

Laboratory Exercise 3

Using an ARM Cortex A9 System

This is an introductory exercise in using the ARM[®] Cortex[®] A9 processor that is included in Intel's Cyclone[®] V SoC devices. The exercise uses the *DEI-SoC Computer*. You can think of this computer system as being a more elaborate version of one of the top-level Verilog files used in Lab 2, like *part5.v*. The *DEI-SoC Computer* includes the ARM processor, memory for holding programs and data, and various I/O ports. The ARM processor accesses these components by using memory-mapped I/O via *address decoding*, similar to the systems that you designed in Lab 2. The address ranges assigned to each of the various devices in the system are specified in the document called *DEI-SoC Computer System with ARM Cortex A9*. This documentation is provided on Quercus along with Lab 3.

In this introductory exercise you will begin learning how to develop programs written in the ARM assembly language, which can be executed in the *DEI-SoC Computer*. You will need to be familiar with the ARM processor architecture and its assembly language. An overview of the ARM processor that is included in Intel's SoC devices can be found in the tutorial *Introduction to the ARM Processor*, which is provided with Lab 3. To develop and “execute” programs for the ARM processor you will use the *CPULATOR* web-based software tool. This tool provides an ARM Assembler that you will use to assemble and run your application programs. The *CPULATOR* uses software techniques to *simulate* the functional behavior of the components in the *DEI-SoC Computer*, including the ARM processor, memory, and a number of I/O devices. This technology is analogous to the *DESim* tool that you used in Labs 1 and 2, but the *CPULATOR* does not use *ModelSim*. The *CPULATOR* is introduced in Part 1, below.

Part I

In this part of the lab exercise you are to watch a short video that provides an introduction to the *CPULATOR* tool. As you will see in the video, the *CPULATOR* is a full-featured software development environment that allows you to compile (assemble) and debug software code for the ARM processor. As mentioned in the video, we should all feel a huge debt-of-gratitude to the author of this tool, Dr. Henry Wong, who developed, and maintains, *CPULATOR* as a volunteer effort on his own personal time. Thank you Henry!!

Use the following URL to access the video from Microsoft Streams, using your *UTOR* credentials:

<https://web.microsoftstream.com/video/15948679-3895-4c1b-91fe-07f0c34c299b>

Part II

Now, we will explore some features of the *CPULATOR* by using a simple application program written in the ARM assembly language. Consider the program given in Figure 1, which finds the largest number in a list of 32-bit integers that is stored in the memory.

Note that some sample data is included in this program. The word (4 bytes) at the label *RESULT* is reserved for storing the result, which will be the largest number found. The next word, *N*, specifies the number of entries in the list. The words that follow give the actual numbers in the list.

```

/* Program that finds the largest number in a list of integers */

        .text                // executable code follows
        .global _start
_start:
        MOV     R4, #RESULT   // R4 points to result location
        LDR     R2, [R4, #4]  // R2 holds number of elements in the list
        MOV     R3, #NUMBERS  // R3 points to the list of integers
        LDR     R0, [R3]      // R0 holds the largest number so far

LOOP:    SUBS    R2, #1        // decrement the loop counter
        BEQ     DONE         // if result is equal to 0, branch
        ADD     R3, #4
        LDR     R1, [R3]      // get the next number
        CMP     R0, R1        // check if larger number found
        BGE     LOOP
        MOV     R0, R1        // update the largest number
        B       LOOP
DONE:    STR     R0, [R4]      // store largest number into result location

END:     B       END

RESULT:  .word    0
N:       .word    7           // number of entries in the list
NUMBERS: .word    4, 5, 3, 6   // the data
        .word    1, 8, 2

        .end

```

Figure 1: Assembly-language program that finds the largest number.

Make sure that you understand the program in Figure 1 and the meaning of each instruction in it. Note the extensive use of comments in the program. You should always include meaningful comments in programs that you will write!

