ECSE 426 - MICROPROCESSOR SYSTEMS

LAB 1 REPORT

ASSEMBLY AND EMBEDDED C

Group 6

Hossam Al-Saati 260169285
Vinod Sridharan 260256868

1. INTRODUCTION

This lab was made of a single library comprised of three functions. One of the functions was defined in assembly, while the other functions were defined in embedded C. Note that for the rest of the lab, the data type unsigned long has been redefined as 'bcd32_t'.

2. FUNCTIONAL SPECIFICATIONS

Function: void bcdadd (bcd32_t *c1, bcd32_t *c2)
Purpose: Performs a Binary Coded Decimal (BCD) addition of two signed 7-digit decimal numbers. A 32-bit representation is used to hold the 7-digit number, a sign bit and an overflow indication flag bit. The output of the addition is written the memory location of c2.
Inputs: c1, c2 – pointers to memory locations storing two 32-bit BCD formatted numbers. c1 and c2 could be positive or negative, or overflow
Output: *c2 = *c1 + *c2. The result of the addition is placed in the memory location pointed to by c2. The result will be of the same format of the two inputs.
Special Cases/Error Conditions:

- Overflow: if *c1 + *c2 is greater than 28 bits, *c = 0x10000000
- The error case when any digit is greater than 9 (e.g. a digit is A to F) is not taken care of by the function. The library assumes that the user can not enter an input which does not evaluate to a BCD number.

Function: void fir(int numberInputs, int filterOrder, bcd32_t *coeffs, bcd32_t *inputs, bcd32_t *output)
Purpose: Performs a filter of order 'filterOrder' on an arbitrary input specified by the user, given the user-supplied coefficients.
Inputs: numberInputs: The number of input signal points; filterOrder: The order of the filter to apply (the number of coefficient values); *coeffs: an array of length filterOrder containing the filter coefficients scaled by 1000 (between 0 and 1000); *inputs: an array of length numberInputs containing the input signal values scaled by 1000 (between 0 and 1000); *output: an array of length numberInputs initialized to 0 (memory allocated).
Output: *output: an array of length numberInputs, where each term is the weighted sum of the previous 'filterOrder' inputs. The weights are provided by the coeffs array.
Special Cases/Error Conditions:
When the input at any index is 0, it indicates a premature end of the signal, and the filter should treat that special case by setting that and further outputs are 0.

Function: void bcdmult(bcd32_t *arg_a, bcd32_t *arg_b, bcd32_t *result)
Purpose: Performs a Binary Coded Decimal (BCD) multiplication of two signed 7-digit decimal numbers.
Inputs: arg_a: a pointer to the address in memory were the multiplier is stored; arg_b: a pointer in memory where the multiplicand is store; result: a pointer to the address in memory where the product is stored. The method requires the bcdadd function to compute the output
Output: *result = *arg_a x *arg_b. The product is stored in the memory location specified by the input argument. The result is a 32 bit BCD number with the same format as the input.
Special Cases/Error Conditions:

- Overflow: if *arg_a x *arg_b is greater than 999, 999: *product = 0x10000000

3. IMPLEMENTATION

The bcdadd function was coded in the Texas Instruments MSP430 IAR assembly language, while the multiplication and filter functions were coded in embedded C. Table 1 below defines the pre-defined BCD format

32-bit unsigned long values. The BCD format basically uses a 7 hexadecimal coded numbers from 0 to 9, 1 sign bit and 1 overflow bit.

Bit Numbers	**Purpose**
31	Sign bit: 0 for a positive value and 1 for a negative value
29-30	Don't care bits
28	Overflow bit: 0 for no overflow and 1 if an overflow occurs
24-27	Most significant decimal value
20-23	6^{th} decimal value
16-19	5^{th} decimal value
12-15	4^{th} decimal value
8-11	3^{rd} decimal value
4-7	2^{nd} decimal value
0-3	Least significant decimal value

Table 1 - 32-bit BCD Format Specification

The BCD adder, programmed in assembly, considered the addresses of inputs specified in R14, and R15. After checking if inputs are overflown, there are 4 main cases for input values into the system that needs to be dealt with: a + b, a – b, -a + b, -a – b; where a and b are the input values to the function.

