FRI-2G.303-1-CCT1-01

THE HOBBY TIME TRAINING APPROACH

Assoc. Prof. Milen Loukantchevsky, PhD, IEEE & ACM Member
Department of Computer Systems & Technologies,
University of Ruse "Angel Kanchev"
phone: 0877 303 850
E-mail: mil@ieee.org

Abstract: *Constructivism asserted that learning arises from building mental models based on experience. The concept of Developer's point of view (DPV) learning approach is considered as "perceive the very solution to the problem as a game", thus, making transition to a next level of gamification.*

The paper introduces the Hobby Time Training (HTT) concept as part of the DPV learning approach. The HTT assumes solving of small, apparently simple problems, which encapsulates deeply hidden potential. The solving takes place during the students' free time and assumes unobtrusive guidance with as little as possible obligatory moments.

Bitwise operations contain the sought-after hidden creative potential, mainly due to the limited support both at the high and low levels. Besides that, bitwise algorithms suppose usage of some special techniques as word-level parallelism, unrolling loops and branch elimination. As an illustration of the hidden deep inner content of the bitwise problems, the attention is focused on the computing parity bit problem. Several sample solutions are discussed: from the Naïve Algorithm, through the Word-Level Parallelism Algorithms to the Hardware Supported Algorithm.

Keywords: *Bitwise, Constructivism, Gamification, Hobby Time, x86/x64, Word-Level Parallelism*

ASJC Codes: *1701, 1708, 1712*

INTRODUCTION

According to Confucius (551–479 B.C.) "What I hear, I forget; What I see, I remember; What I do, I understand.". I guess we have the right to paraphrase this thought in "Do to understand.". That leads us to the constructive model of training, whose roots are in the constructivist philosophy. Constructivism asserted that learning arises from building mental models based on experience (Hadjerrouit, S., 2005).

In (Loukantchevsky, M., 2021) is introduced the concept of Developer's point of view (*DPV*) learning approach. It is described as "*perceive the very solution to the problem as a game*", that is the decision-making process itself becomes a game. Moreover, is assumed the usage of conventional development environments. Thus, making transition to a next level of gamification in comparison to the more primitive perception of gamification as "*make the hard stuff fun*".

We consider the *Hobby Time Training* (*HTT*) concept as part of the *DPV* learning approach. The *HTT* assumes:

- Solving of small, apparently simple problems, which encapsulates deeply hidden potential that could be find out only during the problem solving.
- The problem decision is made in the student's free time, at his discretion.
- The set deadlines for solving are long enough (at least a week, but normally 2-3-4 weeks).
- Unobtrusive guidance by the tutor mainly through a closed social network group (Loukantchevsky, M., 2022.).
- The formulation of the problem will of course contain some strict restrictions, but not other obligatory moments.
- Appropriate incentive system.

The choice of problems set for solving depends mainly on the studied subject. In the area of computer architectures, when viewed as an interface between the high-level languages (*HLL*) and the raw machine, the bitwise algorithms are a good choice. Bitwise operations contain the sought-after hidden creative potential, mainly due to the limited support both at the high and low

levels. Besides that, bitwise algorithms suppose usage of special techniques as word-level parallelism, unrolling loops and branch elimination (Intel Architectures Optimization Reference Manual, 2022).

The scope of bitwise algorithms is very broad [(Anderson, S., 2022), (Knuth, D., 2009), (Warren, H., 2013)]. Let's note some of them:

- Bitwise vs Boolean.
- Bit scan.
- Bit order reversal.
- Compute the minimum and/or maximum without branching.
- Population count.
- Check if an integer is a power of 2.
- Computing parity bit.
- Reverse bit sequence, etc.

As an illustration of the hidden deep inner content of the bitwise problems, we are going to focus on the computing parity bit problem. The code presented is *Windows 32*-bit platform oriented, produced by the *Embarcadero C++ Builder® 11.2* development environment and its classic *BCC32* compiler (Embarcadero, 2022). This choice is optional and therefore not a limitation. In our case, it stems from the practice of the author as corporate developer and from the possibility of transition from *HLL* to Assembly and vice versa.

