

ECE 243S - Computer Organization
January 2020
Laboratory Exercise 1

Introduction to Assembly Language Programming
Using an ARM Cortex A9 System

The goal of this lab is to introduce you to the basic concepts of Assembly Language programming of a processor, which gives you insight into what a computer is actually doing when it executes a program. We will make use of the ARM® Cortex® A9 processor inside the *DE1-SoC Computer*. This computer system is designed for the DE1-SoC board and includes the processor, memory for holding programs and data, and various I/O ports. The ARM processor accesses each component in the system using memory-mapped I/O; the address ranges assigned to each of the various devices in the system are specified in the document called *DE1-SoC Computer System with ARM Cortex A9*, which is provided on Quercus along with Lab 1.

In this introductory exercise you will begin learning how to develop programs written in the ARM assembly language, which can be executed on the *DE1-SoC Computer*. You will need to be familiar with the ARM processor architecture and its assembly language as described in class. An overview of the ARM Cortex A9 processor can be found in the tutorial *Introduction to the ARM Processor*, which is provided with Lab 1.

Since we cannot do in-person labs at this point in the term, to develop and “execute” programs for the ARM processor you will use the *CPULator* software-based simulation tool. It *simulates* the functional behavior of the components in the *DE1-SoC Computer*, including the ARM processor, memory, and a number of I/O devices. You will find that the CPULator is an excellent tool for developing and debugging ARM programs on your home computer—it is easy to use and does not require a DE1-SoC board. The CPULator is introduced in Part I, 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 URL below to access the video from Microsoft Streams, using your *UTOR* credentials. Note that the video refers to using CPULator for the rest of the term, but that will depend on how Covid plays out this term for us, and whether we'll be able to meet in person in the labs.

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

Part II

Now, we will explore some features of the *CPULator* by working with a simple ARM assembly language program. 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-de1soc>). 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.

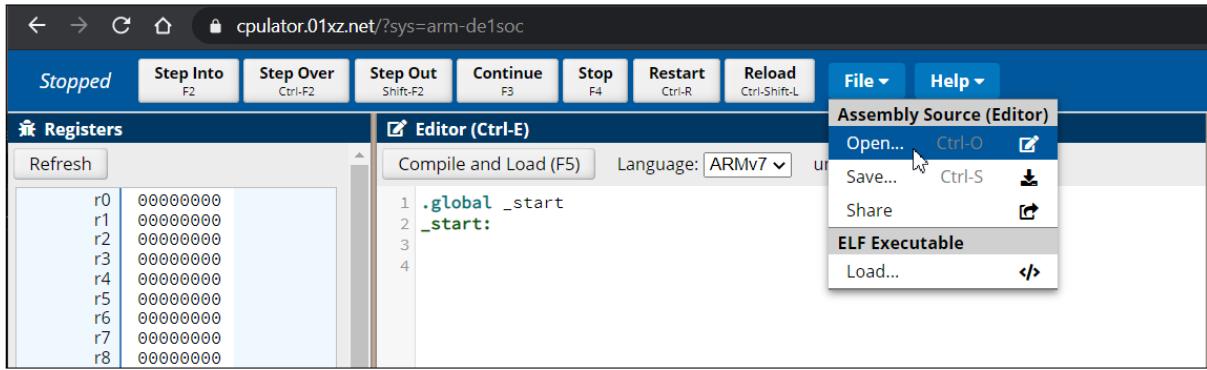


Figure 2: The `File` menu in *CPULator*.

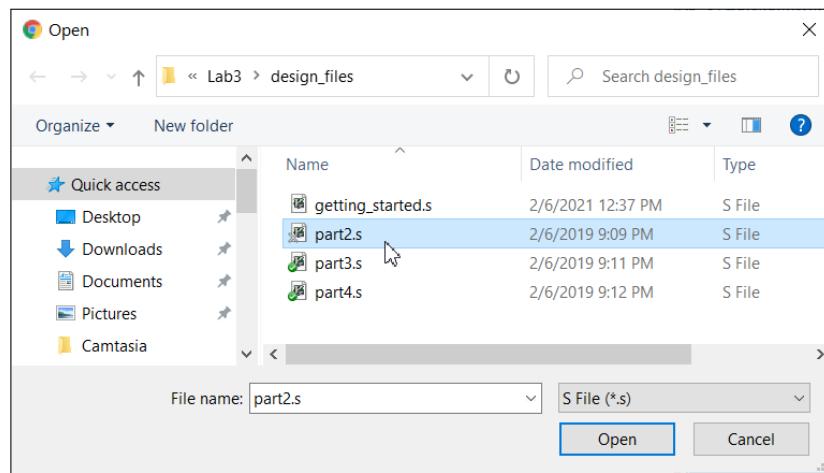


Figure 3: Selecting the *part2.s* file.

