TRANSFER OF AN INTEGER BINARY VALUE INTO DECIMAL ASCII STRING

Abstract

The objective of this article is to present the variety of methods available for transfer of an integer binary number into the decimal ASCII string. The common methods often are not the fastest ones and there are some almost forgotten and still reliable methods (by repetitive subtractions and additions) and some less known ones like division via multiplication or binary tree search for the decimal digits.

Although we write from left to right, the weight places in decimal numbers are organized in reverse order, from right to left, and this require minor modifications and additional code in some algorithms. It is much more convenient on paper or papyrus to write from left to right because the process of writing can be visually controlled and the angle of the pen is higher then 90º which facilitate pen's gliding on paper, while writing on the clay has to be performed in reverse order because the angle of the pen, i.e. wedge shaped stick is less then 90º and therefore it is in much better position to scratch and punch the raw clay table. The reverse order of digits in decimal numbers is the legacy of that ancient era of cuneiform writing on clay tables which became quite obvious during the development of these routines for 2→10 radix conversion. This challenges historical theory that decimal numbers appears firstly in India because they wrote from left to right on fabrics and frescoes and therefore decimal numbers could be much older and with the different origin.

The problem itself is mathematically quite clear and essentially it is consisted of radix transition from radix 2 into the radix 10, but its practical solution is not that simple.

There are 6 basic ways to transfer register's binary value into its decimal ASCII string:

1. Successive subtraction of decimal weight coefficients (1, 2, 4, 8, 16, 32, 64, 128, 256, etc.) and countering till the result is being negative: This method yields the decimal digits in direct order with leading zeroes that should be truncated, but the number of these leading zeroes can be vastly reduced by an estimation of the leading zero positions with the error of eventually a single position only. The set of weight coefficients could be stored in an array and then grabbed with LODSB instruction or on the stack and manipulated via POP one. The average number of iterations per decimal digit is 4.5 loops. The method is quite simple and yet it utilizes only fast instructions for integer addition and subtraction. This method was very popular in the era of 8-bit computers when it was almost only available option due to lack of complex assembly instructions. The routines based on that method are EAX2SUB and ESI2ASC ones. EAX2SUB pulls the weight coefficients from an array while ESI2ASC uses the stack. The array based routine is faster than the stack one because it can finish the task earlier, which is not case with the stack based routine. The stack based ESI2ASC routine has to

pull all the coefficients from the stack to be able to return back with RET instruction while the array based EAX2SUB does not have such limitation and it can cancel its main loop at any time. The array based routine is also suitable for one more optimization: the number of decimal digits can be estimated in regards to the most significant bit of the converted binary number which is determined by the BSR instruction. Thereby the NUMDIG array contains the number of certainly leading zeroes in respect to the most significant bit of the converted binary number and that estimation, as aforesaid, can miss for only one leading zero digit's position which utterly eliminates usage of REP SCASB instruction's pair for truncation of leading zero's decimal digits, just as it is done in the ESI2ASC stack based routine. Disadvantage of the array based method is that it alters more registers then the stack's based one and it also uses CMOVZ instruction to clear the undefined state of destination register of BSR instruction when the source register is zero, and this instruction is not available in most embedded systems because it firstly appeared within Pentium Pro processor.

2. Addition of decimal numbers via AAA instruction for unpacked BCD arithmetic: The essence of this approach is consisted of the fact that we can convert the whole decimal string by adding single decimally digits of the binary positions (1, 2, 4, 8, 16, 32, etc) which can be done by BCD instructions. There is a handful AAA instruction that is utilized here and this is instruction for decimal correction after decimal digit per digit addition of decimally unpacked numbers. Decimally unpacked number is consisted of one decimal digit per a byte while decimally packed number is consisted of two decimal digits per byte where every single nibble of the byte holds a decimal digit. So, if we decimally add all numbers whose denote weight binary numbers (1, 2, 4, 8, 16, 32, 64, 128, etc) for proper binary positions, then we will obtain the decimal string of the whole binary input. Example: 25 is 25D = 11001B = 1D + 8D + 16D, 1 + 8 = 9 + 0·10, 9 + 16 = 9 + 6 + 1·10 = 5 + 1·10 + 1·10 = 25. This method is not very fast but it is extremely suitable for VHDL hardware implementation. These software realizations are carefully coded and therefore the current implementations minimize the number of operation by omission of processing of leading zero positions which is done in an attempt to make this advance method more comparable by speed to other theoretically much simpler ones. The routines that demonstrate this method are EDX2DEC and EAX2BCD ones. EDX2DEC routine is entirely based on the AAA machine instruction and usage of this instruction can be avoided by utilization of an array which might be better solution because BCD arithmetic is obsolete in X64 mode and therefore AAA instruction does not exist in the 64-bit world anymore, and also, such array can instantly hold ASCII values of digits making the routine able to process ASCII digits directly. The routine can operate either by the successive shifting of the originating number one binary position right and then to check whether the carry flag is switched on, or by utilization of the BT instruction – there is chosen algorithm with shifting. This method also extracts digits in direct order

with eventually an extra leading zero which truncation is quite simple. EAX2BCD routine that operates without AAA instruction is derived from EDX2DEC one and it does not use obsolete AAA machine instruction. Instead of AAA instruction EAX2BCD routine utilizes the ADDIT array that holds all necessary decimal positions for addition of two single digit decimal numbers with the decimal carry. The EAX2BCD routine also deals directly with ASCII values of decimal digits because the weight array already contains ASCII digits making this routine faster than the EDX2DEC one based on AAA instruction.

