Lab #5 – Memory Reading and Writing

Goals:

Write simple programs on the Discovery Board using memory load and store instructions

Get acquainted with the Discovery Board graphics display

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.

Remember the 'SVC #0' debugging instruction available with your Discover Board library! That can be inserted in your assembly code any time you need to clarify exactly what is in the registers!

Refer to the Weeks 4 and 5 lecture slides for the following:

Part 1: As shown in lecture, the total address space allocated for Flash read-only memory for storing code instructions in the ARM Cortex M-4 processor runs from addresses 0x00000000 to 0x1FFFFFFF.

a) What is the total number of potential addressable locations in this range, expressed in decimal? **Remember the shortcut** discussed in lecture for converting to decimal a binary number that consists of all 1's over the lower bits of some field: The value is simply the next higher power of 2 binary number position value minus one. For example, for a 16-bit number consisting of the twelve lowest bits set to '1' and the highest four cleared to '0' (0000111111111111), we can easily find that the decimal equivalent is 4095 without having to actually sum 12 position values. Since the 1's range from right to left over the position values of 2^0 to 2^{11}, or 1 to 2048 (using the reference table of powers of two at the top of the page), the binary position value of the first zero to the left is 2^{12}, or 4096, and then subtracting 1 gives 4095 .

b) There are actually only 2MBytes of physical memory functioning in this address space in the processor chip on the Discovery Board. What fraction of the address space is actually in use? (See the lecture slides for Week 1 for the definition of 'M'.)

Part 2: Download the template main C program Lab05-Main.c and the template assembly source code Lab05-func.s from the Canvas files area, and in the LabCode folder. Note that the main program simply calls a function and prints its return value as a hex number.

a) Write assembly code in the function to load into register R0 the contents of the CPUID word for the ARM processor we are using and return it as the function return value to the calling program. The CPUID word is mapped to a memory address of 0xE000ED00 in the STM ARM processors. As shown in lecture, you will need an LDR pseudo-instruction to first load the memory address into some register, and then an actual LDR instruction to load the memory word contents as the function return value into R0. What is the value displayed by the main program? Check it against the expected value from the ARM Cortex-M4 manual:

4.4.2 CPUID base register (CPUID)

Address offset: 0x00

Reset value: 0x410F C241

Required privilege: Privileged

The CPUID register contains the processor part number, version, and implementation information.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Implementer								Variant				Constant			
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PartNo												Revision			
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

Bits 31:24 **Implementer**: Implementer code
0x41: Arm

Bits 23:20 **Variant**: Variant number
The r value in the r*npn* product revision identifier
0x0: revision 0

Bits 19:16 **Constant:** Reads as 0xF

Bits 15:4 **PartNo**: Part number of the processor
0xC24: = Cortex-M4

Bits 3:0 **Revision**: Revision number
The p value in the r*npn* product revision identifier, indicates patch release.
0x1: = patch 1

Now in your workspace command terminal window, manually run the ‘objdump’ command to disassemble the obj/Lab05-func-s.o object file produced by the assembler with the command:

```
arm-none-eabi-objdump -d obj/Lab05-func-s.o
```

In the left column you will see the addresses of the machine code words contributed to the .text section by this source code module. (These are only the addresses relative to the start of this module; the actual physical addresses in the ROM with the other modules are computed later by the linker.) The next column contains the hex representation of the machine code prepared by the assembler. How many instructions are contributed to the .text segment by this module? Are they encoded as 16-bit or 32-bit instructions? At what address is the constant 0xE000ED00, needed by your LDR pseudo-instruction, located in the “literal pool” at the end of the machine code?

b) Refer to the lecture slides describing the memory-mapped graphics display on the Discovery Board. Rewrite the Lab05-func.s code so that now the base address of the graphics memory, 0xD0000000, is loaded into a register for memory access. Then calculate the offset value to be loaded into a second register to read the word corresponding to the pixel at a (column,row) coordinate of (100,150) from the display. Write your code to use an LDR instruction in the form ‘`LDR Rd, [Rn,Rm]`’ to read the

pixel so that its hex word becomes the function return value. The register you use in your code for Rn will hold the base address of the display, and the register you use for Rm will hold the offset into the display array. What is the pixel word reported by the main C calling program, and what color does it correspond to? Is this the color you expect at that pixel coordinate?

c) Modify your program in b) for the pixel (column,row) coordinates (0,0) and report a pixel word and color. Is this the color you expect at that pixel coordinate?

d) Modify your program in b) for the pixel (column,row) coordinates (239,319) and report a pixel word and color. Is this the color you expect at that pixel coordinate?

e) In this step we'll learn about two useful assembler directives, .rept and .endr, for repeating any block of program instructions a fixed number of times. The lines

```
.rept <n>
...
(Any number of assembly instructions)
...
.endr
```

will produce the assembler output equivalent to copying and pasting the assembly lines in the repeat block <n> times in the source code.

Return to using the offset value for coordinates of (100,150) from b) as an initial value in the Rm register, and load the word corresponding to a red pixel into another register. Now use an STR instruction in the form '`STR Rs, [Rn,Rm]`' to store the pixel word into memory, followed by an ADD instruction to increment the *offset* register (not the base address) to index the next pixel in the next column to the right. Use the .rept and .endr directives surrounding the STR and ADD instructions to write the pixel word to 10 consecutive pixels in successive columns including and to the right of (100,150). You should see a short horizontal red line on your graphics screen.

f) Modify your ADD instruction in e) to draw the 10 pixels vertically downward in successive rows. Upload your code and a cell phone photo of your Discovery Board screen.

Part 3: Download and edit the template source code files Lab05-AverageMain.c and Lab05-average.s. Locate the directive for the .data object file section and, as shown in the lecture slides, the four '.word' directives, with the label 'array' labeling the address of the first array word.

a) Go to https://smumustangs.com/sports/football/stats/2021 and pull up the "Game-by-game" tab for the 2021 season. The home attendance is listed in the far right column. Copy and paste the attendance for the four home conference games against South Florida, Tulane, University of Central Florida, and Tulsa into the four .word directives to initialize an array of the attendance figures.

b) Now write the code for the two functions 'average1' and 'average2' to compute the average home conference attendance using two different forms of the LDR instruction. In both functions write code to read the four data words and accumulate their sum in a register. Then use either a UDIV or an LSR instruction to divide the sum by four to compute the average. Arrange the average to be the return value of each function.

In the ‘average1’ function use the LDR instruction form ‘`LDR Rd, [Rn,#<n>]`’ to read each array word into a register. The register you use for Rn, the base memory address, should be set to the ‘array’ address to point to the beginning of the data array. Then use four LDR instructions to load successive array elements into a register for summing, and in each instruction select the fixed offset <n> needed for the array elements 0 through 3.

However, in the ‘average2’ function use the more elaborate LDR instruction form ‘`LDR Rd, [Rn, Rm LSL #<n>]`’ to read each array word into a register. As in the first function, the register you use for Rn, the base memory address, should be set to the ‘array’ address to point to the beginning of the data array. An additional register is used for Rm to serve as an array index. This is initialized to 0 to access the first array element, and incremented by 1 to address each successive logical array element 0 through 3, in the four LDR instructions to load data to sum. What is the required left shift count <n> needed to index each word element?

Run your code and verify that the two functions return the correct average. Upload your code to Canvas.