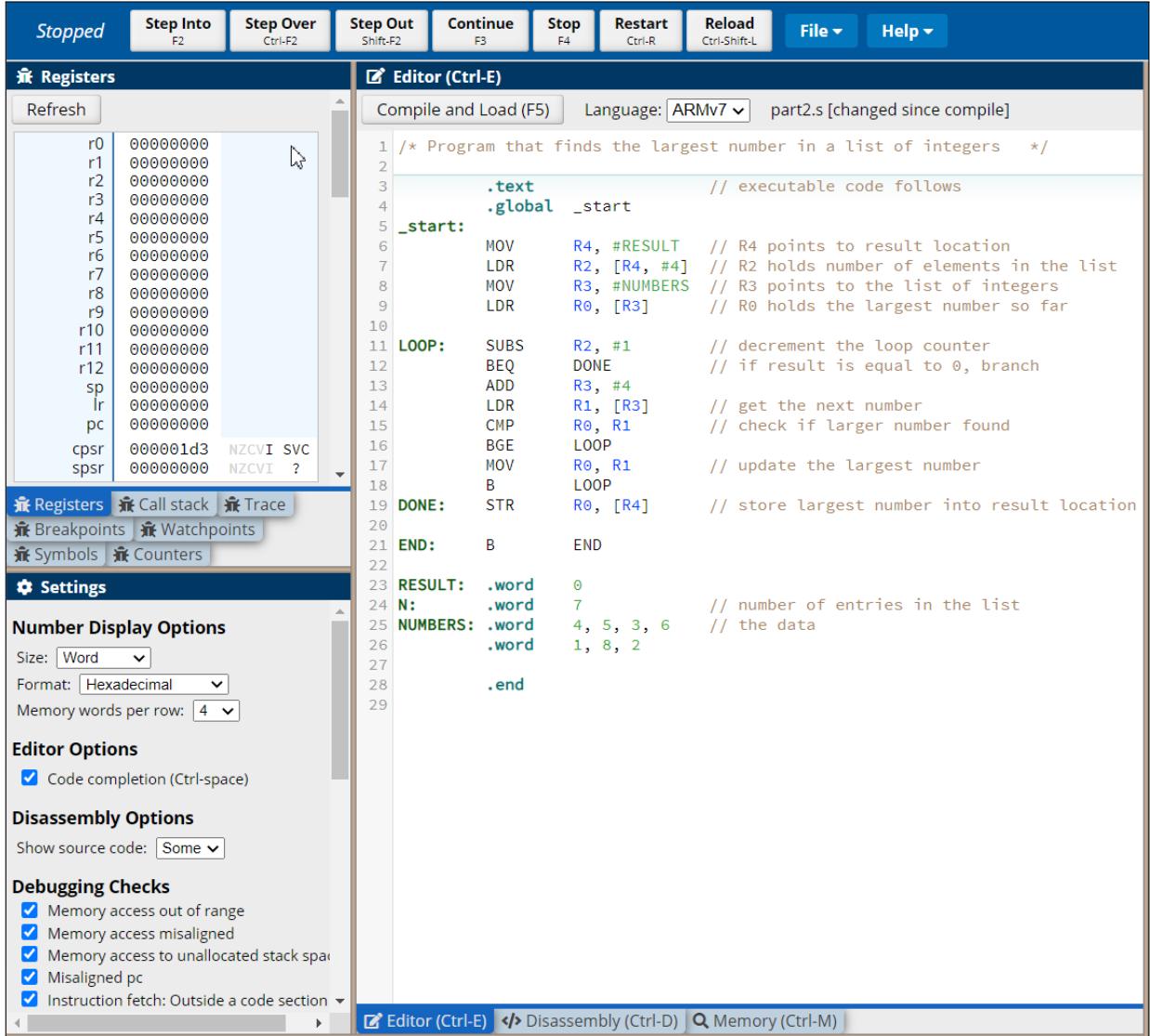
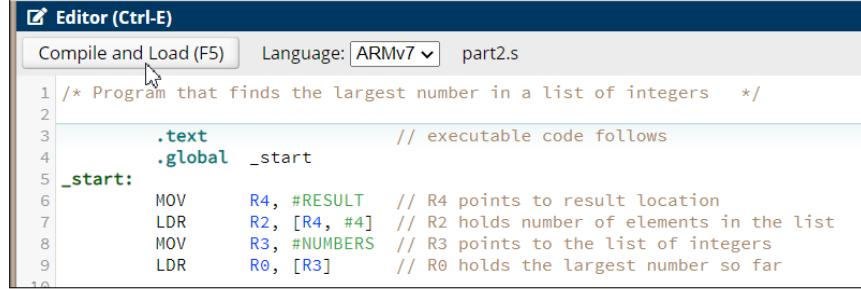


Figure 4: The assembly-language program in the *CPULator* Editor pane.