Perform the following:

1. Create an assembly-language source-code file for the program in Figure 1, and name the file *part2.s* (this file is included in the *design files* for this lab exercise). Then, open the *CPUlator* web-site and set its system parameters to choose the ARMv7 processor and DE1-SoC board (the corresponding URL for the *CPUlator* web page should be <https://cpulator.01xz.net/?sys=arm-del-soc>). As indicated in Figure 2, click on the **File** command near the top of the *CPUlator* window and then select **Open . . .**. Next, in the dialogue depicted in Figure 3, browse in your computer's file-system to choose the *part2.s* file and then click **Open**.

The assembly program should be displayed in the **Editor** pane of the *CPUlator* window, as illustrated in Figure 4. You can make changes to your code in this **Editor** pane if needed by simply selecting text with your mouse and making edits using your keyboard.

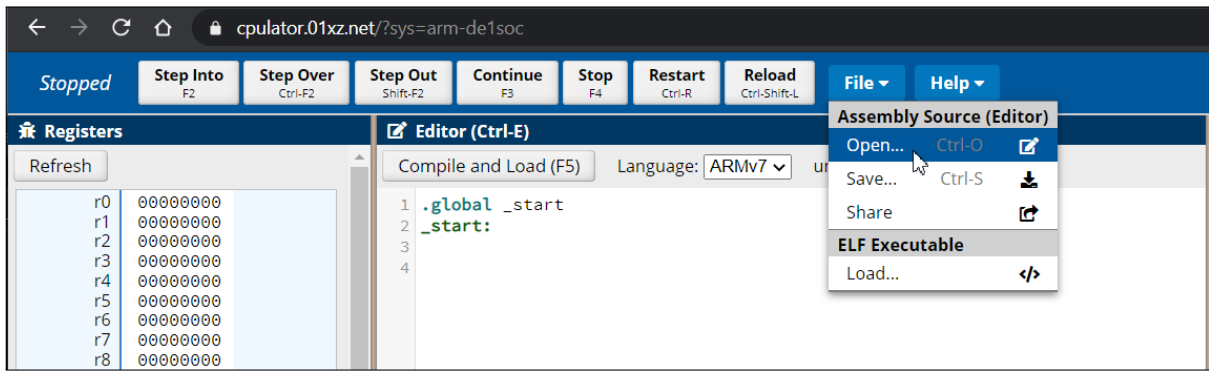


Figure 2: The File menu in CPULator.

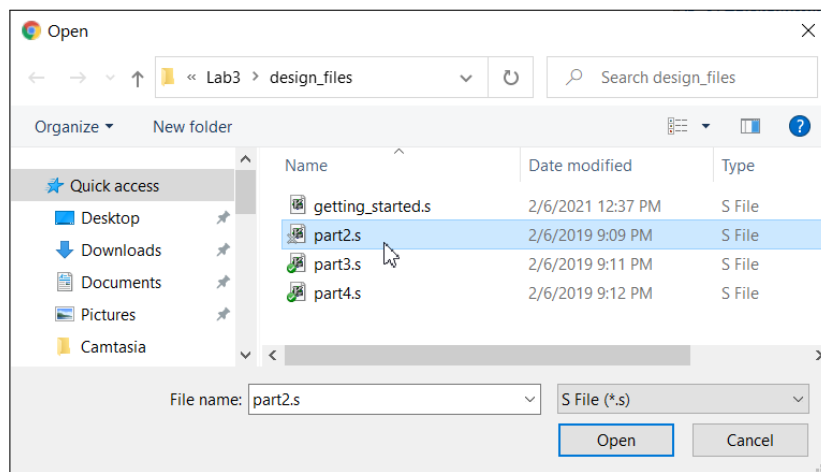


Figure 3: Selecting the *part2.s* file.

