ECSE 426 Lab 01

Group 03

10/11/2011
Akeel Ali - 260275389
Amjad Al-Rikabi - 260143211

Table of Contents

(Part A) BCDADD

Functional Specification

Assembly Description

The *bcdadd* routine, takes two well formatted BCDs in *r0* and *r1*, and places their BCD sum in *r0* *bcdadd*. This is the C function calling convention of the ARM compiler.

C Function Prototype

```
/**
 * bcdadd: Assembly coded function that adds two bcd numbers
 *
 * Functionality: The function adds nibble by nibble (performs 7 such sums)
 *
 * Inputs:
 *              c1, c2: 2 binary coded decimal numbers of type bcd32_t (could be negative)
 *              Encoding: Bit 31: sign, Bit 30: overflow, Bits 29,28: don't care,
 *                          every other 4 bits: decimal digit (0-9)
 *
 * Output: returns a bcd32_t number representing the sum of the 2 inputs
 *
 * Error conditions:
 *                   If any of the passed numbers are already overflown, a 0x30000000 is returned
 *                   An overflow in the sum is signaled in the overflow bit 30
 */
bcd32_t bcdadd(bcd32_t c1, bcd32_t c2);
```

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 binary coded decimal (BCD) sum, and 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 with 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.

   a. BCD Add: *R0 = R0 + R1*
   b. If *R0* overflowed, set the appropriate bit to indicate it

2. $R0$ 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.

   a. BCD Add: $R0 = R0 + R1$
   b. Set the negative sign
   c. If $R0$ overflowed, set the appropriate bit to indicate it

3. $R0$ is +ve and $R1$ is –ve

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

   a. Swap $R0$ and $R1$
   b. Roll off into case 4

4. $R0$ 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.

   a. Take the tens complement of $R0$ and store it in $R0$
   b. BCD Add: $R0 = R0 + R1$
   c. If the original $R0$ was larger than $R1$, we take the tens complement of the result $R0$ and set the negative bit.

BCD Add

This subroutine is where the actual addition takes place once the input case has been categorized and reduced to a simple BCD add.

In general terms, this subroutine works by adding the two registers, $R0$ and $R1$, nibble by nibble. Hence, it performs 7 nibble sums based on the given BCD representation. If any nibble sum turns out larger than 9 (largest allowed digit in BCD), we add 6 to it in order to fix the representation. The carry is then taken on to the next nibble.

The routine uses a mask to extract the nibbles from the original BCD numbers. The nibbles are added without shifting them to the far right, and hence the comparisons are made with a moving 9, and a moving 6 is added in the event the sum is larger than the moving 9.