- 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.

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). Note: you can change the way that code is displayed in the *CPULator* by

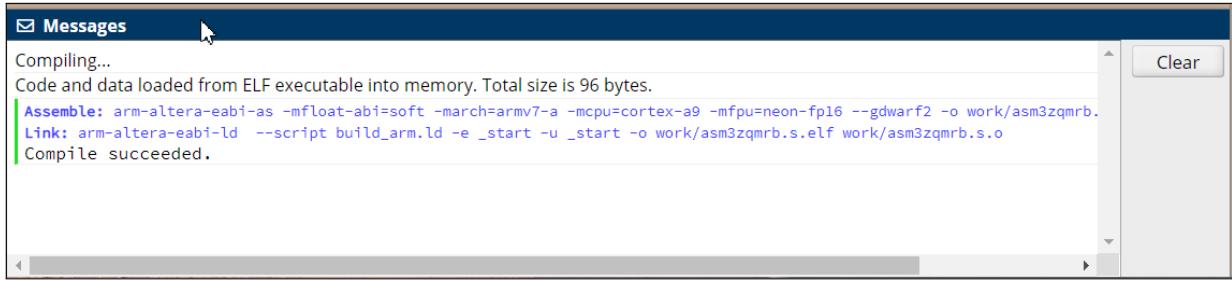
using its **Settings** menu, which is on the left hand side of the *CPUlator* window (for example, changing Disassembly Options from *Some* to *None* will ensure that each instruction is displayed only once).



The screenshot shows the CPUlator Editor window. The title bar says "Editor (Ctrl-E)". The toolbar has "Compile and Load (F5)" and "Language: ARMv7". The file name is "part2.s". The code area contains the following assembly code:

```
1 /* Program that finds the largest number in a list of integers */
2
3     .text          // executable code follows
4     .global _start
5 _start:
6     MOV    R4, #RESULT // R4 points to result location
7     LDR    R2, [R4, #4] // R2 holds number of elements in the list
8     MOV    R3, #NUMBERS // R3 points to the list of integers
9     LDR    R0, [R3]      // R0 holds the largest number so far
10
```

Figure 5: Compiling and loading the program.



The screenshot shows the CPUlator Messages pane. The title bar says "Messages". It displays the following log output:

```
Compiling...
Code and data loaded from ELF executable into memory. Total size is 96 bytes.
Assemble: arm-altera-eabi-as -mfloating-point=soft -march=armv7-a -mcpu=cortex-a9 -mfpu=neon-fp16 --gdwarf2 -o work/asm3zqmrbs.o
Link: arm-altera-eabi-ld --script build_arm.ld -e _start -u _start -o work/asm3zqmrbs.elf work/asm3zqmrbs.o
Compile succeeded.
```

Figure 6: The Messages pane.

