Laboratory Exercise 1

Using an ARM Cortex-A9 System

This is an introductory exercise using the ARM Cortex-A9 processor that is included in Altera's Cyclone V SoC devices. The exercise uses a pre-defined computer system called the *DE0-Nano-SoC Computer*, which includes the A9 processor and various peripheral devices. The system is implemented as a circuit that is downloaded into the FPGA device on an Altera DE0-Nano-SoC board. This exercise illustrates how programs written in the ARM assembly language can be executed on the DE0-Nano-SoC board. We will use the *Altera Monitor Program* software to compile, load, and run the application programs.

For this exercise you have to know the ARM processor architecture and its assembly language. Read the tutorial *Introduction to the ARM Processor*. You also have to become familiar with the Monitor Program; read the tutorial *Altera Monitor Program Tutorial for ARM*. Both tutorials are available in Altera's University Program web site. The Monitor Program tutorial can also be accessed by selecting Help > Tutorial within the Monitor Program software.

Part I

In this part you will use the Altera Monitor Program to set up an ARM software development project using the DE0-Nano-SoC Computer. Perform the following:

1. Make sure that the power is turned on for the Altera DE0-Nano-SoC board.

2. Open the Altera Monitor Program software, which leads to the window in Figure 1.

   To develop ARM software code using the Monitor Program it is necessary to create a new project. Select File > New Project to reach the window in Figure 2. Give the project a name and indicate the folder for the project; we have chosen the project name *lab1_part1* in the folder *Exercise1\Part1*, as indicated in the figure. Use the drop-down menu shown in Figure 2 to set the target architecture to the ARM Cortex-A9 processor. Click Next, to get the window in Figure 3.

3. Now, you can select your own custom computer system (if you have one) or a pre-designed (by Altera) system. Choose the DE0-Nano-SoC Computer and click Next. The display in the window will now show where files that implement the chosen system are located. If you select a computer system that you designed yourself, then you have to provide the locations of the corresponding files. Click Next.

4. In the window in Figure 4 you can specify the type of application programs that you wish to run. They can be written in either assembly language or the C programming language. Specify that an assembly language program will be used. The Altera Monitor Program package contains several sample programs. Select the box Include a sample program with the project. Then, choose the Getting Started program, as indicated in the figure, and click Next.

5. The window in Figure 5 is used to specify the source file(s) that contain the application program(s). Since we have selected the *Getting Started* program, the window indicates the source code file for this
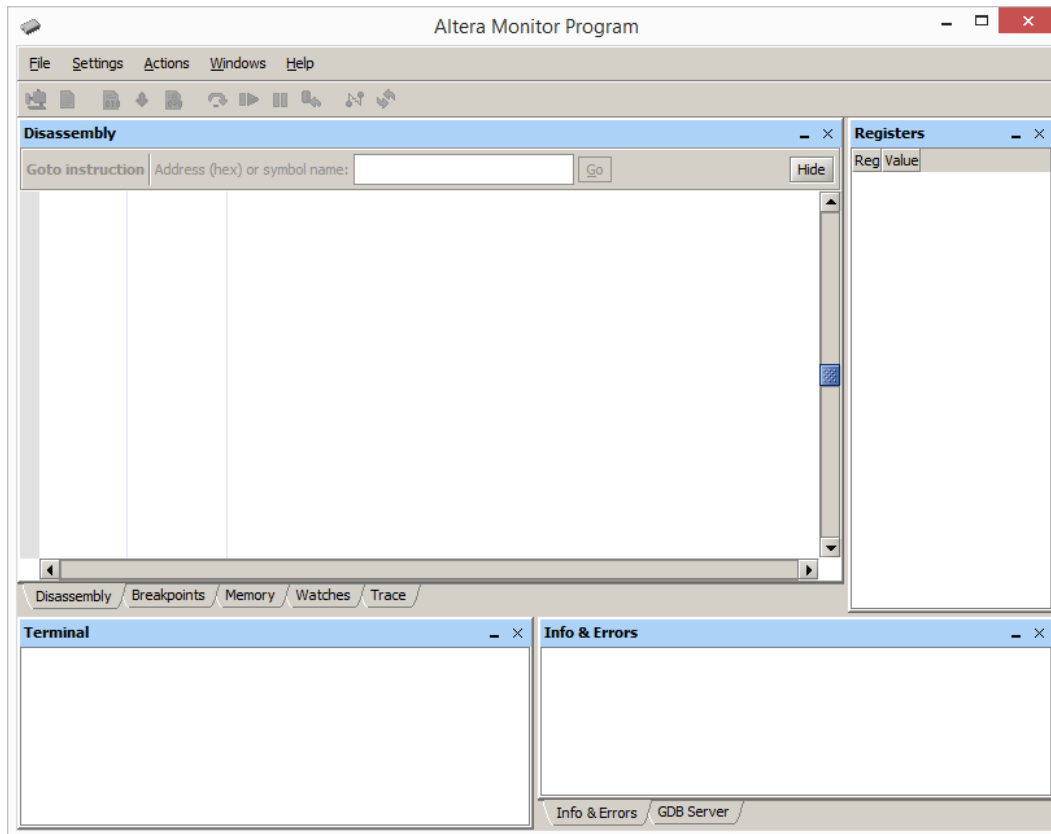
Figure 1: The Altera Monitor Program window.

