CPEN 211 2022W1

LAB 4

DUE: `handin` submission 2022-10-30 23:59 Vancouver time; lab demo: Week 9

1 Introduction

In this lab you get practice writing ARM assembly code using the ARM Cortex-A9 built into the Cyclone V FPGA on your DE1-SoC. The ARM Cortex-A9 is full fledged microprocessor found in several smartphones.

Figure 1 illustrates an example ARM assembly program (a program written using low-level assembly instructions) and Figure 2 showcases the overall system used in Lab 4. Section 2 provides a step-by-step tutorial to download and install the "Altera Monitor Program" then use it to run the code in Figure 1 on the system in Figure 2. After this tutorial Section 3 describes an implementation of the "binary-search" algorithm. The binary serch algorithm helps quickly find the desired element in a sorted array of elements.

The main task of this lab is to write the assembly code for the binary search algorithm described in Section 3 following the ARM calling conventions.

```
       .include  "address_map_arm.s"
       .text
       .globl _start
_start:
       LDR R0, =SW_BASE    // R0 = 0xFF200040
       LDR R1, =LEDR_BASE  // R1 = 0xFF200000
L1:    LDR R2, [R0]        // R2 = value on SW0 through SW9 on DE1-SoC
       MOV R3, R2, LSL #1  // R3 = R2 << 1 (which is 2*R2)
       STR R3, [R1]        // display contents of R3 on red LEDs
       B   L1              // unconditional branch to L1
```

Figure 1: Assembly (`lab4fig1.s`) to read switches, shift to the left, and write to LEDR.

The implementation has the following details:

1. The code in Figure 1 copies a value from the switches to the red LEDs on your DE1-SoC after shifting by one bit to the left (using something called memory-mapped I/O). You will use the Altera Monitor Program to configure the DE1-SoC into system shown in Figure 2 by downloading a prebuilt ".sof".

2. As shown in Figure 2, the address space in Lab 4 is divided between a read-write memory and I/O devices. Memory locations with addresses in the range 0x00000000 through 0x3FFFFFFF are contained in the dynamic random access memory (DRAM) chip on the DE1-SoC. This DRAM memory is in a different chip beside the Cyclone V FPGA. The remaining addresses are used for I/O such as the slider switches and LEDs.

3. Your ARM programs can "read" the value of the switches on the DE1-SoC, by using an LDR instruction to read from address 0xFF200040. In Lab 4 you can access the values on all ten switches (not just the lower 8 switches). Similarly, using a STR instruction you can "write" a value to the LEDs via the address 0xFF200000.

4. In Figure 1, the line, `.include "address_map_arm.s"`, says to use the code in "`address_map_arm.s`". This file is provided by Altera and available on Piazza. It contains the following lines relevant to Figure 1:

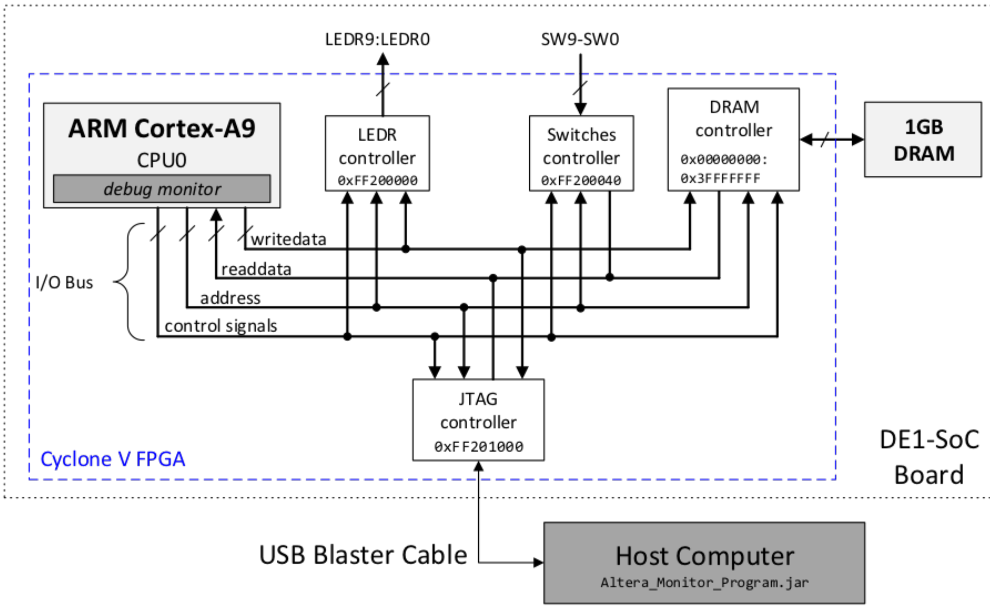


Figure 2: DE1-SoC Computer (Portions Relevant to Lab 4) and Altera Monitor Program

```
.equ   LEDR_BASE,              0xFF200000
.equ   SW_BASE,                0xFF200040
```

These lines specify that `LEDR_BASE` is a constant equal to 0xFF200000 and `SW_BASE` is a constant equal to 0xFF200040.

5. The next line in Figure 1, `.globl _start`, says the program should start at the label "`_start:`". The first instruction, "`LDR R0, =SW_BASE`", says load the value of"`SW_BASE`", which is 0xFF200040 into register R0.

6. When you compile this assembly to run this program on your DE1-SoC (Part 2 in Section 2 below) you will see "`LDR R0, =SW_BASE`" has been transformed by the assembler into the following line:

   ```
   ldr r0, [pc, #16]
   ```

   This line says to add 16 to the value of the PC to form the effective address to use when reading memory. The location that will be read is actually address 24 (0x00000018) instead of 0+16=16 (0x00000010), as one might expect. The reason for this is that, by the time the ARM processor executes the LDR instruction, the value of the program counter (PC) has been incremented by 8 for reasons we will discuss later.

7. In the disassembly window of the Altera Monitor Program debugger you will see that address 0x00000018 contains the following:

   ```
   .word 0xff200040
   ```

   So, the load instruction "`ldr r0, [pc, #16]`" loads the value 0xff200040 into `r0`.

8. The next instruction, "`LDR R1, =LEDR_BASE`", sets R1 to 0xff200000. Next, "`LDR R2, [r0]`" reads the location with address 0xff200040. There is no physical location in the DRAM corresponding to this location. Instead, the controller for the switches shown in Figure 2 recognizes that the address 0xff200040 refers to itself.

9. The “Switches controller” reads all 10 switches on the DE1-SoC at once and places the result in the lower 10-bits of a 32-bit word while setting the upper 22-bits to zero. This value is placed on signal readdata and from their copied into R2 by the LDR instruction.
10. Next, “`MOV R3, R2, LSL #1` reads the value in `R2`, shifts it one bit to the left and then places the result in `R3`.
11. Next, “`STR R3, [R1]`” reads the value in `R3` and writes it to the location with address 0xff200000. Again, there is no physical location in the DRAM corresponding to this location. Instead, the controller for LEDR shown in Figure 2 recognizes that the address 0xff200000 refers to itself and updates a register. The last instruction, “B L1” is an unconditional branch. It branches back to the load that reads the value of the switches.

2 Tutorial

The following steps walk you through the process of running the code from Figure 1 on your DE1-SoC.