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	<code>_start:</code>
00000004	e5942004	<code>mov r4, #56 ; 0x38</code>
00000008	e3a03040	<code>7 LDR R2, [R4, #4] // R2 holds number of elements in the list</code>
0000000c	e5930000	<code>ldr r2, [r4, #4]</code>
00000010	e2522001	<code>8 MOV R3, #NUMBERS // R3 points to the list of integers</code>
00000014	0a000005	<code>9 mov r3, #64 ; 0x40</code>
00000018	e2833004	<code>9 LDR R0, [R3] // R0 holds the largest number so far</code>
0000001c	e5931000	<code>ldr r0, [r3]</code>
00000020	e1500001	<code>11 LOOP: SUBS R2, #1 // decrement the loop counter</code>
00000024	aaffffff9	<code>12 BEQ DONE // if result is equal to 0, branch</code>
00000028	e1a00001	<code>13 breq 0x30 (0x30: DONE)</code>
0000002c	eaffffff7	<code>14 ADD R3, #4</code>
00000030	e5840000	<code>14 ADD R3, #4 ; 0x4</code>
00000034	eafffffe	<code>15 LDR R1, [R3] // get the next number</code>
00000038	00000008	<code>15 ldr r1, [r3]</code>
0000003c	00000007	<code>16 CMP R0, R1 // check if larger number found</code>
00000040	00000004	<code>16 cmp r0, r1</code>
00000044	00000005	<code>17 BGE LOOP</code>
00000048	00000003	<code>17 bge 0x10 (0x10: LOOP)</code>
0000004c	00000006	<code>18 MOV R0, R1 // update the largest number</code>
00000050	00000001	<code>18 mov r0, r1</code>
00000054	00000008	<code>19 DONE: STR R0, [R4] // store largest number into result location</code>
00000058	00000000	<code>19 done: str r0, [r4]</code>
00000062	00000000	<code>20 END: B END</code>
00000066	00000000	<code>20 end: b end</code>
00000070	00000000	<code>END:</code>
00000074	00000000	<code>b 0x34 (0x34: END)</code>
00000078	00000000	<code>RESULT:</code>
00000082	00000000	<code>andeq r0, r0, r8</code>
00000086	00000000	<code>N:</code>
00000090	00000000	<code>andeq r0, r0, r7</code>
00000094	00000000	<code>NUMBERS:</code>
00000098	00000000	<code>andeq r0, r0, r4</code>
0000009c	00000000	<code>andeq r0, r0, r5</code>
000000a0	00000000	<code>andeq r0, r0, r3</code>
000000a4	00000000	<code>andeq r0, r0, r6</code>
000000a8	00000000	<code>andeq r0, r0, r1</code>
000000ac	00000000	<code>andeq r0, r0, r8</code>

Editor (Ctrl-E) Disassembly (Ctrl-D) Memory (Ctrl-M)

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.

Memory (Ctrl-M)								
Go to address, label, or register:								
Address	Memory contents and ASCII							
00000000	e3a04038	e5942004	e3a03040	e5930000	8@.....@0.....			
00000010	e2522001	0a000005	e2833004	e5931000	• R.....0.....			
00000020	e1500001	aaffffff9	e1a00001	eaffffff7	• P.....			
00000030	e5840000	eaffffffe	00000008	00000007	••••••••			
00000040	00000004	00000005	00000003	00000006	••••••••			
00000050	00000001	00000008	00000002	00000000	••••••••			
00000060	aaaaaaaa	aaaaaaaa	aaaaaaaa	aaaaaaaa	••••••••			
00000070	aaaaaaaa	aaaaaaaa	aaaaaaaa	aaaaaaaa	••••••••			
00000080	aaaaaaaa	aaaaaaaa	aaaaaaaa	aaaaaaaa	••••••••			
00000090	aaaaaaaa	aaaaaaaa	aaaaaaaa	aaaaaaaa	••••••••			
000000a0	aaaaaaaa	aaaaaaaa	aaaaaaaa	aaaaaaaa	••••••••			
000000b0	aaaaaaaa	aaaaaaaa	aaaaaaaa	aaaaaaaa	••••••••			
000000c0	aaaaaaaa	aaaaaaaa	aaaaaaaa	aaaaaaaa	••••••••			
000000d0	aaaaaaaa	aaaaaaaa	aaaaaaaa	aaaaaaaa	••••••••			
000000e0	aaaaaaaa	aaaaaaaa	aaaaaaaa	aaaaaaaa	••••••••			
000000f0	aaaaaaaa	aaaaaaaa	aaaaaaaa	aaaaaaaa	••••••••			
00000100	aaaaaaaa	aaaaaaaa	aaaaaaaa	aaaaaaaa	••••••••			
00000110	aaaaaaaa	aaaaaaaa	aaaaaaaa	aaaaaaaa	••••••••			
00000120	aaaaaaaa	aaaaaaaa	aaaaaaaa	aaaaaaaa	••••••••			
00000130	aaaaaaaa	aaaaaaaa	aaaaaaaa	aaaaaaaa	••••••••			
00000140	aaaaaaaa	aaaaaaaa	aaaaaaaa	aaaaaaaa	••••••••			
00000150	aaaaaaaa	aaaaaaaa	aaaaaaaa	aaaaaaaa	••••••••			
00000160	aaaaaaaa	aaaaaaaa	aaaaaaaa	aaaaaaaa	••••••••			
00000170	aaaaaaaa	aaaaaaaa	aaaaaaaa	aaaaaaaa	••••••••			
00000180	aaaaaaaa	aaaaaaaa	aaaaaaaa	aaaaaaaa	••••••••			
00000190	aaaaaaaa	aaaaaaaa	aaaaaaaa	aaaaaaaa	••••••••			
000001a0	aaaaaaaa	aaaaaaaa	aaaaaaaa	aaaaaaaa	••••••••			
000001b0	aaaaaaaa	aaaaaaaa	aaaaaaaa	aaaaaaaa	••••••••			
000001c0	aaaaaaaa	aaaaaaaa	aaaaaaaa	aaaaaaaa	••••••••			
000001d0	aaaaaaaa	aaaaaaaa	aaaaaaaa	aaaaaaaa	••••••••			
000001e0	aaaaaaaa	aaaaaaaa	aaaaaaaa	aaaaaaaa	••••••••			
000001f0	aaaaaaaa	aaaaaaaa	aaaaaaaa	aaaaaaaa	••••••••			
00000200	aaaaaaaa	aaaaaaaa	aaaaaaaa	aaaaaaaa	••••••••			
00000210	aaaaaaaa	aaaaaaaa	aaaaaaaa	aaaaaaaa	••••••••			
00000220	aaaaaaaa	aaaaaaaa	aaaaaaaa	aaaaaaaa	••••••••			
00000230	aaaaaaaa	aaaaaaaa	aaaaaaaa	aaaaaaaa	••••••••			
00000240	aaaaaaaa	aaaaaaaa	aaaaaaaa	aaaaaaaa	••••••••			
00000250	aaaaaaaa	aaaaaaaa	aaaaaaaa	aaaaaaaa	••••••••			
00000260	aaaaaaaa	aaaaaaaa	aaaaaaaa	aaaaaaaa	••••••••			
00000270	aaaaaaaa	aaaaaaaa	aaaaaaaa	aaaaaaaa	••••••••			
00000280	aaaaaaaa	aaaaaaaa	aaaaaaaa	aaaaaaaa	••••••••			
00000290	aaaaaaaa	aaaaaaaa	aaaaaaaa	aaaaaaaa	••••••••			