program. This window also allows the user to specify the starting point in the selected application program. The default symbol is _start, which is used in the selected sample program. Click Next.

6. The window in Figure 6 indicates some system parameters. Note that the figure indicates that the *DE-SoC [USB-1]* cable is selected to provide the connection between the DE0-Nano-SoC board and the host computer. This is name assigned to the Altera USB-Blaster connection between the computer and the board. Click Next.

7. The window in Figure 7 displays the names of Assembly sections that will be used for the program, and allows the user to select a target memory location for each section. In this case only the *.text* section, which corresponds to the program code (and data), is defined. As shown in the figure, the *.text* section is targeted to the DDR3 memory in the DE0-Nano-SoC Computer, starting at address 0. Click Finish to complete the specification of the new project.

8. Since you specified a new project, a pop-up box will appear asking you if you want to download the system associated with this project onto the DE0-Nano-SoC board. Make sure that the power to the board is turned on and click Yes. After the download is complete, a pop-up box will appear informing you that the circuit has been successfully downloaded - click OK. If the circuit is not successfully downloaded, make sure that the USB connection, through which the USB-Blaster communicates, is established and recognized by the host computer. (If there is a problem, a possible remedy may be to unplug the USB cable and then plug it back in.)
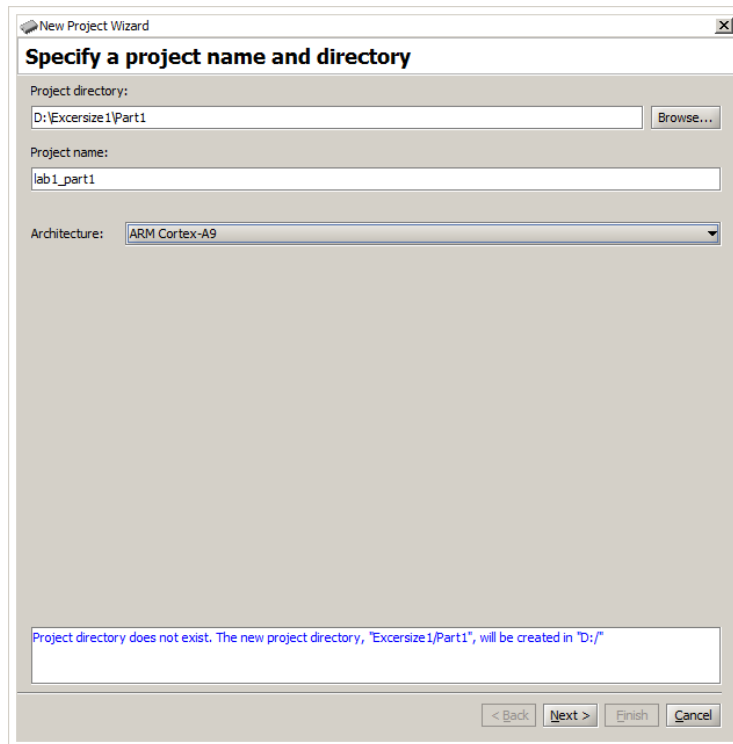
Figure 2: Specify the folder and the name of the project.

