**Project Progress Final Report. Gabriel Facco Bettinelli. 824113076**

**Main code in C >> Calculator Program will prompt user using printf to input 2 numbers to make calculations and select operation desired(1 through 7). Program will store input arguments for operands calculations. Switch stament will lead the program using the selection that user made. From there, each operation will branch to make the operation in ARM Assembly, making them an extern function in C.**

**Program execution: Main function will be wrote in C, even though I have a running version of the main in Assembly, the user interface is way better using C. All functions used in the Main Calculator function were wrote in Assembly, and then called into the C code using the extern declaration. Inputs passed into ARM Assembly using the r0 register and outputs using the r0 too.The pseudo-code was created using C language and then worked on to translate it to Assembly. I also created a C function to print the binary value of the number, so we can display it on the answer to the user.**

**In the first functions( ADD, Mul, Div) were used in C 32 bites data types, unsigned long int, to represent the 32-bits. For the Mean Function, a float return value was used because output is normally not a whole number. For Odd Or Even and Reverse bits were used as well a unsigned long int data type to the return value. In some of the assembly functions I tried to use the LDM and STM instructions to implement pop and push operations just for trying something new, but im not sure if it’s more efficiently. At some of the assembly functions, I tried to implement them using the Frame Pointer too.**

**Specific tool set used: Segger Embedded Studio for ARM V4.22**

**User Guide: Program starts asking the user which operation he will want to execute, the Assembly Calculator will have the following options:**

**i.1. Add two unsigned 32 bit number**

**ii.2. Multiply two unsigned 32 bit number**

**iii.Divide two unsigned 32 bit number**

**iv.Mean of a set of numbers**

**v.Odd Or even Number**

**vi.Reverse the bits**

**vii. OR,**

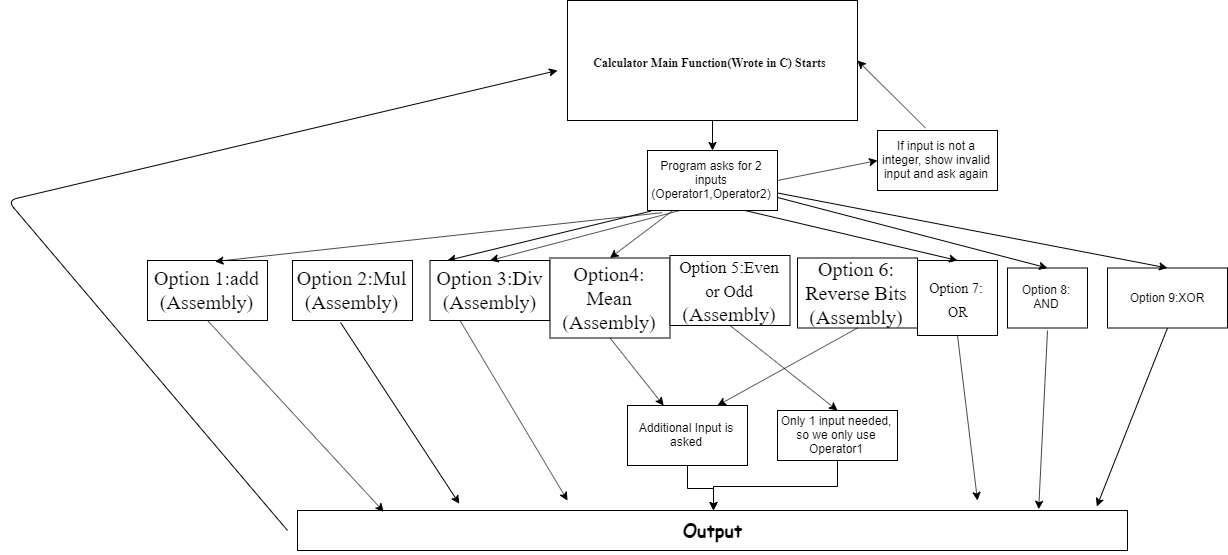
**viii. AND**

**iv. XOR**

**After the user inputs the option, it will ask for 2 inputs regardless of the choice, so if you are using option 4 or 5 you can just put a 0 or any number as input2 that will not change.**