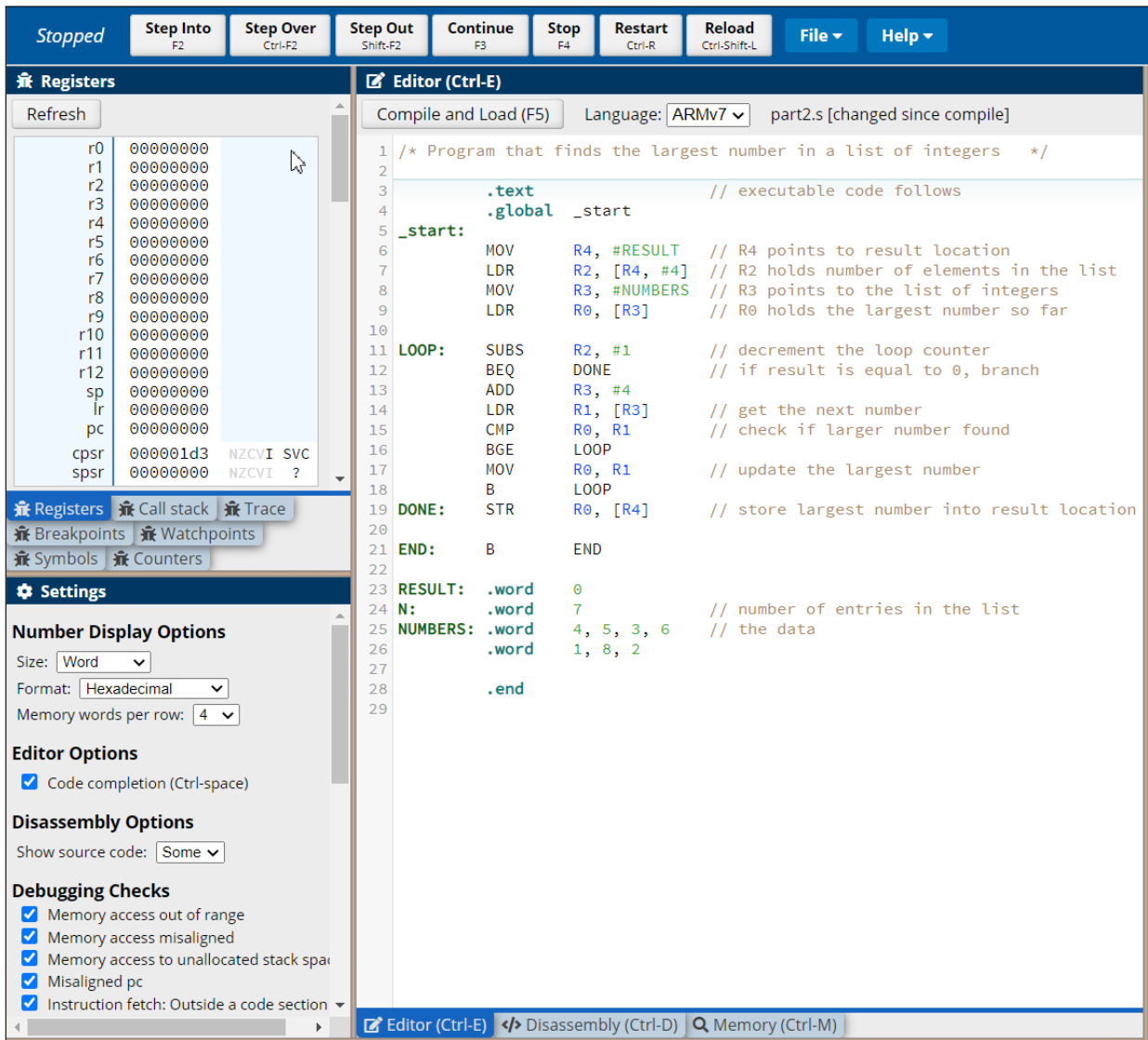


Figure 4: The assembly-language program in the *CPULator* Editor pane.

- As indicated in Figure 5, click on the `Compile and Load` command to *assemble* your program and *load* it into the *memory* that is part of the computer system being simulated within the *CPULator* tool. You should see the message displayed in Figure 6, in the *Messages* pane, which reports a successful compilation result. If not, then you may have inadvertently introduced an error in the program code; fix any such errors and recompile.