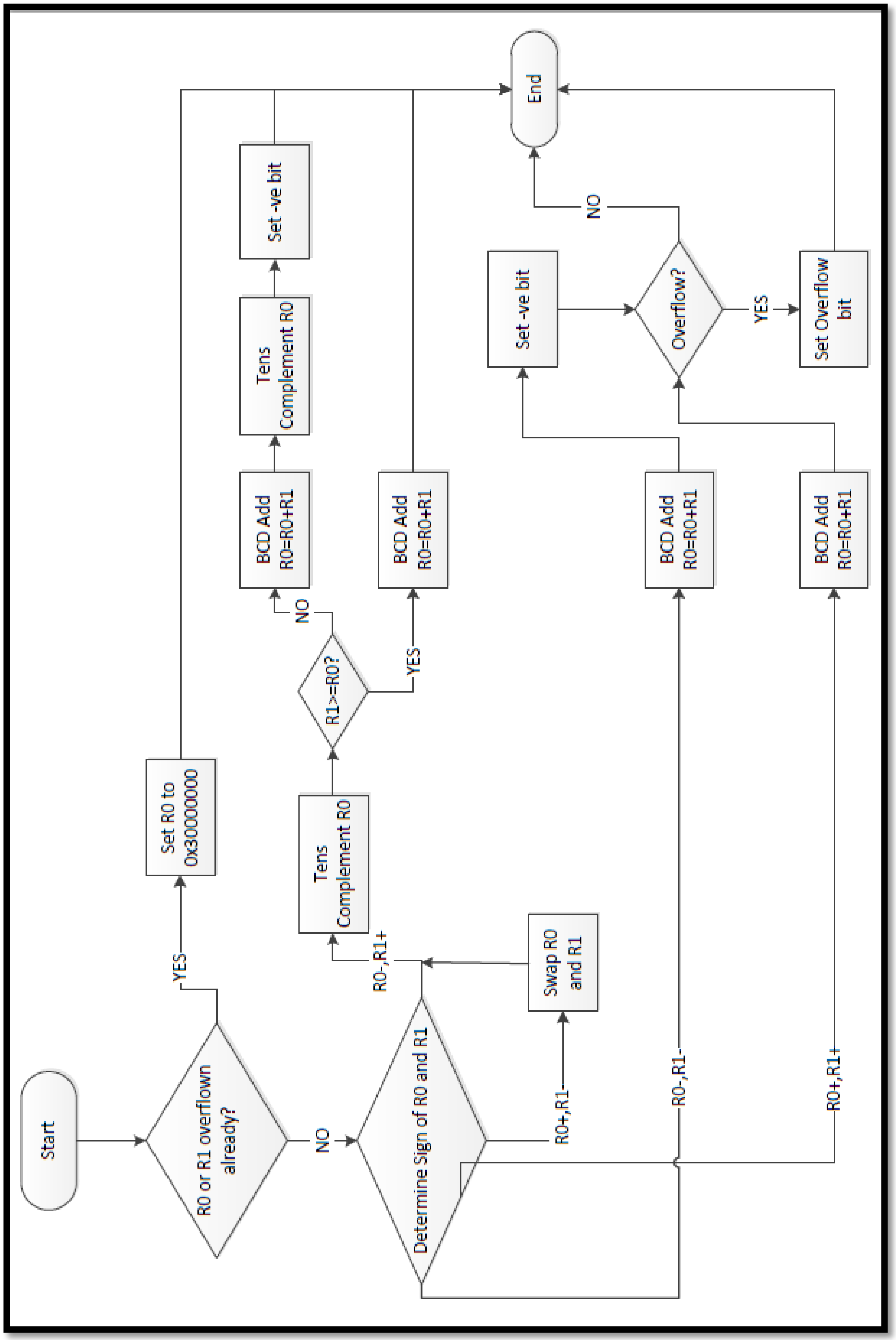

Figure 1 State Diagram of bcdadd

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 1.

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 errors as soon as they are introduced.

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

Table 1 Test cases and their details

Performance

Breakpoints were placed in the wrapper routine right before and after the call to bcdadd for each of the test cases outlined in Table 1. The elapsed time observed in the debugger for each call is noted in Table 2.

Case	Signs	Specs	Time Taken (s)
1	r0+, r1+		20.13μ
2	r0-, r1-		20.00μ
3a	r0+, r1-	(|ro|>|r1|)	38.75μ
3b	r0+, r1-	(|ro|<|r1|)	57.38μ
3c	r0+, r1-	(|ro|=|r1|)	38.75μ
4a	r0-, r1+	(|ro|>|r1|)	57.00μ
4b	r0-, r1+	(|ro|<|r1|)	38.37μ
4c	r0-, r1+	(|ro|=|r1|)	38.37μ
5	r0 , r1	r0 overflown	1.88μ

Table 2 Running time for each test case

Without knowing the frequency of occurrence of each of the test cases, an exact measurement for the average running time cannot be obtained. However, a rough estimate would place the average time taken at about 40 μs.

Finally, one performance tweak that was introduced in the final stages of the design was to place the code handling the swap for case 3 right on top of case 4. This meant that execution would directly roll off into case 4 without the need for an extra branch.

(Part B) Babbage

Functional Specifications

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.