3. The extraction via division by 10: This method is very convenient because DIV instruction simultaneously yields both the result and the remainder. Thereby the decimal digits are extracting in reverse order by repetitive division by 10 – the remainder is a single decimal digit and the result is something that is going to be used once again in the following iteration. The digits are extracting in reverse order and therefore they should be storing backward. The method is demonstrated by the EAX2DEC routine. The disadvantage of the DIV assembly instruction is that it alters Zero flag incorrectly and thus zero flag is not switched on whenever the result of the integer division is zero and therefore one additional TEST instruction must be used to check whether the result is zero because this zero result terminates the decimal digits' extraction.
4. The extraction via division by 10 by multiplication: A strange property of the integer numbers is that division by the constant can be achieved via multiplication with another appropriate constant. For the case of the 32-bit variable the multiplication constant is 858993459 ($x/10 \approx (858993459 \cdot (x+1)) \backslash 2^{33}$), for the case of 16-bit arithmetic the constant is 13107 ($x/10 \approx 13107 \cdot ((x+1)) \backslash 2^{17}$) and for 64-bit arithmetic the corresponding formula is $x/10 \approx (3689348814741910323 \cdot (x+1)) \backslash 2^{65}$. Disadvantage of this method is that there is no remainder denoting single decimal digits and therefore it has to be additionally calculated. The advantage is usage of MUL instruction which is much faster then DIV one and therefore the entire subroutine is much faster too. The routine that demonstrates this is EAX2ASC one. The improvement of the speed by the replacement of DIV instruction is huge although it may happen in the future that it will not be the case – the latency of the DIV instruction may reach the latency of MUL instruction and then this meandering approach will be obsolete making EAX2ASC routine even slower than EAX2DEC one. The extraction of the decimal digits is performing in reverse order and therefore the digits have to be storing reversely.
5. The decimal digits can be extracted from binary numbers by the Binary Tree Search method: The theory says that the number of comparisons in binary tree per digit should be equal to $LOG_2(10)$, and in our case the maximal length of a single branch is maximally 4 comparisons ($LOG_2(10)=3.32$). This method can be implemented via double indirect addressing (routine EAX2BTR) or by utilization of pair of complementary CMOVxx instructions (EAXCMOV). The CMOVxx instructions on some processors have a serious flaw in which the source memory address will be read regardless the condition is fulfilled

or not and then the contents will be transferred to the destination location in respect to the condition. In this particular case it means that Access Violation Fault will be launched because the first CMOVC instruction (from the pair of complementary ones) will affect the register holding the indirect memory address of the second CMOVNC instruction which is not expected to be performed at all according the condition but it will be and that launches this fault – this moment is clearly commented in the EAXCMOV routine. Furthermore, the overflow fault may be also triggered in the situation where the source address is calculated by doubling or quadrupling of the addressing register already altered with wrong content. So, the pair of complementary CMOVC/CMOVNC instructions must be kept isolated with preserved address and that requires usage of an extra register. Although the reason for such persistent ignoring of this error is apparently inscrutable, it might be deliberately done because there is a perceivable reason for this failure – on this way the queue of instructions is not flushed and thereby the execution is faster. This routine is consisted of two stages: on the first stage the number of decimal digits is determining and on the second stage the extraction of these digits is performing. As it is mentioned in the above text, the name of the routine which utilizes the double indirect addressing is EAX2BTR and of the routine which uses CMOVC/CMOVNC instructions is EAXCMOV. The LB array contains the guiding rules for the search in binary tree while D array contains the digits weight coefficients used for subtractions during binary tree search. The method of Binary Tree Search requires substantial effort and dedication to be involved in its design and implementation and therefore the proper debugger and patient are necessary and DOS Box environment may help a lot in such situations.

6. Math coprocessor utilization: There is a method that is based on the utilization of the numeric coprocessor. Although the algorithm itself is hidden behind particular coprocessor instruction, this method still remains as a plausible option. The coprocessor contains one handful FBSTP instruction that yields packed BCD digits of the number. So, the integer should be converted into the double precision real number (i.e. to be put it into a coprocessor's register) and then to be transferred back from coprocessor as the packed BCD number. This is regularly doing by only two coprocessor's instructions: FILD and FBSTP respectively. The rest of the job is unpacking these packed decimal digits and to add 48 (Chr$(48)="0") to each one, i.e. to allocate them all correct ASCII values. This process can be performed on two ways – via the array or by SHRD instruction. The SHRD instruction is able to shift pair of registers simultaneously for a nibble (denoting a packed decimal digit), i.e. 4 bits right. The disadvantage of the array utilization is that it simultaneously yields pair of digits while the disadvantage of the SHRD instruction usage is that this instruction is not able to set zero flag properly when both registers are null, which then requires utilization of an extra register. The routine that utilizes an array is the EAX2AST and the routine based on the SHRD instruction is EAX2AFL. The FBST instruction may be very slow depending of the particular processor (its

latency is 164 clocks on new processors) and the unpacking process also stands for while and therefore the usage of the FBST instruction should be carefully evaluated. The advantage of this method is that it is able to simply convert even 64-bit number. This extraction could be performed in both directions and reverse one is better solution when there is no prior knowledge of number of decimal digits that should be extracted. The table based routine also yields the leading zeroes which should be truncated either by additional REPE SCASB instructions pair or by an estimation of decimal digits as it is done in EAX2SUB routine, and the truncating of leading zeroes is significant disadvantage of this method. The number of digits can be exactly predetermined numerically as $1+INT(LOG_{10}(2)\cdot LOG_2(n))$, which is usually doing by the following sequence of coprocessor instructions: FLDLG2, FILD, FYL2X and FISTP followed by INC integer instruction. The disadvantage of this approach is that it launches an overflow error for the zero input because the logarithm of the zero is infinite. This numerical computation is also slow and should be used only on bigger integers where the additional advantage could be achieved later in the main loop.

7. "Lazy" method: This is the method which is not the real one or at least less decent one because it utilizes existing Windows API function Wsprintf that is able to convert integer contents of a register into corresponding decimal ASCII value. The method is tremendously slow and it should be used only when the speed is less limiting factor than the time itself. The routine based on this API call is EAX2WIN one. The Wsprintf is full of various formatting and rich with options of all sorts which make this routine tremendously slow. This routine really raises the fear that various compilers may routinely use it tomorrow...