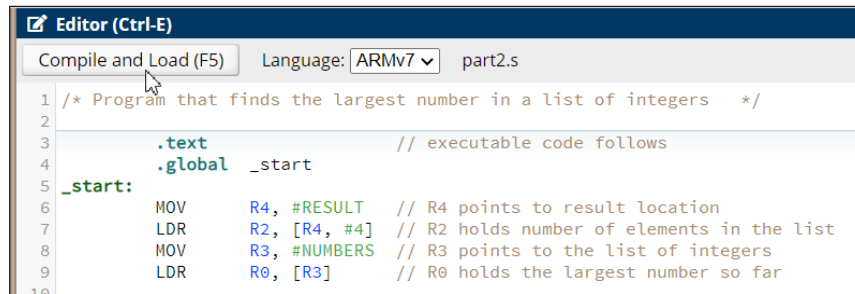


Figure 5: Compiling and loading the program.

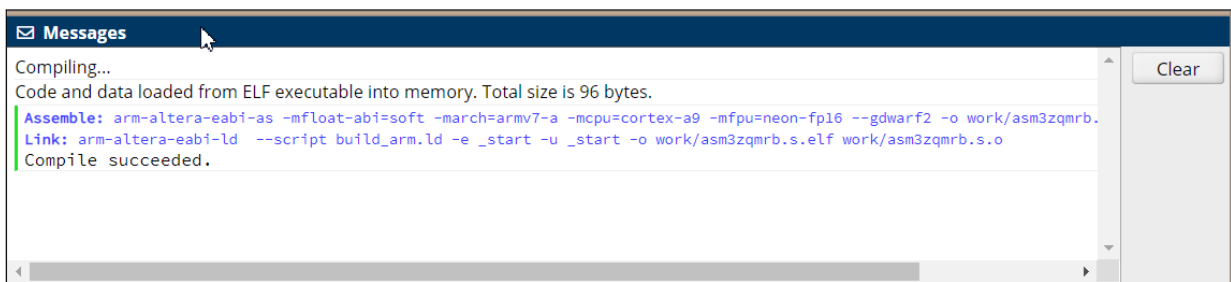


Figure 6: The Messages pane.

Once the compilation is successful, the *CPULATOR* window automatically displays the Disassembly pane, shown in Figure 7. This pane lets you see the machine code for the program and gives the address in the *memory* of each machine-code word. The Disassembly pane shows each instruction in the program twice: once using the original source code and a second time using the actual instruction found by *disassembling* the machine code. This is done because the *implementation* of an instruction may differ, in some cases, from the *specification* of that instruction in the source-code (examples where such differences happen will be shown in class).

3. Select the Continue command near the top of the *CPULATOR* window. This command “executes” the program on the ARM processor that is part of the computer system being simulated within the *CPULATOR* tool. As illustrated in Figure 7, the program runs to the line of code labeled `END`, at memory address `0x34`, where it remains in an endless loop. Select the Stop command to halt the program’s execution. Note that the largest number found in the sample list is 8 as indicated by the contents of register `r0`. This result is also stored in memory at the label `RESULT`.

Disassembly (Ctrl-D)		
Go to address, label, or register: 00000000 Refresh		
Address	Opcode	Disassembly
00000000	e3a04038	_start: mov r4, #56 ; 0x38
00000004	e5942004	7 LDR R2, [R4, #4] // R2 holds number of elements in the list
00000008	e3a03040	8 MOV R3, #NUMBERS // R3 points to the list of integers
0000000c	e5930000	9 LDR R0, [R3] // R0 holds the largest number so far
00000010	e2522001	11 LOOP: SUBS R2, #1 // decrement the loop counter
00000014	0a000005	12 BEQ DONE // if result is equal to 0, branch
00000018	e2833004	13 ADD R3, #4
0000001c	e5931000	14 LDR R1, [R3] // get the next number
00000020	e1500001	15 CMP R0, R1 // check if larger number found
00000024	aa000000	16 BGE LOOP
00000028	e1a00001	17 MOV R0, R1 // update the largest number
0000002c	ea000000	18 B LOOP
00000030	e5840000	19 DONE: STR R0, [R4] // store largest number into result location
00000034	ea000000	21 END: B END
00000038	00000008	RESULT: andeq r0, r0, r8
0000003c	00000007	N: andeq r0, r0, r7
00000040	00000004	NUMBERS: andeq r0, r0, r4
00000044	00000005	andeq r0, r0, r5
00000048	00000003	andeq r0, r0, r3
0000004c	00000006	andeq r0, r0, r6
00000050	00000001	andeq r0, r0, r1
00000054	00000008	andeq r0, r0, r8

