Laboratory Exercise 1

Using a Nios II System

This is an introductory exercise using the Nios II processor. The exercise uses a pre-defined computer system for your DE-series board, which includes the Nios II processor and various peripheral devices. The system is implemented as a circuit that is downloaded into the FPGA device on an Intel DE-series board. This exercise illustrates how programs written in the Nios II assembly language can be executed on the DE-series boards. We will use the *Monitor Program* software to compile, load, and run the application programs.

For this exercise you have to know the Nios II processor architecture and its assembly language. Read the tutorial *Nios II Introduction*. You also have to become familiar with the Monitor Program; read the tutorial *Monitor Program Tutorial for Nios II*. Both tutorials are available in Intel FPGA 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 Monitor Program to set up a Nios II software development project. Perform the following:

1. Make sure that your DE-series board is powered and on.
2. Open the Monitor Program software, which leads to the window in Figure 1.

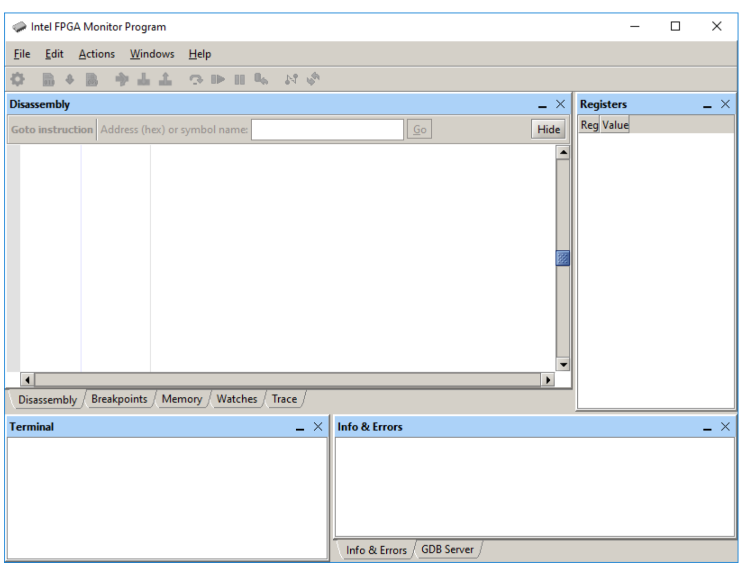

Figure 1: The Monitor Program window.

To develop Nios II 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 Nios II processor. Click Next, to get the window in Figure 3.

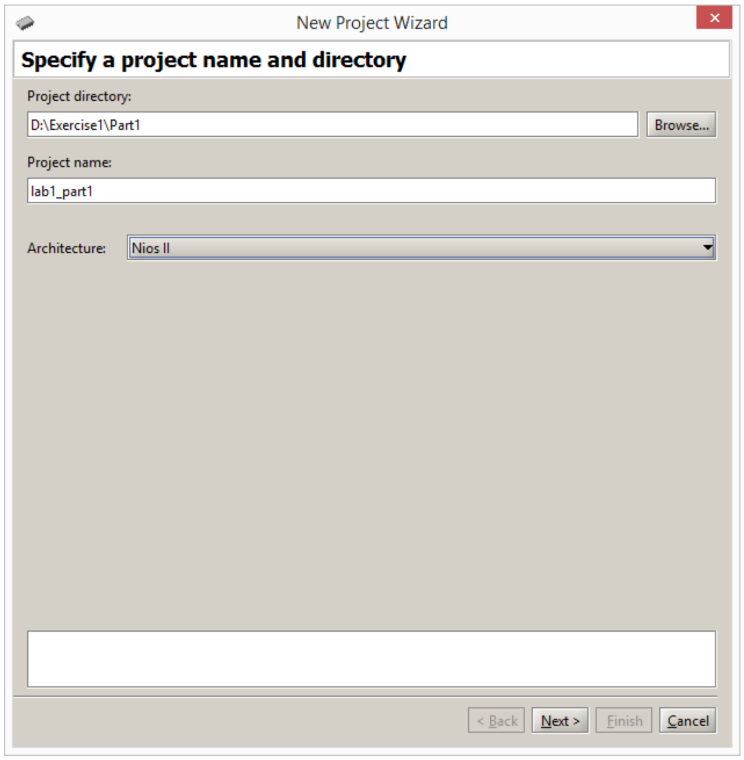

Figure 2: Specify the folder and the name of the project.

