BCD Addition Algorithm

Author: Pooria Lari

This algorithm represents the structured logic of a **16bit Binary-Coded Decimal (BCD) addition** process, derived from an original implementation in **8086 Assembly**.

It performs decimal-accurate addition of two packed BCD operands by handling each nibble individually and applying BCD correction whenever a digit exceeds 9. The algorithm ensures proper carry propagation and produces a fully valid BCD result.

Pseudocode

```
BCD_ADD(A, B)
1   Initialize AX ← 0, CX ← 0
2   REPORT ← 0
3   RESULT_BITS ← 0

4   LOOP:
5      BX ← A              // 16-bit BCD number 1
6      DX ← B              // 16-bit BCD number 2
7      BL ← BL + DL          // add low bytes only (affects AF/CF for D0)
8      Load Status Flags Into AH Register   // LAHF

9      if CL == 0 then
10        goto CHECK_000XH
11     else if CL == 1 then
12        goto CHECK_00X0H
13     end if

14     Store AH into Status Flags         // SAHF
15     BH ← BH + DH + CF               // ADC BH, DH (propagate carry to next nibble)
16     Load Status Flags Into AH Register   // LAHF

17     if CL == 2 then
18        goto CHECK_0X00H
19     else if CL == 3 then
20        goto CHECK_X000H
21     end if

22     AH ← AH AND 01H
23     REPORT ← REPORT OR AH

24     AH ← CH
25     RESULT_BITS ← AX
26     goto END_PROCESS
```

```
------------------------------------------------
CHECK_000XH:
27  CL ← 1
28  AH ← AH AND 10H              // test AF from prior add
29  if AH == 10H then
30     BL ← BL + 06H
31  end if

32  BX ← BX AND 000FH             // isolate least-significant nibble
33  if BX > 0009H then
34     BX ← BX + 0006H           // BCD correction
35     BX ← BX AND 001FH          // keep carry bit if any
36     AL ← AL + BL
37     goto LOOP
38  else
39     AL ← AL + BL
40     goto LOOP
41  end if

------------------------------------------------
CHECK_00X0H:
42  CL ← 2
43  AH ← AH AND 01H              // test CF from previous step
44  if AH == 01H then
45     BL ← BL + 60H
46  end if

47  BX ← BX AND 00F0H             // isolate second nibble
48  if BX > 0090H then
49     BX ← BX + 0060H
50     BX ← BX AND 01F0H
51     AL ← AL + BL
52     CH ← CH + BH
53     goto LOOP
54  else
55     AL ← AL + BL
56     goto LOOP
57  end if

------------------------------------------------
CHECK_0X00H:
58  CL ← 3
59  AH ← AH AND 10H              // test AF
60  if AH == 10H then
61     BH ← BH + 06H
62  end if

63  BX ← BX AND 0F00H             // isolate third nibble
```