9. Having downloaded the DE0-Nano-SoC Computer into the Cyclone V SoC chip on the DE0-Nano-SoC board, we can now load and run the sample program. In the main Monitor Program window, shown in Figure 8, select Actions > Compile & Load to assemble the program and load it into the FPGA chip. Figure 8 shows the Monitor Program window after the sample program has been loaded.

10. Run the program by selecting Actions > Continue or by clicking on the toolbar icon [▷], and observe the patterns displayed on the LEDs.

11. Pause the execution of the sample program by clicking on the icon [⏸], and disconnect from this session by clicking on the icon [⚡],
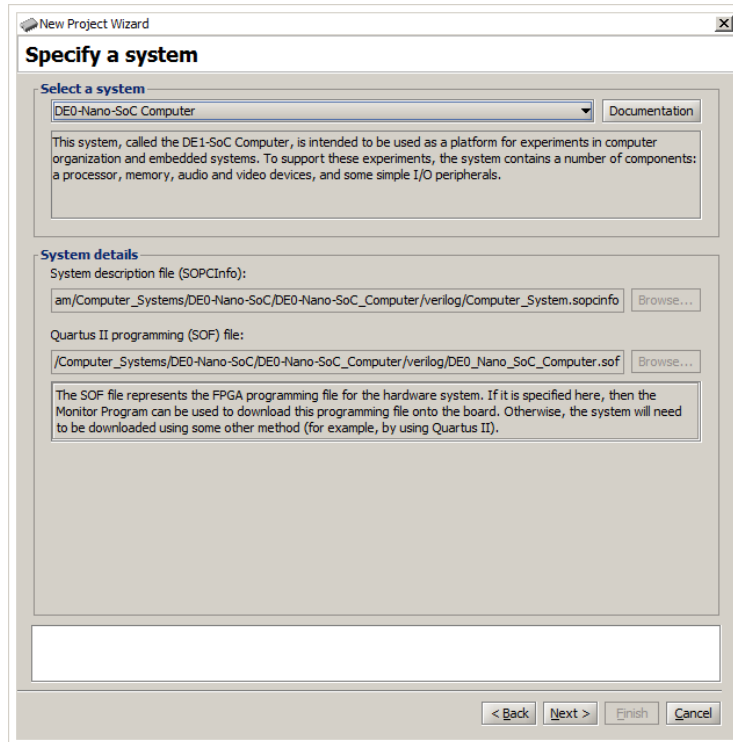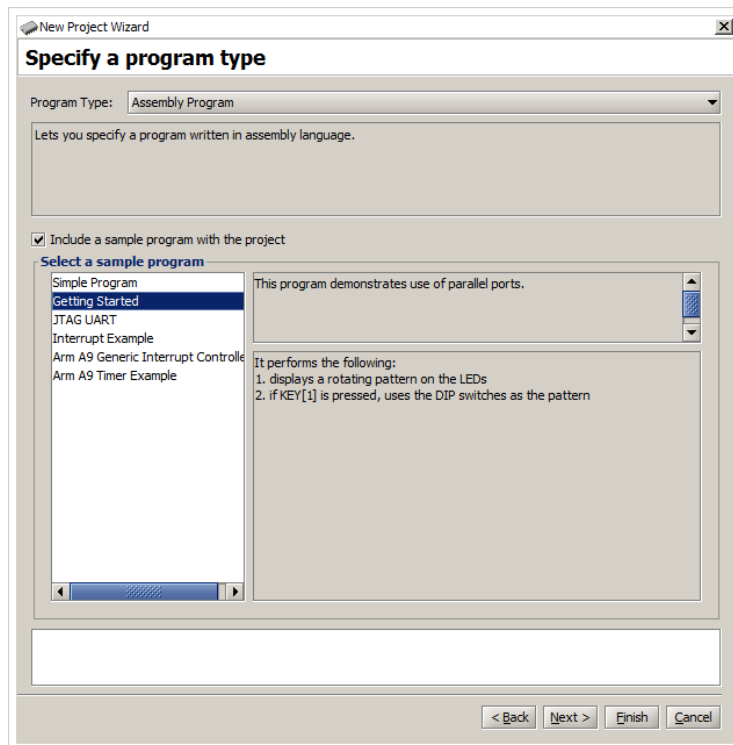
Figure 3: Specification of the system.