Figure 7: The result of executing the program.

The address of the label **RESULT** for this program is 0x00000038, which can be seen near the bottom of Figure 7. Also, you may notice that the Disassembly pane attempts to figure out the machine code at this location, assuming that it represents a processor instruction; it does not, and so the resulting instruction displayed (`andeq`) is not meaningful.

Use the *CPUlator's* Memory pane, as illustrated in Figure 8, to verify that the resulting value 8 is stored in the correct location.

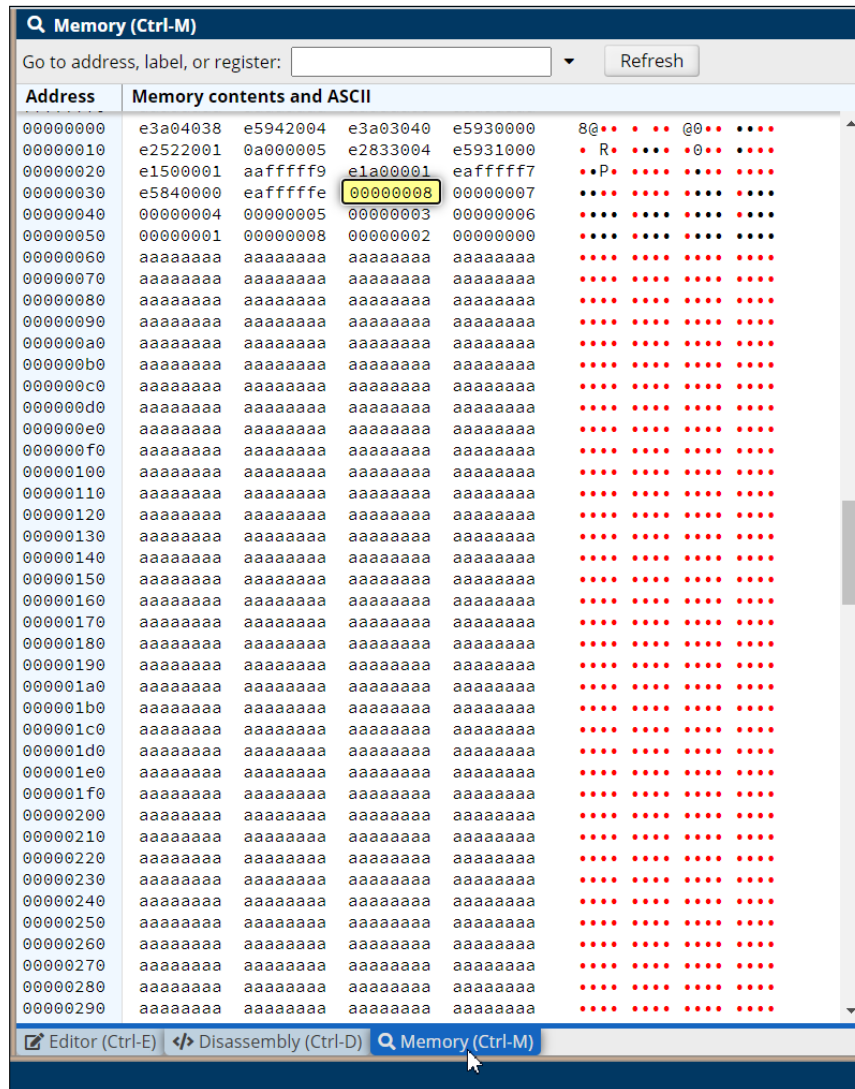


Figure 8: The Memory pane.

