EXPERIMENT 1 – DIGITAL FILTERING

Mathieu Perreault (260158758)
Logan Smyth (260179735)
Lab Group #2

1. FUNCTIONAL REQUIREMENTS

The global deliverables in this experiment constitute a library of functions. In Part A of the experiment, the deliverable is coded in Assembly language. In Part B, the different functions are coded in embedded C. Together, they form the deliverable library.

The deliverables use a special format for data, called the 7-digit binary-coded decimals (BCD). Since it is a custom type made to fit within 32-bit values, a special format was specified in the requirements [1]:

A 32-bit representation example is:
• Bit 31 (MSB) is the sign bit: '0' for '+value', '1' for '-value',
• Bit 30 is the overflow flag: '0' for 'not overflown', '1' for 'overflown',
• Bit 29,28 are don't care bits,
• Bit 27-0 are data magnitude bits.

This format was implemented like specified above with the use of the *unsigned long int* masked as bcd32_t in our purposes. This special type implemented the list of requirements above.

1.1 Part A Requirements

Part A of this experiment had exactly one (1) deliverable.

Function: void bcdadd(bcd32_t* val1, bcd32_t* val2)
Purpose: This function should add two 32-bit 7 digit binary-coded decinal integers and store the result at the address of the 2nd result. It will cover adding two positive or negative values asn well as adding a negative and positive number.
Input: val1, val2 – pointers to two bcd integer values
Output: result = val1 + val2 – the final result is stored in the memory location passed in the second argument and will contain the result of the addition or signal overflow during addition
Special Cases/Error Conditions:

- positive + positive and negative + negative can potentially overflow and will return 0x40000000 if this is the case
- positive + negative cannot overflow but can result in either positive or negative values, which must be accounted for.

1.2 Part B Requirements

Part B of this experiment had multiple deliverables. The final goal was to "implement functions in embedded C that perform simple filtering on a block of data"[1].

Function: void bcdmult(bcd32_t* arg_a, bcd32_t* arg_b, bcd32_t* result)
Purpose: This function should multiply two 32-bit binary-coded decimal (BCD) values and store the result in another 32-bit memory location in BCD form. It should cover all possible corner cases, including multiplication by/of negative numbers as well as overflown numbers.
Input: arg_a, arg_b – pointers to memory locations storing two positive or negative BCD-formatted values.
Output: result = arg_a * arg_b – this memory location should contain the multiplication of the two BCD-coded input arguments. The result should be also coded in BCD form and properly indicated in the case of an overflow with the proper bit set (see representation above).
Special Cases/Error Conditions:

- Follow normal sign conventions – negative * negative = positive, negative*positive = negative, etc.
- If one or more of the operands are overflown (proper bit set), the multiplication will not take place
- If the result or any temporary computation has overflown the proper bit will be set and the function will return.

Function: void fir(unsigned int filterOrder, bcd32_t* coeffs, bcd32_t* inputSamples, bcd32_t* outputValue, unsigned int numInputs)
Purpose: This function implements a digital Finite Impulse Response (FIR) filter on an input stream of data to produce a filtered response. Depending on the coefficients of the filter, this filter may act as a low-pass filter, for example. The formula dictating the filter output is the following [1]:

$$outputValue[n] = coeffs[0]*inputSamples[n] + \ldots + coeffs[N]*inputSamples[n-N]$$

Where n goes from N to $numInputs$ and $N = filterOrder - 1$.

Input:

- filterOrder – unsigned int indicating the order of the filter, also meaning the number of coefficients in the filter.
- coeffs – a pointer to a table of 32-bit BCD-formatted coefficients to the filter. There should be as many as the filterOrder.
- numInputs – an unsigned int indicating how many input samples will be input in the filter system.
- inputSamples – a pointer to a table of 32-bit BCD-formatted inputSamples to the filter. There should be as many as numInputs.

Output:

- outputValue – a pointer to the 32-bit location that will hold the BCD-formatted outputValue to the filter (one at a time).

Special Cases/Error Conditions:

- The function should assume that numInputs is always greater or equal to filterOrder.

- If one or more of the operands are overflown (proper bit set), outputValue will be overflown.

2. IMPLEMENTATION

2.1 Part A Implementation

bcdadd: There are two primary cases that are covered in the algorithm. When adding numbers with the same sign, we much be able to recognize possible overflow. In order to be able to detect overflow in the negative values, we add only the magnitude of the values and watch for overflow into the 8th digit, and then replace the sign for the sign. If overflow is detected, then instead set the overflow bit.