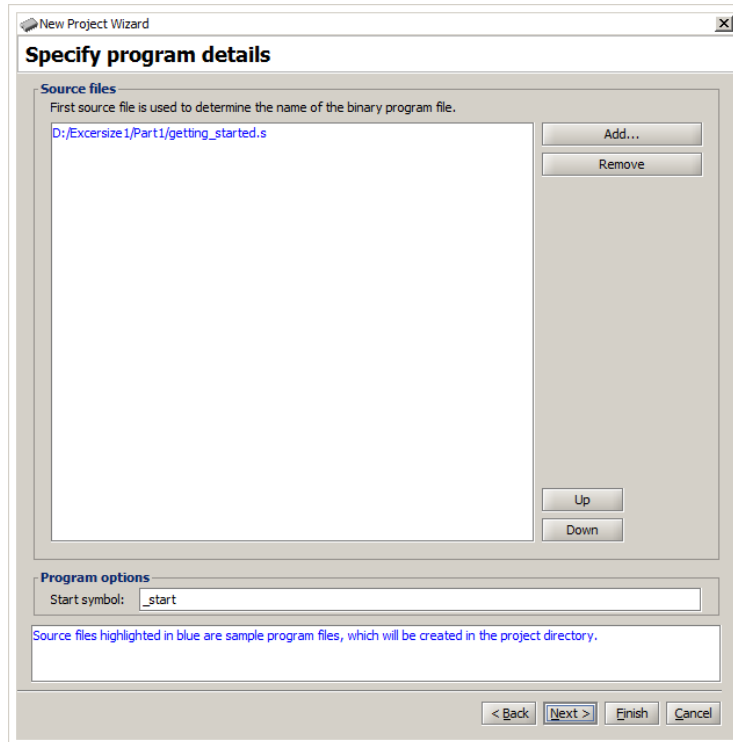Figure 4: Selection of an application program.

