Convert binary number to any base

Mitja Rihtarsic - March 21, 2018

Conversion of binary to decimal numbers is often needed in firmware. And it’s done easily enough if multiplication and division by ten are acceptable. However, these operations, especially division, may be too resource-hungry for a low cost microcontroller (MCU).

I was therefore pleased to learn about the double dabble algorithm, which converts binary to decimal efficiently; it went into my bag of tricks immediately. Still, I want to tailor it to my needs – to work with any base. To do that, I needed to know how it actually works.

The double dabble belongs to the second of two groups of algorithms (shift-adjust and adjust-shift). Both can be used to convert binary numbers to other formats. Among them, conversion to the time or angle format of HMS is especially useful. Additionally, there have been ternary (base-3)-related designs & discussions on EDN recently to which this Design Idea is applicable.

The conversion begins with a binary number:

\[ A = 2^{N-1} a_{N-1} + \ldots + 2^n a_n + \ldots + 2 a_1 + a_0 \]  

(1)

where a's are bits. N bits are needed to store the number A. The a's are either 0 or 1. An example number is 11011011₂.

The number A is defined by a sum:

\[ A = B_M p_M + \ldots + B_m p_m + \ldots + B_1 p_1 + p_0 \]  

(2)

The constants \( B_n \) depend on the numerical system. In decimal, these are products of base constants \( b \) as follows in (3). In the definition (2), the \( p \)'s correspond to decimal places.

\[
\begin{align*}
 b_0 &= 10 & B_1 &= b_0 = 10 \\
 b_1 &= 10 & B_2 &= b_1 B_1 = 100 \\
 & \vdots \\
 b_{m-1} &= 10 & B_m &= b_{m-1} B_{m-1} = 10^m \\
 & \vdots 
\end{align*}
\]  

(3)

The decimal system is not the only interesting system. For example, time, in base 60. Now, \( p_0, p_1, \) and \( p_2 \) are seconds, minutes, and hours respectively.

\[
\begin{align*}
 b_0 &= 60 & B_1 &= b_0 = 60 \\
 b_1 &= 60 & B_2 &= b_1 B_1 = 3600 \\
 & \vdots 
\end{align*}
\]  

(4)
Except on mechanical clock dials, time is displayed by Arabic numerals that end at 9, which is far less than 59. Since it is not necessary that all b’s in the (2) be equal, the solution is simple. To write time with Arabic numerals, the next numerical system (5) is used. Places \( p_0 \) and \( p_1 \) together are seconds in the decimal system, \( p_2 \) and \( p_3 \) are minutes, and \( p_4 \) shows hours.

\[
\begin{align*}
b_0 &= 10 & B_1 &= b_0 = 10 \\
b_1 &= 6 & B_2 &= b_1 B_1 = 60 \\
b_2 &= 10 & B_3 &= b_2 B_2 = 600 \\
b_3 &= 6 & B_4 &= b_3 B_3 = 3600 \\
\end{align*}
\]

Another system’s constants could be \( b_0 = 12 \) and \( b_1 = 3 \); A is length in inches, while \( p_2 \), \( p_1 \), and \( p_0 \) are yards, feet, and inches respectively.

Later in the text, a ternary system will be discussed. It has the following base constants:

\[
\begin{align*}
b_0 &= 3 & B_1 &= b_0 = 3 \\
b_1 &= 3 & B_2 &= b_1 B_1 = 9 \\
& \vdots \\
b_{m-1} &= 3 & B_m &= b_{m-1} B_{m-1} = 3^m \\
& \vdots \\
\end{align*}
\]

When numerical systems are discussed, then the places \( p_m \) are integers, which are limited to the interval

\[0 \leq p_m < b_m\] (7)

\( p_m \) must be less than the corresponding base constant \( b_m \). For decimal, \( p_m \) is an integer: \( 0 \leq p_m < 10 \). But the sum (2) holds true also when places \( p_m \) take values beyond the interval (7)! It is not even necessary that \( p \)’s are integers, but we should keep to the positive integers because, well, MCUs.

Let’s assume that \( A \) is a binary number 0xFF and that base constants \( B_m \) belong to the decimal system (3). The equation (2) holds even when \( p_0 = 255 \), \( p_1 = 0 \), and \( p_2 = 0 \). This is an extreme. It holds also when \( p_0 = 25 \), \( p_1 = 13 \), and \( p_2 = 1 \). But normally, the values are \( p_0 = 5 \), \( p_1 = 5 \), and \( p_2 = 2 \). That is, 0xFF = 255.