In the function bcdadd, register R9 and R10 were used as flag bits to keep track of which inputs are oveflown, and negative. Later in the code, they were used as status bits to indicate the usage of post-addition subroutines such as negation, 10's complement inversion, or an overflow check at the end of the program.

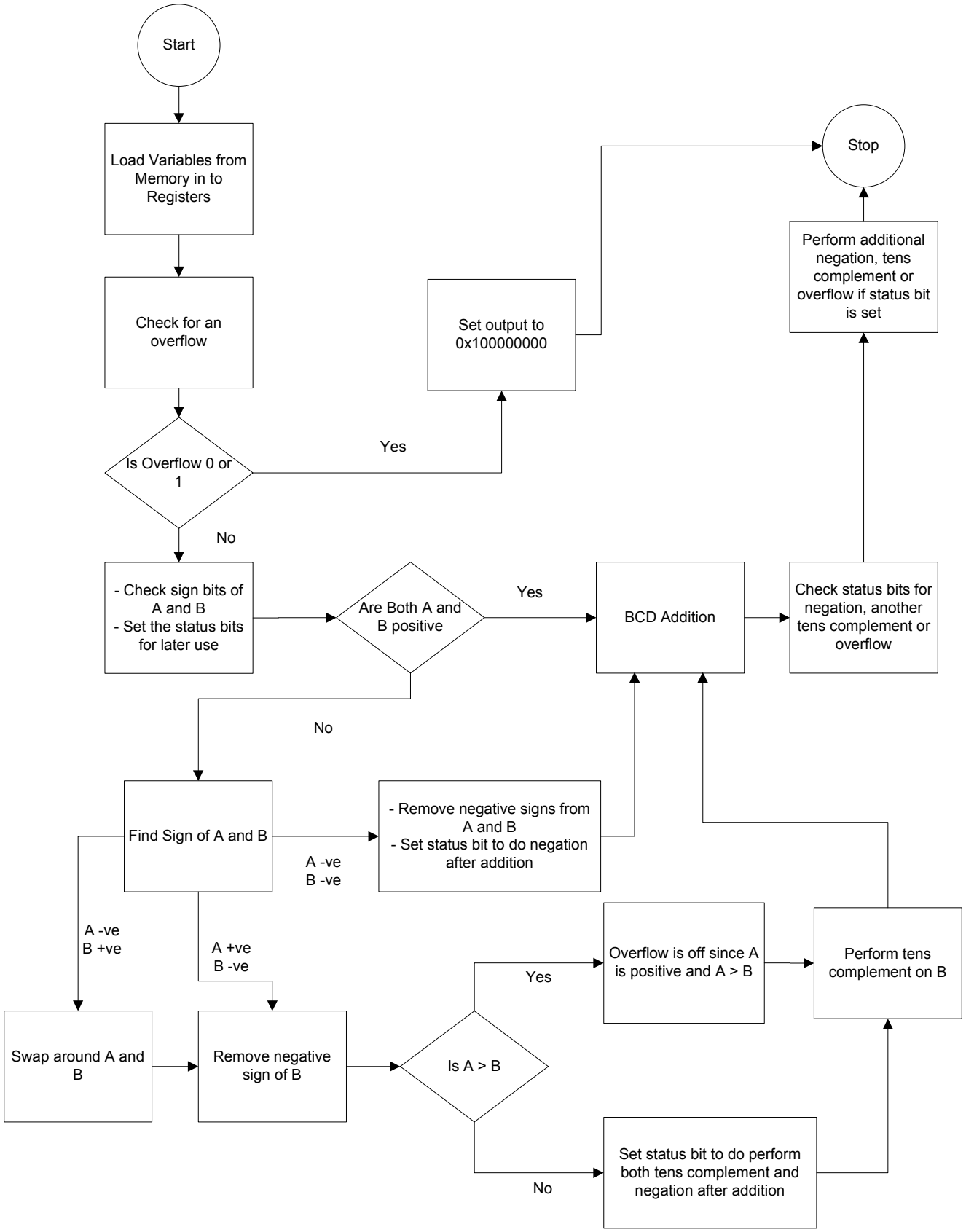

Figure 1: Program flow of bcdadd

The bcdmult function is coded entirely in C, and uses frequent calls to the above defined bcdadd function to perform BCD multiplication. The implementation of the multiplication function is analogous to the long multiplication technique in mathematics.