The demo program also contains the SQR routine which simultaneously yields both the result and the remainder. The explanation of the algorithm is beyond the scope of this article and therefore its explanation of operation will be omitted here, but let say that it is derived from the following formula $(x + \Delta x)^2 = x^2 + 2 \cdot x \cdot \Delta x + (\Delta x)^2$ and basically this is a kind of exhaustion of the quantified interval with Binary Tree Search.

The Windows assembly programming is very meticulous job because Windows environment is much more sensitive than the DOS one and therefore the direction flag must be maintained clear all the time otherwise some Window API call may fail or console print may fail, or some other exotic error may be triggered and for this reason another development environment may help a lot and DOS in DOS Box is close enough.

The another disadvantage of the Windows environment in respect to DOS one is the strong isolation between the DATA and CODE sections which impedes coding of the self modifying routine and also disables the possibility of keeping the routines and their memory variables together in the listing. Also, in an OBJ file usually rather exists .text section than the true .CODE section although existence of the .CODE section should be expected according the common myth available in most of Assembler's textbooks. The ability of keeping the routines gathered with their constants and variables is very important for the Cat & Paste rapid subroutines assembling. These

irrational limitation can be surpassed by the modification of the Text section properties via altering of the .text section's flags trough the section's command of linker: `Link.exe /section:".text",ERW`, which also duly debunks the second myth frequently cited in Assembly textbooks that the self-modifying code is not possible within Windows and on newer processors.

Although this method alters section's read/write property via appropriate linker commands and therefore it should be avoided because its implementation requires certain reader's experience based on the particular assembler's peculiarities (ML, jwasm, etc.) that may distract reader from the essence of the presented algorithms, this particular way is chosen because it offers creation of much more readable listing. All these routines were initially written in DOS and then transferred in Windows environment via WinASm IDE with MASM32 10.0 package and ML 6.14.8444 with Link 5.12.8078. The alternative assembler used here was JWASM v2.04c.

The routines in this text are already coded highly optimized and the numbers of instructions are minimized. Any assembly routine is much faster than any higher language corresponding one just because the contemporary compilers do not utilize all the instructions and do not spread all data trough the available registers. The MMX and XMM registers usually remain unused or just partially populated by the contemporary compilers and the code is exceedingly filled with MOV instructions, thereby carefully crafted assembly routines usually run much faster.