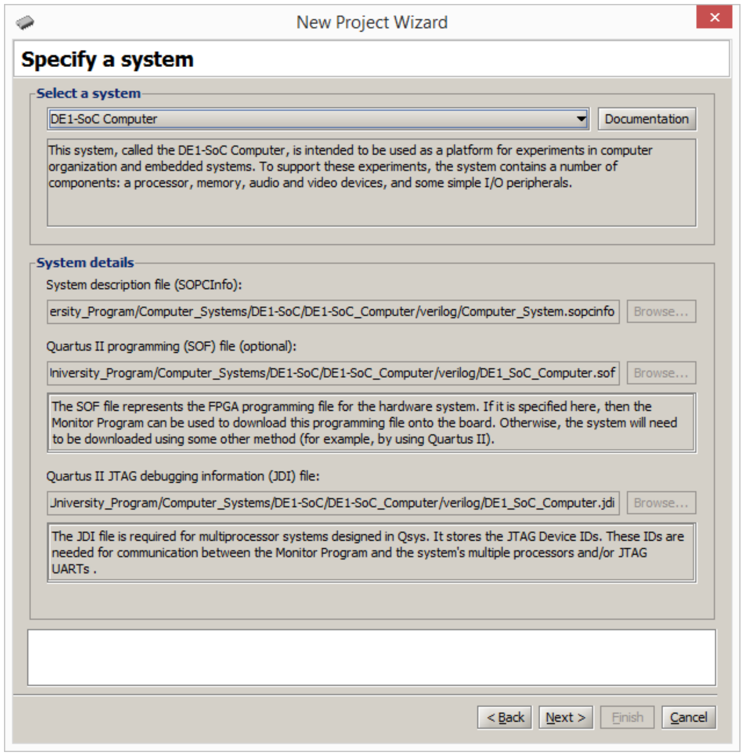

Figure 3: Specification of the system.

3. Now, you can select your own custom computer system (if you have one) or a pre-designed (by Intel) system. Choose the computer for your DE-series board listed in Table 1. The display in the window will now show where files that implement the pre-designed 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.

Board	Computer System
DE0-CV	DE0-CV Computer
DE1-SoC	DE1-SoC Computer
DE2-115	DE2-115 Computer
DE10-Lite	DE10-Lite Computer

Table 1: DE-series board computer systems

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 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.

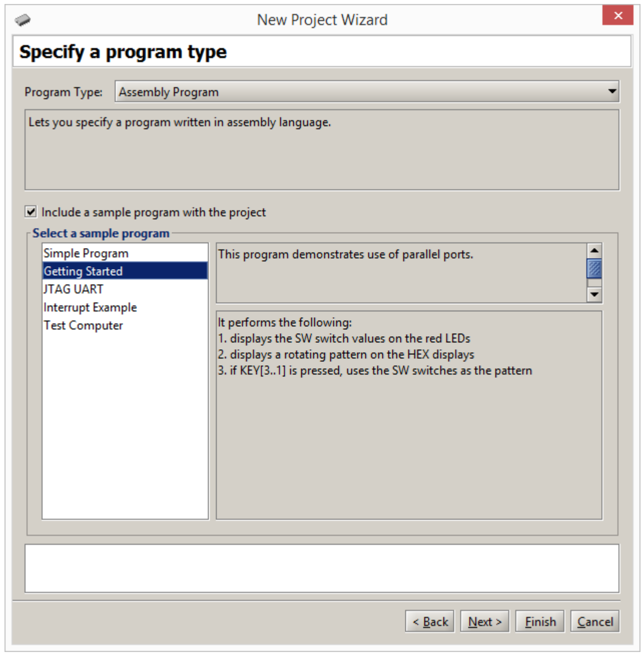

Figure 4: Selection of an application program.

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 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.