In code or hardware for conversion between bases, \( R_m \) bits of memory are allocated for each place \( p_m \). Usually, nibbles (\( R_m = 4 \)) or bytes (\( R_m = 8 \)) are used. The allocated memory is big enough when \( 2^{R_m} \geq b_m \) holds. Still, the allocated chunk of memory should be a bit larger to provide some space for bit carrying. Thus, the place \( p_m \) is an integer that is limited to the interval \( 0 \leq p_m < 2^{R_m} \).

Additional space in \( p_m \) eases bit carrying from \( p_m \) to \( p_{m+1} \). What is this bit carrying? It is executed when \( p_{m+1} \) is incremented by one (a bit) and \( p_m \) is decremented by the base constant \( b_m \). Decrementing is possible as long as \( p_m \geq b_m \). Let us have a binary number \( A = 0xDB \) and base constants \( b_m \) of the decimal system (3). Then, let the places be \( p_0 = 2 \), \( p_1 = 0 \), and \( p_2 = 19 \). We can see that the equation (2) holds. We can also see that at least five bits are necessary to store \( p_0 \). To carry from \( p_0 \) to \( p_1 \), the place \( p_1 \) is incremented by 1 and \( p_0 \) is decremented by \( b_0 = 10 \). The \( p_0 \) and \( p_1 \) are adjusted. Now, the places are \( p_2 = 2 \), \( p_1 = 1 \), and \( p_0 = 9 \). That is, 0xDB = 219. During the bit
carrying, the equation (2) is always valid.

As said, the bit carrying consists of two steps. In one step, the place \( p_{m+1} \) is incremented. In the other step, the base constant \( b_m \) is subtracted from \( p_m \). The two steps can be reduced into one by combining the two places \( p_{m+1} \) and \( p_m \) into one number \( 2^{km} p_{m+1} + p_m \). This number already exists if \( p \)'s are allocated successively in the MCU's memory. That is, two successive bytes form one unsigned int. Then, a carry constant \( 2^{km} - b_m \) is added to this new number. The effect can be seen in (8). The higher byte is incremented by one and the lower byte is decremented by \( b_m \).

\[
p_m \geq b_m ((2^{km} p_{m+1} + p_m) + (2^{km} - b_m) = 2^{km}(p_{m+1} + 1) + (p_m - b_m) \quad (8)
\]

The order of adjusting places \( p_m \) in (2) plays no role. Either pair of places \( p_0, p_1 \) or \( p_m, p_{M-1} \), or any other pair, can be adjusted first. It is because equation (2) holds after any bit carrying. Despite that, it is beneficial that adjustment is executed from \( p_0 \) upwards. This way, no \( p_m \) needs to be adjusted twice. By repeating adjustments, values of individual \( p \)'s are made smaller until the condition (7) is valid for all \( p_m \). The places \( p_m \) are considered adjusted then.

Conversion of binary to some other system is possible by bit carrying only. The initial binary number \( A \) is moved into \( p_0 \), which must be large enough. Then, bit carrying is repeated until the condition (7) is fulfilled for all \( p \)'s. At this point, the number \( A \) is converted. Although this method works, it is inefficient. To speed up the conversion, left shifting, which has the same effect as multiplying by two, is introduced. Then, shifting and bit carrying are executed sequentially. The bit carrying is executed after each shift to keep the needed size of \( p \)'s as small as possible. At the same time, condition (7) is fulfilled. Because of the sequence, this is called a shift-adjust algorithm.

The algorithm runs in iterations. In the initializing iteration \( i = 0 \), all \( p \)'s are cleared. Because of that, equation (9) is zero. Of course, condition (7) holds for all \( p_m \), too.

\[
B_m p_m(i = 0) + \ldots + B_m p_m(i = 0) + B_1 p_1(i = 0) + p_0(i = 0) = 0 \quad (9)
\]

Now, the equation is multiplied by two. At the same time, bit \( a_{N-1} \), which is the MSb of \( A \) in (1), is added to \( p_0 \). In assembler, this is done by left rotate through carry on \( A \) and \( p_0 \). Memory chunks of other \( p \)'s should be large enough so that \( p \)'s do not overflow during the shifting. They are shifted individually.

\[
2(B_m p_m(i = 0) + \ldots + B_m p_m(i = 0) + B_1 p_1(i = 0) + p_0(i = 0)) + a_{N-1} = a_{N-1} \quad (10)
\]