The listing of the aforesaid routines follows:

```
;Author of the program and of all algorithms is Andrija Radovic,
;All Rights Reserved, ©2010.
;This code should be assembled with:
;\masm32\bin\ML.EXE /c /coff /Cp /nologo /I\Masm32\Include ASCIIwin.asm
;and linked with the following line:
;\masm32\bin\LINK.EXE /SUBSYSTEM:CONSOLE /RELEASE /VERSION:4.0 /LIBPATH:"\Masm32\Lib" /section:".text",ERW
ASCIIwin.obj /OUT:ASCIIwin.exe

.686
.MODEL          flat, stdcall
OPTION          casemap :none                                   ;case sensitive
INCLUDE         \masm32\include\windows.inc
INCLUDE         \masm32\include\kernel32.inc
INCLUDELIB      \masm32\lib\kernel32.lib
INCLUDE         \masm32\include\user32.inc
INCLUDELIB      \masm32\lib\user32.lib
INCLUDE         \masm32\include\masm32.inc
INCLUDELIB      \masm32\lib\masm32.lib

;-------------------------------------------------------------------------------------------------------

.DATA
TESTS           DB      "This is the test...", 13, 10, 0
INTRES          DB      0, 0, 0, 0, 0, 0, 0, 0, 0, 0
RESULT          DB      0

;-------------------------------------------------------------------------------------------------------

.STACK  20000

;-------------------------------------------------------------------------------------------------------

.CODE

ASCII:
        INVOKE  StdOut, ADDR TESTS
        CALL    TESTR
        INVOKE  StdOut, ADDR TESTS
        CALL    MAINN
        INVOKE  StdOut, ADDR TESTS
INVOKE  ExitProcess, 0

;-------------------------------------------------------------------------------------------------------

TESTV   DD      0

TESTR   PROC
        MOV     TESTV, 3000000000
        MOV     EDX, TESTV
                CALL    EDX2DEC
        CALL    NEW_ROW
        MOV     EAX, TESTV
                ADD     EAX, 1
                CALL    EAX2SUB
        CALL    NEW_ROW
        MOV     ESI, TESTV
```

```
                ADD     ESI, 2
                CALL    ESI2ASC
        CALL    NEW_ROW
        MOV     EAX, TESTV
                ADD     EAX, 3
                CALL    EAX2ASC
        CALL    NEW_ROW
        MOV     EAX, TESTV
                ADD     EAX, 4
                CALL    EAX2AST
        CALL    NEW_ROW
        MOV     EAX, TESTV
                ADD     EAX, 5
                CALL    EAX2AFL
        CALL    NEW_ROW
        MOV     EAX, TESTV
                ADD     EAX, 6
                CALL    EAX2DEC
        CALL    NEW_ROW
        MOV     EAX, TESTV
                ADD     EAX, 7
                CALL    EAX2BTR
        CALL    NEW_ROW
        MOV     EAX, TESTV
                ADD     EAX, 8
                CALL    EAXCMOV
        CALL    NEW_ROW
        MOV     EAX, TESTV
                ADD     EAX, 9
                CALL    EAX2WIN
        CALL    NEW_ROW
        MOV     EAX, TESTV
                ADD     EAX, 10
                CALL    EAX2BCD
        CALL    NEW_ROW
        RET
TESTR   ENDP

;-------------------------------------------------------------------------------------------------------------

MAINN   PROC
        MOV     TESTV, 1000000000
        LL0:
                ;MOV    EDX, TESTV
                ;CALL   EDX2DEC
                MOV     EAX, TESTV
                ;CALL   EAX2ASC
                ;CALL   EAX2DEC
                ;CALL   EAX2AST
                ;CALL   EAX2AFL
                ;CALL   EAX2BTR
                ;CALL   EAXCMOV
                CALL    EAX2BCD
                CALL    S_EQU
                MOV     EAX, TESTV
                CALL    EAX_SQR
                PUSH    EDX
                ;CALL   EAX2DEC
                ;CALL   EAX2BTR
                ;CALL   EAXCMOV
                CALL    EAX2ASC
                ;CALL   EAX2AST
                ;CALL   EAX2AFL
                ;MOV    ESI, EAX
                ;CALL   ESI2ASC
                CALL    S_PLUS
                ;POP    EAX
                ;CALL   EAX2SUB
                POP     EAX
                ;CALL   EAX2ASC
                CALL    EAX2DEC
                ;CALL   EAX2ASF
                ;CALL   EAX2AST
                ;CALL   EAX2AFL
                CALL    NEW_ROW
                INC     TESTV
                CMP     TESTV, 1000000100
        JLE     LL0
        RET
MAINN   ENDP

;-------------------------------------------------------------------------------------------------------------

SPCL    DB      9, 0

TSPACE  PROC
        INVOKE  StdOut, ADDR SPCL
        RET
TSPACE  ENDP

;-------------------------------------------------------------------------------------------------------------

NEWROWL DB      13, 10, 0

NEW_ROW PROC
        INVOKE  StdOut, ADDR NEWROWL
        RET
NEW_ROW ENDP

;-------------------------------------------------------------------------------------------------------------

EPE     DB      253, " + ", 0

S_PLUS  PROC
        INVOKE  StdOut, ADDR EPE
        RET
S_PLUS  ENDP
```

```
;-----------------------------------------------------------------------------------------------------------

EQE     DB      " = ", 0

S_EQU   PROC
        INVOKE  StdOut, ADDR EQE
        RET
S_EQU   ENDP

;-----------------------------------------------------------------------------------------------------------

D00     DB      02, 1
D01     DB      02, 2
D02     DB      02, 4
D03     DB      03, 8, 0
D04     DB      03, 6, 1
D05     DB      03, 2, 3
D06     DB      04, 4, 6, 0
D07     DB      04, 8, 2, 1
D08     DB      04, 6, 5, 2
D09     DB      05, 2, 1, 5, 0
D10     DB      05, 4, 2, 0, 1
D11     DB      05, 8, 4, 0, 2
D12     DB      05, 6, 9, 0, 4
D13     DB      06, 2, 9, 1, 8, 0
D14     DB      06, 4, 8, 3, 6, 1
D15     DB      06, 8, 6, 7, 2, 3
D16     DB      07, 6, 3, 5, 5, 6, 0
D17     DB      07, 2, 7, 0, 1, 3, 1
D18     DB      07, 4, 4, 1, 2, 6, 2
D19     DB      08, 8, 8, 2, 4, 2, 5, 0
D20     DB      08, 6, 7, 5, 8, 4, 0, 1
D21     DB      08, 2, 5, 1, 7, 9, 0, 2
D22     DB      08, 4, 0, 3, 4, 9, 1, 4
D23     DB      09, 8, 0, 6, 8, 8, 3, 8, 0
D24     DB      09, 6, 1, 2, 7, 7, 7, 6, 1
D25     DB      09, 2, 3, 4, 4, 5, 5, 3, 3
D26     DB      10, 4, 6, 8, 8, 0, 1, 7, 6, 0
D27     DB      10, 8, 2, 7, 7, 1, 2, 4, 3, 1
D28     DB      10, 6, 5, 4, 5, 3, 4, 8, 6, 2
D29     DB      11, 2, 1, 9, 0, 7, 8, 6, 3, 5, 0
D30     DB      11, 4, 2, 8, 1, 4, 7, 3, 7, 0, 1
D31     DB      11, 8, 4, 6, 3, 8, 4, 7, 4, 1, 2

EDX2DEC PROC                                                    ;The routine prints ASCII
        STD                                                     ;decimal content of EDX register
        MOV     DWORD PTR INTRES, 0                             ;via BCD arithmetic AAA
        MOV     DWORD PTR INTRES + 4, 0                         ;instruction.
        MOV     WORD PTR INTRES + 8, 0                          ;Author: Andrija Radovic, ©2011
        LEA     EBX, D00
        LEA     EDI, 9 + INTRES
        PUT_I_DO:
                MOVZX   ECX, BYTE PTR [EBX]
                SHR     EDX, 1
                JNC     PUT_I_END_IF
                        INC     EBX
                        DEC     ECX
                        XOR     AX, AX