**For options 4,5,6 first 2 inputs will be ignored and new input will be asked.**

**Input is checked against bad input, if users input ‘a’ it will go back and ask input again.**

****

**unsigned long int add(unsigned long int a, unsigned long int b)**

**{**

**return a+b; // For Assembly: Create space memory on stack, add variables to a holder**

**And return value to r0.**

**}**

**unsigned int multiply(unsigned long int a, unsigned long int b)**

**{**

**return a\*b; //For Arm: Store fp and make space for variables on stack.**

**// Make Mul operation on registers and return value to r0.**

**}**

**unsigned int divide(unsigned long int a, unsigned long int b)**

**{**

**return a/b;**

**}**

**float mean(int\* set, int n)**

**{**

**signed int sum = 0, i;**

**for(i = 0; i < n; i++)**

**sum += set[i];**

**return sum/n;**

**}**

**int odd\_or\_even(unsigned long int a)**

**{**

**return a % 2;**

**}**

**signed long int reverse\_bits(unsigned long int a)**

**{**

**unsigned int r = 0, i, temp;**

**for (i = 0; i < sizeof(a) \* 8; i++){**

**temp = (a & (1 << i));**

**if(temp)**

**r |= (1 << ((sizeof(a) \* 8 - 1) - i));**

**}**

**return r;**

**}**

**Main C Function and Code:**

**#include <stdint.h>**

**#include <stdio.h>**

**#include <string.h>**

**#include <stdlib.h>**

**#include <limits.h> // For binary print**

**//extern function declarations(ARM Assembly Code)**

**extern signed long int AddinAssembly(signed long int a, signed long int b);**

**extern long int MultInAssembly(signed long int a, signed long int b);**

**extern long int DivInAssembly(signed long int a, signed long int b);**

**extern float MeanInAssembly (signed long int\* set, unsigned long int n);**

**extern long int Odd\_Or\_EvenInAssembly(unsigned long int a);**

**extern long int Reverse\_BitsInAssembly(int a);**

**extern char inputIsANumber(char \*a, int numberHolder);**

**extern long int OrInAssembly(signed long int a, signed long int b);**

**extern long int AndInAssembly(signed long int a, signed long int b);**

**extern long int XORInAssembly(signed long int a, signed long int b);**

**int option;**

**/**

**long int operator1;**

**long int operator2;**

**long int result;**

**long int inputIs; //If input inputIs = 0, means the user inputed a letter. inputIs = 1, user inputed a number.**

**int arraySize; // For Mean calculation;**

**signed long int\* setInputed; // Array to be filled for case n#4**

**int case4holder;**

**int test1;**

**test1 = 10;**

**int switcher1 = 0; // for do loop control**

**int revHolder; // for reverse bits input**

**void main(){**

**//do //Do loop for the calculator restarts**

**// {**

**printf("Welcome to the Calculator, select one of the options:\n");**

**printf(".1. Add two unsigned 32 bit number\n");**

**printf(" 2. Multiply two unsigned 32 bit number \n");**

**printf(" 3.Divide two unsigned 32 bit number \n");**

**printf(" 4.Mean of a set of numbers \n");**

**printf(" 5.Odd Or even Number \n");**

**printf(" 6.Reverse the bits \n");**

**printf(" 7.OR Operation \n");**

**printf(" 8.AND Operation \n");**

**printf(" 9.Xor Operation \n");**

**do // do loop for controling the right input, so until user inputs correctly, the program will re-ask for input**

**{**

**printf("Operation desired: ");**

**scanf("%d", &option);**

**printf("Select Number Inputs:Input1: "); //%lu is the correct format for unsigned long. %li is correct formar for scanf for signed long int**