1. Download the “University Program Installer” from ftp://ftp.intel.com/Pub/fpgaup/pub/Intel_Material/18.1/intel_fpga_upds_setup.exe. If the prior link does not work, try downloading “University Program Installer” from https://www.intel.com/content/www/us/en/programmable/support/training/university/materials-software.html?&ifup_version=18.1 and/or try using a different web browser. Then, install the Intel FPGA Monitor Program by running the installer after it downloads.
2. Create a directory for Lab 4 and save `address_map_arm.s` (available on Piazza) within that directory. Open a text editor and write the code in Figure 1 into a file `lab4fig1.s`.
3. Launch the “Intel FPGA Monitor Program 18.1” installed in Step 1 by clicking on the icon installed on your desktop. If the font size on the Monitor Program is small, see tip #1 in Section 4.2.
4. Select “File”→“New Project...” to open up a new project dialog window.
5. For “Project Directory” select the directory you created in Step 2.
6. For “Project Name” enter “lab4”.
7. For the “Architecture” drop down menu select “ARM Cortex-A9”.
8. Click on “Next”.
9. For the “Select a system” drop down menu choose “DE1-SoC Computer”.
10. (Optional) Clicking on “Documentation” will open a PDF file describing the DE1-SoC Computer.
11. Click on “Next”.
12. For the “Program Type” drop down menu choose “Assembly Program”.
13. Click on “Next”.
14. Click on “Add...”.
15. Highlight “`lab4fig1.s`” then click on “Select”. Notice the part of the screen that says “Program options” and under it “Start symbol:”. The default value here is “`_start`” which matches the label we used on line 4 in Figure 1. Leave this value unchanged.

16. Click on “Next”. If your DE1-SoC is connected and powered on, you should see “`DE1-SoC [USB-1]`” next to “Host connection:”. If not, check your connection and press “Refresh”. You may need to disconnect and reconnect the USB cable and/or power cycle your DE1-SoC.

17. Once “`DE1-SoC [USB-1]`” (or similar) shows, click on “Next” again.

18. The next window shows where our program will be loaded into memory. The default, which you should keep for Lab 4, is to load the program starting at address 0. Click on “Save”. **NOTE:** ***After going through the steps above once for a project you can skip them by going “File”→“Open Recent Project” in the Altera Monitor Program.***

19. A prompt will ask you *“Would you like to download the system associated with this project onto the board?* Click “Yes”. A SOF file is sent over the USB to configure the DE1-SoC. This may take ˜1 minute.

20. A prompt should say, “The system has been successfully downloaded onto the board!” If the download fails correct the issue (e.g., power on, connect USB cable) then try again using “Actions”→“Download System...” and click “Download”.

21. Click “OK”.

22. Select “Actions”→“Compile”. Verify there are no errors in the bottom right window. To edit the assembly you must use another text editor program. To compile you can also press the button with icon that looks like this: 

23. Select “Actions”→“Load” to download the compiled assembly program onto the DE1-SoC Computer. To load the program onto the ARM core on the DE1-SoC you can also press: 

24. Right click in the disassembly window and unselect “Show source code”. The monitor program should look as shown in Figure 3 where we have highlighted five important parts of the screen.

This screen can be interpreted in the following manner:

- The highlighted sections include the disassembled instructions, encoded instructions, instruction addresses, register values and where to click to set a breakpoint for Part 2 below.
- Under the heading “Disassembly” the first column shows the address of each instruction in *hexadecimal.* The next column shows the 32-bit value, shown in hexadecimal, contained in the four bytes of memory starting at the address in the first column. Recall that each ARM instruction is encoded using 32-bits.
- The third column shows the human readable assembly for this instruction. This is the disassembled instruction that exactly matches the instruction placed in memory. One difference you will notice here versus both Figure 1 is with how “`R0, =SW_BASE`” is shown. We discuss this below.
- The portion labeled “register values” indicates the values in the register file. We discuss setting breakpoints using the grey bar below.
- Also of interest here is the tab that says “Memory”. Clicking on this shows you the contents of memory at different addresses.

2.1 Using the Assembly Debugger

This section introduces you to using the Altera Monitor Program debugging features.