When adding a negative and a positive number, we must also account for the fact that a negative or positive result may occur. To accomplish this, we compare the magnitudes of the values, and record whether or not the output will be positive or negative. If either input value is negative, the compliment of the value is taken and used during the addition, and then the sign is reapplied.

In order to take the compliment of the value, we loop though all seven digits and do (9 - #), and then add 1 to the value.

The addition itself is done by making use of the built in DADD assembly instruction that enables 32 bit decimal addition.

2.2 Part B Implementation

bcdmult: To execute in constant time, this function was implemented using a shifting technique. The algorithm takes one operand (let's say, arg_a) and extract its digits turn by turn. For every digit, it does the bcdadd function on arg_b a number of times equivalent to the digit extracted. It then shifts the result according to the position of the extracted digit in arg_a. For example, multiplying 0x00000123 by 0x00000456 would perform the following: (456+456+456) +(456+456)*10 + (456)*100, where the multiplications are in fact shifting left by a certain number of bits.

fir: This simple function was devised according to the specifications in "Functional Requirements," above. It assumes that *numInputs* is at least equal to *filterOrder*. It separately does all the multiplications of the formula and adds the temporary results to a cumulative result. When the algorithm is done for every coefficient of the filter, the *outputValue* is updated.

3. PERFORMANCE ANALYSIS

3.1 Cycles Consumption

Function name	Min cycles	Max cycles	Average number of cycles (10 meas.)
bcdadd	189	223	206
bcdmult	41003	56565	49143
FIR (filterOrder = 3, to produces one outputValue)	531031	542503	535312

Table 1 – Measure of cycles consumptions for different routines

3.2 Test Cases

BCDADD

Input A	Input B	Expected Output	Real Output	Reason
0x00145312	0x07910234	0x08055546	0x08055546	Standard addition, no OF
0x00145300	0x87910200	0x87764900	0x87764900	Add small positive to big negative
0x03145300	0x07910200	0x40000000(overflow bit 30 set)	0x40000000	Positive overflowing
0x83145300	0x87910200	0x40000000(overflow bit 30 set)	0x40000000	Negative overflowing
0x80000345	0x00500400	0x00500055	0x00500055	Small neg. + Large pos.
0x00000345	0x80500400	0x00500055	0x80500055	Small pos. + Large neg.
0x07910234	0x00145312	0x08055546	0x08055546	Same as 1st , swapped

Table 2 – Test cases for the bcdadd routine

BCDMULT

Input A	Input B	Expected Output	Real Output	Reason
0x00000123	0x00000456	0x00056088	0x00056088	Standard mult, no overflow
0x01111111	0x00000009	0x09999999	0x09999999	Multiplying all the digits
0x00000000	0x07237273	0x00000000	0x00000000	Multiply by zero
0x05000000	0x00000002	0x40000000	0x40000000	Simple case of overflow
0x80000001	0x01234567	0x81234567	0x81234567	Multiply by -1
0x80000456	0x00000123	0x80056088	0x80056088	Negative * positive
0x80000456	0x80000123	0x00056088	0x00056088	Negative * negative

Table 3 – Test cases for bcdmult routine

FIR with filterOrder = 3

Coefficients	Input Samples	Expected Outputs	Real Outputs
0x00000333	0x00000400	--------	--------
0x00000333	0x00000200	--------	--------
0x00000333	0x00000100	0x00233100	0x00233100
	0x00000700	0x00333000	0x00333000
	0x00000625	0x00474525	0x00474525

Table 4 – Test case for FIR routine standard 1/3 filter

FIR with filterOrder = 3

Coefficients	Input Samples	Expected Outputs	Real Outputs
0x80000333	0x00000500	--------	--------
0x80000333	0x00000600	--------	--------
0x80000333	0x00000550	0x80549450	0x80549450
	0x00000700	0x80616050	0x80616050
	0x00000625	0x80624375	0x80624375

Table 5 – Test case for FIR routine negative coefficients

Coefficients	Input Samples	Expected Outputs	Real Outputs
0x00000333	0x80000500	--------	--------
0x00000333	0x80000600	--------	--------
0x00000333	0x80000550	0x80549450	0x80549450
	0x80000700	0x80616050	0x80616050
	0x80000625	0x80624375	0x80624375

