# BCDADD

## Functional Specs

Bla Bla

## Algorithm and State Diagram

The designed bcdadd solution divides the given problem into two sub-problems:

1. Reduce any possible input combination of R0 and R1 to a simple sum of two positive bcd numbers.
2. Perform the simple bcd sum, then modify the result depending on the original input case.

In step 1, the 4 general input cases are variations of R0 and R1 being positive or negative. In each one of those cases, we reduce the problem to a simple bcd sum that can later be interpreted to provide us the required result. The 4 cases as well as the means of handling each one of them is outlined below.

1. R0 is +ve and R1 is +ve

This is the simplest case. The only possible issue may be an overflow, which we check at the end before returning.

* 1. BCDADD: R0 = R0 + R1
  2. If R0 overflowed, set the appropriate bit to indicate it

1. R0 is –ve and R1 is –ve

In this case, we disregard the sign and perform a simple sum as in the first case. We then set the negative sign bit to the result, and check for overflow.

* 1. BCDADD: R0 = R0 + R1
  2. Set the negative sign
  3. If R0 overflowed, set the appropriate bit to indicate it

1. R0 is +ve and R1 is –ve

This case is reduced to case 4 by swapping R0 and R1.

* 1. Swap R0 and R1
  2. Roll off into case 4

1. R0 is –ve and R1 is +ve

Tens complement of the negative operand is used to convert a subtraction into a sum. If the absolute value of the negative operand is larger than the other operand, then the tens complement of the sum is our actual result.

* 1. Take the tens complement of R0 and store it in R0
  2. BCDADD: R0 = R0 + R1
  3. If the original R0 was larger than R1, we take the tens complement of the result R0 and set the negative bit.



## Validation

In order to test, validate and debug the bcdadd routine, a wrapper subroutine that includes test cases covering possible scenarios in the state diagram was written. The test cases are outlined in Table \_\_\_.

The wrapper routine was designed such that an incorrect sum arising from bcdadd given one of test case inputs branches the program flow to an error label. If all test cases pass, the program continues to a success label.

This test routine was run after every change to the assembly code in order to catch and debug any introduced error before it evolves into something more serious.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Case | Signs | Specs | R0 | | | R1 | |
|  |  |  | Sign | | Hex | Sign | Hex |
| 1 | r0+, r1+ |  | + | | 0x00762500 | + | 0x00309380 |
| 2 | r0-, r1- |  | - | | 0x80039785 | - | 0x80139962 |
| 3a | r0+, r1- | (|ro|>|r1|) | + | | 0x09656000 | - | 0x87847000 |
| 3b | r0+, r1- | (|ro|<|r1|) | + | | 0x07847000 | - | 0x89656000 |
| 3c | r0+, r1- | (|ro|=|r1|) | + | | 0x09656000 | - | 0x89656000 |
| 4a | r0-, r1+ | (|ro|>|r1|) | - | | 0x89656000 | + | 0x07847000 |
| 4b | r0-, r1+ | (|ro|<|r1|) | - | | 0x87847000 | + | 0x09656000 |
| 4c | r0-, r1+ | (|ro|=|r1|) | - | | 0x89656000 | + | 0x09656000 |
| 5 | r0 , r1 | r0 overflown | - | | 0xF9656000 | + | 0x09656000 |

## Performance

Bla bla

# Babbage

## Functional Specs

the prototype of the babbage function is:

void babbage(unsigned int PolyOrder, unsigned int NumItems, bcd\_t\* Elements)

where the type bcd\_t is defined to be: unsigned long

## Algorithm

Bla Bla

## Validation

Blab la

## Performance

Bla bla

# Appendix