**scanf ("%x", &operator1);**

**printf("\n Select Input2: ");**

**scanf("%x", &operator2);**

**printf("Option choosed: %d \n", option);**

**switch (option)**

**{**

**case 1: printf("Choice is: Add two numbers\n");**

**result = AddinAssembly(operator1,operator2);**

**printf("Result is: %x\n", result);**

**switcher1 == 1; // break from loop**

**break;**

**case 2: printf("Choice is: Multiply two numbers\n");**

**result = MultInAssembly(operator1,operator2);**

**printf("Result is: %x\n", result);**

**switcher1 == 1; // break from loop**

**break;**

**case 3: printf("Choice is Divide two numbers");**

**result = DivInAssembly(operator1,operator2);**

**printf("Result is: %x\n", result);**

**switcher1 == 1; // break from loop**

**break;**

**case 4: printf("Mean of a set of numbers");**

**printf(" Select Size of Array of number: ");**

**scanf("%d", &arraySize);**

**for(int i =0; i<arraySize;i++)**

**{**

**printf("Please select %d array item", i);**

**scanf("%d", &case4holder);**

**setInputed[i] = case4holder;**

**}**

**result = MeanInAssembly(setInputed,arraySize);**

**printf("Mean of the set is: %x\n", result);**

**switcher1 == 1; // break from loop**

**break;**

**case 5: printf("Odd or Even Number\n");**

**result = Odd\_Or\_EvenInAssembly(operator1);**

**if(result == 1)**

**printf("The result is: %x is odd\n", operator1);**

**else**

**printf("The result is: %x is even\n",operator1);**

**switcher1 == 1; // break from loop**

**break;**

**case 6: printf("Reverse the bits\n");**

**printf("Input Number to reverse bits: ");**

**scanf("%d", &revHolder);**

**result = Reverse\_BitsInAssembly(revHolder);**

**printf("Number to be translated: %d, in binary:",revHolder);**

**print\_bin(revHolder);**

**printf("\nReversing the bits of input Number we get: ");**

**print\_bin(result);**

**switcher1 == 1; // break from loop**

**break;**

**case 7: printf("Or Operation");**

**result = OrInAssembly(operator1,operator2);**

**printf("A | B = %x\n", result);**

**switcher1 == 1; // break from loop**

**break;**

**case 8: printf("AND Operation");**

**result = AndInAssembly(operator1,operator2);**

**printf("A & B = %x\n", result);**

**printf("Result in binary:");**

**print\_bin(result);**

**printf("\n");**

**switcher1 == 1; // break from loop**

**break;**

**case 9: printf("XOR Operation");**

**result = XORInAssembly(operator1,operator2);**

**printf("A ^ B = %x\n", result);**

**switcher1 == 1; // break from loop**

**break;**

**default: printf("Not a valid option selected");**

**}**

**} while( switcher1 == 0);**

**}**

**//function to print binary value of int.**

**void print\_bin(unsigned char byte)**

**{**

**int i = CHAR\_BIT; /\* however many bits are in a byte on your platform \*/**

**while(i--)**

**{**

**putchar('0' + ((byte >> i) & 1)); /\* loop through and print the bits \*/**

**}**

**}**

**ARM Code:**

**//Gabriel Facco Bettinelli. RedId: 824113076**

**//Final Project ARM Assembly code for Comp-E 271: Prof. Kenneth Arnold (12/1/2019)**

**//frame pointer serves the base of stack frame while stack pointer refers to top location of the stack.**

**.syntax unified**

**.global AddinAssembly**

**.data**

**// declare any global variables here**

**.text**

**//each register is 32 bits.**

**// 32 bits = 4 bytes = 0xFFDF .i.e.