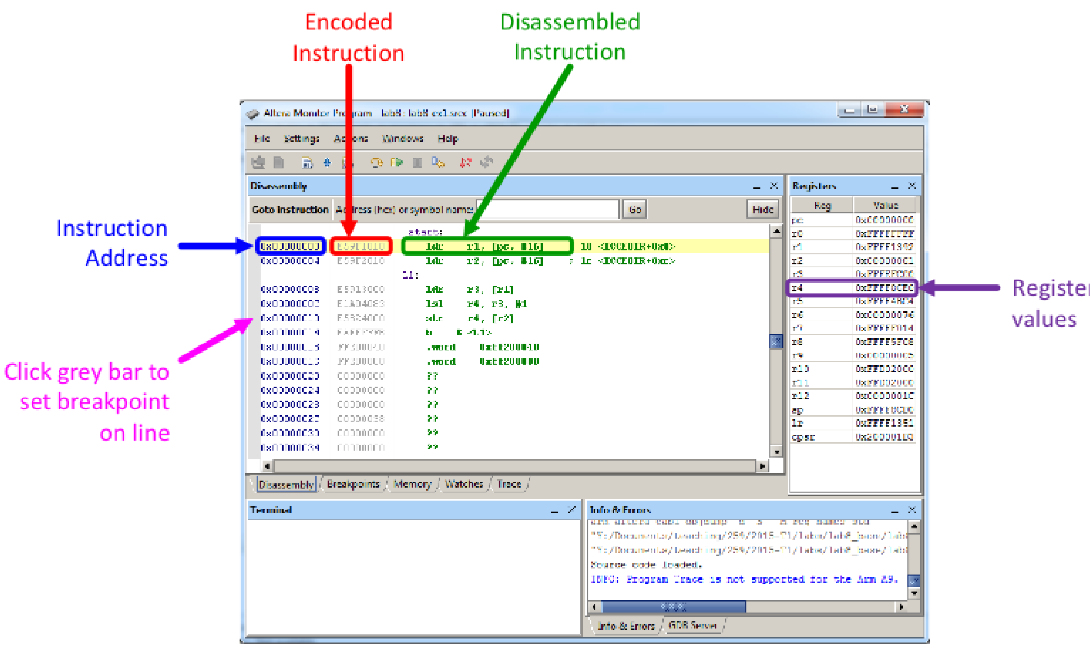


Figure 3: Altera Monitor Program after Loading Program onto DE1-SoC

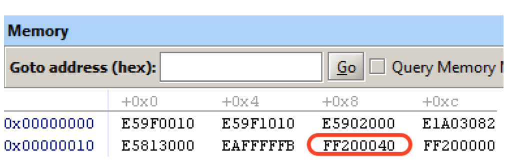
2.1.1 Single Stepping

1. Each instruction in an ARM program appears to execute one at a time in program order. This means we can step through the program one operation at a time to see what happens after each instruction executes. After completing the steps above, the program in Figure 1 should be loaded into memory and the Altera Monitor Program should have the program ready to execute the very first instruction at address 0.

2. To execute just this instruction, which is shown as "`ldr r0, [pc, #16]`", select "Actions"→"Single Step". Notice the registers display on the right updates after each instruction. The registers that change when executing the instruction are highlighted in red. In this case both the program counter ("pc") and the destination register of the load instruction ("r0") are updated.

3. If you double-click on a cell under "Value" in the "Registers" window you can edit the value. If you do this, the Altera Monitor Program running on your desktop or laptop computer sends the new value over the USB cable and it is written into the register file contained inside the Cortex-A9 inside the Cyclone V chip on your DE1-SoC.

4. How did the first instruction write the value `0xFF200040` placed into `r0`? As discussed in Section 1, this is the value loaded into memory at address 0x00000018, which is shown as "`.word 0xff200040`" in the disassembly window. If you click on the tab that says "Memory" you should see:



	+0x0	+0x4	+0x8	+0xc
0x00000000	E59F0010	E59F1010	E5902000	E1A03082
0x00000010	E5813000	EAFFFFFB	FF200040	FF200000

5. In the memory tab you can see the actual values in memory for your program. Notice that the hexadecimal value 0xE59F0010 corresponding to the first instruction is at address 0x00000000 and

the value 0xFF200040 is at address 0x00000018 (highlighted in red).