If the product of two numbers ends up being greater than 9, 999, 999, then the bcdadd function automatically detects the overflow, and sets the product to 0x10000000. Consequently, the bcdmult function does not require an explicit flag set for overflow. It should be noted that the case the output is only negative when the inputs have opposite signs. Consequently, the xor of the signs of arg_a and arg_b will naturally generate the sign of the output.

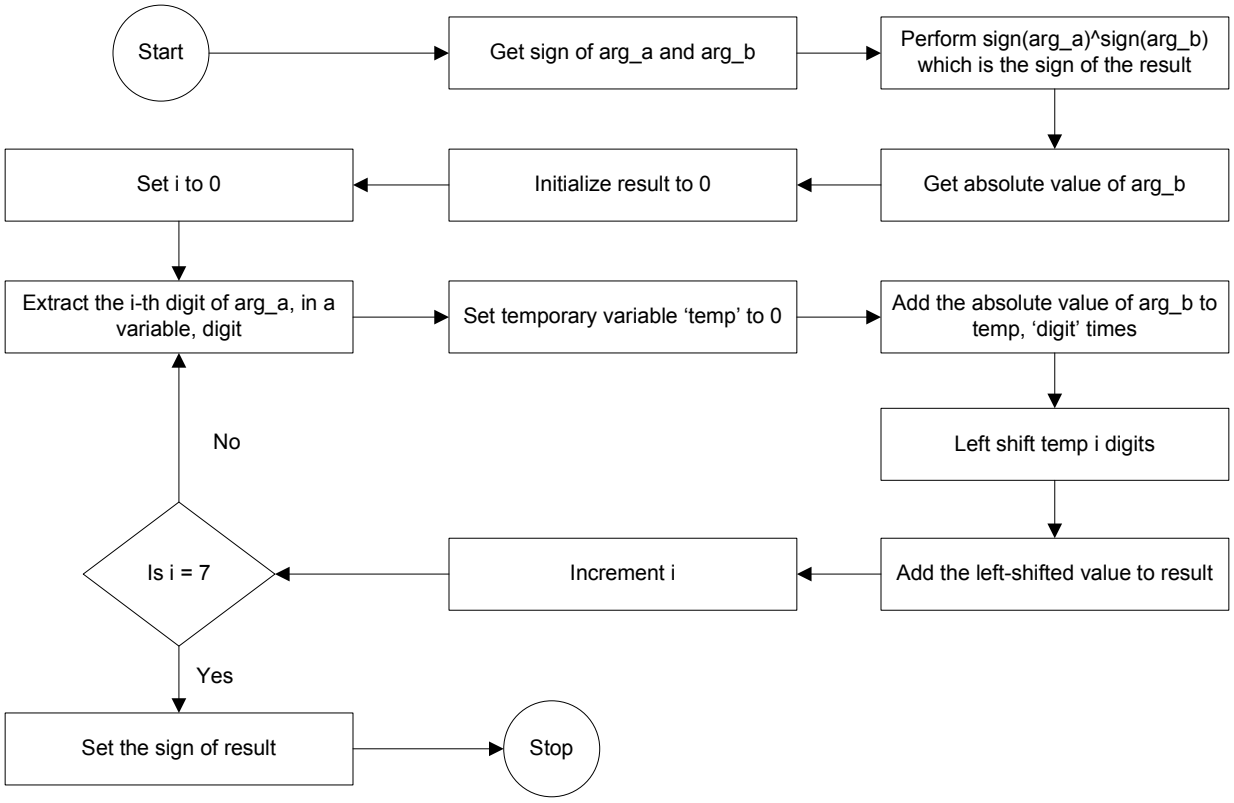

Figure 2: Program flow of bcdmult

The fir function generates a filter output for an arbitrary input of length numberInputs. For a filter of order filterOrder, every output at position 'i' is a weighted sum of the inputs from (i-filterOrder) to i, assuming (i-filterOrder) >= 0. If the condition does not hold true, then the output value equals the input value. The weights themselves were provided as an input to the function as the '*coeff' argument. It should be noted that all inputs and coefficients are scaled by 1000 (e.g. 0.4 is scaled to 400). Consequently, the output is 1000 times greater than the scaling of the input. Therefore, the output was shifted right 3 places to make sure that the input and output were scaled the same amount.