**

**AddinAssembly:**

**mov r12,r13 // save stack pointer into register r12**

**sub sp,#32 // reserve 32 bytes of space for local variables**

**push {lr} // push link register onto stack -- make sure you pop it out before you return**

**str fp, [sp, #-4]! // store frame pointer in stack -4**

**add fp, sp, #0 // add stack to frame pointer**

**sub sp, sp, #12 // make space on stack for 3 registers**

**str r0, [fp, #-8] // store r0 into frame -8**

**str r1, [fp, #-12] // store r1 into fp -12**

**ldr r2, [fp, #-8] // load fp-8 to r2**

**ldr r3, [fp, #-12] // load fp -12 to r3**

**add r3, r2, r3 //Adding the parameters**

**mov r0, r3 //Assigning return value to r0**

**add sp, fp, #0 //deallocate stack space**

**ldmfd sp!, {fp}**

**pop {lr} // pop link register from stack**

**mov sp,r12 // restore the stack pointer -- Please note stack pointer should be equal to the**

**// value it had when you entered the function .**

**bx lr // return from the function by copying link register into program counter**

**.global MultInAssembly**

**.data**

**// declare any global variables here**

**.text**

**MultInAssembly:**

**mov r12,r13 // save stack pointer into register r12**

**sub sp,#32 // reserve 32 bytes of space for local variables**

**push {lr} // push link register onto stack -- make sure you pop it out before you return**

**str fp, [sp, #-4]! // store frame pointer into stack -4**

**add fp, sp, #0 // add stack to fp**

**sub sp, sp, #12 // make space for 3 registers in the stack**

**str r0, [fp, #-8] // store r0 into loc -8**

**str r1, [fp, #-12] // store r1 into loc -12**

**ldr r3, [fp, #-8] // load loc -8 to r3**

**ldr r2, [fp, #-12] // load loc -12 to r2**

**mul r3, r2, r3 // multiply arguments**

**mov r0, r3 // get result of calculation into return register r0**

**add sp, fp, #0 //deallocate stack space**

**ldmfd sp!, {fp}**

**pop {lr} // pop link register from stack**

**mov sp,r12 // restore the stack pointer -- Please note stack pointer should be equal to the**

**// value it had when you entered the function .**

**bx lr // return from the function by copying link register into program counter**

**//function to verify that user inputs a number, not a letter**

**.global inputIsANumber**

**.data**

**// declare any global variables here**

**.text**

**inputIsANumber:**

**mov r12,r13 // save stack pointer into register r12**

**sub sp,#32 // reserve 32 bytes of space for local variables**

**push {lr} // push link register onto stack -- make sure you pop it out before you return**

**// Your solution here**

**// r0= argument char \*c and return value**

**// r2= 48, r3= 57**

**mov r2, #48 //make r2=48 and r3= 57, comparison values.**

**mov r3, #57**

**cmp r0,r3 // if every input data is between 48 and 57, then is a number.**

**bge isNotNumber //if r1>r3(57( go to isNotNumber**

**cmp r0,r2**

**ble isNotNumber //if r1<48 go to isNotNumber**

**b isNumber //if passed both r1>57 and r1< 48, it is a number.**

**isNumber:**

**mov r0, #1**

**pop {lr}**

**mov pc, lr //return 1 if argument is a number.**

**isNotNumber:**

**mov r0, #0**

**pop {lr}**

**mov pc,lr //return 0 if argument is not a number.**

**pop {lr} // pop link register from stack**

**mov sp,r12 // restore the stack pointer -- Please note stack pointer should be equal to the**

**// value it had when you entered the function .**

**mov pc,lr // return from the function by copying link register into program counter**

**.global DivInAssembly**

**.data**

**// declare any global variables here**

**.text**

**DivInAssembly:**

**mov r12,r13 // save stack pointer into register r12**

**sub sp,#32 // reserve 32 bytes of space for local variables**

**push {lr} // push link register onto stack -- make sure you pop it out before you return**

**stmfd sp!, {fp, lr} // here we are using a stack type full descending (using load and store multiple instructions to implement push and pop operations to a registers in a single instruction)**