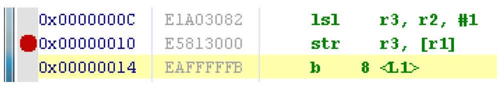
6. If your program ever starts acting very strange (especially in Lab 4) you should check the memory tab to see if the program has overwritten the instructions in memory with data! The reason you need to check the memory tab is that the view of the program in the disassembly tab is not refreshed if your program accidentally overwrites the program itself.

7. To single step you can also click on the button with the following icon: 

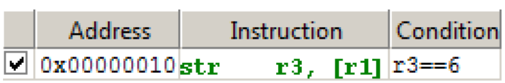
8. Single step at least until the branch instruction at address 0x00000014.

2.1.2 Setting a breakpoint

1. If you single click on grey bar to the left of the instruction address shown in the figure above you will set a breakpoint. Set a breakpoint on the store instruction at address 0x00000010. You should see the red circle shown below if you set the breakpoint:



2. After setting the breakpoint, select the “Breakpoints” tab to inspect and modify the breakpoint. Double click under the heading “Condition” and enter “r3==6” (without quotes). The result should look like this:



3. This will cause the breakpoint to only be triggered if r3 is equal to 6, which will happen if only SW0 and SW1 are in the up position. Select the “Dissassembly” table. Then, select “Actions”→“Continue” to allow the program to run until reaching this point. You can also click on the following icon:

4. Move SW0 and SW1 to the up position and the program should stop right before executing the store instruction. Single step over the store instruction and only LEDR1 and LEDR2 should be on. Also note that when the program stops at a breakpoint the registers and memory values are updated.

3 Binary Search

1. You will implement the “C function” in Figure 4 in ARM assembly which is described below. This function is used as a reference code and it will help you in writing your ARM assembly code.

2. If you find any part of the explanation below unclear then try compiling the C code in Figure 4 using the Microsoft Visual Studio (the programming environment you used in APSC 160) then “single-step” through each line of code in Figure 4 using the integrated debugger in Visual Studio.

   If you have forgotten how to do the latter you may want to refer to the following article: https://msdn.microsoft.com/en-us/library/y740d9d3.aspx.

3. As the name implies, the function `binary_search()` uses a binary search algorithm to look through a sorted array of positive integers (`numbers`) for a specific number (`key`) and returns the position of that number in the sorted array (`keyIndex`).

```
1  int binary_search(int *numbers, int key, int length)
2  {
3    int startIndex = 0;
4    int endIndex = length - 1;
5    int middleIndex = endIndex/2;
6    int keyIndex = -1;
7    int NumIters = 1;
8    while (keyIndex == -1) {
9      if (startIndex > endIndex)
10       break;
11     else if (numbers[middleIndex] == key)
12       keyIndex = middleIndex;
13     else if (numbers[middleIndex] > key) {
14       endIndex = middleIndex-1;
15     } else {
16       startIndex = middleIndex+1;
17     }
18     numbers[ middleIndex ] = -NumIters;
19     middleIndex = startIndex + (endIndex - startIndex)/2;
20     NumIters ++;
21   }
22   return keyIndex;
23 }
```

Figure 4: Binary Search Algorithm

4. The start of the array is passed into `binary_search()` using the 32-bit integer pointer argument `numbers`. The syntax "**`int`** `*numbers`" means that "`numbers`" is a pointer to an integer. A pointer is just a memory address. We treat this pointer as the base of an array using the syntax "`numbers[middleIndex]`". The syntax "`numbers[i]`" where `numbers` has type "pointer to int" means access (read or write) the i-th element of the integer array with base address given by the value of "`numbers`".