After shifting, (10), the bit carries are repeated until the conditions in (7) are met for all \( m \). Adjusting is not yet needed after the first shift. But to keep to the shift-adjust algorithm, adjusting of \( p \)'s by bit carrying takes place after each shift. After the first iteration is completed, (7) holds for all \( m \). An equation (11) is obtained, which is an input to the second iteration.

\[
B_m p_m(i = 1) + \ldots + B_m p_m(i = 1) + B_1 p_1(i = 1) + p_0(i = 1) = a_{N-1} \quad (11)
\]

Now, shifting and adjusting take place again.

\[
B_m p_m(i = 2) + \ldots + B_m p_m(i = 2) + B_1 p_1(i = 2) + p_0(i = 2) = 2a_{N-1} + a_{N-2} \quad (12)
\]

The sequence is repeated \( N \)-times.

\[
B_m p_m(i = N) + \ldots + B_m p_m(i = N) + B_1 p_1(i = N) + p_0(i = N) = 2^{N-1} a_{N-1} + \ldots + 2 a_1 + a_0 \quad (13)
\]

After the last bit carrying is over, binary number \( A \) is on the right side of (13). On the left side, \( A \) is
converted into numerals of the desired numerical system. These numerals are stored in the p's.

**Examples, ternary, & code**

**Examples**

In **Example 1**, a binary number $A = 0xDB$ is converted into decimal, using the structure $p_2,p_1,p_0:A$. For each $p$, we allocate one byte (it is convenient to allocate bytes in 8-bit MCUs).

As we can see in **shift 1**, the MSb from binary number $A$ is shifted into $p_0$. After this shift, no bit carrying is needed in **adjust**, because conditions (7) for all $p$'s are already fulfilled. For the first time, bit carrying is executed in the **adjust 4**. There, a carry constant, $2^n - 10$, is added to a two-byte structure $0x000D$ to get $0x0103$. This way, the bit carrying or adjusting is done on places $p_0$ to $p_1$ as explained in (8). After **adjust 8**, there are decimal digits in the $p$'s.

**Example 1** Converting $0xDB$ to 219 by the shift-adjust algorithm.
<table>
<thead>
<tr>
<th>shift</th>
<th>00.00.00:DB</th>
</tr>
</thead>
<tbody>
<tr>
<td>adjust</td>
<td>00.00.01:B6</td>
</tr>
<tr>
<td>shift</td>
<td>00.00.03:6C</td>
</tr>
<tr>
<td>adjust</td>
<td>00.00.06:D8</td>
</tr>
<tr>
<td>shift</td>
<td>00.00.0D:B0</td>
</tr>
<tr>
<td>adjust</td>
<td>00.02.07:60</td>
</tr>
<tr>
<td>shift</td>
<td>00.04.0E:C0</td>
</tr>
<tr>
<td>adjust</td>
<td>00.0A.09:80</td>
</tr>
<tr>
<td>adjust</td>
<td>01.00.09:80</td>
</tr>
<tr>
<td>shift</td>
<td>02.00.13:00</td>
</tr>
<tr>
<td>adjust</td>
<td>02.01.09:00</td>
</tr>
</tbody>
</table>

We now have a procedure for converting binary numbers to other bases, but it can be made even more code-efficient if we switch *shifting* and *adjusting*, which we’ll call an adjust-shift algorithm. In
each iteration, all necessary bit carrying is executed first, then positions $p_m$ are shifted. After both steps are over, the condition (7) must be fulfilled. Since the last step in an iteration is shifting, $p_m$'s must be properly adjusted before that. Therefore, the bit carry must be executed if $p_m(i) \geq b_m/2$. In this case, the carry constant $(2^{Rm} - b_m)/2$ is added to $p_m$, as in (14).

$$p_m \geq b_m/2, \quad p_m + (2^{Rm} - b_m)/2 = 2^{Rm-1} + (2 p_m - b_m)/2$$

(14)

At the beginning of each iteration of the adjust-shift algorithm, all places $p_m$ comply to (7). Because of that, the $(2 p_m - b_m)/2$ from (14) is always less than $b_m/2$. Furthermore, the $b_m/2$ is less than $2^{Rm-1}$. Consequently, the $p_m$ does not overflow when the carry constant $(2^{Rm} - b_m)/2$ is added to it. That is, the right side of (14) is always less than $2^{Rm}$. At the same time, adding the carry constant always sets an MSb, as we can see in (14). This MSb increments $p_{m+1}$ during the following shifting phase of the iteration. The MSb from $p_m$ must be carried to the LSb of $p_{m+1}$. Because of that, places of $p_m$ in adjust-shift algorithms must be shifted as one big number.