**add fp, sp, #4 // add frame pointer to loc +4**

**sub sp, sp, #8 // make space on stack for 2 variables**

**str r0, [fp, #-8] // store r0 into loc -8**

**str r1, [fp, #-12] // store r1 into loc -12**

**ldr r0, [fp, #-8] // load loc -8 into r0**

**ldr r1, [fp, #-12] // load loc -12 to r1**

**bl \_\_aeabi\_uidiv // to proceed with division >> \_\_aeabi\_idiv0 raises SIGFPE with an additional argument, DIVBYZERO.**

**mov r3, r0 // get result from r0 and store into r3**

**mov r0, r3 //**

**sub sp, fp, #4**

**ldmfd sp!, {fp, pc} // finishes first instruction**

**pop {lr} // pop link register from stack**

**mov sp,r12 // restore the stack pointer -- Please note stack pointer should be equal to the**

**// value it had when you entered the function .**

**bx lr // return from the function by copying link register into program counter**

**.global MeanInAssembly**

**.data**

**// declare any global variables here**

**.text**

**MeanInAssembly:**

**mov r12,r13 // save stack pointer into register r12**

**sub sp,#32 // reserve 32 bytes of space for local variables**

**push {lr} // push link register onto stack -- make sure you pop it out before you return**

**stmfd sp!, {fp, lr} // Stmfd call to push space in the stack for one more variable**

**add fp, sp, #4 // make frame pointer = stack + 4**

**sub sp, sp, #16 // make space for 4 variables on the stack**

**str r0, [fp, #-16] // store r0 into loc -16**

**str r1, [fp, #-20] // store r1 into loc -20**

**mov r3, #0 // make r3 = 0**

**str r3, [fp, #-8] // store r3 into loc -8**

**mov r3, #0 // make r3 = 0**

**str r3, [fp, #-12] // store r3 into loc -12**

**b .L5 // branch to '.L5' branch. It's called as a subroutine, but doesn't return control (unless it's cut off in your snippet and actually continues below).**