                        LEA     EDI, 9 + INTRES
                        PUT_I_DO1:
                                MOVZX   AX, AH
                                ADD     AL, BYTE PTR [EDI]
                                ADD     AL, BYTE PTR [EBX]
                                AAA
                                STOSB
                                INC     EBX
                        LOOPD   PUT_I_DO1
                PUT_I_END_IF:
                ADD     EBX, ECX
                TEST    EDX, EDX
        JNZ     PUT_I_DO
        INC     EDI
        OR      DWORD PTR INTRES, "0000"
        OR      DWORD PTR INTRES + 4, "0000"
        OR      WORD PTR INTRES + 8, "00"
        XOR     EDX, EDX
        CMP     BYTE PTR [EDI], "0"
        SETZ    DL
        ADD     EDX, EDI
        CMP     EDX, OFFSET RESULT
        SETZ    CL
        SUB     EDX, ECX
        CLD
        INVOKE  StdOut, EDX
        RET
EDX2DEC ENDP

;-----------------------------------------------------------------------------------------------------------

ADDIT   DW      "00", "01", "02", "03", "04", "05", "06", "07", "08"
        DW      "09", "10", "11", "12", "13", "14", "15", "16", "17"
        DW      "18", "19", "20", "21", "22", "23", "24", "25", "26"
        DW      "27", "28", "29", "30"
WC      DB      02, "1"
        DB      02, "2"
        DB      02, "4"
        DB      03, "8", "0"
        DB      03, "6", "1"
        DB      03, "2", "3"
        DB      04, "4", "6", "0"
        DB      04, "8", "2", "1"
        DB      04, "6", "5", "2"
        DB      05, "2", "1", "5", "0"
        DB      05, "4", "2", "0", "1"
        DB      05, "8", "4", "0", "2"
        DB      05, "6", "9", "0", "4"
```

```
        DB      06, "2", "9", "1", "8", "0"
        DB      06, "4", "8", "3", "6", "1"
        DB      06, "8", "6", "7", "2", "3"
        DB      07, "6", "3", "5", "5", "6", "0"
        DB      07, "2", "7", "0", "1", "3", "1"
        DB      07, "4", "4", "1", "2", "6", "2"
        DB      08, "8", "8", "2", "4", "2", "5", "0"
        DB      08, "6", "7", "5", "8", "4", "0", "1"
        DB      08, "2", "5", "1", "7", "9", "0", "2"
        DB      08, "4", "0", "3", "4", "9", "1", "4"
        DB      09, "8", "0", "6", "8", "8", "3", "8", "0"
        DB      09, "6", "1", "2", "7", "7", "7", "6", "1"
        DB      09, "2", "3", "4", "4", "5", "5", "3", "3"
        DB      10, "4", "6", "8", "8", "0", "1", "7", "6", "0"
        DB      10, "8", "2", "7", "7", "1", "2", "4", "3", "1"
        DB      10, "6", "5", "4", "5", "3", "4", "8", "6", "2"
        DB      11, "2", "1", "9", "0", "7", "8", "6", "3", "5", "0"
        DB      11, "4", "2", "8", "1", "4", "7", "3", "7", "0", "1"
        DB      11, "8", "4", "6", "3", "8", "4", "7", "4", "1", "2"

EAX2BCD PROC                                                    ;The routine prints ASCII
        MOV     DWORD PTR INTRES, "0000"                        ;decimal content of EDX register
        MOV     DWORD PTR INTRES + 4, "0000"                    ;via BCD arithmetics on the
        MOV     WORD PTR INTRES + 8, "00"                       ;ADDIT array.
        LEA     EBX, WC                                         ;No specific instruction is used.
        LEA     EDI, 9 + INTRES                                 ;Author: Andrija Radovic, ©2011
        EAX2BCD_DO:
                MOVZX   ECX, BYTE PTR [EBX]
                SHR     EAX, 1
                JNC     EAX2BCD_END_IF
                        INC     EBX
                        DEC     ECX
                        MOV     DH, "0"
                        LEA     EDI, 9 + INTRES
                        SUB     EDI, ECX
                        EAX2BCD_DO1:
                                MOVZX   EDX, DH
                                ADD     DL, BYTE PTR [EDI + ECX]
                                ADD     DL, BYTE PTR [EBX]
                                MOV     DX, WORD PTR [2 * EDX + OFFSET ADDIT - 6 * "0"]
                                MOV     BYTE PTR [EDI + ECX], DL
                                INC     EBX
                        LOOPD   EAX2BCD_DO1
                EAX2BCD_END_IF:
                ADD     EBX, ECX
                TEST    EAX, EAX
        JNZ     EAX2BCD_DO
        INC     EDI
        XOR     EDX, EDX
        CMP     BYTE PTR [EDI], "0"
        SETZ    DL
        ADD     EDX, EDI
        CMP     EDX, OFFSET RESULT
        SETZ    CL
        SUB     EDX, ECX
        INVOKE  StdOut, EDX
        RET
EAX2BCD ENDP

;-------------------------------------------------------------------------------------------------------

BASES   DD      1000000000, 100000000, 10000000, 1000000, 100000, 10000, 1000, 100, 10, 1
NUMDIG  DB      9, 9, 9, 8, 8, 8, 7, 7, 7, 6, 6, 6, 6, 5, 5, 5, 4, 4, 4, 3, 3, 3, 3, 2, 2, 2
        DB      1, 1, 1, 0, 0, 0

EAX2SUB PROC                                                    ;The routine prints ASCII
        MOV     DX, "00" - 101H                                 ;decimal content of EAX register
        BSR     EDI, EAX                                        ;by repetitive subtractions
        CMOVZ   EDI, EAX                                        ;with coefficients pulled from
        MOVZX   EDI, BYTE PTR [EDI + OFFSET NUMDIG]             ;array with reducted number
        PUSH    EDI                                             ;of iteration.
        EAX2SUB_DO1:                                            ;Author: Andrija Radovic, ©2011
                MOV     EBX, DWORD PTR [4 * EDI + OFFSET BASES]
                EBX_DO2:
                        INC     DL
                        SUB     EAX, EBX
                JNC     EBX_DO2
                MOV     BYTE PTR [EDI + OFFSET INTRES], DL
                INC     EDI
                MOV     DL, DH
                ADD     EAX, EBX
        JNZ     EAX2SUB_DO1
        POP     EDI
        CMP     BYTE PTR [EDI + OFFSET INTRES], "0"
        SETZ    AL
        LEA     EDI, [EDI + EAX + OFFSET INTRES]
        CMP     EDI, OFFSET INTRES + 10
        SETZ    AL
        SUB     EDI, EAX
        INVOKE  StdOut, EDI
        RET
EAX2SUB ENDP

;-------------------------------------------------------------------------------------------------------

ESI2ASC PROC                                                    ;The routine prints ASCII
        LEA     EDI, INTRES                                     ;decimal content of ESI register
        MOV     AX, "00" - 101H                                 ;by repetitive subtractions
        PUSH    DWORD PTR 0                                     ;with coefficients pulled from
        PUSH    DWORD PTR 1                                     ;stack without counter.
        PUSH    DWORD PTR 10                                    ;Author: Andrija Radovic, ©2011
        PUSH    DWORD PTR 100
        PUSH    DWORD PTR 1000
        PUSH    DWORD PTR 10000
        PUSH    DWORD PTR 100000
        PUSH    DWORD PTR 1000000
        PUSH    DWORD PTR 10000000
```

```
        PUSH    DWORD PTR 100000000
        MOV     EBX, 1000000000
        CLD
        EAS_DO1:
                EAS_DO2:
                        INC     AX
                        SUB     ESI, EBX
                JNC     EAS_DO2
                ADD     ESI, EBX
                STOSB
                MOV     AL, AH
                POP     EBX
                TEST    EBX, EBX
        JNZ     EAS_DO1
        MOV     AL, "0"
        MOV     ECX, 10
        LEA     EDI, INTRES
        REPE    SCASB
        DEC     EDI
        INVOKE  StdOut, EDI
        RET
ESI2ASC ENDP

;-------------------------------------------------------------------------------------------------------------------

ASCII_TABLE     DW      "00", "10", "20", "30", "40", "50", "60", "70", "80", "90", "00", "00"
                DW      "00", "00", "00", "00", "01", "11", "21", "31", "41", "51", "61", "71"
                DW      "81", "91", "00", "00", "00", "00", "00", "00", "02", "12", "22", "32"
                DW      "42", "52", "62", "72", "82", "92", "00", "00", "00", "00", "00", "00"
                DW      "03", "13", "23", "33", "43", "53", "63", "73", "83", "93", "00", "00"
                DW      "00", "00", "00", "00", "04", "14", "24", "34", "44", "54", "64", "74"
                DW      "84", "94", "00", "00", "00", "00", "00", "00", "05", "15", "25", "35"
                DW      "45", "55", "65", "75", "85", "95", "00", "00", "00", "00", "00", "00"
                DW      "06", "16", "26", "36", "46", "56", "66", "76", "86", "96", "00", "00"
                DW      "00", "00", "00", "00", "07", "17", "27", "37", "47", "57", "67", "77"
                DW      "87", "97", "00", "00", "00", "00", "00", "00", "08", "18", "28", "38"
                DW      "48", "58", "68", "78", "88", "98", "00", "00", "00", "00", "00", "00"
                DW      "09", "19", "29", "39", "49", "59", "69", "79", "89", "99", "00", "00"
BCRESUL         DT      0

EAX2AST PROC                                                    ;The routine prints ASCII
        MOV     DWORD PTR BCRESUL, EAX                          ;decimal content of EAX register
        MOV     DWORD PTR BCRESUL + 4, 0                        ;by usage of coprocessor FBSTP
        FILD    QWORD PTR BCRESUL                               ;instruction which converts
        FBSTP   BCRESUL                                         ;number into the packed BCD.
        LEA     EDI, INTRES                                     ;Unpacking of BCD is done via
        MOV     ECX, 5                                          ;the table of unpacked values.
        CLD                                                     ;Author: Andrija Radovic, ©2011
        EAX2AST_DO:
                MOVZX   EAX, BYTE PTR [ECX + OFFSET BCRESUL - 1]
                MOV     AX, WORD PTR [2 * EAX + OFFSET ASCII_TABLE]
                STOSW
        LOOPD   EAX2AST_DO
        MOV     AL, "0"
        MOV     ECX, 10
        LEA     EDI, INTRES
        REPE    SCASB
        DEC     EDI
        INVOKE  StdOut, EDI
        RET
EAX2AST ENDP

;-------------------------------------------------------------------------------------------------------------------

BCDEX   DT      0

EAX2AFL PROC                                                    ;The routine prints ASCII
        STD                                                     ;decimal content of EAX register
        MOV     DWORD PTR BCDEX, EAX                            ;by usage of coprocessor FBSTP
        MOV     DWORD PTR BCDEX + 4, 0                          ;instruction which converts
        FILD    QWORD PTR BCDEX                                 ;number into the packed BCD.
        FBSTP   BCDEX                                           ;Unpacking of BCD is done via
        MOV     EDX, DWORD PTR BCDEX                            ;SHR and SHRD instructions.
        MOVZX   EBX, WORD PTR BCDEX + 4                         ;Author: Andrija Radovic, ©2011
        LEA     EDI, 9 + INTRES
        EAX2AFL_DO:
                MOV     EAX, EDX
                AND     EAX, 15
                OR      AL, "0"
                STOSB
                SHRD    EDX, EBX, 4
                SHR     EBX, 4
                MOV     EAX, EDX
                OR      EAX, EBX
        JNZ     EAX2AFL_DO
        LEA     EDX, [EDI + 1]
        CLD
        INVOKE  StdOut, EDX
        RET
EAX2AFL ENDP

;-------------------------------------------------------------------------------------------------------------------

EAX2DEC PROC                                                    ;This routine prints ASCII
        LEA     ECX, INTRES + 9                                 ;decimal content of EAX register
        MOV     EBX, 10                                         ;by its repetitive dividing by
        EAX2DEC_DO:                                             ;10 using DIV instruction that
                XOR     EDX, EDX                                ;simultaneously yields result
                DIV     EBX                                     ;and remainder which denotes
                OR      EDX, "0"                                ;current decimal digit.
                MOV     BYTE PTR [ECX], DL                      ;Author: Andrija Radovic, ©2011
                TEST    EAX, EAX
        LOOPNZD EAX2DEC_DO
        LEA     EDX, [ECX + 1]
        INVOKE  StdOut, EDX
        RET
EAX2DEC ENDP
```

```
;-------------------------------------------------------------------------------------------------------

EAX2ASC PROC                                                            ;This routine prints ASCII
        LEA     ECX, INTRES + 9                                         ;decimal content of EAX register
        MOV     EDI, 858993459                                          ;by its repetitive dividing by
        MOV     EBX, EAX                                                ;10 using MUL instruction to
        AX2ASC_DO:                                                      ;divide by 10 via multiplication
                LEA     EAX, [EBX + 1]                                  ;with the appropriate constant.
                MUL     EDI                                             ;Author: Andrija Radovic, ©2011
                SHR     EDX, 1
                LEA     EAX, [4 * EDX + EDX]
                NEG     EAX
                LEA     EAX, [EBX + 2 * EAX + "0"]
                MOV     BYTE PTR [ECX], AL
                MOV     EBX, EDX
        LOOPNZD AX2ASC_DO
        LEA     EDX, [ECX + 1]
        INVOKE  StdOut, EDX
        RET
EAX2ASC ENDP

;-------------------------------------------------------------------------------------------------------

LB      DB      0F0H,0F1H, 1,3, 0F2H,4, 0F3H,0F4H, 2,7, 0F5H,0F6H, 6,8, 0F7H,9, 0F8H,0F9H
D       DD      0,1,2,3,4,5,6,7,8,9
        DD      00,10,20,30,40,50,60,70,80,90
        DD      000,100,200,300,400,500,600,700,800,900
        DD      0000,1000,2000,3000,4000,5000,6000,7000,8000,9000
        DD      00000,10000,20000,30000,40000,50000,60000,70000,80000,90000
        DD      000000,100000,200000,300000,400000,500000,600000,700000,800000,900000
        DD      0000000,1000000,2000000,3000000,4000000,5000000,6000000,7000000,8000000,9000000
