# BCD Addition without Carry Propagation

**Sukrith B**
**A. Sreekumar**
Cochin

**Article History:** Received: 10 November 2020; Revised 12 January 2021 Accepted: 27 January 2021; Published online: 5 April 2021

**Abstract:** Binary Coded Decimal addition is less used due to the computational overhead pertaining to it. We propose an algorithm to reduce the computational overhead caused by removing carry propagation time delay by doing a prefixed set of instructions for a block of 64-bit digits, speeding up the computation up by four times in the ideal case.

**Keywords:** BCD Addition, Algorithm Complexity, Computer arithmetic, Algorithm Adder.

## 1. Introduction

Binary Coded Decimal (BCD) numbers are usually represented in a byte (8 bit). There is also a packed BCD concept where a digit is stored as a nibble (4 bits). Results of the addition of any two BCD digits require the additional overhead of adding by six if the resultant digit is greater than $9_{10}$ (Liu, 1986)**.** Packed BCD requires an extra bit for Carry Flag (CF) to detect the incoming carry. Hence, the actual number of bits becomes five per digit. The addition of large numbers requires propagation of this carry and the delay caused by it. To mitigate the requirement of CF an algorithm used by Michael Wiedeking of MATHEMA Software GMBH reduces eight-bit processing to seven bits, thus reserving the MSB (Most Significant Bit) for carry (Jonas, 1999). Thus, reducing carry propagation but does not incorporate incoming or outgoing carry. The proposed algorithm is more straightforward and incorporates the incoming and outgoing carry; therefore could be utilised in various scenarios.

BCD is preferable because it is in human-readable format, and ease of conversion prevail in computing and electronic communications (O. Al-Khaleel, 2011). But the additional overhead of the correction mechanism curtails the widespread usage of BCD. Many error correction methods and carry chain optimisation by (M. Vazquez, 2009) are used in hardware implementation to reduce computational time and complexity. The pre-processing method of correction is proposed by (Z. T. Sworna, 2016), but these operations cause much delay, whereas more research works are on post-process processing when output exceeds $9_{10}$ (Haque, 2018).

## 2.Proposed Method

Addition using a packed BCD format has its nuances. If a digit-wise addition generates a carry, it falls to the LSB (Least Significant Bit) of high order digit. Hence carry should be stored separately to avoid interfering with the higher-order digit. Storing and then processing each digit that is nibble by nibble will consume unnecessary waiting of resources.

The proposed algorithm mitigates the waiting of propagating carry created by digit-wise addition. Here numbers are represented as unsigned integers in packed BCD format of 64-bit size, say *A* and *B*. Thus, addition operates block-wise with the size of 64-bit, that is, 16 digits at a time. This algorithm can be used for any number of digits packed in this format by considering it as a block sequence. Adding each block requires an input and output parameter, namely, carry from the previous block of digits and carry generated by the current block-addition. Leading blocks should be added with *CF* = FALSE. Algorithm 2 explains this process with an example illustrated in table 2.

### 2.1. Block-wise Addition

Initially, a (*CF*) from low order block is observed and if true *A* is incremented by one. Then we compute Generated Carry, *GC* = ~(*A* ⊕ *B*). In *GC*, the LSB of each digit is only of our concern. This is to check even parity of corresponding digits of the sum of *A* and *B*.
New *A* is computed by incrementing it by six and *B* digit-wise (line 6 of algorithm 1). It will produce carry exactly when decimal addition of two operands would develop a carry (Jonas, 1999). A hindrance to this procedure is that there will be an excess of six in all digits that does not generate a carry, whereas the digits that produced a carry will have the correct value.

```
Algorithm 1 BCD addition Block wise
Require: 64 bit Integers $A$, $B$ and Boolean CF
 1: if CF= TRUE then
 2:     $A \leftarrow A + 1$
 3: end if
 4: $GC \leftarrow\sim (A \oplus B)$          /*parity of individual digits of sum ;Assumption no incoming carry*/
 5: $A \leftarrow A + B + 0x6666666666666666$     /*on the assumption that individual digitsum generates carry*/
 6: $GC \leftarrow (GC \oplus A)$ bitwise and $0x1111111111111111$   /*digits which doesnt actually generate carry*/
 7: $GC \leftarrow \text{shiftright4}(GC) \times 6$
 8: if $A \leq B$ then
 9:     CF ← TRUE
10: else
11:     CF ← FALSE
12:     $GC \leftarrow GC | 0x6000000000000000$
13: end if
14: $A \leftarrow A - GC$
Ensure: Output: $A$, CF
```

```
Algorithm 2 BCD Addition Arbitrary
Require: $A$, $B$
 1: $i \leftarrow A.len$ ; $j \leftarrow B.len, k = 0$; CF =FALSE
 2: $s = \min(i, j)$
 3: for $k = 0; k < s; k++$ do
 4:     (Sum[$k$],CF) = BCD(A[$k$],B[$k$],CF)     /*calls algorithm 1*/
 5: end for
 6: if s = i then
 7:     $R \leftarrow B$                /*A is smaller than B remaining of B is considered*/
 8: else
 9:     $R \leftarrow A$, j ← i          /* B is smaller than A remaining of A is considered*/
10: end if
11: repeat
12:     (Sum[$k$],CF) = Inc(R[$k$],CF), $k++$      /*Next block incremented if Carry is TRUE*/
13: until CF = FALSE |($k = j$)
14: if CF = TRUE and ($k = j$) then
15:     Sum[$k++$] ← 1                /*Length of sum more than length of numbers*/
16: else
17:     repeat
18:         Sum[$k$] ← R[$k$], $k++$         /*Rest of the blocks (if any) are copied */
19:     until $k = j$
20: end if
21: Sum.len ←k
Ensure: Output: Sum
```