**.L6:**

**ldr r3, [fp, #-12] // load loc -12 to r3**

**mov r3, r3, asl #2 // make r3 = r3 Arith shift left, #2, so we get next value**

**ldr r2, [fp, #-16] // load value of loc -16 to r2**

**add r3, r2, r3 // make r3 = r2+ next value**

**ldr r3, [r3, #0] //**

**ldr r2, [fp, #-8] // load value from loc -8 to r2**

**add r3, r2, r3 // r3 = r3+r2**

**str r3, [fp, #-8] // store r4 into loc -8**

**ldr r3, [fp, #-12] // load value from loc -12 to r3**

**add r3, r3, #1 // make r3= r3 +1( get next value)**

**str r3, [fp, #-12] // store r3 into stack loc -12**

**.L5:**

**ldr r2, [fp, #-20] // load value from loc -20 to r2**

**ldr r3, [fp, #-12] // load value value form loc -12 to r3**

**cmp r2, r3 // compare r2 to r3**

**bhi .L6 // if r2 bigger than r3(unsigned) branch to .L6**

**ldr r3, [fp, #-20] // load value from stack loc -20 to r3**

**ldr r0, [fp, #-8] // load value from loc -8 to r0**

**mov r1, r3 // r1= value**

**bl \_\_aeabi\_uidiv // branch to Division function**

**mov r3, r0 // make r3 = r0**

**mov r0, r3 //**

**bl \_\_aeabi\_ui2f**

**mov r3, r0**

**mov r0, r3**

**sub sp, fp, #4**

**ldmfd sp!, {fp, pc} // ldmfd to pop space from the stack**

**pop {lr} // pop link register from stack**

**mov sp,r12 // restore the stack pointer -- Please note stack pointer should be equal to the**

**// value it had when you entered the function .**

**bx lr // return from the function by copying link register into program counter**

**.global Odd\_Or\_EvenInAssembly**

**.data**

**// declare any global variables here**

**.text**

**Odd\_Or\_EvenInAssembly:**

**mov r12,r13 // save stack pointer into register r12**

**sub sp,#32 // reserve 32 bytes of space for local variables**

**push {lr} // push link register onto stack -- make sure you pop it out before you return**

**str fp, [sp, #-4]! // store frame pointer into stack loc -4**

**add fp, sp, #0 //**

**sub sp, sp, #12 // make space on stack for 3 variables**

**str r0, [fp, #-8] // store value of r0 into loc -8**

**ldr r3, [fp, #-8] // load value from loc -8 to r3**

**and r3, r3, #1 // and r3 with 1, so we can know if its odd or even, returns 1 if argument is even**

**mov r0, r3 // make return value = r3**

**add sp, fp, #0 // deallocate stack register memory**

**ldmfd sp!, {fp} //**

**pop {lr} // pop link register from stack**

**mov sp,r12 // restore the stack pointer -- Please note stack pointer should be equal to the**

**// value it had when you entered the function .**

**bx lr // return from the function by copying link register into program counter**

**.global Reverse\_BitsInAssembly**

**.data**

**// declare any global variables here**

**.text**

**Reverse\_BitsInAssembly:**

**mov r12,r13 // save stack pointer into register r12**

**sub sp,#32 // reserve 32 bytes of space for local variables**

**push {lr} // push link register onto stack -- make sure you pop it out before you return**

**//mvn Rd, Rm**

**//MVN{S}{cond} Rd, Operand2**

**mvn r1,r0 //The MVN instruction takes the value in Rm, performs a bitwise logical NOT operation on the value, and places the result in Rd.**