Editor (Ctrl-E) Disassembly (Ctrl-D) Memory (Ctrl-M)

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 run the program, which will stop at the breakpoint in each iteration of the loop, and monitor the contents of register *r0* as the program searches for the largest number in the list.

Disassembly (Ctrl-D)

Go to address, label, or register: 00000000 Refresh

Address	Opcode	Disassembly
		4 .global _start 5 _start: 6 MOV R4, #RESULT // R4 points to result location
00000000	e3a04038	_start: 7 mov r4, #56 ; 0x38
00000004	e5942004	8 LDR R2, [R4, #4] // R2 holds number of elements in the list ldr r2, [r4, #4]
00000008	e3a03040	9 MOV R3, #NUMBERS // R3 points to the list of integers mov r3, #64 ; 0x40
0000000c	e5930000	10 LDR R0, [R3] // R0 holds the largest number so far ldr r0, [r3]
		11 LOOP: SUBS R2, #1 // decrement the loop counter
		LOOP:
00000010	e2522001	12 subs r2, r2, #1 ; 0x1 13 BEQ DONE // if result is equal to 0, branch beq 0x30 (0x30: DONE)
00000014	0a000005	14 ADD R3, #4 add r3, r3, #4 ; 0x4
00000018	e2833004	15 LDR R1, [R3] // get the next number ldr r1, [r3]
0000001c	e5931000	16 CMP R0, R1 // check if larger number found cmp r0, r1
00000020	e1500001	17 BGE LOOP bge 0x10 (0x10: LOOP)
00000024	aaffffff9	18 MOV R0, R1 // update the largest number mov r0, r1
00000028	e1a00001	19 B LOOP b 0x10 (0x10: LOOP)
0000002c	eafffff7	20 DONE: STR R0, [R4] // store largest number into result location
00000030	e5840000	21 END: B END
00000034	eafffffe	END: 22 b 0x34 (0x34: END)
00000038	00000008	RESULT: 23 andeq r0, r0, r8
0000003c	00000007	N: 24 andeq r0, r0, r7
00000040	00000004	NUMBERS: 25 andeq r0, r0, r4
00000044	00000005	26 andeq r0, r0, r5
00000048	00000003	27 andeq r0, r0, r3

Editor (Ctrl-E) Disassembly (Ctrl-D) Memory (Ctrl-M)

Figure 9: Setting a breakpoint.

Be prepared to show your ability to single step and set breakpoints when you demonstrate other parts of this lab to your TA.

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. Demonstrate the working program to your TA.

```

/* 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.