DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING
UNIVERSITY OF BRITISH COLUMBIA
CPEN 211 Introduction to Microcomputers, Fall 2019
Lab 9: Recursive Binary Search
Due to Remembrance Day, this assignment will marked by autograder only.
The handin deadline is November 14 at 9:59 PM for all lab sections.

REMINDER: *As outlined in the CPEN 211 Lab Academic Integrity Policy, until all students involved have a grade for Lab 9 you must* **NOT** *share or describe any code you write for this assignment with anyone except your* **authorized** *lab partner for Lab 9,* **NOT** *ask for or use any code offered to you by anyone (other than your authorized lab partner) and* **NOT** *look at or use solution code from anywhere. If you are repeating CPEN 211 you may not reuse any of your code submitted in a prior year.* **Using a compiler to generate ARM from C code for this lab, or Lab 10 or 11 is considered cheating. Use of a compiler will be considered cheating regardless of whether the compiler generated code (or even a portion of it) is used directly, adapted with changes, or a student "just looks at" the compiler generated code to compare with their own.** *Promptly report cases of misconduct you have first-hand knowledge of to the instructor. Your partner is "authorized" to work with you for Lab 9 if https://cpen211.ece.ubc.ca/cwl/ lab_partners.php says they are your "current lab partner" at the time you start working together on Lab 9 up until you demo your code. The deadline to sign up or change lab partners using the above URL is 96 hours before your lab section. Your code* **will** *be checked for plagiarism using very effective plagiarism detection tools. As per UBC policy, all suspected cases of academic misconduct must be reported to the APSC Dean's office. Examples of outcomes for misconduct cases at UBC can be found online. See e.g.: https://universitycounsel.ubc.ca/files/2016/07/SD-2014-2015.pdf.*

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 many smartphones. Figure 1 illustrates an example ARM assembly program and Figure 2 illustrates the overall system used in Lab 9. 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 a recursive implementation of the binary search algorithm. Writing assembly code for the recursive binary search algorithm described in Section 3 following the ARM calling conventions is the main task for this lab.

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

Figure 1: Assembly (`lab9fig1.s`) to read switches, shift to the left, and write to LEDR.

Similar to the example program for Stage 3 from Lab 7 the code in Figure 1 simply copies a value from the switches to the red LEDs on your DE1-SoC after shifting by one bit to the left. You will use the Altera Monitor Program to configure the DE1-SoC into system shown in Figure 2 by downloading a prebuilt ".sof". As shown in Figure 2, and much like in Lab 7, the address space in Lab 9 through 11 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. Just as in Stage 3 of Lab 7, your ARM programs can "read" the value of the switches on the DE1-SoC, by using an LDR instruction to read from address 0xFF200040. Unlike Lab 7 in Lab 9 to 11 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.