**mov r0,r1 // get return value**

**pop {lr} // pop link register from stack**

**mov sp,r12 // restore the stack pointer -- Please note stack pointer should be equal to the**

**// value it had when you entered the function .**

**bx lr // return from the function by copying link register into program counter**

**.global OrInAssembly**

**.data**

**// declare any global variables here**

**.text**

**OrInAssembly:**

**//r0 = input1 and return value, r1= input2,r2 holder**

**sub sp,sp,#12 // make space on stack for 3 registers**

**str r0, [sp, #-8] // save r0 on stack**

**str r1, [sp, #-4] // save r1 on stack**

**str r2, [sp] // save r2 on stack**

**orr r0,r1,r0 // Or r0(input1) with r1(input2) and store result in r0.**

**ldr r2, [sp] // restore r2 value**

**ldr r1, [sp, #-4]**

**ldr r0, [sp,#-8]**

**add sp, sp, #12 // deallocate stack space**

**mov pc, lr // return function to caller**

**.global AndInAssembly**

**.data**

**// declare any global variables here**

**.text**

**AndInAssembly:**

**//r0 = input1 and return value, r1= input2,r2 holder**

**ldr r3,= 0x00000000**

**and r3,r0,r1 // AND r0(input1) with r1(input2) and store result in r3.**

**mov r0,r3 // get return value**

**mov pc, lr**

**// add sp, sp, #12 // deallocate stack space**

**// mov pc, lr // return function to caller**

**.global XORInAssembly**

**.data**

**// declare any global variables here**

**.text**

**XORInAssembly:**

**//r0 = input1 and return value, r1= input2,r2 holder**

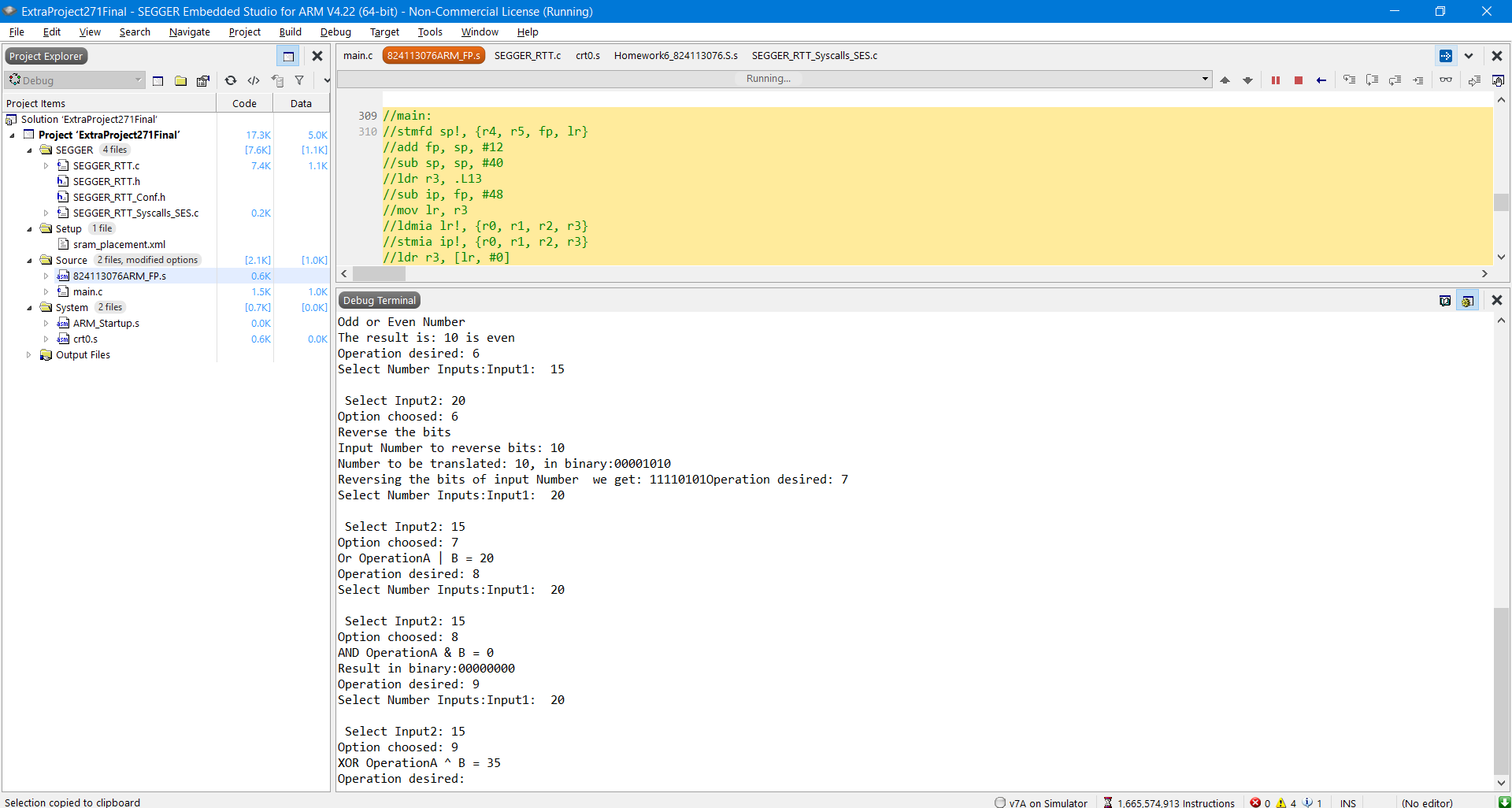
**ldr r3,= 0x00000000**

**EOR r3,r0,r1 // XOR r0(input1) with r1(input2) and store result in r0.**

**mov r0,r3**

**mov pc, lr // return function to caller**

**Video show included in blackboard submission**

****

**13. Spent +- 13-15 hours in the project.**

**Conclusion: It was really hard and challenging some of the parts, going back and forth between C and Assembly and trying them to work together, had to learn to deal with a lot of different kind’s of data. Tried inputing all different kinds of input to see how program would react, and was safe against 80% of the attempts to crash, using letters or symbols.**

References:

<http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0068b/Cacbgchh.html>

<http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.kui0097a/armcc_cjaddjea.htm>

<http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.kui0097a/armcc_cjaddjea.htm>

<http://www.keil.com/support/man/docs/armasm/armasm_dom1359731152499.htm>

<http://mrjester.hapisan.com/04_MC68/Sect04Part06/Index.html>

<http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0068b/BABJAJIB.html>