Lab #4 – Shifts, Multiply, Divide

Goals:

Practice binary bit shifting

Write simple programs on the Discovery Board exploring shifts, multiply, and divide instructions

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.

Refer to the Week 4 lecture slides for the following:

Part 1:

a) Write the hexadecimal number 0xC3C3C3C3 in 32-bit binary.

b) Shift the bits in the binary number from a) **left** one position, imitating what the LSL instruction would do. What is this binary number expressed in hex?

c) Shift the bits in the binary number from b) back to the **right** one position, as the LSR instruction would. What is this binary number expressed in hex?

d) Shift the bits in the binary number from b) again back to the **right** one position, but this time as the ASR instruction would. What is this binary number expressed in hex?

e) Start again with the binary number in a) but this time shift it two positions to the **left**, as the LSL instruction would. What is this number in hex?

f) Shift the bits in the binary number from e) back to the **right** two positions, as the LSR instruction would. What is this binary number expressed in hex?

g) Shift the bits in the binary number from e) again back to the **right** two positions, but this time as the ASR instruction would. What is this binary number expressed in hex?

h) Divide the decimal numbers 368 by 42, and find the integer quotient and the integer remainder. Just use a calculator or calculator app for this.

Part 2:

a) Download the template main C program Lab04-Main.c and the template assembly source code Lab04-func.s from the Canvas files area, and in the LabCode folder.

Note that in lines 9 and 10 of Lab04-Main.c, two prototypes are declared for functions `ufunc()` and `sfunc()`. The function `ufunc()` takes two arguments declared as as 32-bit **unsigned** type ‘uint32_t’, and returns an unsigned integer, but the function `sfunc()` takes two arguments declared as 32-bit **signed** integers of the ‘int32_t’ type, and returns a signed integer. Variables holding the two function arguments and their return values of the appropriate types are declared in lines 13 and 14.

Both functions in Lab04-func.s are called in the C program, and in Lab04-Main.c their return values are displayed on the Discovery Board screen with printf() calls, using the matching formatted output types. You will need to complete the assignment statements for the four function arguments (two arguments for each function) a few times in the following exercises and run the program each time.

b) Download and edit the general framework of the assembly code in the file Lab04-func.s available in the Canvas files section. This code defines the two functions `ufunc` and `sfunc` to be called by the C main program. You will need to complete the assembly code for each function, between the function name labels and the '`bx lr`' return instruction, a few times and run the program on the Discovery Board each time. According to the C function calling conventions, you should only make use of registers R0 through R3 in your assembly code.

Of course, the C main program will load the values of the two argument variables for each function in registers R0 and R1 before each function is called.

c) Complete the assembly code in the function '`ufunc`' to:

Perform a **logical** shift of the bits of the first function argument to the left by a shift count specified by the second function argument with a form of the `LSL` instruction
Insert a debugging breakpoint with the '`SVC #0`' instruction
Perform a **logical** shift of the bits of the first shift back to the right by a shift count specified by the second function argument with a form of the `LSR` instruction
Return the results of the second shift as the function return value

Complete the assembly code in the function '`sfunc`' to:

Perform a **logical** shift of the bits of the first function argument to the left by a shift count specified by the second function argument with a form of the `LSL` instruction
Insert a debugging breakpoint with the '`SVC #0`' instruction
Perform an **arithmetic** shift of the bits of the first shift back to the right by a shift count specified by the second function argument with a form of the `ASR` instruction
Return the results of the second shift as the function return value

Modify all the assignment statements in the C main program to apply the hex number in part 1a above for a shift count of one bit to both functions. Run the program. Compare the hex numbers appearing on the two debugging screens and the two hex function return values to the hex numbers you predicted in parts 1b, 1c, and 1d above. Upload your code to Canvas.

d) Repeat part 2c, but this time passing a shift count of two bits with the function arguments in the C code. Compare the hex numbers appearing on the two debugging screens and the two hex function return values to the hex numbers you predicted in parts 1e, 1f, and 1g above.

e) In the **first quarter** of play in the game against Lamar University last Saturday, the victorious SMU Mustangs scored 14 points, and Lamar gained -16 yards (in other words, lost 16 yards) on a play where they fumbled the ball to the Mustangs. Use the assembly code to predict what these two statistics would have been had they kept up the same rate of scoring and fumbling, respectively, throughout the entire game. First comment out the `LSR` instruction from `ufunc` code and the `ASR` instruction from the `sfunc` code. Next modify the assignment statements in the C main function to pass the appropriate

arguments to the two functions, that is, 14 to `ufunc` and -16 to `sfunc`. What should the two shift count arguments be to extrapolate these two statistics over four quarters? Run your program and check the decimal return values against expected values.

f) Now rewrite the assembly code for each of the two functions to compute an unsigned and a signed **remainder** from integer division. The remainder left over by dividing two integers a and b is $r = a - \left[\left(\frac{a}{b}\right) \times b\right]$ Note that for integer division, the quotient $\frac{a}{b}$ is an integer representing the number of whole times *b* can be divided into *a*, with no fractional part. Therefore $\left(\frac{a}{b}\right) \times b$ will in general be ≤ *a*, and the remainder *r* will be the part left over after whole division. Your two rewritten functions should return the remainder value for both unsigned and signed division as the function values. (Remember to only use registers R0-R3 as per the calling convention.) You can of course make use of the debugging breakpoint instructions as needed for debugging, but you need not include them for the final result. Modify the assignment statements in the C main program to pass the decimal values 368 and 42 as *a* and *b* to the `ufunc` function, and the values -368 and -42 as *a* and *b* to the `sfunc` function and run the program. Compare the two decimal function return values to the numbers you predicted in part 1h above. Upload your code to Canvas.