together along with the contents of the nybble-carry flag. If the result of this addition is greater than 9, then we subtract 10 from this result and set the main carry (C) flag to 1; otherwise we leave the result as-is and clear the main carry (C) flag to 0.
3) If the signs of the two numbers to be added together are different (one is positive and the other is negative), then the overflow flag is automatically cleared to 0, irrespective of the sign of the result. However, if the signs of the numbers to be added together are the same (both positive or both negative), then if the sign of the resulting value in the accumulator is different to the sign of the original value in the accumulator, then the overflow flag is set to 1; otherwise it is cleared to 0.

The only difference between the `DADD` and `DADDC` instructions is that, with regard to step 1 above, in the case of a `DADDC`, the two least-significant nybbles are added together along with the current contents of the main carry (C) flag (as opposed to a carry-in of 0).

In the case of multi-byte BCD additions, the LS bytes are added using a `DADD` instruction, and then subsequent bytes are added using `DADDC` instructions.

Unsigned and tens complement BCD additions work in exactly the same way. If the values being added together are considered to represent unsigned BCD quantities, then – at the end of a multi-byte addition – the overflow (O) flag is ignored and the carry (C) flag being cleared to 0 indicates that the result was small enough to fit in the allocated field, while a carry-out of 1 indicates that the result was too large. By comparison, if the values being added together are considered to represent signed (tens complement) BCD quantities, then – at the end of a multi-byte addition – the carry flag is ignored and the overflow flag being cleared to 0 indicates that the result could fit in the allocated field, while an overflow of 1 indicates that the result was too large or two small to fit in the field.

As a worked example, consider the following (assume these represent – and are stored inside the computer as – signed [tens complement] BCD values):

$$\begin{array}{r} 1362 \\ \underline{+\ 2257} \\ =3619 \end{array}$$

In this case, the way we would perform this operation would be as follows:

Copyright © 2005 Clive “Max” Maxfield and Alvin Brown. All rights reserved.

- Load the accumulator with $62.
- Use a `DADD` instruction to add $57 with a carry-in of 0.
- Store this LS byte of the result.
- Load the accumulator with $13.
- Use a `DADDC` instruction to add $22 along with the current contents of the carry flag.
- Store this MS byte of the result.

The DSUB and DSUBC Instructions

When the `DSUB` instruction requests a new byte to be subtracted from the existing contents of the accumulator:

1) The new value is first subtracted from a value of $99 to generate its nines complement (see notes below).
2) The LS nybble from the nines complement value is added to the LS nybble in the accumulator together with a carry-in of 1 (see notes below) and the result is stored in the LS nybble of the accumulator. If the result of this addition is greater than 9, then we subtract 10 from this result and set the nybble-carry flag to 1; otherwise we leave the result as-is and clear the nybble-carry flag to 0.
3) The MS nybble from the nines complement value is added to the MS nybble in the accumulator along with the contents of the nybble-carry flag. If the result of this addition is greater than 9, then we subtract 10 from this result and set the main carry (C) flag to 1; otherwise we leave the result as-is and clear the main carry (C) flag to 0.
4) If the signs of the two original numbers (before we took the nines complement) are the same (both positive or both negative), then the overflow flag is automatically cleared to 0, irrespective of the sign of the result. However, if the signs of the numbers to be subtracted are different (one positive and the other negative negative), then if the sign of the resulting value in the accumulator is different to the sign of the original value in the accumulator, then the overflow flag is set to 1; otherwise it is cleared to 0.

The only difference between the `DSUB` and `DSUBC` instructions is that, with regard to step 1 above, in the case of a `DSUBC`, the two least-significant nybbles are added together along with the current contents of the main carry (C) flag (as opposed to a carry-in of 1).

In the case of multi-byte BCD subtractions, the LS bytes are subtracted using a `DSUB` instruction, and then subsequent bytes are subtracted using `DSUBC` instructions.

Unsigned and tens complement BCD subtractions work in exactly the same way. If the values being subtracted are considered to represent unsigned BCD quantities, then – at the end of a multi-byte subtraction – the overflow (O) flag is ignored and the carry (C) flag being set to 1 indicates that the result could fit in the allocated field, while a carry-out of 0 indicates that the result could not fit in the allocated field. By comparison, if the values being subtracted are considered to represent signed (tens complement) BCD quantities, then – at the end of a multi-byte subtraction – the carry flag is ignored and the overflow flag being cleared to 0 indicates that the result could fit in the allocated field, while an overflow of 1 indicates that the result was too large or two small to fit in the field.

Copyright © 2005 Clive “Max” Maxfield and Alvin Brown. All rights reserved.

As a worked example, consider the following (assume these represent – and are stored inside the computer as – signed [tens complement] BCD values):

```
  6735
– 2352
= 4383
```

- o Load the accumulator with $35.
- o Use a `DSUB` instruction to subtract $52 with a carry-in of 1.
- o Store this LS byte of the result.
- o Load the accumulator with $67.
- o Use a `DSUBC` instruction to subtract $23 along with the current contents of the carry flag.
- o Store this MS byte of the result.