Table 6 – Test case for FIR routine positive coefficients but negative input

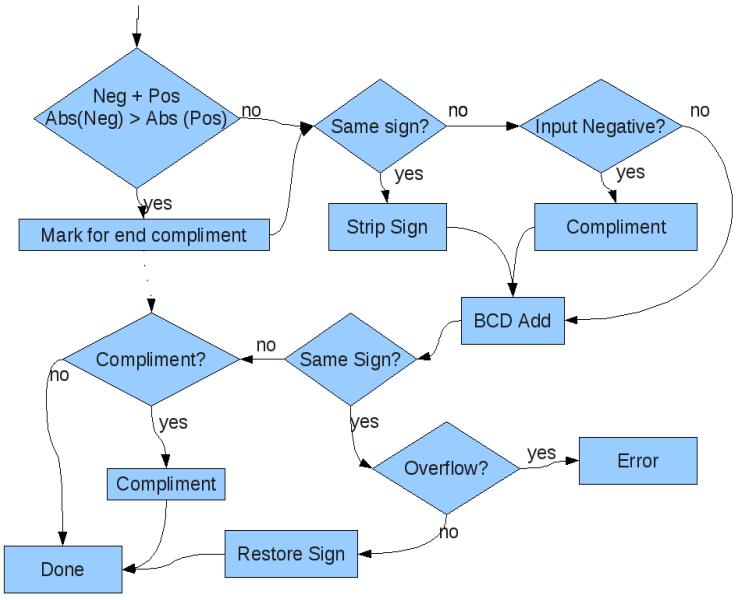
Coefficients	Input Samples	Expected Outputs	Real Outputs
0x00000200	0x80000500	--------	--------
0x00000200	0x80000500	--------	--------
0x00000200	0x80000500	--------	--------
0x00000200	0x80000700	--------	--------
0x00000200	0x80000700	0x80580000	0x80580000
	0x80000700	0x80620000	0x80620000
	0x80000700	0x80660000	0x80660000

Table 7 – Test case for filterOrder 5 and 7 input samples

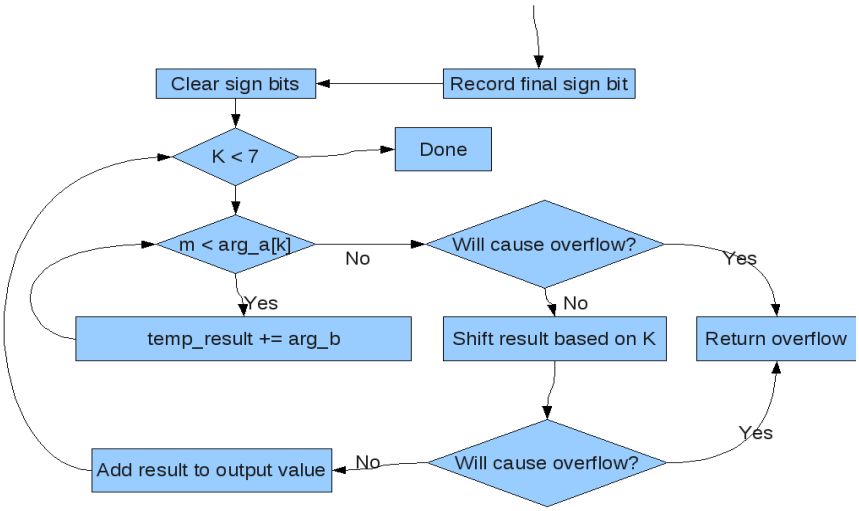
Coefficients	Input Samples	Expected Outputs	Real Outputs
0x00000333	0x00000500	--------	--------
0x00000333	0x00000500	--------	--------
0x00000333	0x00000500	0x00499500	0x00499500
	0x00000700	0x00566100	0x00566100
	0x00000700	0x00632700	0x00632700