EXPOSITION

1. Problem Definition and Naive Solution

The problem is to find a parity bit of an operand of size byte, word (16-bits) or doubleword (32-bits). Let us assume, for simplicity, that the operand X is of size byte, i.e. $n = 8$ (Eq. 1).

$$X = x_7x_6x_5x_4x_3x_2x_1x_0 \tag{1}$$

The parity bit P must complement to an even number the number of ones contained in the operand. Thus, for the size $n = 8$ of the operand X given, we obtain the next Eq. 2

$$P = x_7 \oplus x_6 \oplus x_5 \oplus x_4 \oplus x_3 \oplus x_2 \oplus x_1 \oplus x_0 \tag{2}$$

The solution should be capsulated in a function with prototype $BOOL\ \_get\_parity(BYTE\ x)$. The same for 16- and 32-bits cases but replacing the type $BYTE$ with $WORD$ and $DWORD$, respectively.

```
30 BOOL __get_parity(BYTE x)
   {
   //  BOOL result;

       __asm
       {
               //  Init
               mov BL, 8
               mov AH, [x]
               xor AL, AL
40
       L_LOOP: xor AL, AH
               shr AH, 1

               dec BL
               jnz L_LOOP

               and EAX, 0x00000001

   //            mov [result], EAX
50     }
```

Fig. 1. Naive Algorithm (*x86* Assembly Version, 8-bits)

Seemingly obvious (do we really have any other way?) is to XOR successively n times following Eq. 2. Eventually we will get the Assembly solution, shown at Fig. 1. The extension from byte to word and double word is as simple as changing the number of cycles at row 37.

As simple as this solution may seem, it raises several questions:

- Why the returned variable $result$ is ignored?
- Well, if we use *8*-bits registers, why we should return the result by a *32*-bits register?

The answer of the first question is in the way classic *BCC32* compiler returns values - by default is used register *EAX*. And the attentive student will notice this peculiarity on examining the identical *C* version. The answer of the second question is in the real length of the Boolean operand – it is *32*-bits (!) with the high *31* bits reset.

The drawbacks:

- The estimated performance is $O(n)$.
- Short base block, containing only three instructions (at rows 41, 42 and 44) is blocking the instruction reordering mechanism (*Out-Of-Order Execution*, *OOO* in Intel's notation) [(Intel Architectures Optimization Reference Manual, 2022), (Intel Architectures Software Developer's Manual, 2022)].
- Control stalls will appear because of the branch instruction.

And while the first drawback is obvious, the other two require knowledge of the modern pipelined superscalar computer architectures vs traditional scalar ones.

2. Word-Level Parallelism at High Level (Shift-Lookup Algorithm/WLP-1)

As already noted above, bitwise operations contain hidden creative potential. One manifestation of it is the *Word-Level Parallelism* [(Anderson, S., 2022), (Warren, H., 2013)].

Word-level Parallelism is a special case of data-level parallelism on scalar units – bytes (in our case), words, double words. With one bitwise *XOR* instruction we could *XOR*-ed not one but *8*, *16*, *32* Booleans at once!

We should decompose our operand *X* from Eq. 1 to two *4*-bits operands $x_3x_2x_1x_0$ and $x_7x_6x_5x_4$ using a shift right, then *XOR* them in parallel (Eq. 3).

$$(c_3 = x_7 \oplus x_3)||(c_2 = x_6 \oplus x_2)||(c_1 = x_5 \oplus x_1)||(c_0 = x_4 \oplus x_0) \quad (3)$$

As result we get a *4*-bits code $c_3c_2c_1c_0$. We consider this code as a lookup-code in the parity table from Fig. 2. The parity table in its classical representation is a vector by size *16*. Each element of the vector contains Boolean *0* or *1* according to the appropriate value of the parity bit.