The parameters of the function are:

-*PolyOrder* is the order of the polynomial

-*NumItems* is the number of elements in the array (number of elements to compute + initial elements). New calculations are only performed to the right of the current point

-The *Elements* array is passed via a pointer to the first element. It contains *PolyOrder*+1 items, which are non-zero. Other elements are zero and must be filled by the Babbage routine

Algorithm

1. Copy the initial values from the Elements array into a temporary array to be restored later.
2. Subtract each of the Element's array members from its predecessor and store the difference in the place of Element's member. This process is implemented in a expanding buffer data structure as is described below.

Implementation of Buffer:

For the sake of memory and performance efficiency, it was decided to use an expanding buffer as the data structure to store the newly calculated parameters of the Babbage difference engine. This buffer originally held the element entries that were passed to the function. These entries are later overwritten by the differences between consecutive elements. This process is repeated as many times as the value of the PolyOrder. This make the buffer have entries where the nth difference is stored in (PolyOrder-n) position in the array (in case in the example below, this should correspond to the 3rd iteration since PolyOrder is 3). The content of the buffer at the different stages is illustrated in the diagram below:

Buffer Index	0	1	2	3	4
f(x)	*1*	*8*	*17*	*34*	*NULL*
1st Iteration	7	9	17	34	*NULL*
2nd Iteration	2	8	17	34	*NULL*
3rd Iteration	6	8	17	34	*NULL*
Intermediate Stages	…	…	…	…	…
End of Summation	*6*	*8*	*17*	*34*	**65**

Figure 3 Contents of Buffer at Different Stages of Execution (Based on the Lab Handout Example)

Note: The red colour in the Figure 2 above denotes where new values have overwritten the original values of the buffer. The green color denotes entries that are appended to the end of the buffer thereby filling it up with the function results.

3. Sum up the values in the buffer to produce the next consecutive f(x) value. Append this value of f(x) to the end of the buffer (this can be seen in the figure above where the entry that extended the buffer is printed in green). Repeat this process until the buffer is filled up with all the required number of items.
4. Copy back the initial values of the elements from the temporary buffer back to the original buffer.
5. Before returning the pointer to the result buffer, we free memory of the temporary buffer. This is not necessarily needed for the purposes of this lab as there is no other threads/resources that are running concurrently which need to share the memory, hence we might be using a few extra cycles at the end of the Babbage function for cleanup. However, we still decided to go through with it as it was cleaner to do so in case we did chose to employ this function along with other concurrent threads in the future. We recognize that this is highly recommended since it is a given that memory space is constrained in embedded systems.

Validation

The validation of the Babbage function was done by calling it from the main function that passed given structs containing different polynomials and their corresponding entries (test cases supplied by Umaid Imran). The calculated results of those polynomials were then compared with those of the expected results to ensure the proper functionality of the function.

Performance

In the implementation of the Babbage difference engine, the larger share of the calculation was handled by the bcdadd routine, hence the rest of the function was designed around that in order to make it more efficient. The expanding buffer described above was thought to be the most suitable data structure for the algorithm. Since the buffer was implemented as a C-language array, this implicitly used C pointers to access elements of the buffer which is a speedy implementation. Since some of the values stored in the buffer are used in future calculations, it was also deemed to be a more efficient use of memory. In addition, the memory consumed by the Babbage function for the temporary array is restored and reclaimed for use at the end of the execution.