The way this operation is performed inside the CPU is quite interesting. Remember that 6,735 – 2,352 is the same as saying 6,735 + (–2,352), and that this is the same as saying 6,735 + (the tens complement of 2,352).

Also remember that we can generate the nines complement of a decimal value by subtracting it from a value of all nines; thus, the nines complement of 2,352 is 9,999 – 2,352 = 7,647. Furthermore, the tens complement of a decimal value is its nines complement + 1; thus, the tens complement of 2,352 = 7,647 + 1 = 7,648.

So now we have the following progression:

```
        6.735 – 2,352
=       6.735 + (–2,352)
=       6.735 + (the tens complement of 2,352)
=       6,753 + (the nines complement of 2,352 + 1)
=       6.735 + 7647 + 1
=      14,383
=       4,383 (when we drop the final carry)
```

Keeping this in mind, the way the CPU actually performs the above operations is as follows

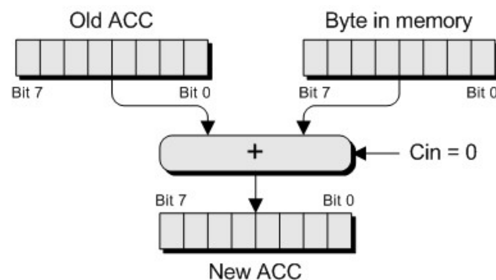
- o Load the accumulator with $35.
- o Use a `DSUB` instruction to subtract $52 with a carry-in of 1.
  - Internally, the CPU subtracts $52 from $99 to give $47, which is the nines complement of $52. The CPU then adds this value to the accumulator (using a BCD add of course) along with a carry-in set to 1 (where this carry-in of 1 is what gives us the tens complement of the number to be subtracted).
- o Store this LS byte of the result.
- o Load the accumulator with $67.
- o Use a `DSUBC` instruction to subtract $23 along with the current contents of the carry flag.
  - Internally, the CPU subtracts $23 from $99 to give $76, which is the nines complement of $23. The CPU then adds this value to the accumulator (using a BCD add of course) along with the current contents of the carry flag.
- o Store this MS byte of the result.

 Copyright © 2005 Clive “Max” Maxfield and Alvin Brown. All rights reserved.

DADD (Decimal [BCD] add without carry)

Description

This instruction adds the contents of a byte of data in memory to the current contents of the accumulator and stores the result in the accumulator (the contents of the memory are not affected). Both data values are assumed to be in BCD format and they are added using a BCD process as discussed earlier in this document. Note that the result is not affected by the contents of the carry flag, because the carry-in to the ALU is forced to logic 0. See also the corresponding `DADDC` instruction.



Addressing modes

Mode	#Bytes	Opcode	Example code	Comments
imm	2	$48	`DADD $03`	BCD add $03 to the ACC.
abs	3	$49	`DADD [$4C76]`	BCD add the contents of memory location $4C76 to the ACC
abs-x	3	$4A	`DADD [$4C76,X]`	BCD add the contents of a memory location to the ACC, where the address of the memory location is $4C76 plus the contents of the X register

Flags affected

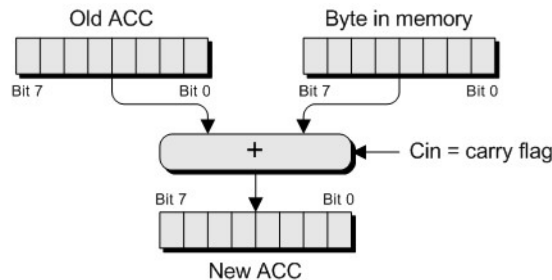
O	Set to 1 if the result overflows; otherwise cleared to 0
N	Set to 1 if the result is >= $50; otherwise cleared to 0
Z	Set to 1 if both nybbles in the result are 0; otherwise cleared to 0
C	Set to 1 if there is a carry out from the addition; otherwise cleared to 0

Copyright © 2005 Clive "Max" Maxfield and Alvin Brown. All rights reserved.

DADDC (Decimal [BCD] add with carry)

Description

This instruction adds the contents of a byte of data in memory (along with the current contents of the carry flag) to the current contents of the accumulator and stores the result in the accumulator (the contents of the memory are not affected). Both data values are assumed to be in BCD format and they are added using a BCD process as discussed earlier in this document. See also the corresponding `DADD` instruction.