C_3 C_2 C_1 C_0	0 1 1 0 1 0 0 1 1 0 0 1 0 1 1 0
0 0 0 0	X
0 0 0 1	X
0 0 1 0	X
0 0 1 1	X
0 1 0 0	X
. . .	. . .
1 1 0 1	X
1 1 1 0	X
1 1 1 1	X

Fig. 2. Parity Table

Instead of placing such a table into the memory we compress it into *16*-bits word 0110 1001 1001 0110 in binary or 6996 in hexadecimal. And replace the lookup operation with right shift of the above word-sized table by $c_3c_2c_1c_0$ bits. This algorithm could be implemented at *HLL* as *C* in our case or in *Assembly*. Here we prefer the *HLL* but translate it to *Assembly* to analyze the end code and eventually to compare with *Assembly* language edition as at Fig. 3.

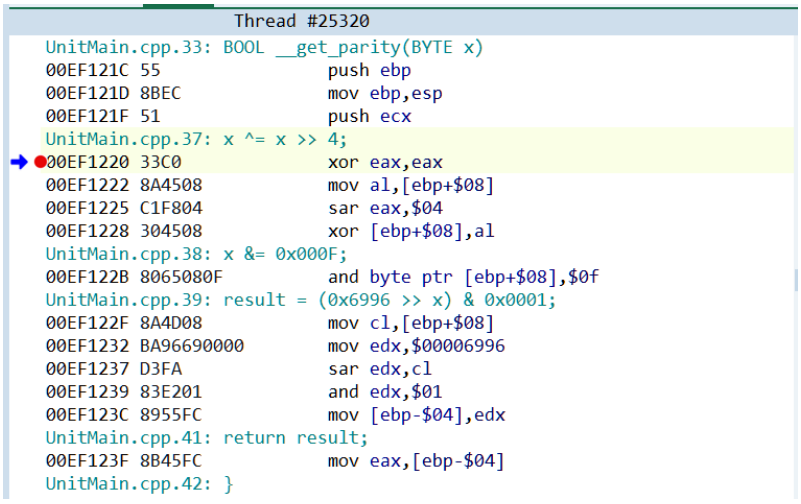

Fig. 3. Translated Version of the Shift-Lookup Algorithm (8-bits)

Really beautiful variant free from all the drawbacks of the naïve solution. Impressive to see that what is considered obvious eventually turns out to be the most inappropriate.

The extension from byte to word and double word requires additional pairs of *SHR* and *XOR* operations: one additional such a pair for word and yet another for double word.

3. Word-Level Parallelism at Low Level (Throughout-Shift Algorithm/WLP-2)

The *Throughout-Shift Algorithm* is very close to the above discussed *Shift-Lookup Algorithm*. There are two differences: the way of grouping is changed; the lookup table is omitted.

For *8*-bits case we will need three pairs of *SHR* and *XOR* instructions following the Eq. 4, Eq. 5 and finally Eq. 6.

$$(x'_6 = x_7 \oplus x_6) || (x'_4 = x_5 \oplus x_4) || (x'_2 = x_3 \oplus x_2) || (x'_0 = x_1 \oplus x_0), \quad (4)$$

$$(x''_4 = x'_6 \oplus x'_4) || (x''_0 = x'_2 \oplus x'_0), \quad (5)$$

$$(P = x'''_0 = x''_4 \oplus x''_0). \quad (6)$$

At Fig. 4 is shown the *8*-bits version in *x86 Assembly*. The groups of highest bits are formed in register DX, and the groups of lowest bits are formed in register AX.

Could be calculated that the number of these pairs will be $log_2 n$. Hence, the estimated performance of this algorithm is $O(log_2 n)$. It is quite better than the *Naïve Algorithm*. On another hand, the *Throughout-Shift Algorithm* is quite a bit less effective than the *Shift-Lookup Algorithm*, because of one additional *SHR*/*XOR* pair needed.

4. The Importance of Hardware Support

After everything done up to now, one cannot help but say to oneself: "Eureka! Well, the machine forms the parity flag automatically!". Yes, but…

At first, we should know how to set/reset the parity flag *PF*. Second, we should know how to access them, and of course no way how with *HLL*.