Elements in Array	Time Taken (s)
9	619.00μ
6	414.25μ
5	34.75μ
5	350.50μ
6	524.13μ
11	1657.75μ

Table 3 Running time for each test case with the Babbage Function

Appendix

bcdadd.s

```
        AREA    PROGRAM, CODE, READONLY

        EXPORT wrapper
        EXPORT bcdadd
        ENTRY

;wrapper to the bcdadd routine that sends it operands and tests the returned sum
wrapper
                        ;test cases
                        ;CASE 1: ro+, r1+
                        LDR r1, =0x00762500
                        LDR r0, =0x00309380

                        BL bcdadd

                        LDR r2, =0x01071880
                        CMP r0, r2
                        BNE error

                        ;CASE 2: r0-,r1-
                        LDR r1, =0x80039785
                        LDR r0, =0x80139962

                        BL bcdadd

                        LDR r2, =0x80179747
                        CMP r0, r2
                        BNE error


                        ;CASE 3a: r0+, r1- (|ro|>|r1|)
                        LDR r0, =0x09656000
                        LDR r1, =0x87847000

                        BL bcdadd

                        LDR r2, =0x01809000
                        CMP r0, r2
                        BNE error

                        ;CASE 3b: r0+, r1- (|ro|<|r1|)
                        LDR r0, =0x07847000
                        LDR r1, =0x89656000

                        BL bcdadd
```

```
LDR r2, =0x81809000
CMP r0, r2
BNE error

;CASE 3c: r0+, r1- (|ro|=|r1|)
LDR r0, =0x09656000
LDR r1, =0x89656000

BL bcdadd

LDR r2, =0x00000000
CMP r0, r2
BNE error


;CASE 4a: r0-, r1+ (|ro|>|r1|)
LDR r0, =0x89656000
LDR r1, =0x07847000

BL bcdadd

LDR r2, =0x81809000
CMP r0, r2
BNE error

;CASE 4b: r0-, r1+ (|ro|<|r1|)
LDR r0, =0x87847000
LDR r1, =0x09656000

BL bcdadd

LDR r2, =0x01809000
CMP r0, r2
BNE error

;CASE 4c: r0-, r1+ (|ro|=|r1|)
LDR r0, =0x89656000
LDR r1, =0x09656000

BL bcdadd

LDR r2, =0x00000000
CMP r0, r2
BNE error

;CASE 5: r0 , r1 have an overflow to start with
LDR r0, =0xF9656000
LDR r1, =0x09656000
```

```
                    BL bcdadd

                    LDR r2, =0x30000000
                    CMP r0, r2
                    BNE error

                    B success

error   B error

success         B success



;bcdadd routine, takes two well formatted BCDs in r0 and r1, and places their BCD sum in r0
bcdadd
        push {LR}

        ;determine if any of r0 and r1 have an overflow
        TST r0, #0x40000000
        BNE overflow
        TST r1, #0x40000000
        BNE overflow

        ;determine sign of r0
        TST r0, #0x80000000
        BNE r0Negative
        B r0Positive

r0Negative
        ;determine sign of r1
        TST r1, #0x80000000
        BNE r0N_r1N
        B r0N_r1P

r0Positive
        ;determine sign of r1
        TST r1, #0x80000000
        BNE r0P_r1N
        B r0P_r1P

;CASE 1: r0 Positive, r1 Positive
r0P_r1P
        BL add
        B checkOverflow ;check overflow only if r0 and r1 are both same sign

;CASE 2: r0 Negative, r1 Negative
r0N_r1N
        BL add
```

```
            ORR r0, #0x80000000 ;set the negative bit
            B checkOverflow             ;check overflow only if r0 and r1 are both same sign

;CASE 3: r0 Positive, r1 Negative
r0P_r1N