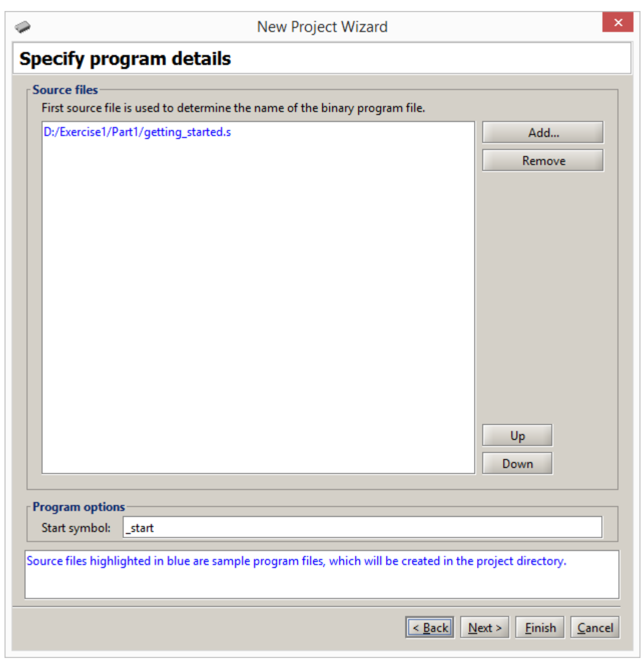

Figure 5: Source files used by the application program.

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 DE1-SoC board and the host computer. This is the name assigned to the USB-Blaster connection between the computer and the DE1-SoC board. For the other boards, the connection may be called *USB-Blaster [USB-0]*. Click Next.

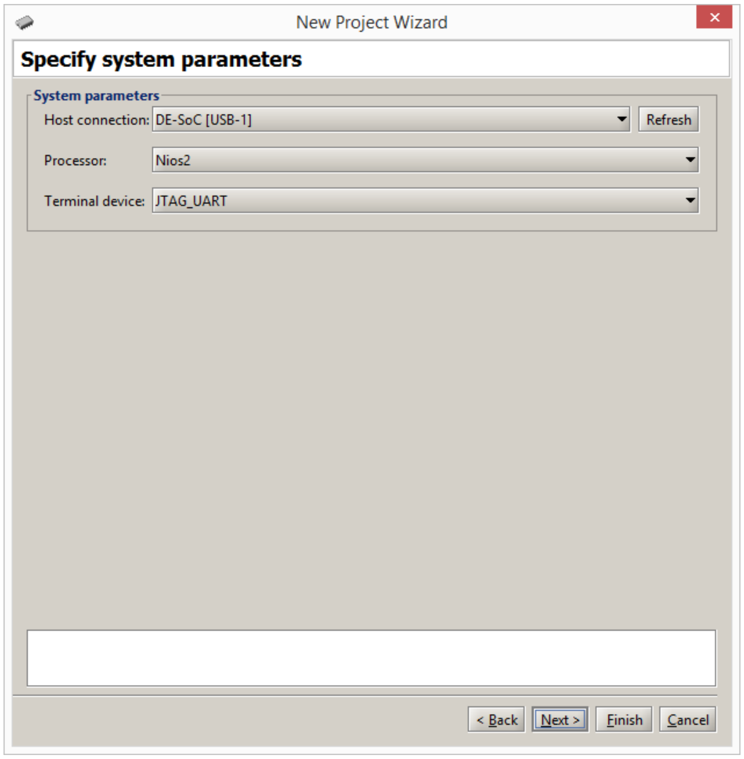

Figure 6: Specify the system parameters.

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 SDRAM memory in the Computer, starting at address 0. Click Finish to complete the specification of the new project.