$$output[i] = \sum_{j=0}^{filterOrder-1} Coeff[j] * input[i - filterOrder + j] for\ (i - filterOrder) \geq 0$$

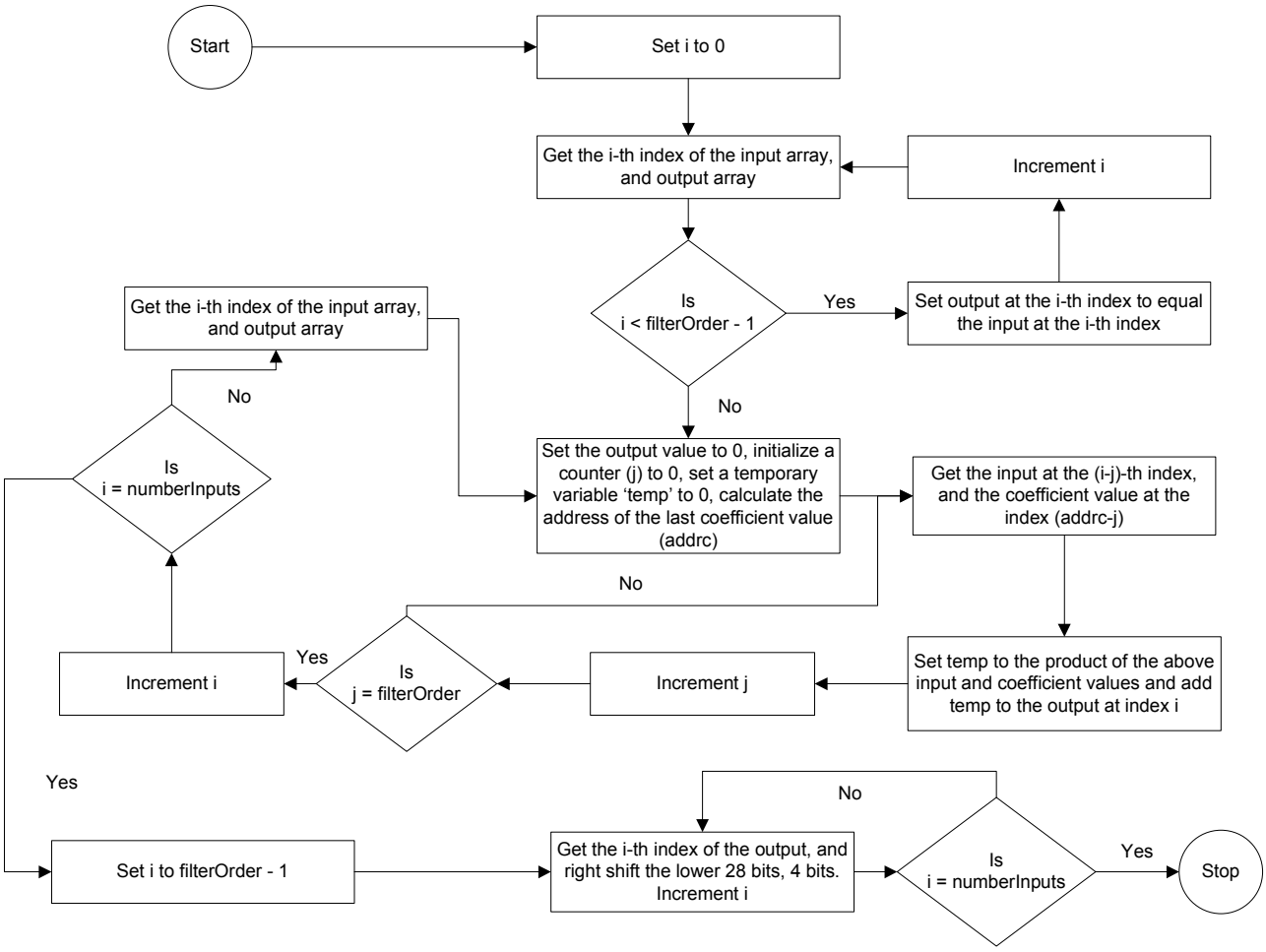

Figure 3: Program flow for fir function

4. PERFORMANCE ANALYSIS

In order to verify the integrity of the functions designed, each component of the library was tested extensively. For the bcdadd function, each of the 5 input cases, a + b, a - b, -a + b, -a - b, and the cases where the inputs had overflow, were tested. The table below shows each of the five test cases, and the output that the function returned. The efficiency of the method in each case is also highlighted