            ;swap and move on to case 4 (no branch needed, just roll off)
            MOV r2, r1
            MOV r1, r0
            MOV r0, r2

;CASE 4a: r0 Negative, r1 Positive (r1>=|r0|)
r0N_r1P
            AND r0, #0x0fffffff ;clear last nibble since now we've already processed overflow and sign
flags
            AND r1, #0x0fffffff

            CMP r0, r1          ;moved comparison before the first tensComplement (didn't work after)
            BGT r0N_r1P_GT

            BL tensComplement
            BL add ;r1 needs to be larger than or equal to r0 (before complement operation) for the result
to be positive
            B stop

;CASE 4b: r0 Negative, r1 Positive (r1<|r0|)
r0N_r1P_GT
            BL tensComplement
            BL add
            BL tensComplement
            ;add negative sign to result
            ORR r0, #0x80000000 ;set the negative bit
            B stop


;normal exits
checkOverflow
            ;determine if overflow, and set overflow bit if the case
            TST r0,#0x10000000
            MOVNE r0,#0x40000000 ;set the overflow bit if there was an overflow

stop
            ;clear don't care bits for uniformity
            AND r0, #0xCFFFFFFF

            pop {LR}
            BX LR

;abnormal exits
overflow
```

```
            LDR r0, =0x30000000 ;return special value if passed values have overflow to begin with

            pop {LR}
            BX LR

 ;computes tensComplement of r0 and stores it in r0
 ;assumes properly formatted BCD in r0
tensComplement
            push {r1,LR}

            ;constructing the number #0x09999999
            LDR r2, =0x09999999

            ;9's complement
            RSB r0, r0, r2

            ;add one using our adder routine
            MOV r1, #1
            BL add

            pop {r1,LR}
            BX LR
;assumes well formatted BCDs (i.e. no nibble over 9)
 ;adds +ve BCDs of 7 nibbles stored in r0 and r1 and returns result in r0
add
            push {r2-r7} ;save context of used scratch registers

            MOV r4, #0 ;r4 will contain the result
            MOV r3, #0 ;r2 and r3 are for additions (r3 contains result)
            MOV     r2, #0

            ;values used for special purposes (shifted left after every nibble addition)
            MOV r5, #0x0000000f ;used as mask to extract nibble
            MOV r6, #0x00000009   ;used to determine if sum greater than 9
            MOV r7, #0x00000006 ;used to add 6 to a sum if greater than 9

nibble_add_loop
            ;isolate the nimbles to be added
            AND r2, r0, r5
            AND r3, r1, r5

            ;add the isolated nibbles
            ADD r3, r2, r3

            ;add the new nibble sum to the overall sum
            ADD r4, r4, r3

            CMP r4, r6 ;perform comparison here in order to include carry from previous nibble add
            ADDGT r4, r7
	;shifting of special values
	;note that shifting is not done during operation (using barrel shifter) because the shifted value
is not persisted
	LSL r5, #4
	LSL r6, #4 ;shift of r6 replicates the 9's for comparison purposes with r4
	ADD r6, #9
	LSL r7, #4

	;determine if we need to pursue adding nibbles
	CMP	r5, #0xf0000000
	BNE nibble_add_loop ;when the comparison produces a result of 0, don't branch again

	MOV r0, r4	;store result in r0
	pop {r2-r7} ;restore context

	BX LR

END
```