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:

```
.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. 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. 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 when we talk about pipelining and 8+16=24. 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.

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
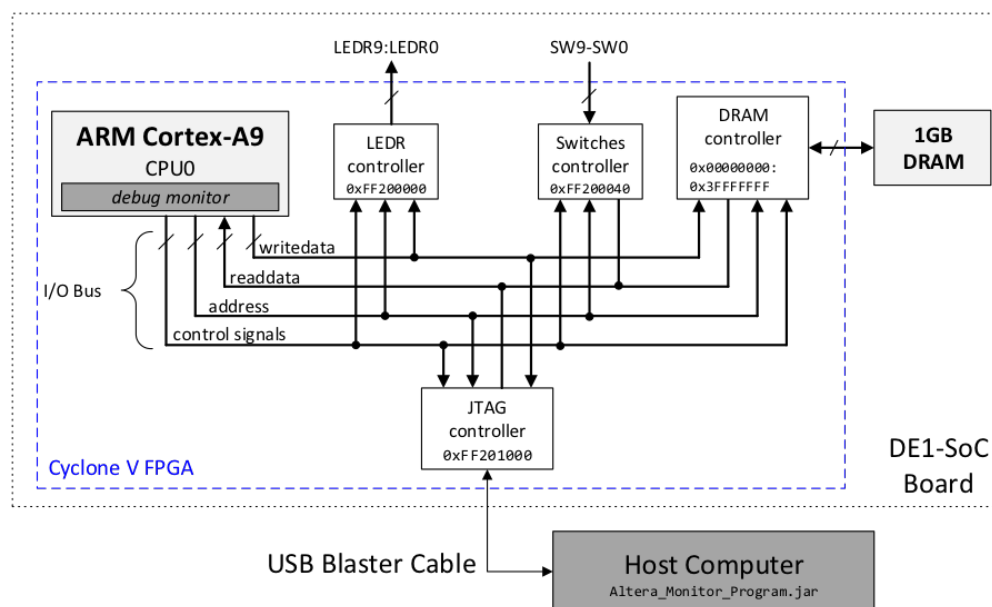

Figure 2: DE1-SoC Computer (Portions Relevant to Lab 8) and Altera Monitor Program

location. Instead, the controller for the switches shown in Figure 2 recognizes that the address 0xff200040 refers to itself. This hardware operates exactly like the hardware you designed for Stage 3 of Lab 7. 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. 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. 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 much like you did in Stage 3 of Lab 7. The last instruction, "B L1" is an unconditional branch back to the load that reads the value of the switches.

2   Tutorial

The following walks 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 Altera Monitor Program by running the installer after it downloads.

2. Create a directory for Lab 9 (e.g., Z:\cpen211\Lab9-L1A) and save `address_map_arm.s` (available on Piazza) there. Open a text editor (vi, emacs, notepad, ModelSim, Quartus, etc...) and write the code in Figure 1 into a file `lab9fig1.s`.

3. Launch the "Altera Monitor Program" installed in Step 1 (e.g., Start > "All Programs" > "Altera UP 15.0" > "Altera Monitor Program" > "Altera Monitor Program").

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 "lab9".

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 "`lab9fig1.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 9, is to load the program starting at address 0. We will change the "Linker Section Presets" in Lab 10 when we work with interrupts. Click on "Finish". **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. (If you want to be considered a serious electrical or computer engineer one day, try "vim" or "emacs".) 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.

The view you see in Figure 3 is very similar to that shown in the ARMSim simulator shown in class. 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 (and also ARMSim) is with how "R0, =SW_BASE" is shown. We discuss this below. The portion labeled "register values" indicates the values in the register file. The register file in the Cortex-A9 is similar to the one you added in Lab 5 except that it contains more registers and each register is 32-bits. 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 (this memory is like the RAM module you added in Lab 7).

2.1 Using the Assembly Debugger

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

2.1.1 Single Stepping

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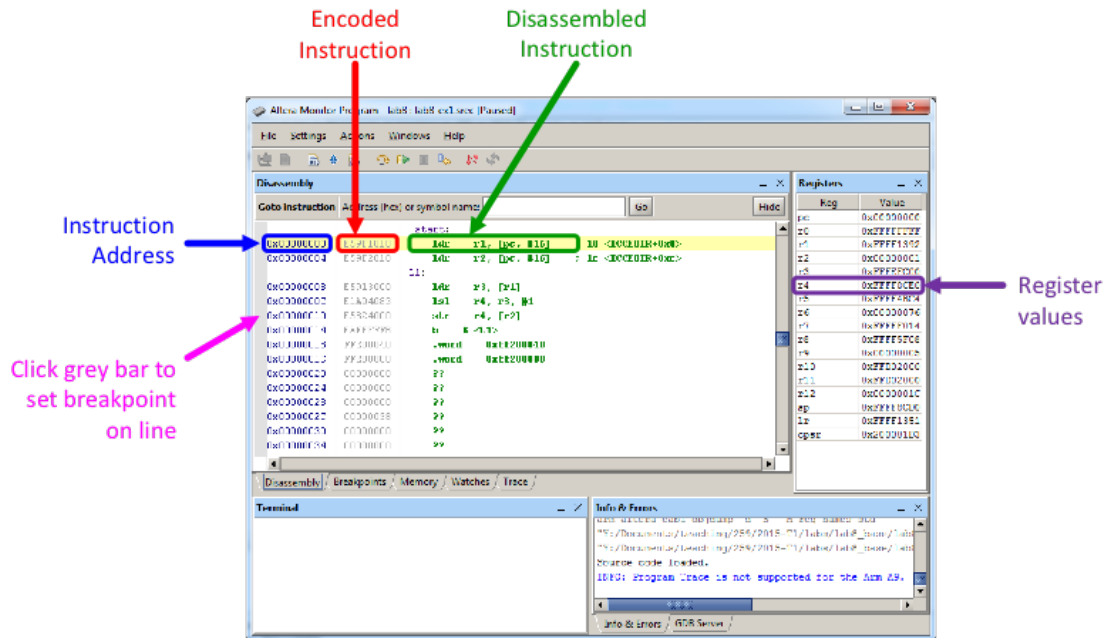
Figure 3: Altera Monitor Program after Loading Program onto DE1-SoC

of the load instruction ("r0") are updated. 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.

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:

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). If your program ever starts acting very strange (especially in Lab 9) 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. To single step you can also click on the button with the following icon:

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

2.1.2 Setting a breakpoint

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:

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:

	Address	Instruction	Condition
✔	0x00000010	str r3, [r1]	r3==6

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:

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 Recursion

You will implement the C function in Figure 4 in ARM assembly and call it from a `main()` function of your own devising to test it. You must follow the ARM calling conventions described in class or your code will not pass the autograder.

Below the operation of the code in Figure 4 is described. 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.

