CMPE-250 Assembly and Embedded Programming

Laboratory Exercise 5

Secure String I/O and Number Output

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: 09-29-20

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

Lecture Section: 1
Lecture Instructor: Melton

Results

A screen capture of the memory contents after program execution was taken.

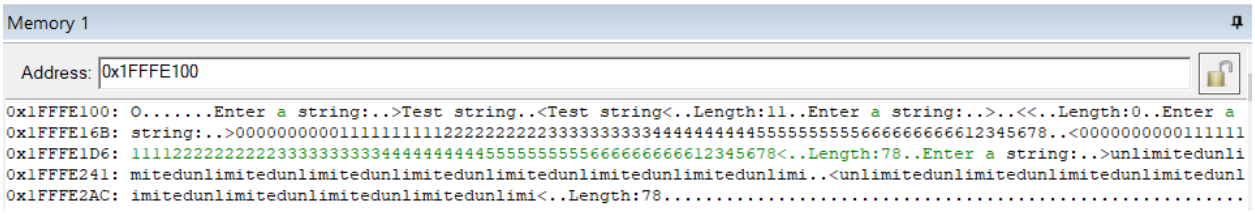

Figure 1: Memory after program execution

Figure 1 shows the memory contents of the program after execution. Because `Exercise05_Lib.lib` attached a large portion of RAM to `0x1FFFE100`, the command-line output was stored in memory at the offset shown.

A capture of the register values was taken at the end of execution.

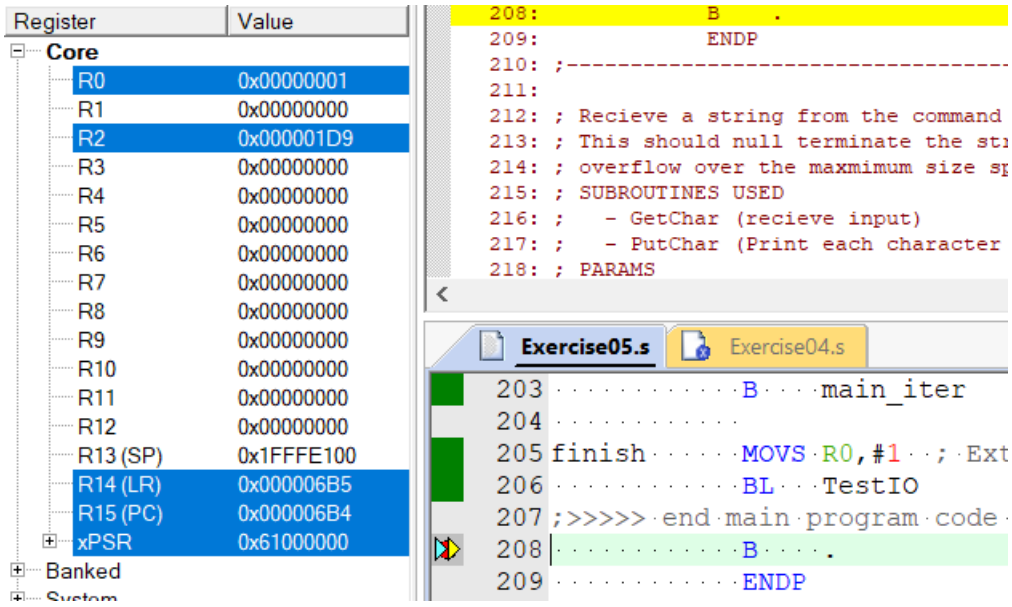

Figure 2: Register values after program executation.

Figure 2 shows the register values after the code was executed. The registers to note are `R1` and `R2`. `R1`'s value of `0x0` indicate that the code is testing data using the extra-credit version of `PutNumU`. `R2`'s value of `0x1D9` indicates that the program has valid output given the extra-credit `PutNumU` subroutine was used.

A screen capture of the listing as well as part of the memory map was taken.

```
207 00000064                ;>>>>> end main program code <<<<<
208 00000064 E7FE           B               .
209 00000066                ENDP
```

Figure 3: Code offset of main program end.

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

```
  Load Region LR_1 (Base: 0x00000000, Size: 0x00000d58, Max: 0xffffffff, ABSOLUTE)

    Execution Region ER_RO (Exec base: 0x00000000, Load base: 0x00000000, Size: 0x0000080c, Max: 0xffffffff, ABSOLUTE)

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

    0x00000000   0x00000000   0x000000c0   Data   RO            2    RESET               exercise05.o
    0x000000c0   0x000000c0   0x00000590   Code   RO           11    Exercise05_Lib      Exercise05_Lib.lib(exercise05_lib.o)
    0x00000650   0x00000650   0x0000019c   Code   RO            1  * MyCode              exercise05.o
    0x000007ec   0x000007ec   0x00000020   Data   RO            3    MyConst             exercise05.o


    Execution Region ER_RW (Exec base: 0x1fffe000, Load base: 0x0000080c, Size: 0x0000054c, Max: 0xffffffff, ABSOLUTE)

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

    0x1fffe000   0x0000080c   0x00000100   Data   RW            4    .ARM.__at_0x1FFFE000  exercise05.o
    0x1fffe100   0x0000090c   0x000003fc   Data   RW           12    Exercise05_Lib_RAM  Exercise05_Lib.lib(exercise05_lib.o)
    0x1fffe4fc   0x00000d08   0x0000004f   Data   RW            5    MyData              exercise05.o
```

Figure 4: Memory map of the assembly program.

Looking at Figure 3 and Figure 4 the main program starts at address `0x650` and ends `0x66` bytes later at `0x6b6` because the last instruction in of the main program has an offset of `0x66`.

According to Figure 4 the `MyCode AREA` starts at `0x650` and ends at `0x7ec`. Constants starts immediately after at `0x7ec` and ends 32 bytes after at `0x80c`.

The variables in this program don't start at the usual `0x1fffe100` because the `Exercise05_Lib.lib` attached its own variables at this location. The memory defined by `Exercise05_Lib.lib` includes a large buffer used for string output to create a virtual terminal output. Instead the variables defined in `Exercise05.s` start at `0x1ffe4fc` and end at `0x1ffe54b`.

A screen capture was taken to outline the code sizes of the subroutines.

```
GetStringSB ..... 0x000006b7   Thumb Code   48   exercise05.o(MyCode)
PutStringSB ..... 0x000006e7   Thumb Code   32   exercise05.o(MyCode)
PutNumU ......... 0x00000707   Thumb Code   50   exercise05.o(MyCode)
```

Figure 5: Code size of written subroutines.

Figure 5 shows the code size of `GetStringSB`, `PutStringSB` and `PutNumU` (with extra-credit). Sizes are listed in decimal and are 48, 32, and 50 for `GetStringSB`, `PutStringSB` and `PutNumU` respectively.

Questions

1. Why does the number of input characters stored in the string have to be one fewer than the number of bytes allocated for the string?
   The standard for a string is to add a null terminator at the end of the string. This is so that variable length string can be stored into an array of constant size and the its length does not need to be held elsewhere. The number of characters stored must include the null terminator and therefore the number of readable characters must be 1 less than the buffer size.
2. Why does `MAX_STRING` need to be defined as an EQUate rather than as a DCD in the `MyConst AREA`, (i.e., a constant)?
   `MAX_STRING` must be used at compile time. The `MyConst AREA` is stored in ReadOnly RAM and the assembler will not read RAM at compile time. The `EQU` instruction is a directive expanded by the assembler and can be used as a compile time constant.