Babbage.c

```
/*
// A Main Program to call the subroutines
*/
#include <stdlib.h>
#include <stdio.h>
#include <stm32f10x_lib.h>
#include "bcd.h" // declare bdc_t – could be just unsigned long or alike

//Define to print to stdOutput Debug:
#define PRINT_DEBUG
//Define to Free memory before exiting the babbage function:
#define FREE_MEM

typedef struct TestCase {
            int PolyOrder;
            int NumItems;
            bcd32_t *elements;
            bcd32_t *expected;
} TestCase;

/* AA The following definition of elements and expected are outside main to give them a constant
address */
bcd32_t elements1[9] = { 0x1, 0x8, 0x17, 0x34, 0x0, 0x0, 0x0, 0x0, 0x0 };
bcd32_t expected1[9] = { 0x1, 0x8, 0x17, 0x34, 0x65, 0x116, 0x193, 0x302, 0x449 };

bcd32_t elements2[6] = { 0x80000001, 0x80000002, 0x80000005, 0x2, 0x31, 0x116 };
bcd32_t expected2[6] = { 0x80000001, 0x80000002, 0x80000005, 0x2, 0x31, 0x94 };

bcd32_t elements3[5] = { 0x6, 0x6, 0x1, 0x2, 0x0 };
bcd32_t expected3[5] = { 0x6, 0x6, 0x6, 0x6, 0x6 };

bcd32_t elements4[5] = { 0x4, 0x8, 0x36, 0x112, 0x0 };
bcd32_t expected4[5] = { 0x4, 0x8, 0x36, 0x112, 0x260 };

bcd32_t elements5[6] = { 0x80000005, 0x80000011, 0x80000023, 0x80000047, 0x0, 0x0 };
bcd32_t expected5[6] = { 0x80000005, 0x80000011, 0x80000023, 0x80000047, 0x80000089,
0x80000155 };

bcd32_t elements6[11] = { 0x1, 0x2, 0x129, 0x2188, 0x16385, 0x78126, 0x279937, 0x823544, 0x0, 0x0,
0x0 };
bcd32_t expected6[11] = { 0x1, 0x2, 0x129, 0x2188, 0x16385, 0x78126, 0x279937, 0x823544,
0x2097153, 0x4782970, 0x40000000 };

void main(void){ // main neither has arguments nor returns anything

            TestCase tests[6] = {{3, 9, &(elements1[0]), &(expected1[0])},
                        {3, 6, &(elements2[0]), &(expected2[0])},
                        {0, 5, &(elements3[0]), &(expected3[0])},
                        {3, 5, &(elements4[0]), &(expected4[0])},
                        {3, 6, &(elements5[0]), &(expected5[0])},
                        {7, 11, &(elements6[0]), &(expected6[0])}
                        };

            int i;
            int j;

            for (i=0;i<6;i++){
                    printf("Expected: ");
                    for (j=0;j<tests[i].NumItems;j++)
                            printf("%x ", tests[i].expected[j]);
                    printf("\n");

                    babbage(tests[i].PolyOrder,tests[i].NumItems,tests[i].elements);
          printf("");
            }
}

// babbage uses finite differences to evaluate the function value x=f(p)
// list parameters here as bcd numbers
void babbage(unsigned int PolyOrder, unsigned int NumItems, bcd32_t* Elements) { // Your code
            //must perform checks that all arguments are properly sent

            //this method uses the passed Elements array to store differences and sums (and eventually
results)
            //a temporary array is used to store initial values
            //uses a moving window of 2 elements

            //declarations
            int endI=PolyOrder;
            int i;
            int resultsI; //where to store the computed sum
            //scratch x for iteration
            int x;

            //copy the initial values from Elements into an array (as they will be overwritten with
diffs/sums)
            bcd32_t* tempInitial;
            tempInitial=(bcd32_t*)malloc(sizeof(bcd32_t)*PolyOrder+1);
            for (x=0;x<PolyOrder+1;x++){
                    tempInitial[x]=Elements[x];
            }

            //obtain the differences till constant row
            while(endI>0){
                    for (i=0;i<endI;i++){
                            //Elements[i]=Elements[i+1]-Elements[i];
                            //*-1 for subtraction
```

```
				Elements[i]=bcdadd(Elements[i+1],Elements[i] ^ 0x80000000 );
			}
			endI--;
		}

		//perform sums to obtain new results (result will be placed in last non-zero cell)
		resultsI=PolyOrder+1;
		while (resultsI<NumItems){
			for(i=0;i<PolyOrder;i++){
				Elements[i+1]=bcdadd(Elements[i],Elements[i+1]);
			}
			//copy last computed sum which will be a result of the function
			Elements[resultsI]=Elements[i];
			resultsI++;
		}

		//copy over to Elements the initial values that were overwritten
		for (x=0;x<PolyOrder+1;x++){
			Elements[x]=tempInitial[x];
		}

	#ifdef PRINT_DEBUG
		printf("Results: ");
			for (x=0;x<NumItems;x++)
				printf("%x ", Elements[x]);
		printf("\n");
	#endif

	#ifdef FREE_MEM
		free(tempInitial);
	#endif FREE_MEM
	}
```