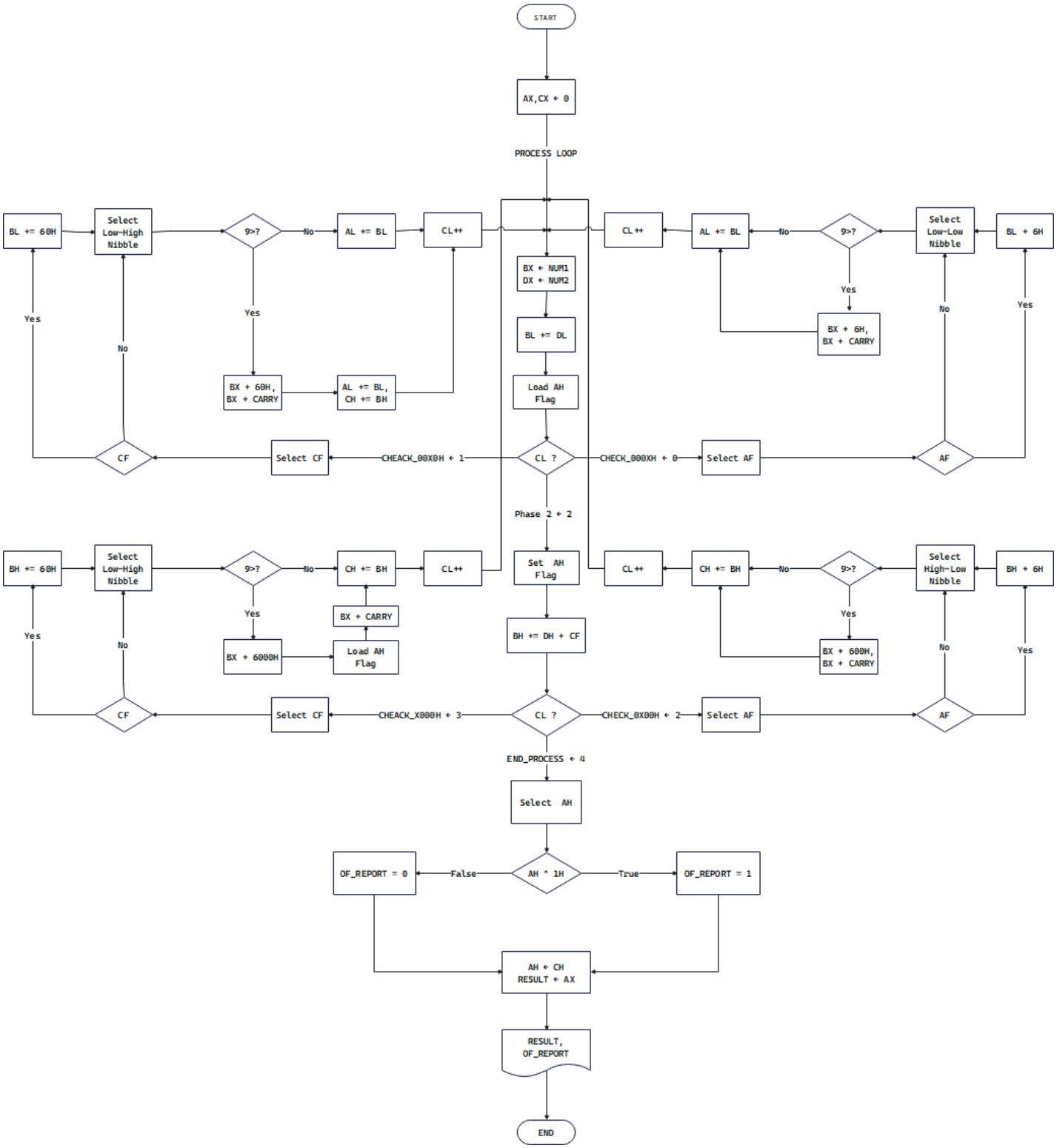
```
64  if BX > 0900H then
65     BX ← BX + 0600H
66     BX ← BX AND 1F00H
67     CH ← CH + BH
68     goto LOOP
69  else
70     CH ← CH + BH
71     goto LOOP
72  end if

-------------------------------------------------
CHECK_X000H:
73  CL ← 4
74  AH ← AH AND 01H              // test CF
75  if AH == 01H then
76     BH ← BH + 60H
77  end if

78  BX ← BX AND 0F000H           // isolate most significant nibble
79  if BX > 9000H then
80     BX ← BX + 6000H
81     Load Status Flags Into AH Register    // LAHF
82     BX ← BX AND 0F000H
83     CH ← CH + BH
84     goto LOOP
85  else
86     CH ← CH + BH
87     goto LOOP
88  end if

-------------------------------------------------
END_PROCESS:
89  RESULT_BITS ← AX
90  Return RESULT_BITS, REPORT
```

Flowchart


START
AX,CX ← 0
PROCESS LOOP
BX ← NUM1
DX ← NUM2
BL += DL
Load AH Flag
CL ?
CHECK_000XH ← 0
CHEACK_00X0H ← 1
Select AF
Select CF
AF
CF
BL += 60H
Select Low-High Nibble
9>?
No
AL += BL
CL++
Yes
BX + 60H, BX + CARRY
AL += BL, CH += BH
Select Low-Low Nibble
BL + 6H
BX + 6H, BX + CARRY
Phase 2 ← 2
Set AH Flag
BH += DH + CF
CHECK_0X00H ← 2
CHEACK_X000H ← 3
BH += 60H
CH += BH
BX + CARRY
BX + 6000H
Load AH Flag
Select High-Low Nibble
BH + 6H
BX + 600H, BX + CARRY
END_PROCESS ← 4
Select AH
AH ^ 1H
True
False
OF_REPORT = 1
OF_REPORT = 0
AH ← CH
RESULT ← AX
RESULT, OF_REPORT
END