**Table 1.** Algorithm 1, Example.

| | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ***A*** | input | 7 | 9 | 8 | 4 | 1 | 4 | 6 | 3 | 7 | 3 | 8 | 1 | 1 | 3 | 5 | 9 |
| ***B*** | input | 3 | 8 | 2 | 4 | 8 | 5 | 3 | 4 | 2 | 6 | 2 | 2 | 8 | 6 | 4 | 8 |
| ***GC*** | line 4 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| ***A*** | line 5 | 1 | 8 | 0 | E | F | F | F | E | 0 | 0 | 0 | A | 0 | 0 | 0 | 7 |
| ***GC*** | line 6 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 |

| | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ***GC*** | line 8 | 0 | 0 | 0 | 6 | 6 | 6 | 6 | 6 | 0 | 0 | 0 | 6 | 0 | 0 | 0 | 0 |
| ***A*** | line 15 | 1 | 8 | 0 | 8 | 9 | 9 | 9 | 8 | 0 | 0 | 0 | 4 | 0 | 0 | 0 | 7 |

Hence, *GC* is again computed by *GC* ⊕ *A* to find the positions where carry is not generated and is then reduced by six in each of these positions, thus getting the required result. We then compare the outcome with any of the operands, and if it's of lesser value, then *CF* holds TRUE, else FALSE. Instead, *CF* could be set as TRUE or FALSE after the above operation by observing the Carry Flag register.

**Table 2.** Algorithm 2, Example.

| | | | | | |
|---|---|---|---|---|---|
| ***A*** | input | | 9999999999999999 | 1234567890123456 | 8546215647826546 |
| ***B*** | input | | | 8765432109876543 | 2354875468245123 |
| ***Sum*** | line 5 | | | 0000000000000000 | 0901091116071669 |
| ***R*** | line 10 | | 9999999999999999 | | |
| ***Sum*** | line 13 | | 0000000000000000 | 0000000000000000 | 0901091116071669 |
| ***Sum*** | line 15 | 1 | 0000000000000000 | 0000000000000000 | 0901091116071669 |

## 3. Conclusion

The proposed algorithm mitigates the delay caused by carry propagation and utilizes the complete 64 bit each time. As there are no loops in it, it could be implemented easily in hardware platforms as well as parallelize in a multi-core architecture. The algorithm proposed here is considering a single-core 64-bit processor, but the algorithm could be scaled to the availability of the resources. In the ideal case, we can gain speed up to four times.

## My Appendix

| notation | Meaning |
|---|---|
| ⊕ | XOR operator |
| shiftright4 | shift right the operand 4 times |
| A← B | assigns the value of B to A |
| ~A | Negation |
| \| | OR operator |
| *A.len* | length of number A |

## References

[1]. Haque, M. U. (2018). A fast fpga-based bcd adder. *Circuits, Systems, and Signal Processing*, 4384--4408.

[2]. Jonas, D. W. (1999). *BCD Arithmetic a tutorial*. Retrieved from The Arithmetic tutorial collection: http://homepage.divms.uiowa.edu/~jones/bcd/bcd.html#packed

[3]. Liu, Y.-C. a. (1986). *Microcomputer systems: the 8086/8088 family architecture, programming and design.* Prentice Hall, Inc., Old Tappan, NJ.

[4]. M. Vazquez, G. S. (2009). Decimal Adders/Subtractors in FPGA: Efficient 6-input LUT Implementations. *International Conference on Reconfigurable Computing and FPGAs*, (pp. 42-47).

[5]. O. Al-Khaleel, M. A.-K.-Q. (2011). Fast binary/decimal adder/subtractor with a novel correction-free BCD addition. *2011 18th IEEE International Conference on Electronics, Circuits, and Systems*, (pp. 455-459).

[6]. Z. T. Sworna, M. U. (2016). Low-power and area efficient binary coded decimal adder design using a look up table-based field programmable gate array. *IET Circuits, Devices Systems, 10*(3), 163-172.