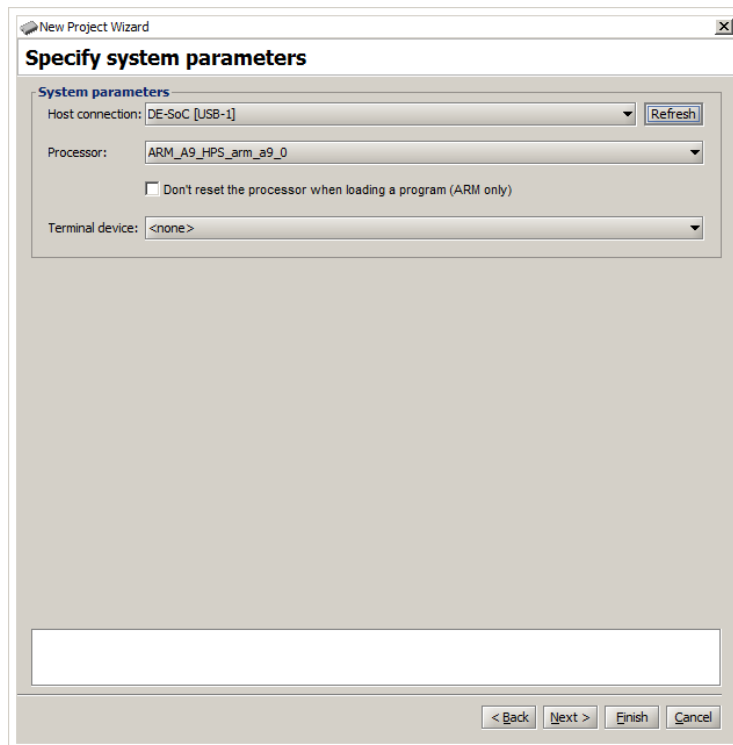Figure 5: Source files used by the application program.
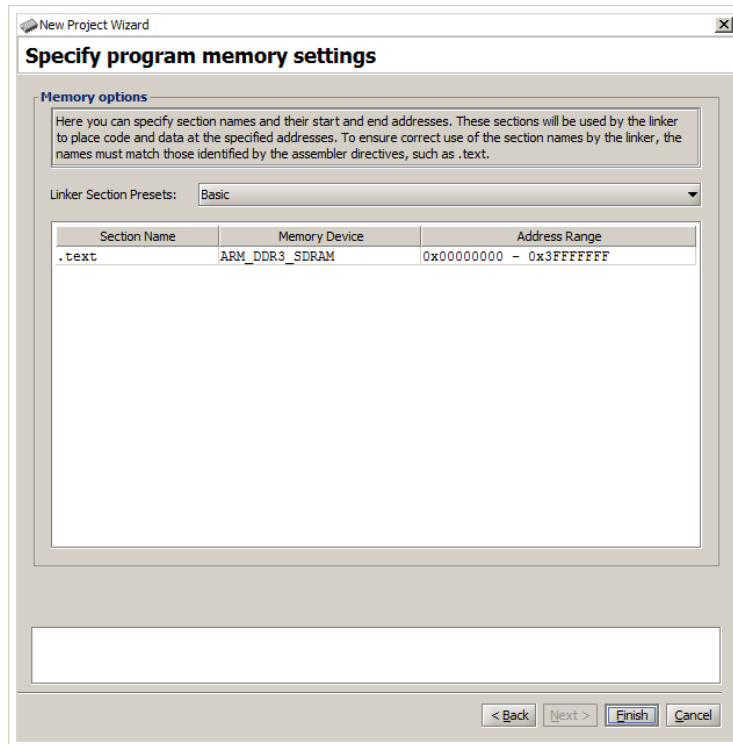

Figure 6: Specify the system parameters.

Figure 7: Specify the program memory settings.
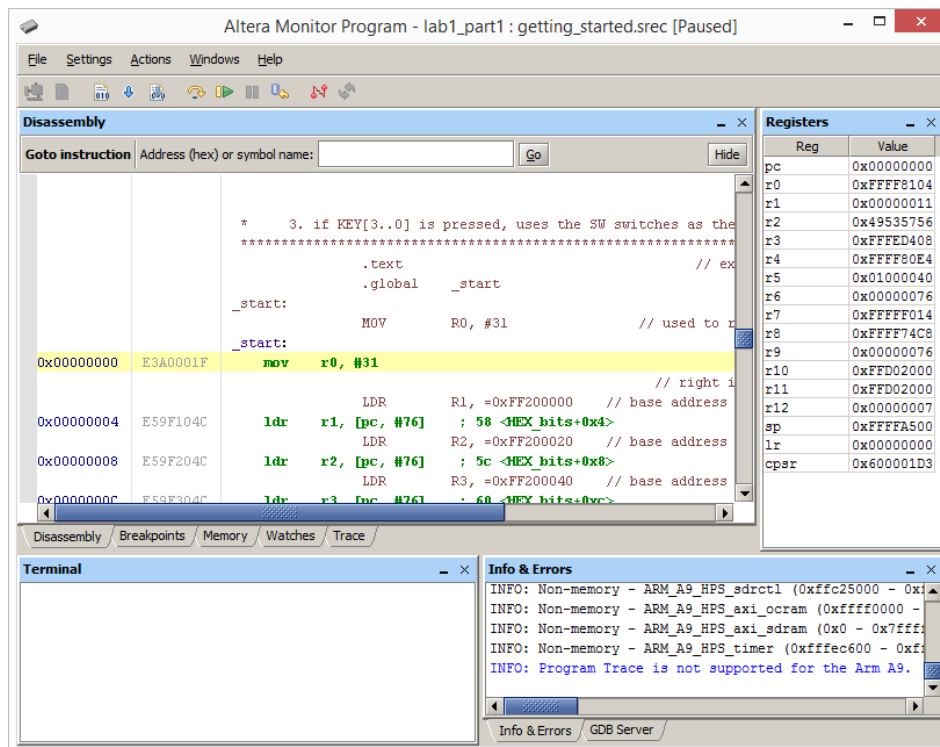

Figure 8: The monitor window showing the loaded sample program.

