CMPE-250 Assembly and Embedded Programming

Laboratory Exercise 4

Iteration and Subroutines

By submitting this report, I attest that its contents are wholly my individual writing about this exercise and that they reflect the submitted code. I further acknowledge that permitted collaboration for this exercise consists only of discussions of concepts with course staff and fellow students. Other than code provided by the instructor for this exercise, all code was developed by me.

Andrei Tumbar
Submitted: 09-22-20

Lab Section:	5
Instructor:	Gordon Werner
TA:	Tianran Cui
	Anthony Bacchetta
Lecture Section:	1
Lecture Instructor:	Melton

Abstract

This laboratory exercise investigated subroutines and more in-depth usage of branching and iteration. The exercise involved development of an unsigned integer algorithm. The goal was to compute a quotient and remainder for an arbitrary dividend and divisor. In the case that division by zero is attempted, the operands should remain unchanged and the C flag should be set. Using an external library, inputs and outputs of the division subroutine are tested. Using this library, results were results successful as the testing subroutines found no errors.

Procedure

A subroutine (`DIVU`) to perform unsigned integer division was written. Input/Output of `DIVU` are as follows:

$$R1 \div R0 = R0 \text{ remainder } R1 \tag{1}$$

Equaton 1 shows that the input dividend and divisors are R1 and R0 respectively. The outputs are stored back into R0 and R1 where R0 will be the quotient and R1 will be the remainder. The `DIVU` subroutine was tested using an external library that had subroutines to load and test input data.

The division algorithm has mulitple valid implementations. The simplest algorithm is to continously subtract the divisor from the dividend and increment the quotient until the dividend is less than the divisor. The result in the dividend is known as the remainder. The issue with this algorithm is that as the dividend gets larger and the divisor smaller, the subroutine will need to iterate a greater number of times. A better "fast" division algorithm was developed in C so that the magnitude of the inputs would not raise the maximum number of iterators needed for division.

```c
#define LEFT_MASK 0x80000000

void test_div(int N, int D, int* Q, int* R) {
    *R = 0;
    *Q = 0;

    for (int i = 31; i >= 0; i--) {
        *R = *R << 1;
        *R |= (N & LEFT_MASK) >> 31;
        N = N << 1;
        if (*R >= D) {
            *R = *R - D;
            *Q |= 1 << i;
        }
    }
}
```

The above function will compute `N / D` and `N % D` and store the results in the memory at `Q` and `R` respectively. The algorithm is based on long division and essentially subtracts the largest possible multiple of each digit in the divisor as it moves from left to right. After verifying the validity of that this algorithm by comparing the results of the function to simple division expanded by the compiler, the algorithm was developed in an assembly routine.

The assembly subroutine was tested by linking an external library with `InitData`, `LoadData`, and `TestData` subroutines defined. The `TestData` subroutine verified the results of the `DIVU` subroutine and incremented R6 every time it found an error. This means that after the program finishes executing, valid results are indicated by a zeroed R6 register.

Results

A screen capture was taken of the register values after program execution:

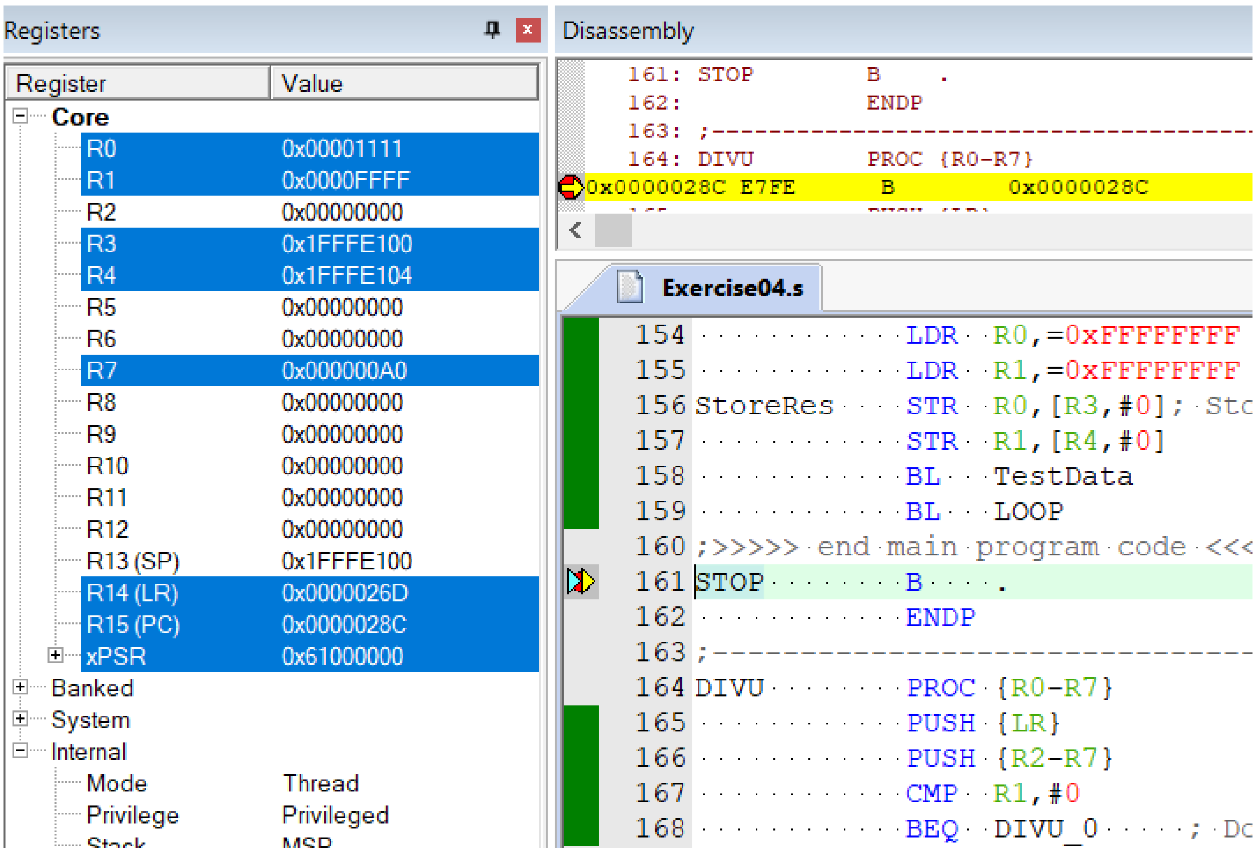

Figure 1: Debugger results after program execution.

Figure 1 shows the register values after program execution. The important register to note is R6 which is used by `TestData` as the an error output. This capture shows that a valid `DIVU` was written as the value of R6 indicates to errors occured.

Another screen capture was taken of the memory map after compiling the assembly code to determine the memory regions of the generated machine-code.

```
Execution Region ER_RO (Exec base: 0x00000000, Load base: 0x00000000, Size: 0x0000032c, Max: 0xffffffff, ABSOLUTE)

Exec Addr    Load Addr    Size         Type   Attr      Idx    E Section Name        Object

0x00000000   0x00000000   0x000000c0   Data   RO            2    RESET               exercise04.o
0x000000c0   0x000000c0   0x000001a4   Code   RO           11    Exercise04_Lib      Exercise04_Lib.lib(exercise04_lib.o)
0x00000264   0x00000264   0x000000c8   Code   RO            1  * MyCode              exercise04.o

Execution Region ER_RW (Exec base: 0x1fffe000, Load base: 0x0000032c, Size: 0x000001d0, Max: 0xffffffff, ABSOLUTE)

Exec Addr    Load Addr    Size         Type   Attr      Idx    E Section Name        Object

0x1fffe000   0x0000032c   0x00000100   Data   RW            4    .ARM.__at_0x1FFFE000  exercise04.o
0x1fffe100   0x0000042c   0x000000d0   Data   RW            5    MyData              exercise04.o
```

Figure 2: Memory map of the assembly program.

The code written for the main program is labelled `MyCode` starting at `0x264`. According to the map file this section has a size of `0xc8` or 200 bytes. This however includes the instruction inside the `DIVU` subroutine. Looking at the listing file shows that the final instruction in the main part of the program:

```
162  0000002A                    ENDP
```

This line tells us that the offset of the final instruction in the main of the program is `0x2A` meaning that the main program has a size of 42 bytes. The ending address address of `MyCode` is `0x000002A6`.

The `Exercise04_Lib` which includes the `InitData`, `LoadData` and `TestData` subroutines starts at `0xC0` and ends `0x264` with a size of 420 bytes.

Looking at the assembly source code, the section directly before the `MyData` section labelled `.ARM.__at_0x1FFFE000` holds the stack data. The symbol `__initial_sp` is defined at the end of this section because the stack grows upward toward the beginning of the memory area. The stack's size is defined by an `EQU` to be 0x100 or 256 bytes. The stack area starts at `0x1FFFE000` and ends at `0x1FFFE100`.

The section after the stack area (`MyData` starts where the stack ends at `0x1FFFE100`. This section holds the variables P, Q, and the $Results$ array. P and Q are both word variables. The $Results$ array contains $2 \cdot MaxData$ word variables where $MaxData$ is defined at 25. Therefore the size of `MyData` is 208 bytes or `0xD0`. This section ends at the `0x1FFFE1D0` address.

Conclusion

This lab exercise successfully introduced students to the use of subroutines and linking external libraries. In addition to this it taught students how to implement binary unsigned integer division. This lab exercise was successful as the subroutine to verify the results of the division subroute did not indicate any errors were found.