Function: Condition	**Input**	**Output**	**Cycles**
bcdadd(&a,&b): a + b	a = 0x00145312 b = 0x07910234	b = 0x08055546	90
bcdadd(&a,&b): -a + b	a =0x88945312 b =0x03910234	b = 0x85035078	134
bcdadd(&a,&b): a - b	a = 0x00145300 b = 0x87910200	b = 0x87764900	134
bcdadd(&a,&b): -a - b	a = 0x80045300 b = 0x80010200	b = 0x80055500	108
bcdadd(&a,&b): a + b (overflow)	a = 0x03145300 b = 0x07910200	b = 0x11055500	90

Table 2: Test cases and performance of bcdadd

For the bcdmult function, the cases to consider were exactly the same, and in each case the results and the efficiency are shown below:

Function: Condition	Input	Output	Cycles
bcdmult(&a,&b,&c): +a * +b	a = 0x00005880 b = 0x00062439	c =0x10000000	62,157
bcdmult(&a,&b,&c): -a * +b	a = 0x80093433 b = 0x00000443	c =0x5312819	52,554
bcdmult(&a,&b,&c): +a * -b	a = 0x00051932 b = 0x00003411	c =0x10000000	71,719
bcdmult(&a,&b,&c): -a * -b	a = 0x800343322 b = 0x00000022	c =0x80036630	67,818
bcdmult(&a,&b,&c): a * b (overflow)	a = 0x023422241 b = 0x03431331	c =0x82834869	127,568

Table 3: Test cases and performance of bcdmult

Testing the fir function was more extensive and required more test cases. Since inputs range from 0V to 1V, and coefficients also range from 0 to 1, the highest possible output is 999*999 which is 998001, and consequently cases of overflow need not be tested by the fir function. The table below shows an input of length 12:

500	500	500	700	700	700	400	100	800	1000	0	0

Table 4: Test input for the fir function

The above input was designed to test all possible orders (up to the order 10) and test the case where the signal ended prematurely, in which case the fir function was supposed to stop computing filter values, and set the output to 0 for those respective indices. The following table shows various filter coefficients, and the efficiency of each case

Filter Order	Coefficients	Output	Cycles
3	{333, 333, 333}	{500, 500, 490, 560, 630, 690, 590, 390, 430, 630, 590, 330}	951,298
5	{300, 500, -100, 400, 600}	{500, 500, 500, 700, 1050, 1030, 950, 710, 1040, 1320, 490, 330}	985,460
2	{400, 600}	{500, 500, 500, 620, 700, 700, 520, 220, 520, 920, 400, 0}	898,677
7	{-100, -100, -100,-100,-100,-100,-100}	{500, 500, 500, 700, 700, 700, -400, -360, -390, -440, -370, -300}	959,545

Table 5: Test cases and performance of fir

4. REFERENCES

[1] 304-426A Microprocessor Systems Fall 2009 Lab1: Assembly and Embedded C, McGill University, Fall 2009
[2] Texas Instruments, MSP430x1xx Family User's Guide, Mixed Signal Products, 2006