Table 8 – Test case for the standard test in Experiment documentation

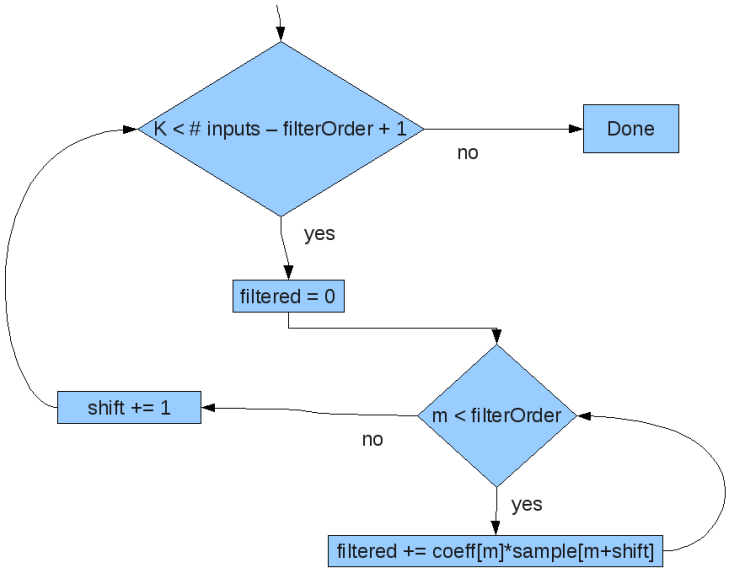
Appendix A: Algorithmic Diagrams



Drawing 1: Flow Chart of Addition Algorithm



Drawing 2: Flow Chart of the Multiplication Algorithm



Drawing 3: Flow Chart of the Filtering Algorithm

REFERENCES

[1] J.S. Chenard, “Lab 1: Assembly and Embedded C,” pp. 1-6, September 8th, 2009.