Let’s highlight the differences between the shift-adjust and adjust-shift algorithms. In the shift-adjust algorithm, two successive $p_m$'s take place in each bit carrying. On the other hand, $p_m$'s are adjusted individually in adjust-shift algorithm. Shifting is a different story. In the shift-adjust algorithm, each $p_m$ is shifted individually. An exception is $p_0$, which is shifted together with $A$. On the other hand, all $p$'s and $A$ are shifted as one big binary number in adjust-shift.

The carry constant $(2^{Rm} - b_m)/2$ is the reason why the adjust-shift algorithm is suitable for even bases only. If the $b_m$ is odd as in the ternary system, the carry constant would not be integer anymore.

The double dabble algorithm is an adjust-shift algorithm. It converts from binary to decimal ($b_m = 10$) and its places $p$ are stored in nibbles ($R_m = 4$). Therefore, bit carry must be executed when $p_m \geq 5$ and the carry constant is $(2^{Rm} - b_m)/2 = 3$. That is why the double dabble is called the add 3 algorithm sometimes. In Example 2, the binary number 0xDB is converted to decimal by the double dabble algorithm.

**Example 2** Converting 0xDB to 219 by the double dabble algorithm.
Usage of the double dabble in 8-bit MCUs has one weakness. Namely, instructions in these MCUs are made to manipulate bytes. Therefore, we are converting from bytes to nibbles and from nibbles
to bytes all the time. But, if bytes are used instead of nibbles to store places $p_n$ ($R_m = 8$), the code would simplify. On the other hand, twice as much memory is needed. But that is not too high a cost.

One such conversion is shown in Example 3, where the binary number represents time in seconds, and will be converted into minutes. The base constants are not equal, now. They are $b_0 = 10$, $b_1 = 6$, and $b_2 = 10$. The place $p_0$ is stored in a byte ($R_0 = 8$). Because of that, its base constant is $b_0 = 10$ and the bit carrying takes place when $p_0 \geq 5$. The carry constant is $(2^{R_0} - b_0)/2 = 0x7B$. The base constant of $p_1$ is $b_1 = 6$. Therefore, the bit carry takes place when $p_1 \geq 3$ and the carry constant is $(2^{R_1} - b_1)/2 = 0x7D$. The base constant of $p_2$ is $b_2 = 10$. The adjustment is like that of $p_0$.

**Example 3** Converting 0xDB to 3 minutes and 39 seconds by the adjust-shift algorithm.
An interesting step in Example 3 is adjust_8. In this step, two bit carries take place. In \( p_1 \), constant 0xFD is added to 0x04. \( p_o \) has a different base constant. Therefore, 0xFB is added to 0x09. We can
see that both additions occur inside a byte. The actual bit carries, from $p_0$ to $p_1$ and from $p_1$ to $p_2$, happen during shift.

At the end of Example 3, the binary number 0xDB is converted to 3 minutes and 39 seconds.

**Ternary**

The ternary system (base-3) is not very common, but it has its uses. For instance, binary to ternary conversion is needed for a ternary DAC (e.g., see [Ternary DAC: Greater resolution, less bits](#)). Each DAC input can accept three states: 0, hi-Z, or 1. These translate to 0, 1, and 2 respectively (though that can vary by DAC design).

Since base-3 is odd, the shift-adjust algorithm must be used. Consequently, each place $p_m$ is at least three bits long. In **Example 4**, a whole byte ($R_m = 8$) is allocated for each $p_m$. The initial binary number $A$ is one byte long. To convert it to ternary, six places of $p_m$ are needed. The carry constant is $2^{R_m} - b_m = 0xFD$.