        DD      00000000,10000000,20000000,30000000,40000000,50000000,60000000,70000000,80000000,90000000
        DD      000000000,100000000,200000000,300000000,400000000,500000000,600000000,700000000,800000000
        DD      900000000
        DD      0000000000,1000000000,2000000000,3000000000,4000000000,4294967295,4294967295,4294967295
        DD      4294967295,4294967295
        DD      1,10,100,1000,10000,100000,1000000,10000000,100000000,1000000000

EAX2BTR PROC                                                            ;This routine prints ASCII
        CLD                                                             ;decimal content of EAX register
        MOV     ESI, 400                                                ;by binary tree digits
        MOV     EBX, EAX                                                ;extraction.
        MOV     EAX, 5                                                  ;No specific instruction is used.
        EAX2BTR_DO1:                                                    ;Author: Andrija Radovic, ©2011
                CMP     EBX, DWORD PTR [4 * EAX + ESI + OFFSET D]
                SBB     EDX, EDX
                MOV     AL, BYTE PTR [2 * EAX + EDX + (OFFSET LB - 1)]
                TEST    AL, AL
        JNS     EAX2BTR_DO1
        AND     AX, 15
        LEA     EDI, 9 + INTRES
        SUB     EDI, EAX
        PUSH    EDI
        LEA     ESI, [EAX + 4 * EAX]
        MOV     EAX, 5
        SHL     ESI, 3
        EAX2BTR_DO3:
                CMP     EBX, DWORD PTR [4 * EAX + ESI + OFFSET D]
                SBB     EDX, EDX
                MOV     AL, BYTE PTR [2 * EAX + EDX + (OFFSET LB - 1)]
                TEST    AL, AL
        JNS     EAX2BTR_DO3
                AND     EAX, 15
                SUB     EBX, DWORD PTR [4 * EAX + ESI + OFFSET D]
                OR      AL, "0"
                STOSB
                MOV     EAX, 5
                SUB     ESI, 40
        JNC     EAX2BTR_DO3
        POP     EDX
        INVOKE  StdOut, EDX
        RET
EAX2BTR ENDP

;-------------------------------------------------------------------------------------------------------

LC      DW      0FF00H,0FF01H, 1,3, 0FF02H,4, 0FF03H,0FF04H, 2,7, 0FF05H,0FF06H, 6,8, 0FF07H,9
        DW      0FF08H,0FF09H
DC      DD      0,1,2,3,4,5,6,7,8,9
        DD      00,10,20,30,40,50,60,70,80,90
        DD      000,100,200,300,400,500,600,700,800,900
        DD      0000,1000,2000,3000,4000,5000,6000,7000,8000,9000
        DD      00000,10000,20000,30000,40000,50000,60000,70000,80000,90000
        DD      000000,100000,200000,300000,400000,500000,600000,700000,800000,900000
        DD      0000000,1000000,2000000,3000000,4000000,5000000,6000000,7000000,8000000,9000000
        DD      00000000,10000000,20000000,30000000,40000000,50000000,60000000,70000000,80000000,90000000
        DD      000000000,100000000,200000000,300000000,400000000,500000000,600000000,700000000,800000000
        DD      900000000
        DD      0000000000,1000000000,2000000000,3000000000,4000000000,4294967295,4294967295,4294967295
        DD      4294967295,4294967295
        DD      1,10,100,1000,10000,100000,1000000,10000000,100000000,1000000000

EAXCMOV PROC                                                            ;This routine prints ASCII
        CLD                                                             ;decimal content of EAX register
        MOV     ESI, 400                                                ;by binary tree digits
        MOV     EBX, EAX                                                ;extraction.
        MOV     EAX, 5                                                  ;CMOV instruction is used.
        EAXCMOV_DO1:                                                    ;Author: Andrija Radovic, ©2011
                CMP     EBX, DWORD PTR [4 * EAX + ESI + OFFSET DC]
;               CMOVC   AX, WORD PTR [4 * EAX + (OFFSET LC - 4)]        ;These two instructions
;               CMOVNC  AX, WORD PTR [4 * EAX + (OFFSET LC - 2)]        ;work on some processors...

                CMOVC   DX, WORD PTR [4 * EAX + (OFFSET LC - 4)]        ;Malfunction of x86 CMOV
                CMOVNC  DX, WORD PTR [4 * EAX + (OFFSET LC - 2)]        ;requires these ones instead
                MOV     AX, DX                                          ;of above two instructions.
```

```
                TEST    AX, AX
        JNS     EAXCMOV_DO1
        MOVZX   EAX, AL
        LEA     EDI, 9 + INTRES
        SUB     EDI, EAX
        PUSH    EDI
        LEA     ESI, [EAX + 4 * EAX]
        MOV     EAX, 5
        SHL     ESI, 3
        EAXCMOV_DO3:
                CMP     EBX, DWORD PTR [4 * EAX + ESI + OFFSET DC]
;               CMOVC   AX, WORD PTR [4 * EAX + (OFFSET LC - 4)]        ;These two instructions
;               CMOVNC  AX, WORD PTR [4 * EAX + (OFFSET LC - 2)]        ;work on some processors...

                CMOVC   DX, WORD PTR [4 * EAX + (OFFSET LC - 4)]        ;Malfunction of x86 CMOV
                CMOVNC  DX, WORD PTR [4 * EAX + (OFFSET LC - 2)]        ;requires these ones instead
                MOV     AX, DX                                          ;of above two instructions.

                TEST    AX, AX
        JNS     EAXCMOV_DO3
                MOVZX   EAX, AL
                SUB     EBX, DWORD PTR [4 * EAX + ESI + OFFSET DC]
                OR      AL, "0"
                STOSB
                MOV     EAX, 5
                SUB     ESI, 40
        JNC     EAXCMOV_DO3
        POP     EDX
        INVOKE  StdOut, EDX
        RET
EAXCMOV ENDP

;-------------------------------------------------------------------------------------------------------------

NumFormat       DB      "%u",0
BufferW         DB      32 DUP(0)

EAX2WIN PROC                                                            ;This routine prints ASCII EAX.
        INVOKE  wsprintf, ADDR BufferW, ADDR NumFormat, EAX             ;It utilizes native Windows NT
        INVOKE  StdOut, ADDR BufferW                                    ;subroutine.
        RET
EAX2WIN ENDP

;-------------------------------------------------------------------------------------------------------------

EAX_SQR PROC                                                            ;EAX = SQR(EAX), square rooting
        XOR     ESI, ESI                                                ;routine.
        XOR     EDX, EDX                                                ;EDX is remainder.
        MOV     EBX, 1073741824                                         ;Author: Andrija Radovic, ©2011
        SQRT_DO:
                LEA     EDI, [EBX + ESI]
                ADD     EDI, EDX
                SHR     EDX, 1
                CMP     EAX, EDI
                JC      SQRT_END_IF
                        MOV     ESI, EDI
                        ADD     EDX, EBX
                SQRT_END_IF:
                SHR     EBX, 2
        JNZ     SQRT_DO
        SUB     EAX, ESI
        XCHG    EAX, EDX
        RET
EAX_SQR ENDP

;-------------------------------------------------------------------------------------------------------------

END     ASCII
```