4. You can return control of the program to the start by clicking on the `Restart` command in *CPULATOR*. Do this and then *single-step* through the program by (repeatedly) selecting the `Step Into` command. Observe how each instruction that is executed affects the contents of the ARM processor's registers.
5. Double-click on the `pc` register in the *CPULATOR* and then change the value of the program counter to 0. This action has the same effect as selecting the `Restart` command.
6. Now set a breakpoint at address `0x0000002C` by clicking on the gray bar to the left of this address, as illustrated in Figure 9. Select the `Continue` command to run the program again and observe the contents of register `r0` when the instruction at the breakpoint, which is `B LOOP`, is reached. Use `Continue` to repeatedly reach the breakpoint, and monitor the contents of register `r0` as the program searches for the largest number in the list.

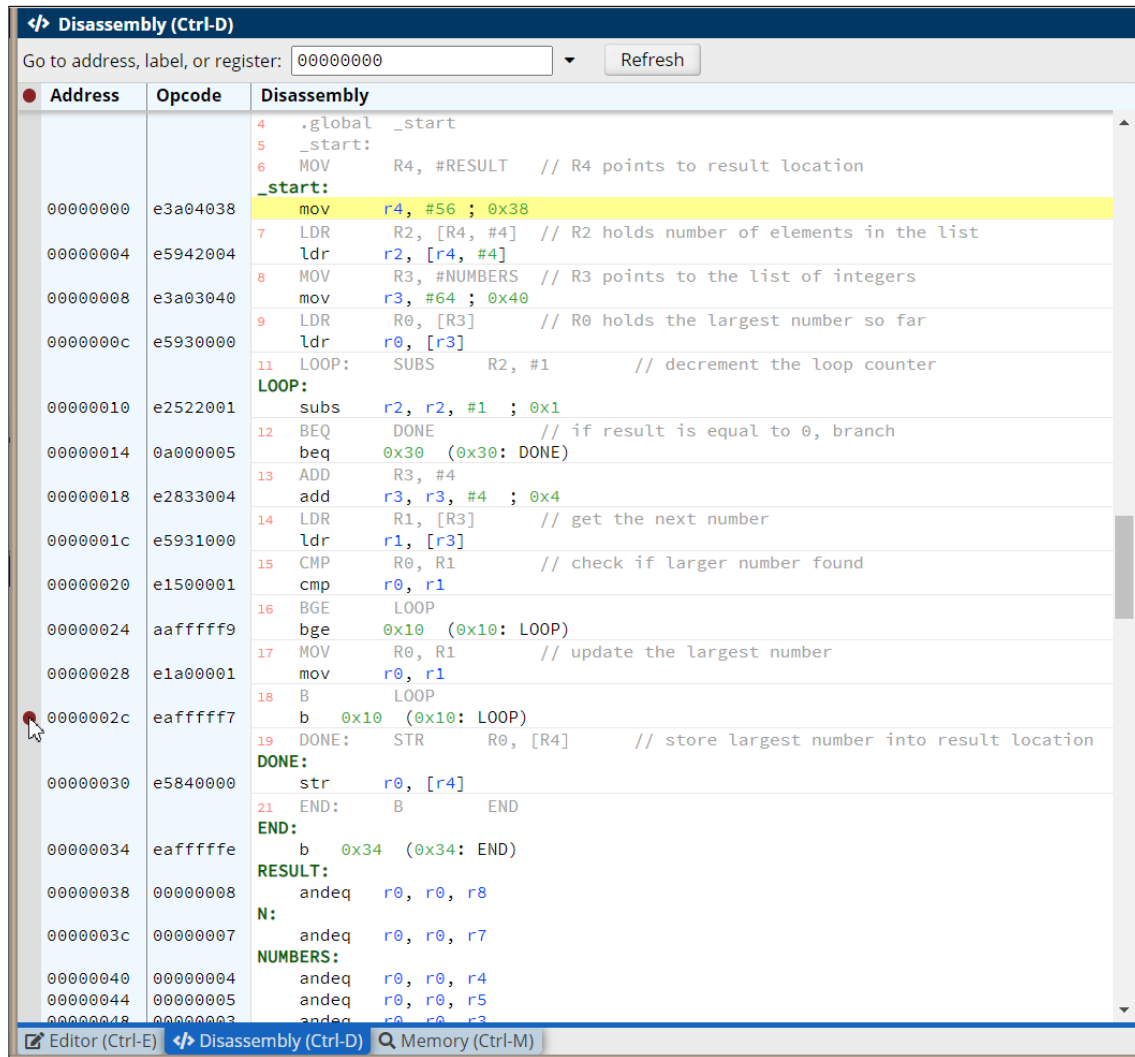


Figure 9: Setting a breakpoint.

Part III

Implement the task in Part II by modifying the program in Figure 1 so that it uses a subroutine. The subroutine, `LARGE`, has to find the largest number in a list. The main program passes the number of entries and the address of the start of the list as parameters to the subroutine via registers `r0` and `r1`. The subroutine returns the value of the largest number to the calling program via register `r0`. A suitable main program is given in Figure 10.

Use the *CPUlator* tool to assemble, load, execute, and debug (as needed!) your program.


```

/* Program that finds the largest number in a list of integers */

        .text                // executable code follows
        .global _start
_start:
        MOV     R4, #RESULT   // R4 points to result location
        LDR     R0, [R4, #4]  // R0 holds the number of elements in the list
        MOV     R1, #NUMBERS  // R1 points to the start of the list
        BL      LARGE
        STR     R0, [R4]      // R0 holds the subroutine return value

END:     B       END

/* Subroutine to find the largest integer in a list
 * Parameters: R0 has the number of elements in the list
 *            R1 has the address of the start of the list