**Example 4** Converting 0xDB to 22010, by the shift-adjust algorithm.
<table>
<thead>
<tr>
<th></th>
<th>00.00.00.00.00:DB</th>
</tr>
</thead>
<tbody>
<tr>
<td>shift_1</td>
<td>00.00.00.00.01:B6</td>
</tr>
<tr>
<td>adjust_1</td>
<td>00.00.00.00.01:B6</td>
</tr>
<tr>
<td>shift_2</td>
<td>00.00.00.00.03:6C</td>
</tr>
<tr>
<td>adjust_2</td>
<td>00.00.00.00.01.00:6C</td>
</tr>
<tr>
<td>shift_3</td>
<td>00.00.00.00.02.00:D8</td>
</tr>
<tr>
<td>adjust_3</td>
<td>00.00.00.00.02.00:D8</td>
</tr>
<tr>
<td>shift_4</td>
<td>00.00.00.00.04.01:B0</td>
</tr>
<tr>
<td>adjust_4</td>
<td>00.00.00.01.01.01:B0</td>
</tr>
<tr>
<td>shift_5</td>
<td>00.00.00.02.02.03:60</td>
</tr>
<tr>
<td>adjust_5</td>
<td>00.00.00.02.03.00:60</td>
</tr>
<tr>
<td>adjust_5</td>
<td>00.00.00.03.00.00:60</td>
</tr>
<tr>
<td>adjust_5</td>
<td>00.00.01.00.00.00:60</td>
</tr>
<tr>
<td>shift_6</td>
<td>00.00.02.00.00.00:C0</td>
</tr>
<tr>
<td>adjust_6</td>
<td>00.00.02.00.00.00:C0</td>
</tr>
<tr>
<td>shift_7</td>
<td>00.00.04.00.00.01:80</td>
</tr>
<tr>
<td>adjust_7</td>
<td>00.01.01.00.00.01:80</td>
</tr>
<tr>
<td>shift_8</td>
<td>00.02.02.00.00.03:00</td>
</tr>
<tr>
<td>adjust_8</td>
<td>00.02.02.00.01.00:00</td>
</tr>
</tbody>
</table>
In Example 4, \(0xDB\) is converted to \(022010_3\). The adjustment in the conversion is special because the bit carrying is repeated three times. First, the place \(p_0\) is adjusted. The bit carry from it to \(p_1\) makes \(p_1 \geq 3\). Because of that, \(p_1\) must be adjusted although it was fine after shift. The story repeats. The bit carry from \(p_1\) to \(p_2\) makes \(p_2 \geq 3\). Only after the third repetition are the \(p\)'s are ready for shift.

The repetition of bit carrying is a peculiarity of conversion from binary to odd bases. Namely, the highest possible numeral in place \(p_m\) of a number in an odd base is even. After shifting, which is the same as multiplying by two, and before adjusting, all places \(p\) are even. If a carry bit is added to this even value, it becomes odd. Then, it is possible that some \(p_m\) become equal to \(b_m\), which are odd. Consequently, these places \(p_m\) must be adjusted. Contrarily, the highest possible numeral in the place \(p_m\) of a number in an even base is odd. After shifting and before adjusting, all \(p_m\) are even. If a carry bit is added to them, they become odd. The condition \(p_m < b_m\) remains fulfilled. Therefore, no repetitions are needed in conversion to even systems.

**Ternary implementation**

As said, an MCU input/output pins can have three possible states: low, hi-Z, and high. The pin is controlled by data and direction bits. With a Microchip PIC, these bits are in LAT (output latch) and TRIS (data direction) registers. TRIS=1 puts the pin into input mode. Each \(p_m\) consists of one bit from LAT and one bit from TRIS registers.

For conversion from binary to ternary \((b_m = 3)\) by shift-adjust, each place \(p_m\) must be at least three bits long \((R_m = 3)\). The bits are named \(c_m\), \(l_m\) and \(t_m\). The bits \(l_m\) and \(t_m\) correspond to flags in LAT and TRIS registers respectively. The place \(p_m\) equals \(p_m = 4c_m + 2l_m + t_m\). These bits are located in three bytes \(C\), \(L\), and \(T\). With the 8-bit MCU, whole bytes are allocated, although only six places \(p_m\) of ternary are needed for conversion of one-byte binary number \(A\).

\[
C = 2^7 c_7 + \ldots + 2^1 c_1 + 2^0 c_0 \\
L = 2^7 l_7 + \ldots + 2^1 l_1 + 2^0 l_0 \\
T = 2^7 t_7 + \ldots + 2^1 t_1 + 2^0 t_0
\]  

(15)

Because each place \(p_m\) is stored in three bits \((R_m = 3)\), the carry constant is \(2^{8m} - b_m = 5\).

The shift-adjust algorithm here is iterative. In each iteration, places \(p_m\) are shifted first. Then, the places are adjusted. The shift is executed simply by placing the byte \(L\) to \(C\) and byte \(T\) to \(L\). Then, the byte \(T\) is cleared. Although carry over from original binary number \(A\) usually takes place in this shift step, it is going to be postponed into the adjust step. Because the ternary system is odd, adjustment may consist of several repetitions. Logic of the first adjustment repetition is shown in **Figure 1**, and logic of later ones is shown in **Figure 2**.
The bits of individual $p_m$ are scattered between bytes. On the other hand, bits of different $p_m$'s are collected in bytes. This is why adjusting is done by logical operations, as in Figures 1 and 2, rather than by comparing and adding. This logic is the basis for Listing 1. In the loop of the code example, there are logic instructions.