Above routines should be assembled rather from the command line than from some assembler's IDE because most of IDEs usually do not offer full control of the parameters passed to assembler and linker, just as is required by the linking process with removed section's read-only property.

Therefore they should be assembled manually from the command line:

```
C:\masm32\bin\ML.EXE /c /coff /Cp /nologo
/I"\Masm32\Include" ASCIIwin.asm
```

And it should be linked with the following single line which is spread to two lines of text although this actually should be only one uninterrupted line in the console:

```
C:\masm32\bin\LINK.EXE /SUBSYSTEM:CONSOLE /RELEASE
/LIBPATH:"\Masm32\Lib" /section:".text",ERW ASCIIwin.obj
```

Above two lines make EXE program with writable Code section. The benefit of such hand crafted linking is ability to write self modifying code.

Although it is highly discouraged by the MASM manual, this is something that should not be instantly rejected! Actually, all the recommendations those allegedly improve prettiness of the code by forcing real limitations to programmers should be immediately rejected! The self modifying code is very useful and sometimes may shrink code a lot and rejection of such option just because somebody found that it is not fancy enough seems to be quite wrong way of action. The second ability to keep all the routines' peaces tightly together is also very important for quick and efficient coding, which all requires section with the code to be writable.

CONCLUSION

All presented methods suffer from some sorts of disadvantages that elongate their executions. The misery of the task is consisted of the fact that set of 2 elements should be translated into the set of only 10 elements with up to 10 decimal positions and this exceptionally small number of decimal digits and decimal positions (32-bit integer has 10 decimal positions) really limits the efficiency of all of the applied optimization schemes, highly challenging coding skills because the number of instructions in optimization schemes reach the number of instructions performed in main loops.

Although the algorithm EDX2DEC based on the AAA instruction altogether with the EAX2BTR based on binary tree search are duly incomprehensible at the glance, this give it special charm making them quite fashionable. Despite the fact that EAX2BTR utilizes theoretically very promising Binary Tree Search, it is not much better than much simpler algorithms like EAX2SUB based on the repetitive subtractions. The one non-trivial algorithm with comparable performance with simplest ones is the EAX2ASC routine that utilizes division trough multiplication with only 8 instructions within main loop, which is going to be quite fast on the new processors with MUL instruction able to accomplish it in just of a couple of clock cycles. The algorithm EAX2DEC based on the DIV instruction is absolute champion of shortness – only five instructions within the main loop, but DIV instruction may be very slow depending on the particular processor. The situation is quite reversal when these algorithms should be transferred in VHDL hardware, then the binary tree search and BCD AAA instruction based algorithms are the most promising ones offering tremendous opportunities for hardware parallelization – they can reach even execution in a single clock cycle only.

Author:
Dipl.-Ing. Andrija Radović

andrija_radovic@hotmail.com