Addressing modes

Mode	#Bytes	Opcode	Example code	Comments
imm	2	$68	`DADDC $03`	BCD add $03 to the ACC
abs	3	$69	`DADDC [$4C76]`	BCD add the contents of memory location $4C76 to the ACC
abs-x	3	$6A	`DADDC [$4C76,X]`	BCD add the contents of a memory location to the ACC, where the address of the memory location is $4C76 plus the contents of the X register

Flags affected

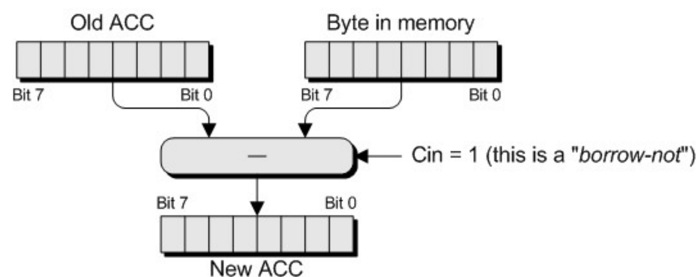
O	Set to 1 if the result overflows; otherwise cleared to 0
N	Set to 1 if the result is >= $50; otherwise cleared to 0
Z	Set to 1 if both nybbles in the result are 0; otherwise cleared to 0
C	Set to 1 if there is a carry out from the addition; otherwise cleared to 0

 Copyright © 2005 Clive “Max” Maxfield and Alvin Brown. All rights reserved.

DSUB (Decimal [BCD] subtract without carry)

Description

This instruction subtracts the contents of a byte of data in memory from the current contents of the accumulator and stores the result in the accumulator (the contents of the memory are not affected). Both data values are assumed to be in BCD format and they are subtracted using a BCD process as discussed earlier in this document. Note that the result is not affected by the contents of the carry flag, because the carry-in to the ALU is forced to logic 1 (the carry-in is really a "*borrow-not*" in this case). See also the corresponding `DSUBC` instruction.



Note: The diagram above is a stylized representation of the action of the `SUB` instruction. In reality, the CPU doesn't have a "subtractor block," so the actual operation that is performed to achieve the desired result is as follows (where the +1 on the far right-hand side of the equation comes from the `Cin` ("carry-in") signal):

```
new_ACC[7:0] = old_ACC[7:0] + ($99 – byte_in_memory[7:0]) + 1
```

Addressing modes

Mode	#Bytes	Opcode	Example code	Comments
imm	2	$88	`DSUB $03`	BCD subtract $03 from the ACC.
abs	3	$89	`DSUB [$4C76]`	BCD subtract the contents of memory location $4C76 from the ACC
abs-x	3	$8A	`DSUB [$4C76,X]`	BCD subtract the contents of a memory location from the ACC, where the address of the memory location is $4C76 plus the contents of the X register

Flags affected

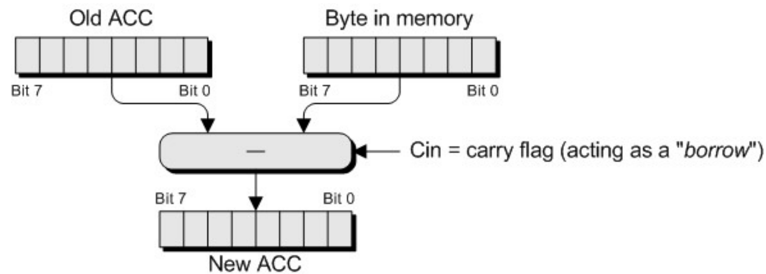
O	Set to 1 if the result overflows; otherwise cleared to 0
N	Set to 1 if the result is >= $50; otherwise cleared to 0
Z	Set to 1 if both nybbles in the result are 0; otherwise cleared to 0
C	Set to 1 if there is a carry out (really a "*borrow-not*") from the subtraction; otherwise cleared to 0 (which indicates a "*borrow*")

 Copyright © 2005 Clive "Max" Maxfield and Alvin Brown. All rights reserved.

DSUBC (Decimal [BCD] subtract with carry)

Description

This instruction subtracts the contents of a byte of data in memory (along with the current contents of the carry flag) from the current contents of the accumulator and stores the result in the accumulator (the contents of the memory are not affected). Both data values are assumed to be in BCD format and they are subtracted using a BCD process as discussed earlier in this document. See also the corresponding `DSUB` instruction.



Note: The diagram above is a stylized representation of the action of the `SUBC` instruction. In reality, the CPU doesn't have a "subtractor block," so the actual operation that is performed to achieve the desired result is as follows:

```
new_ACC[7:0] = old_ACC[7:0] + ($99 – byte_in_memory[7:0]) + Cin
```

Addressing modes

Mode	#Bytes	Opcode	Example code	Comments
imm	2	$B8	`DSUBC $03`	BCD subtract $03 from the ACC
abs	3	$B9	`DSUBC [$4C76]`	BCD subtract the contents of memory location $4C76 from the ACC
abs-x	3	$BA	`DSUBC [$4C76,X]`	BCD subtract the contents of a memory location from the ACC, where the address of the memory location is $4C76 plus the contents of the X register

Flags affected

O	Set to 1 if the result overflows; otherwise cleared to 0
N	Set to 1 if the result is >= $50; otherwise cleared to 0
Z	Set to 1 if both nybbles in the result are 0; otherwise cleared to 0
C	Set to 1 if there is a carry out (really a "*borrow-not*") from the subtraction; otherwise cleared to 0 (which indicates a "*borrow*")

 Copyright © 2005 Clive "Max" Maxfield and Alvin Brown. All rights reserved.