5. The positions in array `numbers` are numbered 0 to `endIndex`. If the number `key` is not in the array then -1 is returned. In addition, the number of times the loop iterates is recorded in the variable "`NumCalls`".

6. To see how the code operates, consider Figure 5 where we pass `binary_search()` the global array "numbers" which contains 100 elements numbered 0 through 99 and search an element with value equal to 418. The C keyword **`volatile`** in this code just means that the compiler should not assume it can allocate the location pointed to by `ledr` in a register. The result returned to `main()` by `binary_search()` is 43. Also, the contents of the array "numbers" are modified is shown in Figure 6. For example, the number 488 shown in Figure 4 has been replaced by -1 in Figure 6. This occurred when executing Line 18 in Figure 4 with the value `NumCalls` equal to 1. Similarly, the value 418 in Figure 5 has been replaced by -6 in Figure 6 when executing Line 18 in Figure 4 with `NumCalls` equal to 6. Notice how the value of `NumCalls` is increased on Line 20 in Figure 4.

7. When you write ARM code for this program you can verify you implemented this part correctly by examining the contents of the array "numbers" in memory in the "memory" tab of the Altera Monitor Program. This tab in the Altera Monitor Program allows you to examine the contents of the DRAM memory at any given address. Thus, you can use it to check if the elements of the array have been examined by your calls to `binary_search()`. You can also see the order in which your program examined them.

4 Lab Procedure

1. In file `lab4.s` implement the `binary_search()` function described in Section 3 in ARM assembly code.
2. Note you are required to directly write your own assembly. **Using a compiler (e.g., gcc, g++, Visual Studio, ARM DS-5, etc...) to generate ARM from C code for this lab is considered cheating regardless of how you use the compiler generated code.**
3. A simple example of a main program is provided on Piazza in "main.s". The code in "main.s" puts the address of numbers in R0, the value to search for in R1 and the length of the array in R2. It calls your binary search using a special "branch and link" instruction, `BL binary_search`.
4. You can return to main() from your code by adding a line "`MOV PC,LR`" at the end and ensuring you do not modify R14 (the "link register") inside your binary search code.
5. This MOV instruction acts like a "return" in C but does not pass the result value to the caller (main). Instead, the main.s code assumes the index is returned in R0. We will see more about how function calls work in lectures, but this should be enough knowledge for you to get the main.s to work with your lab4.s.

```
int numbers[100] = {
  28, 37, 44, 60, 85, 99, 121, 127, 129, 138,
  143, 155, 162, 164, 175, 179, 205, 212, 217, 231,
  235, 238, 242, 248, 250, 258, 283, 286, 305, 311,
  316, 322, 326, 351, 355, 364, 366, 376, 391, 398,
  408, 410, 415, 418, 425, 437, 441, 452, 474, 488,
  506, 507, 526, 532, 534, 547, 548, 583, 585, 595,
  603, 621, 640, 661, 666, 690, 692, 713, 719, 750,
  755, 768, 775, 776, 784, 785, 791, 797, 798, 804,
  828, 842, 846, 858, 884, 887, 890, 893, 908, 936,
  939, 953, 960, 970, 978, 979, 981, 990, 1002, 1007 };

int main(void)
{
  volatile int *ledr = (int*) 0xFF200000;   // recall LEDR_BASE is 0xFF200000
  int index = binary_search(numbers,418,100);
  *ledr = index; // display the *final* index value on the red LEDs as in Lab 8
}
```

Figure 5: Example input and usage

```
 28   37   44   60   85   99  121  127  129  138
143  155  162  164  175  179  205  212  217  231
235  238  242  248   -2  258  283  286  305  311
316  322  326  351  355  364   -3  376  391  398
408  410   -4   -6  425   -5  441  452  474   -1
506  507  526  532  534  547  548  583  585  595
603  621  640  661  666  690  692  713  719  750
755  768  775  776  784  785  791  797  798  804
828  842  846  858  884  887  890  893  908  936
939  953  960  970  978  979  981  990 1002 1007
```