Algorithm Logic

1. **Initialization**
   - All registers (`AX, BX, DX, CX`) are cleared to ensure no previous data affects the computation.
   - The variables `RESULT_BITS` and `REPORT` are initialized to zero.
   - The step counter (`CL`) is set to zero to control the sequence of processing stages.
2. **Loading the Data**
   - The first operand (`A`) is loaded into register `BX`, and the second operand (`B`) is loaded into `DX`.
   - Both operands are 16-bit numbers stored in packed BCD format in memory.
3. **Initial Binary Addition**
   - The least significant bytes of both operands are added together (`BL + DL`).
   - The result may generate an internal carry or overflow at the nibble level.
   - The processor's status flags (`AF` and `CF`) are saved for later use in the correction phase.
4. **Stage Determination**
   - The value of the step counter `CL` determines which nibble of the number is currently being processed:
     - `CL = 0` → least significant digit (`CHECK_000XH`)
     - `CL = 1` → second digit (`CHECK_00X0H`)
     - `CL = 2` → third digit (`CHECK_0X00H`)
     - `CL = 3` → most significant digit (`CHECK_X000H`)
   - After completing each stage, the counter is incremented to move to the next nibble.
5. **BCD Digit Correction**
   - At each stage, the corresponding nibble is isolated using a bitwise **AND** operation with the appropriate mask.
   - If the nibble value exceeds 9, or if the **Auxiliary Carry Flag (AF)** is active, the value **6 (0x6)** is added to that nibble for BCD correction.
   - This ensures that every nibble remains a valid decimal digit (0–9).
   - If a carry occurs during correction, the **Carry Flag (CF)** is set and propagated to the next stage.
6. **Carry Propagation Between Nibbles**
   - The carry flag is transferred between stages using the logical addition instruction (`ADC`), ensuring proper carry handling in the next nibble.
   - If a new carry is generated in the current step, it will be automatically included in the addition of the following nibble.
7. **Flag Update and Result Storage**
   - After each stage, the `AF` and `CF` flags are saved and restored using the `LAHF` and `SAHF` instructions through register `AH`.
   - The final carry flag is combined into the `REPORT` variable using a logical OR operation.
   - The final computed sum stored in `AX` is then saved to the variable `RESULT_BITS`.
8. **End of Algorithm**
   - When the step counter (`CL`) reaches 4, all four nibbles have been processed.
   - At this point, the algorithm terminates and returns the two final outputs: `RESULT_BITS` (the 16bit BCD result) and `REPORT` (the final carry status).

Registers Usage Table

Register	Purpose
AX	Holds the final combined 16-bit result of the BCD addition.
BX	Holds first operand
DX	Holds second operand
CH	High byte accumulator
CL	Step counter for operations
AH	Flag checking and temporary operations
AL	Low byte accumulator

Flag Handling

Register	Purpose
CF (Carry Flag)	Indicates carry propagation between nibbles and detects overflow at the byte level.
AF (Auxiliary Carry Flag)	Detects carry between the two internal nibbles of a byte. When this flag is set, a correction of **+6** must be applied to maintain valid BCD digits.
Other Flags (ZF, SF, PF)	These flags may change during arithmetic operations but have no effect on the core logic of the algorithm.

Input / Output Variables

Variable	Type / Size	Role and Description
A	16-bit (BCD)	First input operand. Represents a packed 4-digit BCD number (each nibble corresponds to one decimal digit).
B	16-bit (BCD)	Second input operand. Also a packed 4-digit BCD number to be added to `A`.
REPORT	8-bit	Output flag variable that indicates whether a carry occurred after processing the most significant nibble. `00H` = No carry, `01H` = Carry occurred.
RESULT_BITS	16-bit	Output variable holding the final BCD-corrected result after all four nibble corrections. Represents the sum of `A` and `B` in valid BCD format.

Performance Evaluation

Assumptions

- Processor: Intel 8086
- Clock Frequency: 5 MHz
- Each nibble correction loop executes once per digit (4 total).
- Average instruction timing is based on Intel's standard cycle counts.
- No I/O or interrupt overhead is considered.

The **time complexity** of this algorithm is **O(1)**, since it performs a fixed number of operations — four nibble-processing stages — independent of the input values.
Similarly, the **space complexity** is **O(1)**, because it uses only a constant number of registers and memory locations throughout execution, without allocating any additional storage.

Calculation

Each correction stage (one nibble) requires approximately **80–90 clock cycles**.
Since the algorithm processes four nibbles:

$$320 - 360\ cycles = 4 \times (80\ to\ 90) = T_{total}$$

At a 5 MHz clock frequency:

$$t = \frac{Total\ Cycles}{Clock\ Frequency}$$

$$t = \frac{360}{5{,}000{,}000} = 0.000072\ s = 72\ \mu s$$

≈ 70–80 microseconds on a standard 8086 @ 5 MHz

Notes

Based on the 8086 Assembly structure (.MODEL SMALL, .DATA, .CODE). Suitable for educational use in microprocessor or computer organization labs. Output logic and flags may vary depending on specific implementation details.

The complete assembly implementation can be found in the project repository

https://github.com/Pooria-Lari/16bit-BCD-Addition

Pooria Lari