```
1    int binary_search(int *numbers, int key, int startIndex, int endIndex, int NumCalls)
2    {
3      int middleIndex = startIndex + (endIndex - startIndex)/2;
4      int keyIndex;
5      NumCalls++;
6      if (startIndex > endIndex)
7        return -1;
8      else if (numbers[middleIndex] == key)
9        keyIndex = middleIndex;
10     else if (numbers[middleIndex] > key)
11       keyIndex = binary_search(numbers, key, startIndex, middleIndex-1, NumCalls);
12     else
13       keyIndex = binary_search(numbers, key, middleIndex+1, endIndex, NumCalls);
14     numbers[ middleIndex ] = -NumCalls;
15     return keyIndex;
16   }
```

Figure 4: Recursive Binary Search Algorithm

As the name implies, the function `binary_search()` uses a recursive binary search 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`). 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. Recall from class (Slide Set 9) that 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`". The positions in array `numbers` are numbered 0 to `endIndex`. If the number `key` is not in the array then -1 is returned on line 7. In addition, the number of times `binary_search()` is called is recorded in the local variable "`NumCalls`", which is also the fifth

input argument. As discussed in Slide Set 9, only the first four arguments to a function are passed into the function using registers. This means `NumCalls` should *not* be passed into `binary_search()` using a register but rather *must* be placed on the call stack *before* calling `binary_search()`.

The input arguments to a C function are *local variables*. Recall a change to a local variable should not be visible from within another function. This also applies to recursive function calls even though the recursive calls are to the "same" function because each call to a function executes differently and has its own copy of any local variables on the stack. Now, consider these two facts: **Fact 1:** *In C, if you modify an input argument, which is a local variable, as we do on line 5, it should **not** change the value of that input argument as seen by the caller.* For example, suppose we first execute line 11 with `NumCalls` equal to 1 and during the recursive call to `binary_search()` we execute line 5. Then, when we return to line 11 the value of `NumCalls` in the caller should still be 1 even though the *version* of NumCalls seen inside the callee, namely the recursive call to `binary_search()`, was modified to have the value 2. **Fact 2:** *The ARM calling convention requires we pass the first four arguments to a function in registers R0 through R3 and does not ensure these registers have the same values after the call returns.* Notice that when writing assembly you can, if needed, resolve the apparent "contradiction" implied between **Fact 1** and **Fact 2** by making a backup copy of any input argument that is needed after a function call in another location *before* making the function call on the stack. You may want to review the example for `fact()` given in Slide Set 9 and described in detail in Section 2.8 of COD4e to see what this might look like in practice. After the element in the array at `middleIndex` is examined by the code it is replaced with the value "-NumCalls" on line 14. Negative numbers are used here so as to more easily distinguish them from the original numbers, which should all be positive.

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 as 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 the line 14 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 14 in Figure 4 with `NumCalls` equal to 6. Notice how the value of `NumCalls` is increased on line 5 in Figure 4 *before* the recursive calls to `binary_search()` on lines 11 and 13. 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

Implement `binary_search()` function described in Section 3 in ARM assembly code using the conventions described in class. 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, Lab 10 or 11 is considered cheating regardless of how you use the compiler generated code.** Your code MUST implement recursive function calls using the branch and link instruction (BL) and use the stack to save any registers used by the caller. A simple example of a main program is provided on Piazza in "main.s". This code illustrates how the autograder will call your `binary_search` function but simply getting your code to work with "main.s" does not ensure your code will not lose marks. You must also test your code and verify it works. How you do this is up to you.

4.1 HINTS

1. Line 3 in Figure 4 requires 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.

2. 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.

3. 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:

```
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,418,0,99,0);
  *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()

```
        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
```

4. 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.

5. 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: 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: http://cpulator.01xz.net/?sys=arm. Henry is a former MASc student of Tor's who developed these tools while a teaching assistant during his PHD 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 9-11.

2. Debugging tip #2: Use breakpoints to skip over code. You can press the continue button to let the program run until it hits a breakpoint.

3. Debugging tip #3: 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 14 in Figure 4 works correctly.

4. Debugging tip #4: "My program branches when executing an instruction that isn't a branch!" This usually happens when you either forget to initialize the stack pointer, or your code for line 14 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 (we will learn more about this when we talk about "interrupts"). 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 3.

5 Marking Scheme

Your mark will be computed using an autograder by running several tests that verify (a) that the value returned by your `binary_search` function returns the correct value in R0, (b) that your code modifies the `numbers` array correctly (e.g., as illustrated in Figure 6, (c) that you implement a stack that saves and restores registers used by the callee as per the ARM calling convention described in class, (d) that your code for `binary_search` implements binary search by calling itself recursively using the BL instruction and saving/restoring the return PC from/to link register to/from the stack at the begin/end of every call to `binary_search` as is done for the factorial example in Slide Set 9. Marks will be deducted for any of the autograder test cases that fail. To avoid losing marks you should ensure both that you have followed the ARM calling convention, implemented binary search using recursion (**not** a loop), and written extensive tests to verify that your `binary_search` function works for any valid input arguments.

6 Lab Submission

Submit all files by the deadline on Page 1 using handin. Use the same procedure outlined at the end of the Lab 3 handout except that now you are submitting Lab 9, so use:

```
handin cpen211 Lab9-<section>
```

where `<section>` should be replaced by your lab section. Remember you can overwrite previous or trial submissions using `-o`.

Your lab9.zip file MUST include a file lab9.s and your lab9.s must contain a label `binary_search:` or you will get zero for this lab. Your lab9.s may also contain code you use for testing such as that provided on Piazza in "main.s".