5. APPENDIX

Appendix 1 – main.c code

```
// A main program to call the subroutines
#include <msp430x14x.h>
#include <cross_studio_io.h>
#include "bcd.h"
void main(void){ //main neither has arguments nor returns anything
bcd32_t a[] = {0x00145312, 0x88945312, 0x00145300, 0x80045300, 0x03145300};
bcd32_t b[] = {0x07910234, 0x03910234, 0x87910200, 0x80010200, 0x07910200};
bcd32_t a_mult[]={0x00005880, 0x80093433, 0x00051932, 0x800343322, 0x023422241};
bcd32_t b_mult[] = {0x00062439, 0x00000443, 0x00003411, 0x00000022, 0x03431331};
bcd32_t result[] = {0x00, 0x00, 0x00, 0x00, 0x00};
bcd32_t input[] = {0x500, 0x500, 0x500, 0x700, 0x700, 0x700, 0x400, 0x100, 0x800,
0x1000, 0x00, 0x00};
bcd32_t coeffs[] = {0x80000100, 0x80000100, 0x80000100, 0x80000100, 0x80000100,
0x80000100, 0x80000100};
bcd32_t output[] = {0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00};
int j;
for(j = 0; j < 5; j++){
bcdadd(&a[j],&b[j]);
debug_printf("Addtion \n %8.8lX \n",b[j]);
}
for(j = 0; j < 5; j++){
bcdmult(&a_mult[j],&b_mult[j],&result[j]);
debug_printf("Multiplication \n %8.8lX \n",result[j]);
}
for(j = 0; j < 12; j++){
debug_printf("%08lx \t",input[j]);
}
debug_printf("\n");
f1r(12,7UL, &coeffs[0],&input[0],&output[0]);
for(j = 0; j < 12; j++){
debug_printf("%lx \t",output[j]);
}
debug_printf("done\n");
return;
}
//bcdmult multiplies two 7 digit BCD numbers and returns the product in
//works similar to long multiplication, and uses a series of additions
//to achieve the result
void bcdmult(bcd32_t *arg_a, bcd32_t *arg_b, bcd32_t *result){
//arg_a * arg_b needs to be performed
int i,j,digit; bcd32_t temp,signoutput,numb;
temp = *arg_a & 0x80000000; //extract sign of A
signoutput = *arg_b & 0x80000000; //extract sign of B
signoutput = signoutput ^ temp; // 1 if negative 0 for positive
*result = 0; //initialise the result (just in case)
numb = *arg_b & 0x1FFFFFFF;//get abs of B
for(i = 0; i < 7; i++){ //iterate through each digit
digit = (*arg_a & (0xF << (i*4)))>>(i*4); //extracts digit of A
temp = 0;
for(j = 0; j < digit; j++){
bcdadd(&numb,&temp); //multiply
```



```
}
temp = temp << (i*4); //shift back accordingly
bcdadd(&temp, result); //add to result
}
*result = *result | signoutput; //fix the sign of the output
}
// Comments explain your fir function.
void f1r(unsigned int numberInputs, unsigned int filterOrder, bcd32_t *coeffs,
bcd32_t *inputSamples, bcd32_t
int i,j; bcd32_t *addri, *addrc, *addro; bcd32_t temp,temp2; //temporary variables
used everywhere
for(i= 0; i < filterOrder-1; i++){ //if there aren't enough previous data points,
input = output
addri = inputSamples + i;
addro = outputValue + i;
*addro = *addri;
}
for(i=filterOrder-1;i<numberInputs;i++){ //iterate through each data point till the
end of the array (signified
addri = inputSamples + i;
addro = outputValue + i;
addrc = coeffs + (filterOrder-1);
*addro = 0; //set the value at that point to be 0
for(j = 0; j < filterOrder; j++){ //iterate through all the points on the filter
temp = 0; //set the product initially to be zero
bcdmult(addrc-j,addri-j,&temp); // multiply
bcdadd(&temp,addro); //add the product to the output
}
addri = inputSamples + i; //get the next input
addro = outputValue + i; //get the next output
}
addro = outputValue + (filterOrder - 1);
for(i=filterOrder-1; i < numberInputs;i++){
temp = (*addro & 0x0FFF0000) >> 12;
temp2 = (*addro & 0xF0000000);
*addro = temp | temp2;
addro = addro + 1;
}
}
```

Appendix 2 – bcd.h code

```
//creating a new type called bcd32_t that represents the unsigned long type used in
bcd problems
typedef unsigned long bcd32_t;
//adds 2 BCD numbers (signed 7 digit)
extern void bcdadd(unsigned long *c1, unsigned long *c2);
// function is coded in assembly
// performs signed multiplication between 2 7 digit BCD numbers
void bcdmult(bcd32_t* arg_a, bcd32_t* arg_b, bcd32_t* result);
void f1r(unsigned int numberInputs, unsigned int filterOrder, bcd32_t* coeffs,
bcd32_t* inputSamples, bcd32_t* outputVale);
```