 * Returns: R0 returns the largest item in the list */
LARGE:   . . .
        . . .

RESULT:  .word    0
N:       .word    7           // number of entries in the list
NUMBERS: .word    4, 5, 3, 6  // the data
        .word    1, 8, 2

        .end

```

Figure 10: Main program for Part III.

Part IV

The program shown in Figure 11 converts a binary number into two decimal digits. The binary number is loaded from memory at the location *N*, and the two decimal digits that are extracted from *N* are stored into memory in two bytes starting at the location *Digits*. For the value $N = 76$ ($0x4c$) shown in the figure, the code sets *Digits* to 00000706.

Make sure that you understand how the code in Figure 11 works. Then, extend the code so that it converts the binary number to four decimal digits, supporting decimal values up to 9999. You should modify the *DIVIDE* subroutine so that it can use any divisor, rather than only a divisor of 10. Pass the divisor to the subroutine in register *r1*.

If you run your code with the value $N = 9876$ ($0x2694$), then *Digits* should be set to 09080706. Use the *CPULator* tool to develop your program, and to demonstrate that it works correctly.

```

/* Program that converts a binary number to decimal */

        .text                // executable code follows
        .global _start
_start:
        MOV     R4, #N
        MOV     R5, #Digits // R5 points to the decimal digits storage location
        LDR     R4, [R4]    // R4 holds N
        MOV     R0, R4      // parameter for DIVIDE goes in R0
        BL      DIVIDE
        STRB    R1, [R5, #1] // Tens digit is now in R1
        STRB    R0, [R5]    // Ones digit is in R0
END:     B      END

/* Subroutine to perform the integer division R0 / 10.
 * Returns: quotient in R1, and remainder in R0 */
DIVIDE:  MOV     R2, #0
CONT:    CMP     R0, #10
        BLT     DIV_END
        SUB     R0, #10
        ADD     R2, #1
        B      CONT
DIV_END: MOV     R1, R2      // quotient in R1 (remainder in R0)
        MOV     PC, LR

N:       .word   76         // the decimal number to be converted
Digits:  .space  4         // storage space for the decimal digits

        .end

```

Figure 11: A program that converts a binary number into two decimal digits.