Part II

Now, we will explore some features of the Monitor Program by using a simple application program written in the ARM assembly language. Consider the program in Figure 9, which finds the largest number in a list of 32-bit integers that is stored in the memory.

```
/* Program that finds the largest number in a list of integers */
            .text
            .global     _start
_start:
            LDR     R4, =RESULT     // R4 points to result location
            LDR     R2, [R4, #4]    // R2 holds the number of elements in the list
            ADD     R3, R4, #8      // R3 points to the first number
            LDR     R0, [R3]        // R0 holds the largest number so far

LOOP:       SUBS    R2, R2, #1      // decrement the loop counter
            BEQ     DONE
            ADD     R3, R3, #4
            LDR     R1, [R3]        // get the next number
            CMP     R0, R1          // check if larger number found
            BGE     LOOP
            MOV     R0, R1          // update the largerst 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 9: Assembly-language program that finds the largest number.

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.

Make sure that you understand the program in Figure 9 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 a new folder for this part of the exercise, with a name such as *Part2*. Create a file named *part2.s* and enter the code from Figure 9 into this file. Use the Monitor Program to create a new project in this folder; we have chosen the project name *part2*. When you reach the window in Figure 4 choose Assembly Program but do not select a sample program. Click Next.

2. Upon reaching the window in Figure 5, you have to specify the source code file for your program. Click Add and in the pop-up box that appears indicate the desired file name, *part2.s*. Click Next to get to the window in Figure 6. Again click Next to get to the window in Figure 7. Notice that the DDR3_SDRAM is selected as the memory device. Your program will be loaded starting at address 0 in this memory. Click Finish.

3. Compile and load the program.