Appendix – Code

main.c

```
#include <stdio.h>
#include "bcd.h"//routines.h"
#include <cross_studio_io.h>
#include  <msp430x14x.h>

// A Main Program to call the subroutines

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

 int i;
 bcd32_t multResult = 0x0;

 bcd32_t bcdadd_tc[] = {
  0x00145312, 0x07910234, 0x08055546,
  0x87910200, 0x00145300, 0x87764900,  // carry test //this one doesn't work
  0x03145300, 0x07910200, 0x40000000,  // sum test
  0x83145300, 0x87910200, 0x40000000,  // subtraction test
  0x80000345, 0x00000400, 0x00000055,  // small neg + large pos
  0x80000345, 0x00000200, 0x80000145,  // big neg + small pos
  0x00003480, 0x80345800, 0x80342320,  // small pos + big neg
  0x00003000, 0x80000800, 0x00002200,  // big pos + small neg
  0x89999990, 0x80000010, 0x40000000,  // overflow negative
  0x09999990, 0x00000010, 0x40000000   // overflow positive
 };

 bcd32_t bcdmult_tc[] = {
  0x00000123, 0x00000456, 0x00056088, /*mult that doesn't overflow*/
  0x01111111, 0x00000009, 0x09999999, /*looking at all the digits*/
  0x00000000, 0x07237273, 0x00000000,
  0x05000000, 0x00000002, 0x40000000,
  0x80000001, 0x01234567, 0x81234567,
  0x80000456, 0x00000123, 0x80056088,
  0x80000456, 0x80000123, 0x00056088
 };

 bcd32_t coefficients[] = {
  0x00000333, 0x00000333, 0x00000333
 };
 bcd32_t inputSamples[] = {
  0x80000500, 0x80000600, 0x80000550, 0x80000700, 0x80000625
 };


 for (i = 0;i < sizeof(bcdadd_tc)/sizeof(bcd32_t); i = i + 3) { //testing the addition
  bcdadd(bcdadd_tc + i, bcdadd_tc+i+1);
```

```
    if (bcdadd_tc[i+1] & 0x40000000) debug_printf("Addition overflowed ");
    if (bcdadd_tc[i+1] == bcdadd_tc[i+2]) debug_printf("%dth case: gave 0x%1lX, expected 0x%1lX =>OK\n", i/3,
bcdadd_tc[i+1], bcdadd_tc[i+2] );
    else debug_printf("%dth case bcdadd => NOT OK, gave 0x%1lX\n",i/3, bcdadd_tc[i+1]);
  }

  for (i = 0; i < sizeof(bcdmult_tc)/sizeof(bcd32_t); i = i + 3) { //testing the multiplication
    multResult = 0;
    bcdmult(&bcdmult_tc[i], &bcdmult_tc[i + 1], &multResult); //result in multResult
    //compare with multiplication results
    if (multResult & 0x40000000) debug_printf("Multiplication overflowed\n");
    if (multResult == bcdmult_tc[i+2]) debug_printf("bcdmult =>OK\n");

  }

  fir(3, coefficients, inputSamples, &multResult, sizeof(inputSamples)/sizeof(bcd32_t));

  return;
}


// Comments explaining your bcdmult function
void bcdmult( bcd32_t* arg_a, bcd32_t* arg_b, bcd32_t* result){
  int k = 0;
  int m = 0;
  int t = 0;
  int aisneg = 0;
  int bisneg = 0;
  bcd32_t tempdigit = 0x0;
  bcd32_t tempresult = 0x0;
  //extract both sign bits to see the resulting
  unsigned long int sign = ((*arg_a) & 0x80000000) ^ ((*arg_b) & 0x80000000); //if sign is 1 the number is negative
  debug_printf("***Multiplying: 0x%1lX and 0x%1lX\n",*arg_a, *arg_b);

  if (sign != 0) if (*arg_a & 0x80000000) aisneg = 1; else bisneg = 1;

  *arg_a &= ~0x80000000;
  *arg_b &= ~0x80000000; //clearing their MSBs
  *result = 0;
  if (*arg_a & 0x40000000 || *arg_b & 0x40000000){ //one of them is overflown
    //debug_printf("One number is overflown. 0x%1lX or 0x%1lX\n",*arg_a, *arg_b);//what should we do ?
  }


  for (;k < 7; k++){
    tempdigit = (*arg_a >> k*4) & 0xF ;
    //debug_printf("tempDigit %d is 0x%1lX\n", k, tempdigit);
    if (tempdigit){ //if arg_a has a digit in the kth position
```

```
      tempresult = 0x0;
      for (m = 0; m < tempdigit; m++){
        //debug_printf("Adding 0x%1lX to tempresult 0x%1lX\n",*arg_b, tempresult);
        bcdadd(arg_b, &tempresult);
      }
      if ( tempresult & 0xF0000000 ) { //overflown in the tempresult we are about to add
        debug_printf("Addition/shift overflown 0x%1lX\n", tempresult);
        *result = 0x40000000;
        return;
      }
      for (t = 0; t < k; t++){
        tempresult =tempresult << 4;
        if ( tempresult & 0xF0000000 ) { //overflown in the tempresult we are about to add
          debug_printf("Addition/shift overflown 0x%1lX\n", tempresult);
          *result = 0x40000000;
          return;
        }
      }
      //debug_printf("tempResult %d is 0x%1lX\n", k, tempresult);
      bcdadd(&tempresult,result);

    }
  }
  if (sign != 0) {
    *result |= 0x80000000;//if sign was one (negative number) just flip the sign bit
    if (aisneg) *arg_a |= 0x80000000;
    if (bisneg) *arg_b |= 0x80000000;
  }
  debug_printf(" gives 0x%1lX\n", *result);

}
```

```
// Comments explain your fir function.
void fir(unsigned int filterOrder, bcd32_t* coeffs, bcd32_t* inputSamples, bcd32_t* outputValue, unsigned int numInputs) {
  int k = 0;
  int m = 0;
  int numIterations = numInputs - filterOrder +1;
  bcd32_t tempFilterOutput = 0x0;
  bcd32_t tempMultOutput = 0x0;

  for (; k < numIterations; k++){ //let's say from 0 to (20-5) = 15
    tempFilterOutput = 0;
    for (m = 0; m < filterOrder; m++){ //from 0 to 4, let's say
      tempMultOutput = 0;
      bcdmult(coeffs + m,inputSamples + m,&tempMultOutput);
      if (tempMultOutput & 0x40000000){
        debug_printf("Overflow ");
      }; //FIXME if tempMultOutput has overflown ???

      bcdadd(&tempMultOutput, &tempFilterOutput); //result gets put in tempFilterOutput
      if (tempFilterOutput & 0x40000000){
        debug_printf("\nOverflow\n");
      } //FIXME if tempFilterOutput has overflown ???
    }

    *outputValue = tempFilterOutput; //we should have the full output here
    debug_printf("FIR output : 0x%1lX\n", *outputValue);
    inputSamples += 1; //increment the inputSamples pointer by 1, going to next input
    if (k ==0){} ; //dummy line
  }
}
```