Before the shift in the current iteration, value of places $p_m$ is adjusted so that (7) holds. That is, the value in $p_m$ is 0, 1, or 2. After the shift, it is doubled to 0, 2, or 4. This even value is an input to the adjustment logic in Figure 1. The adjustment by bit carrying is executed if $p_m \geq 3$. Therefore, it is executed when $p_m = 4$ only. To execute it, a carry constant of 5 is added as in equation (8). If 5 is added to a structure $3 + p_m$, then $p_m+1$ is incremented by one and $p_m$ set from $p_m = 4$ to $p_m = 1$. Besides the bit carrying, the bit from original number $A$ is added to place $p_0$ during the first repetition of adjustment. Since the number $A$ is shifted left in each iteration, the MSb of $A$ is always
the bit that is added.

After the first repetition, $p_m$ can be 0, 1, 2, or 3. If $p_m < 3$ for all $p_m$'s, then the new repetition is not needed and the next iteration of conversion may begin. In case any of $p_m = 3$, another bit carrying must be executed. If the carry constant of 5 is added to $p_m = 3$, $p_m$ becomes zero and $p_{m+1}$ is incremented. This is done as it is shown in Figure 2. Again, the result can be 0, 1, 2, or 3. So the adjustment executed by logic in Figure 2 is repeated until all $p_m \leq 3$.

When all bits of the number A are shifted to the logic in Figure 1, the conversion of binary number $A$ to ternary is done. The value from byte $t$ is copied to the TRIS register and byte $l$ is copied to LAT. Ideally, both copies should take place at the same time to minimize glitches.

The widespread usage of the venerable double dabble algorithm proves that it is very effective at converting numbers from binary to decimal. As we have seen here, its idea can be expanded to shift-adjust and adjust-shift algorithms, which convert to various numerical systems.

**Related articles:**
- Perform hexadecimal-to-BCD conversion in firmware
- Conversion circuit handles binary or BCD
- C routine speeds decimal-to-binary conversion
- Ternary DAC: Greater resolution, less bits
- Technique increases low-cost DAC's resolution
- Trits vs. Bits: 5-Trit ternary DAC has (almost) 8-bit resolution and high PSRR

—Mitja Rihtarsič has an MSc degree and has been designing BLDC motor controllers and electronics used in compressed air equipment.

**Listing 1** PIC18 binary-to-ternary assembler code

```asm
;**** Code Example: Conversion from binary to ternary ****

;byte “numberA” is input to the conversion routine

movlw 0x08
movwf iCnt,ACCESS ;initializing the iteration counter

clrf byteL,ACCESS ;places p (l, t) are set to zero initially

clrf byteT,ACCESS

iterationLoop:
```


movff byteL,byteC ;shifting
movff byteT,byteL ;shifting
movff byteC,byteT ;Prepare input to xor gate 1 in Figure 1
comf byteC,W,ACCESS
andwf byteL,F,ACCESS ;Prepare input to xor gate 2 in Figure 1
rlcf numberA,F,ACCESS ;bit from number A is put into STATUS.C

adjustmentLoop:

rlcf byteC,F,ACCESS ;carry from pm to pm+1
movf byteC,W,ACCESS
andwf byteT,W,ACCESS ;and gate 3 in Figure 1 and Figure 2
xorwf byteL,F,ACCESS ;xor gate 2 in Figure 1 and Figure 2
movf byteC,W,ACCESS
xorwf byteT,F,ACCESS ;xor gate 1 in Figure 1 and Figure 2
movf byteL,W,ACCESS
andwf byteT,W,ACCESS ;Testing condition p < 3 (all pm)
bz adjustmentDone

movwf byteC,ACCESS
comf WREG,W,ACCESS
andwf byteL,F,ACCESS ;Prepare input to xor gate 2 in Figure 2
andwf byteT,F,ACCESS ;Prepare input to xor gate 1 in Figure 2
bra adjustmentLoop ;STATUS.C remained cleared from “rlcf”

adjustmentDone:

decfsz iCnt,F,ACCESS
bra iterationLoop

;Done. Bytes byteL and byteT are prepared to move to LAT and TRIS registers