4. The Monitor Program will display a disassembled view of the machine code loaded in the memory, as indicated in Figure 10. Note that the pseudo instruction **LDR R4, =RESULT** from your source code has been implemented by using the instruction, **LDR R4, [PC, #84]**. This instruction loads the 32-bit address of the label RESULT into register R4. After this instruction has been executed, the content of register R4 will be 0x00000038, because this is the address in the memory of the label RESULT.

   The **LDR R4, [PC, #84]** instruction loads the required 32-bit constant 0x00000038 from the *literal pool*, where this value has been placed by the assembler/linker. The address in the literal pool is calculated as [pc] + 8 + *OFFSET*, where *OFFSET*=0x54 in this case (84 in decimal). The reason that 8 is added has to do with the way that the ARM processor automatically increments its program counter register as instructions are being executed. Hence, the location in the literal pool where the processor gets the constant 0x00000038 in this case is 0 + 8 + 0x54 = 0x0000005C.

   You can use the Monitor Program Disassembly tab (or the Memory tab) to verify that the constant 0x00000038 is in the literal pool at the address 0x0000005C. Figure 11 shows the literal-pool constant in the Disassembly window. You can single-step the instruction **LDR R4, =RESULT** in the Monitor Program to verify that it sets R4 to the value 0x00000038.

5. Execute the program. When the code is running, you will not be able to see any changes (such as the contents of registers or memory locations) in the Monitor Program windows, because the Monitor Program cannot communicate with the ARM processor while code is being executed. But, if you pause the program then the Monitor Program windows will be updated. Pause the program using the icon ▯▯ and observe that the processor stops within the endless loop **END: B END**. 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. As discussed above, the address of the label RESULT for this program is 0x00000038. Use the Monitor Program's Memory tab, as illustrated in Figure 12, to verify that the resulting value 8 is stored in the correct location.

6. You can return control of the program to the start by clicking on the icon 🔖, or by selecting Actions > Restart. Do this and then single-step through the program by clicking on the icon 🔁. Watch how the instructions change the data in the processor's registers.
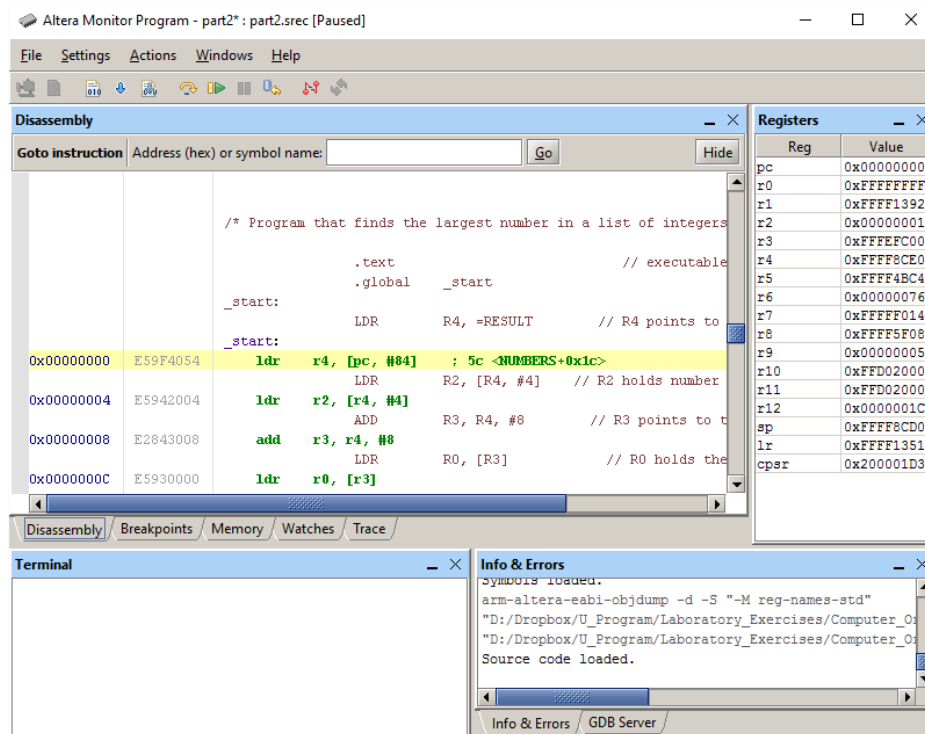
Figure 10: The disassembled view of the program in Figure 9.