At Fig. 5 is shown *8*-bits version of the *Hardware Supported Algorithm* in *x86 Assembly*. It is straight enough, but even in these three-four instructions there are its own peculiarities [(Intel Architectures Software Developer's Manual, 2022), (Microsoft technical documentation, 2022)]:

- The *PF* reflects the parity only of the least significant byte of the result, hereof the operand is placed in register *AL*.

- The *PF* is accessed with different instruction in *16/32* and in *64*-bit modes.
- The *PF* is placed not in position *0* of the status register, but in position *2*.

```
BOOL __get_parity(BYTE x)
{
//  BOOL result;

    __asm
    {
            //  Init
            movzx EAX, [x]

            //  Xor 4 groups of 1 bit
            mov DX, AX
            shr DX, 1
            and DX, 0x55
            and AX, 0x55
            xor AX, DX

            //  Xor 2 groups of 2 bits
            mov DX, AX
            shr DX, 2
            and DX, 0x33
            and AX, 0x33
            xor AX, DX

            //  Xor 1 group of 4 bits
            mov DX, AX
            shr DX, 4
            and DX, 0x0F
            and AX, 0x0F
            xor AX, DX

//            mov [result], EAX
    }
```

Fig. 4. Throughout-Shift Algorithm (x86 Assembly Version, 8-bits)

```
BOOL __get_parity(BYTE x)
{
//  BOOL result;

    __asm
    {
        mov AL, [x]

        //  Version 1
        and  AL, AL     // Set PF according to AL
        lahf            // Load AH with FLAGS
        mov AL, AH

//      //  Version 2
//      //  pushf is slow but in 64-bit only possible
//      and  AL, AL   // Set PF according to AL
//      pushf           // Push lower 16 bits of EFLAGS
//      pop AX          // Pop lower 16 bits of EFLAGS into AX

        shr EAX, 2      // PF is at index 2

        and EAX, 0x01

//      mov [result], EAX
    }
```

Fig. 5. Hardware Supported Algorithm (*x86* Assembly Version, 8-bits)

The extension from byte to word and double word requires processing of every byte due to the mentioned peculiarity of the *PF* - reflecting the parity only of the least significant byte of the result. At Fig. 6 is shown *16*-bits version of the *Hardware Supported Algorithm* in *x86 Assembly*. This simple example is a good illustration of the importance of the available hardware support.

At Fig. 7 is shown complexity estimation for byte, word, and double word variants of the four algorithms discussed. The *Naïve Algorithm* is worst in all cases. The *Hardware Supported Algorithm* is best for byte and word processing. And the *Shift-Lookup Algorithm* is best for double

word processing. The *Throughout-Shift Algorithm* is close to them. Except the *Naïve Algorithm* the others do not restrict instruction reordering (*OOO*) and are branch free, i.e. control stalls free.

```
BOOL __get_parity(WORD x)
{
//  BOOL result;

    __asm
    {
        // --- Byte 0 ---
        mov DX, [x]
        mov AX, DX
        and  AX, AX   // Set PF according to AL
        lahf          // Load AH with FLAGS
        shr AH, 2     // PF is at index 2
        mov BL, AH

        // --- Byte 1 ---
        shr DX, 8
        mov AX, DX
        and  AX, AX   // Set PF according to AL
        lahf          // Load AH with FLAGS
        shr AH, 2     // PF is at index 2
        xor BL, AH

        mov AL, BL
        and EAX, 0x01
//      mov [result], EAX
    }
```

Fig. 6. Hardware Supported Algorithm (*x86* Assembly Version,16-bits)