Appendix 3 – bcdadd.s43 code

```
; bcdadd.s43
; bcdadd adds 2 signed 7 digit bcd numbers whose addresses are
; in registers 14 and 15.
; Convention: bit 0 to 27: 7 digits of the number
; bit 28: overflow bit
; bit 31: sign bit
PUBLIC _bcdadd ; Declare symbol to be exported
RSEG CODE ; Code is relocatable
_bcdadd: ; BCD ADD assembly routine
save_context: push R4 ; MSB of A
push R5 ; LSB of A
push R6 ; MSB of B
push R7 ; LSB of B
push R8 ; scratch reg - used to track a negative number, , overflow and post-
addition funcations
push R9 ; scratch reg
push R10 ; scratch reg
; for bcdadd(&a, &b), &a is in R15, and &b is in R14
assign_variables: mov @R15, R5 ;moves B into R4-R5
mov 2(R15), R4
mov @R14, R7 ;moves A into R6-R7
mov 2(R14),R6
clr R12 ;Sign of result
clr R8 ;sign status of A and B
overflow_check: mov #0x1000, R9 ; check if inputs are overflown
mov R4, R10
and R9, R10 ;check if B is overflown
cmp R9,R10
jeq overflown
mov R6, R10
and R9, R10
cmp R9, R10
jeq overflown
signB: mov #0x8000, R9 ;mask to get the sign
mov R4, R10
and R9, R10 ;get the MSB of B
cmp R9, R10 ;if B is -ve, cmp returns 0
jne signA ;if B +ve, move to test A
add #0x0001, R8 ;else B is negative
signA: mov R6,R10
and R9,R10 ;get MSB of A
cmp R9,R10 ;if A -ve, cmp returns 0
jne divide ;if A positive go to tester
add #0x0010, R8 ;else A negative
divide: clr R9
cmp R8,R9 ;if 00, then A + B
jeq addition
mov #0x0001, R9
cmp R8,R9 ;if 01, then A - B
jeq AminusB
mov #0x0010, R9
cmp R8, R9 ;if 10, then B - A
jeq BminusA
negativeAdd: ;else all negative
```



```
mov #0x8000, R12 ;set sign of ans -ve
mov #0x0FFF, R9
and R9, R4 ;clear sign of B
and R9, R6 ;clear sign of A
addition: clrc
clrn
dadd R7, R5 ; add R5 = R5 + R7
dadd R6, R4 ; add R4 = R6 + R4 + carry
bis R12, R4 ;set sign if negative
jmp restore_memory
AminusB: mov R4, R9 ; switch A with B and continue below
mov R5, R10
mov R6, R4
mov R7, R5
mov R9, R6
mov R10, R7
BminusA: mov #0x0FFF, R9 ; get sign mask
and R9, R6 ;remove sign of A
mov #0x9999, R9 ; sign mask
sub R7, R9
mov R9, R7
mov #0x0999, R9
sub R6, R9 ;calculate A'
mov R9, R6
clrc
dadd #0x01, R7
dadd #0x00, R6 ; A' is done
clrc
dadd R7, R5
dadd R6, R4 ; add the two
mov #0x1000, R12
mov R4, R13
and R12, R13
cmp R12, R13
jeq allsfine ;if carry is set, ans is +ve so continue
mov #0x9999,R9 ;else ans is -ve, take 10s complement and set sign
sub R5, R9
mov R9, R5
mov #0x0999, R9
sub R4, R9
mov R9, R4
clrc
dadd #0x01, R5
dadd #0x00, R4 ; A'' is done
mov #0x8000, R12 ; set the sign bit
bis R12,R4
jmp restore_memory
overflown: mov #0x0000, R5
mov #0x1000, R4
jmp restore_memory
allsfine: mov #0x0FFF, R12
and R12, R4
restore_memory: mov R5, @R14
mov R4, 2(R14)
restore_context: pop R10
pop R9
pop R8
```

```
pop R7
pop R6
pop R5
pop
```