7. Double-click on the pc register in the Monitor Program and then set the program counter to 0. Note that this action has the same effect as clicking on the restart icon ![icon].

8. Now set a breakpoint at address 0x0000002C (by clicking on the gray bar to the left of this address), so that the program will automatically stop executing whenever the branch instruction at this location is about to be executed. Restart the program and run it again. Observe the contents of register R0 each time the breakpoint is reached.

Part III

Implement the task in Part II by modifying the program in Figure 9 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 13.

Create a new folder and a new Monitor Program project to compile and download your program. Run your program to verify its correctness.
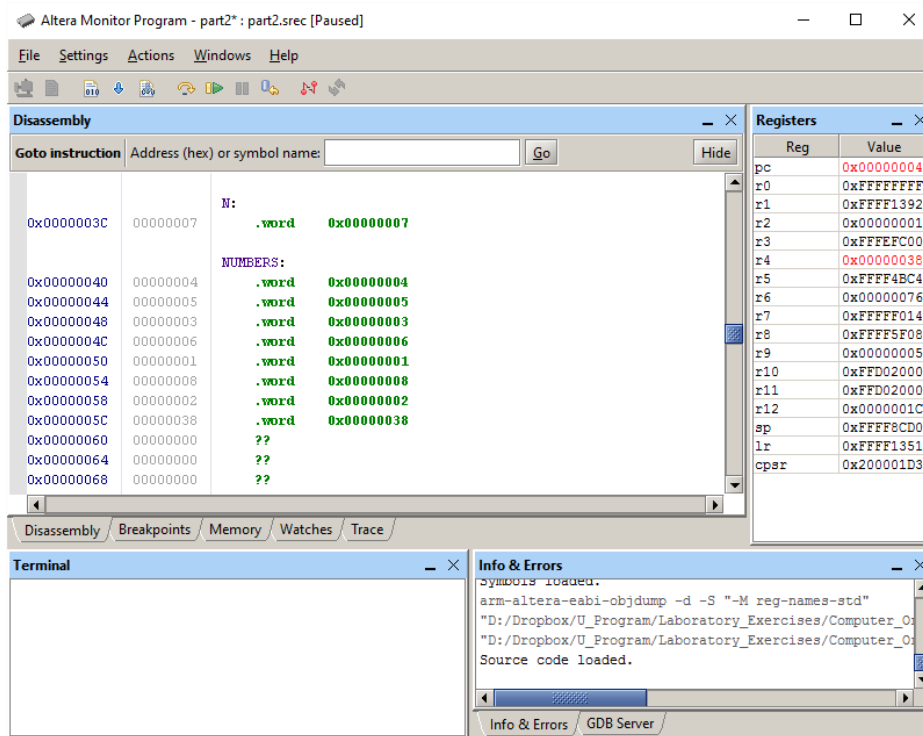
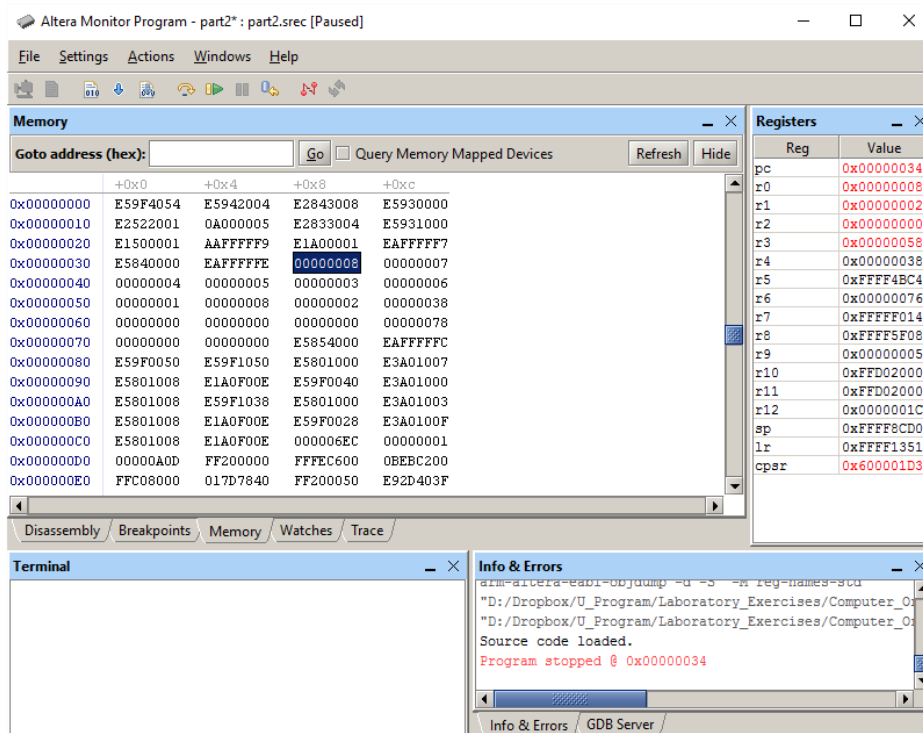Figure 11: The constant 0x00000038 in the literal pool at address 0x0000005C.

Figure 12: Displaying the result in the memory tab.

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

```
            .text
            .global     _start
_start:
            LDR         R4, =RESULT    // R4 points to result location
            LDR         R0, [R4, #4]   // R0 holds the number of elements in the list
            ADD         R1, R4, #8     // R1 points to the first number
            BL          LARGE
            STR         R0, [R4]       // R0 holds the subroutine return value

END:        B           END

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 13: Main program for Part III.

Part IV

The program shown in Figure 14 converts a binary number to 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 14 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.

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

```
                .text
                .global     _start
_start:
                LDR         R4, =N
                ADD         R5, R4, #4          // 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 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              // return quotient in R1 (remainder is in R0)
                BX          LR

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

                .end
```

Figure 14: A program that converts a binary number to two decimal digits.