Figure 6: Contents of numbers after call to `binary_search()`

6. You must also test your code and verify it works. How you do this is up to you.

4.1 HINTS

1. For the `BL binary_search` in main.s to work you must add the following lines at the top of lab4.s before your code:

```
.globl binary_search
binary_search:
```

2. Lines 5 and 19 in Figure 4 require you to divide by 2. Page 231 in COD4e (Appendix A1) says "While there are many versions of ARM, the classic ARM instruction set had no divide instruction. That tradition continues with the ARMv7A, although the ARMv7R and AMv7M include signed integer divide (SDIV) and unsigned integer divide (UDIV) instructions." The version of ARMv7 in your DE1-SoC is ARMv7A so SDIV and UDIV are *not* supported.

   However, you *can* divide an unsigned (non-negative) integer by any power of 2 by right shifting an appropriate number of times. This should be sufficient for the divide by 2 you need to perform for this lab. You must be a bit careful, because right shift only performs power of 2 division for *unsigned* (non-negative) integers. Performing arithmetic right shift on -1 leaves you with -1 instead of 0 as you would get if you did "-1/2" using C code. For more details see page 262 in Section 3.8 in the PDF of COD4e on Canvas.

3. The easiest way to write an arbitrary constant value larger than 255 into a register is to place the value in memory then load it using LDR as we did for `LEDR_BASE` in Figure 1.

4. You can put a negative number into a register in many ways, but at first none of these may be obvious. One is using reverse subtract with an immediate operand value of zero. For example, you put the value "-R3", the negative of whatever value is in register R3, into register R0 using:

```
RSB R0, R3, #0
```

   You can put -3 into R1 using:

```
MOV R1, #-3
```

   which the compiler will encode using "MVN" as follows (recall the definition of MVN from Lab 6 and how 2's complement works):

```
MVN R1, #2
```

   For larger negative numbers (e.g., -1000) that do not fit into the rotated immediate format you can use:

```
LDR R1,=-1000
```

5. Alternatively, note (see appendices B1 and B2 or http://www.davespace.co.uk/arm/introduction-to-arm/immediates.html) that ARM organizes the 12 bits immediate field for "mov" as an 8-bit immediate field (lower 8-bits) combined with four bits used to "rotate" these 8-bits by an even number of bits. Using these "rotated immediates" you can use "mov" with values such as 0x000000FF (255), 0x00000FF0 (4080), 0x0000FF00 (65280), etc... notice each value has two contiguous non-zero hex digits and they are offset by an even number of bits from the least significant bit.

6. How can you load data into an array in ARM assembly? Consider the following C code:

```
int my_array[2] = {10, 20};
```

This declares a global array of integers called `my_array`. The array `my_array` is initialized before the start of execution so that `my_array[0]` contains the value 10 and `my_array[1]` contains the value 20. How can we do this in ARM assembly? You can effectively declare and initialize the array `my_array` by using a label followed by as many lines as there are elements in the array and with each element initialized using the assembly directive "`.word`":

```
my_array:   .word 10
            .word 20
```

Then, to access "`my_array[i]`" at some point in your code you could use:

```
LDR r0,=my_array
```

This sets `r0` to contain the *base address* of array `my_array`. Recall that if `r0` contains the base address of array `my_array`, and the variables `f` and `i` are in registers `r1` and `r2`, then the C code "`f=my_array[i];`" can be implemented in ARM using:

```
LDR r1, [r0, r2, LSL#2]
```

4.2 Debugging Tips!

Below are a collection of tips gleaned from helping students with this lab in prior years:

1. Debugging tip #1: "Small fonts in Monitor Program". A student in 2019W, Andrew Hanlon, worked out the following procedure to increase the font size in the Monitor Program when run on Windows 10, which you may find helpful:
   (a) Right click the shortcut to the monitor program on your desktop.
   (b) Click on "Properties".
   (c) Go to the "Compatibility" tab in the properties window that opens up.
   (d) Click on "Change high DPI settings" near the bottom of the window.
   (e) A new window will come up. Near the bottom of it, check the box saying "Override high DPI scaling behaviour."
   (f) Change the option just below that checkbox to "System."
   (g) Press OK in that window.
   (h) Press OK in the properties window.
   (i) You may have to restart the Altera monitor program.
2. Debugging tip #2: Single step through your ARM code using the Monitor program. Alternatively, you may want to try using Henry Wong's online simulator for the DE1-SoC running the Altera Monitor Program and the "DE1-SoC Computer", which is available here: https://cpulator.01xz.net/?sys=arm-de1soc. To load a program composed of multiple files to the simulator use "File -¿ Load ELF" and upload the .axf file generated when compiled by the Monitor Program. Henry is a former MASc student of Tor's who developed these tools while a teaching assistant during his Ph.D. at the University of Toronto. At the time of writing Henry's simulator does not yet support conditional breakpoints, but does just about anything else you would need to do for Labs 4.
3. Debugging tip #3: Use breakpoints to skip over code. You can press the continue button to let the program run until it hits a breakpoint.

4. Debugging tip #4: Use the “Memory” tab in the Monitor program to examine the contents of memory. You should do this to ensure your spill code and code for line 18 in Figure 4 works correctly.
5. Debugging tip #5: “My program branches when executing an instruction that isn’t a branch!” This usually happens when your code for line 18 in Figure 4 is incorrect. In either case you may accidentally overwrite the code for your program with data (which is bad)! If the data you wrote into memory is later fetched as an instruction it will be executed! If it cannot be executed because the value cannot be interpreted as a valid instruction the ARM processor will “trap” to address 0x00000004 (also known as an “interrupt”). If you jump to address 0x00000010 when executing an LDR or STR instruction check that the effective address is a multiple of 4. See also debugging tip 4.

5 Marking Scheme

Your mark will be:

0/10 marks	If you did not submit any code.
1/10 marks	If the submitted code is incomplete (missing instructions for some of the code in Figure 4).
2/10 marks	If the submitted code is complete but has compilation errors.
3/10 marks	If the submitted code runs but returns incorrect results for all values you search for or crashes/hangs for any input.
5/10 marks	If the submitted code runs but only returns correct results when searching for the value at the initial middle index, or cannot perform more than one search with the provided main.s without needing to download the code to your DE1-SoC again, or your code only works in simulation and not on your DE1-SoC.
7/10 marks	If the submitted code runs and returns correct results for at least two values and you can perform these searches without downloading your compiled code to your DE1-SoC more than once, but it does not appear to update memory correctly.
8/10 marks	If the submitted code runs and returns correct results for at least two values and you can perform these searches without downloading your compiled code to your DE1-SoC more than once, and it appear you do update memory correctly.
9/10 marks	If the submitted code runs and returns correct results for at least four values and you can perform these repeated searches without downloading your compiled code to your DE1-SoC more than once, and it appear you do update memory correctly.
10/10 marks	If the submitted code appears to work on your DE1-SoC using the provided main.s without any issues.

6 Lab Submission

If you are working with a partner, your submission **MUST** include a file called "'CONTRIBUTIONS.txt" that describes each student's contributions to each file that was added or modified. If either partner contributed less than one third to the solution (e.g., in lines of code), you must state this in your CONTRIBUTIONS file and inform the marking TAs.

Note that submitted files may be stored on servers outside of Canada. Thus, you may omit personal information (e.g., your name, SN) from your files and refer to "Partner 1" and "Partner 2" in CONTRIBUTIONS. Submit your code using "handin" as described in the Handin tutorial on Piazza.