bcd.h

```
#define bcd32_t unsigned long int

/**
 * Adds two 32-bit 7 digit signed BCD values by complimenting and using the built in DADD instruction
 * Coded in assembly and linked in later
 * Inputs: *arg1 and *arg2, two signed BCD vales
 * Outputs: *arg2 => the sum of the two values
 * Special Cases: When overflow is encountered, the 30th bit will be set to to a vlaue of one
 */
extern void bcdadd(bcd32_t *arg1, bcd32_t *arg2);

/**
 * Multiplies two 32-bit binary-coded decimal (BCD) values and stores the result in another 32-bit memory location in BCD form. It should cover all possible corner cases, including multiplication by/of negative numbers as well as overflown numbers.
 * Inputs: *arg_a and *arg_b, two signed BCD vales
 * Outputs: *result, a signed BCD values that is the result of the multiplication, or overflow
 * Special Cases: There is a good chance that overflow will occur and the overflow bit on the result will be set
 */
void bcdmult( bcd32_t* arg_a, bcd32_t* arg_b, bcd32_t* result);

/**
 *  implements a digital Finite Impulse Response (FIR) filter on an input stream of data to produce a filtered response. Depending on the coefficients of the filter, this filter may act as a low-pass filter, for example.
 * Inputs:  filterOrder - The order of the filter function
 *          coeffs - The address of the coefficient values of the function
 *          inputSamples - The address of the input signal discreet values
 *          numInputs - The number of values stored in the inputSamples array
 * Outputs: outputValue - The address to store the output values
 * Special Cases: Assumes a numInputs value larger that the filterOrder value
 */
void fir(unsigned int filterOrder, bcd32_t* coeffs, bcd32_t* inputSamples, bcd32_t* outputValue, unsigned int numInputs);
```