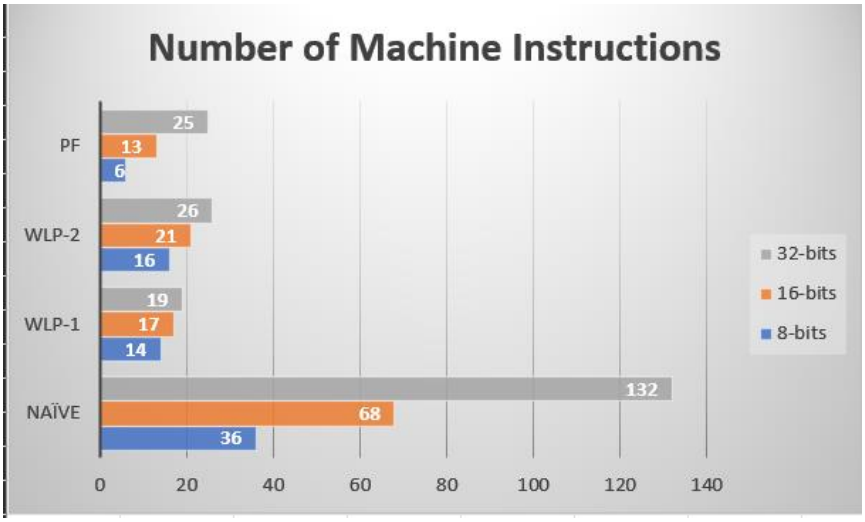

Fig. 7. Complexity estimation

Finally, we observe the following *phenomenon*: usually, what is considered obvious eventually turns out to be the most inappropriate.

CONCLUSION

The professional software developer perceives the very solution of the problem as a game. To reach this highest level of gamification is proposed the concept of the *Hobby Time Training* (*HTT*) concept. The *HTT* is part of the *DPV* learning approach and assumes solving of small, apparently simple problems, which encapsulates deeply hidden potential. Solving takes place during the students' free time and assumes unobtrusive guidance from the tutor with as little as possible obligatory moments.

In the area of computer architectures, viewed as an interface between the high-level languages (*HLL*) and the raw machine, the bitwise algorithms are a good choice. Bitwise

operations contain the sought-after hidden creative potential, mainly due to the limited support both at the high and low levels. Besides that, bitwise algorithms suppose usage of special techniques.

For illustration, from the long list of algorithms requiring bitwise operation is selected the *parity bit* calculation problem. This is a very good example because of its apparent simplicity. In practice it turns out that the task is not so simple at all.

From one side, you cannot be innovative in the big if you never even tried to be innovative in the small. And from another, we encounter the phenomenon of what is considered as obvious as a solution eventually turns out to be the most inappropriate one.

REFERENCES

Anderson, S. (2022) Bit Twiddling Hacks. Retrieved September 17, 2022 from http://graphics.stanford.edu/~seander/bithacks.html#ParityParallel

Embarcadero. (2022). RAD Studio Docwiki: C++ Compilers. Retrieved September 23, 2022 from https://docwiki.embarcadero.com/RADStudio/Alexandria/en/C%2B%2B_Compilers

Intel. (2022). Intel® 64 and IA-32 Architectures Optimization Reference Manual. Order Number: 248966-045

Intel. (2022). Intel® 64 and IA-32 Architectures Software Developer's Manual. Order Number: 325462-077US

Hadjerrouit, S. (2005). Constructivism as Guiding Philosophy for Software Engineering Education. ACM SIGCSE Bulletin, Vol. 37, Num. 4. DOI: https:// doi.org/10.1145/1113847.1113875

Knuth, D. (2009). The Art of Computer Programming, Volume 4, Fascicle 1: Bitwise Tricks & Techniques; Binary Decision Diagrams. 1st Ed. AddisonWesley Professional. ISBN 978-0321580504

Loukantchevsky, M. (2022). Hobby Time: Bit Order Reversal. Retrieved September 21, 2022 from https://t.me/c/1569632971/106

Loukantchevsky, M. (2021). Solving Classical Problem in New Context as Constructive Model of Training: Active Memory Array of Concurrent Processes Concept. IN: CompSysTech '21, New York, NY, USA, ACM, pp. 191-195, https://doi.org/10.1145/3472410.3472430

Microsoft technical documentation. (2022) x86 Instructions. Retrieved September 17, 2022 from https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/x86-instructions

Warren, H. (2013). Hacker's Delight. 2nd Ed. Addison-Wesley Professional. ISBN 978-0321842688

