Lab #3 – Signed arithmetic

Goals:

Practice 2's complement and signed binary arithmetic

Write a simple program on the Discovery Board using ADDS and SUBS instructions to explore 2's complementing

Reference: 2^0=1 2^1=2 2^2=4 2^3=8 2^4=16 2^5=32 2^6=64 2^7=128 2^8=256 2^9=512 2^{10}=1024 2^{11}=2048 2^{12}=4096 2^{13}=8192 2^{14}=16384 2^{15}=32768 2^{16}=65536 …

For each step enter the necessary responses into the Canvas assignment text entry tool to let the TA follow your work.

Part 1: Refer to the Week 3 lecture slides for the following:

a) What are the two steps for finding the 2's complement of any binary number?

b) What do you expect from the subtraction operation 42 – 42? Using 8-bit binary arithmetic, simulate the subtraction of one register holding 42 and another also holding 42 by ***adding*** 42 + the 2's complement of 42. (The corresponding binary numbers are already in the slides!) What result would be in an 8 bit register? What is the carry bit out of the most significant bit?

c) On lecture slide 19, an example addition is shown that just causes a signed integer overflow in the positive direction by adding 86+42. Repeat this example in 8-bit binary with the addition of 85+42, which should result in the largest positive number representable in an 8-bit signed integer. What is the sign bit?

d) Repeat the idea of c) with the example shown on slide 20. That is, follow that example and find the subtraction of (-86) – (42). This should result in the largest (absolute value) negative number in an 8-bit signed integer. What is the sign bit?

Part 2:

a) Go to https://smumustangs.com/sports/football/stats/2021 and look up the total number of offensive plays run by both the Mustangs and by their opponents last season.

b) Convert each decimal number to binary, using 12 bit fields.

c) Find the 2's complement of each binary number.

d) Find how many plays the Mustangs ran **more than** their opponents by **summing** the appropriate two binary numbers from b) or c) using 12-bit arithmetic.

e) Convert the difference back into decimal.

f) Compare your result from e) with the difference of the two decimal numbers in a).

g) Look up the statistics for Rushing yards gained for the Mustangs and their opponents. Can either or both of these numbers be represented as a 12-bit signed integer? Why or why not? (See the reference line of powers of 2 at the top of this page.)

Part 3:

a) Download the template main C program Lab03-DiffMain.c from the Canvas files area, and in the LabCode folder:

```
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include <stdarg.h>
#include <string.h>
#include <math.h>
#include "library.h"

extern int32_t diff(int32_t,int32_t);

int main(void) {
  int32_t a,b,r;

  InitializeHardware(HEADER,"Subtract");
  a = ;  // Complete this statement
  b = ;  // Complete this statement
  printf("difference (hex) =\n  0x%lX\n",a - b);
  printf("difference (signed decimal) =\n  %ld\n",a - b);
  r = diff(a,b);
  printf("diff() return value (hex) =\n  0x%lX\n",r);
  printf("diff() return value (signed dec) =\n  %ld\n",r);
}
```

Note that the function return value and all the variables are now declared as 32-bit signed integers of the ‘int32_t’ type (as opposed to the 32-bit unsigned type ‘uint32_t’ in Lab 2 on addition). This will allow the values to be printed as signed values by the calls to printf().

b) Download and edit the general framework of the assembly code in the file Lab03-diff.s available in the Canvas files section:

```
          .syntax   unified
          .cpu      cortex-m4
          .text

          .global        diff
          .thumb_func
          .align
diff:                          // Function entry point
          svc            #0             // Debug breakpoint
          mov                           // Move constant 0 to R2
          svc            #0             // Debug breakpoint
          subs                          // Subtract R1 from R2
```

```
svc             #0          // Debug breakpoint
adds                        // Add R2 to R0 and store in R3
svc             #0          // Debug breakpoint
subs                        // Subtract R1 from R0
svc             #0          // Debug breakpoint
bx   lr                     // Return to calling program

.end
```

Locate label statement diff: , the entry point for the functions coded in assembly language, and the 'bx lr' instruction that returns execution from the function back to the calling program. You will be completing the assembly instructions between the entry points and the returns for two functions. Note that there are already debug breakpoint 'svc #0' instructions before and after every active instruction. Of course, the C main program will load the values of the argument variables a and b in registers R0 and R1, respectively, before this function is called.

c) Complete the assembly code in the function 'diff' to:

Move a constant 0 to register R2
Subtract register R1 (the second function argument) from R2 and leave the result in R2
Add R2 to R0 and store the result in R3
Subtract R1 from R0 (the first function argument) and leave the result in R0 as the function return value

Let's work through these instructions to see what we expect to happen in this function. What is in register R2 before the first 'subs' instruction executes? After the first 'subs' instruction runs, there should be a 2's complement value in R2, the 2's complement of what value? Therefore what should be the value in R3 after the 'adds' instruction executes? After the second 'subs' instruction runs, how do you expect R0 and R3 to compare? What do you expect the function return value to compute? (Hint: Look at what the C main program computes as a preview value before it calls the diff() function!) **Upload your two completed source code files Lab03-DiffMain.c and Lab03-diff.s into the file upload entry in Canvas.**

d) Now complete the two assignment statements in the C main program, where values are assigned to the variables a and b, to replicate the example shown in the lecture slides on page 13, that is, the subtraction of 17-42. Build the program and run the program on your board, stepping through the instructions. Do the register values agree with your expectations of part c)? Remember, the example done in class on page 13 was done with 8-bit binary values for simplicity. Are the 32-bit hex values you see in the registers consistent with the class example? What condition flags are set in the xPSR register after the second 'subs' instruction? (Look at the first hex digit showing in the xPSR register value in the debug window.) What is the function return value?

e) Now let's see if we can intentionally cause a signed arithmetic overflow. Change the two assignment statements in the C main program to assign 2147483000 to 'a' and -648 to 'b'. What decimal number should result from subtracting a – b, if we had an unlimited bit width to perform our arithmetic? Compare this number to the maximum allowable signed positive number in 32-bit arithmetic. Rebuild and rerun your program. As you step through the instructions, now what condition flags are set in the xPSR register after the second 'subs' instruction? What is the function return value?