bcdadd.s43

```
  PUBLIC _bcdadd
  RSEG CODE

_bcdadd
  ; Push registers onto stack to preserve state!
  PUSH R4
  PUSH R5
  PUSH R6
  PUSH R7
  PUSH R8
  PUSH R9
  PUSH R10
  PUSH R11
  PUSH R12
  PUSH R15

  ; Make space on stack
  CLRC
  SUB.W #8,SP

  MOV #0, R9

  ; Move value from R15 into Stack
  MOV.W 0(R15),0(SP)
  MOV.W 2(R15),2(SP)
  MOV.B 3(R15),R5
  AND.B #0x0F,3(SP)

  ; Move value from R14 into Stack
  MOV.W 0(R14),4(SP)
  MOV.W 2(R14),6(SP)
  MOV.B 3(R14),R4
  AND.B #0x0F,7(SP)

  ; Initialize to 0, tracks if the final result should be complimented
  MOV.W #0x0,R11

  ; If the value R14 is negative and R15 is positive, check if R14's magnitude is greater
  ; If so, then the final result will be negative, so it will need to be complimented
  AND.B #0x80,R4
  JZ notneg_r14
    AND.B #0x80,R5
    JNZ neg_r15
      MOV #1, R9
      ; R14 neg, R15 pos
      CMP.W 2(SP),6(SP); R14-R15
      JL start_add
```

```
      JNE mark_compliment1

      CMP.W 0(SP),4(SP); R14-R15
      JL start_add
mark_compliment1
      MOV.W #0x1,R11
      JMP start_add
neg_r15
notneg_r14

  ; If the value R15 is negative and R14 is positive, check if R15's magnitude is greater
  ; If so, then the final result will be negative, so it will need to be complimented
  AND.B #0x80,R4
  JNZ neg_r14
    AND.B #0x80,R5
    JZ notneg_r15
      MOV #1, R9

      ; R14 pos, R15 pos
      CMP.W 6(SP),2(SP); R15-R14
      JL start_add
      JNE mark_compliment2
        CMP.W 4(SP),0(SP); R15-R14
        JL start_add
mark_compliment2
      MOV.W #0x1,R11
      JMP start_add
notneg_r15
neg_r14

start_add

  ; If the 2 values are the same sign, then there is a possibility of overflow
  ; And skip code for complimenting, since the code will simply add magnitudes instead
  CMP.B #1, R9
  JNZ solve_same_sign_eq

  ; Initialize sign counter to 0
  MOV.W #0,R7

  ; Check sign of R14 and complement the value if needed
  AND.B #0x80,R4
  JZ no_compl1
    ADD.W #0x1,R7
    PUSH R15
    MOV.W SP,R8
    ADD.W #6,R8
    MOV.W R8,R15
```

```
    CALL #_bcdcompliment
    AND.B #0x0F,3(R15)
    POP R15
no_compl1

  ; Check sign of R15 and complement the value if needed
  AND.B #0x80,R5
  JZ no_compl2
    ADD.W #0x1,R7
    PUSH R15
    MOV.W SP,R8
    ADD.W #2,R8
    MOV.W R8,R15
    CALL #_bcdcompliment
    AND.B #0x0F,3(R15)
    POP R15
no_compl2

  JMP add_values

  ; Handles adding 2 values with the same sign
  ; Removes signs and marks for later replacing signs
solve_same_sign_eq
  BIT.B R4,R5
  JGE add_values
    AND.B #0x0F, 3(SP)
    AND.B #0x0F, 7(SP)
    MOV #1,R9 ; Mark so sign will be replaced later


  ; All code converges again here
add_values
  ; Perform the bcd addition
  CLRC
  DADD.W 0(SP),4(SP)
  DADD.W 2(SP),6(SP)

  ; Move final value into @R14 and clear sign/overflow bits
  MOV.W 4(SP),0(R14)
  MOV.W 6(SP),2(R14)
  MOV.B 3(R14),R8
  AND.B #0x0F,3(R14)

  ; Return space from stack
  CLRC
  ADD.W #8,SP

  ; Prepare sign bits for comparison
```

```
  AND.B #0x80,R4
  AND.B #0x80,R5

  CMP.B R4,R5
  JNZ no_invert_sign_after ; If both are same sign
    CMP.B #1,R9
    JNZ both_positive ; If we removed the sign
      ADD.B #0x80,3(R14) ; restore the sign
both_positive
    BIT.B #0x10,R8
    JZ no_invert_sign_after ; If overflow!
      ; Just set the value to 0x40000000
      MOV.W #0x4000,2(R14)
      MOV.W #0x0000,0(R14)

no_invert_sign_after

  ; If the compliment flag was set at the beginning of the function
  ; then compliment the final value
  CMP.W #0x1,R11
  JNZ no_compl3
    PUSH R15
    MOV.W R14,R15
    CALL #_bcdcompliment
    POP R15
no_compl3


  ; Restore the program state to it's previous values
  POP R15
  POP R12
  POP R11
  POP R10
  POP R9
  POP R8
  POP R7
  POP R6
  POP R5
  POP R4
  RET
; Subfunction to handle complimenting the bcd value passed into it
  PUBLIC _bcdcompliment
  RSEG CODE

_bcdcompliment
  PUSH R4 ; Loop index
  PUSH R5 ; Address backup
  PUSH R6 ; Current byte lower
```

```
  PUSH R7 ; Current byte upper shifted
  PUSH R8 ; temp
  PUSH R9 ; temp

  MOV.W #3,R4
  MOV.W R15,R5

  ; Loop over the 4 bytes of the BCD value
start
    MOV.B @R15+,R6

    ; Pull out the upper and lower digit from the current byte into R6 and R7
    MOV.B R6,R7
    RRA.B R7
    RRA.B R7
    RRA.B R7
    RRA.B R7
    AND.B #0x0F,R6
    AND.B #0x0F,R7

    ; Subtract the upper digit from 9 and shift back to correct location at MSB
    MOV.B #9,R8
    SUB.B R7,R8
    ADD.B R8,R8
    ADD.B R8,R8
    ADD.B R8,R8
    ADD.B R8,R8

    ; Subtract the digit lower digit from 9
    MOV.B #9,R9
    SUB.B R6,R9

    ; Add the 2 digits together and push onto the stack for later use
    ADD.B R8,R9
    PUSH R9

    DEC.W R4
  JGE start

  MOV.W R5,R15 ; Restore dest address

  ; Back up the sign bits of the input
  MOV.B 3(R15),R8

  ; Pop digits off of stack and set values in return address
  POP R4
  POP R5
  POP R6
```

```
POP R7
MOV.B R7,0(R15)
MOV.B R6,1(R15)
MOV.B R5,2(R15)
MOV.B R4,3(R15)

; Add one to the complimented values
CLRC
DADD.W #1,0(R15)
DADD.W #0,2(R15)

; Clear sign and overflow bits
AND.B #0x0F,3(R15)

; toggle the sign bit from it's previous state
AND.B #0x80,R8
XOR.B #0x80,R8
CLRC
ADD.B R8,3(R15)

; Restore program context
POP R9
POP R8
POP R7
POP R6
POP R5
POP R4

RET

END
```