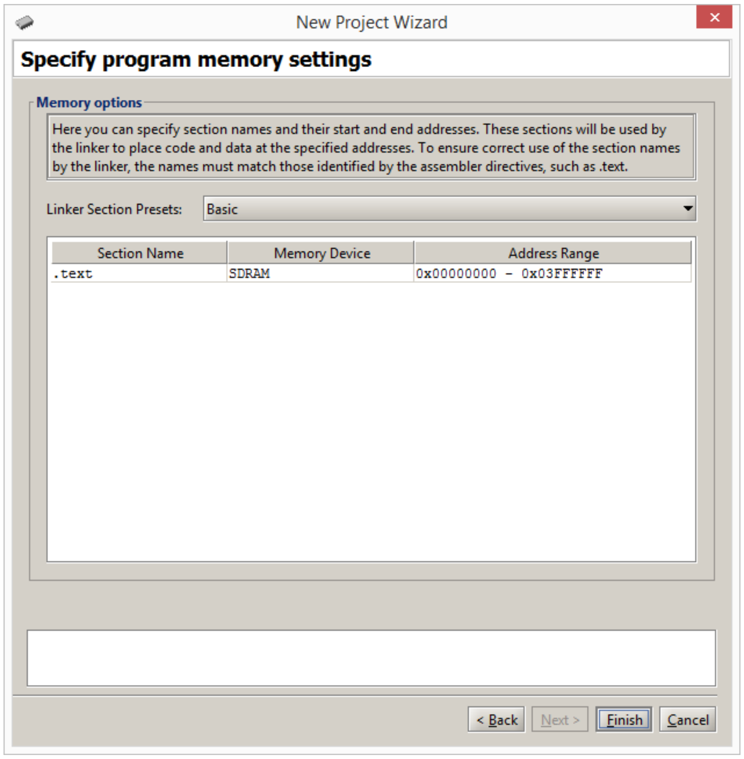

Figure 7: Specify the program memory settings.

8. Since you specified a new project, a pop-up box will appear asking if you want to download the system associated with this project onto your DE-series 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.)

9. Having downloaded the Computer into the FPGA on your DE-series 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 Nios II program and then 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 and 7-segment displays.

11. Pause the execution of the sample program by clicking on the icon , and disconnect from this session by clicking on the icon ,

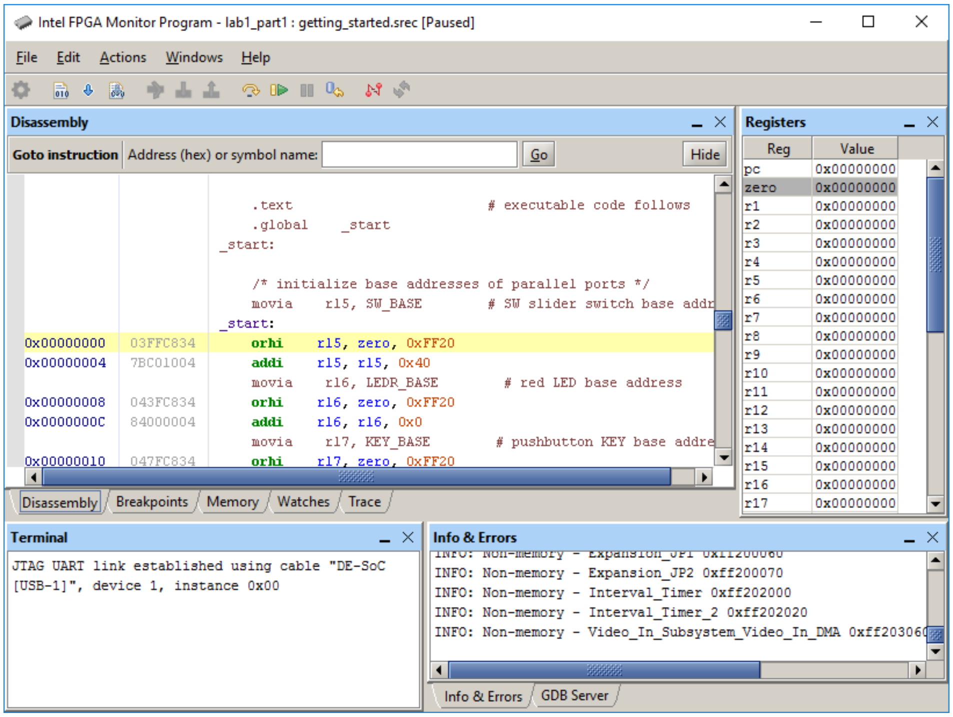

Figure 8: The monitor window showing the loaded sample program.

Part II

Now, we will explore some features of the Monitor Program by using a simple application program written in the Nios II 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:
            movia     r8, RESULT      # r8 points to result location
            ldw       r4, 4(r8)       # r4 is a counter, initialize it with N
            addi      r5, r8, 8       # r5 points to the first number
            ldw       r2, (r5)        # r2 holds the largest number found so far
LOOP:       subi      r4, r4, 1       # decrement the counter
            beq       r4, r0, DONE    # finished if r4 is equal to 0
            addi      r5, r5, 4       # increment the list pointer
            ldw       r6, (r5)        # get the next number
            bge       r2, r6, LOOP    # check if larger number found
            mov       r2, r6          # update the largest number found
            br        LOOP
DONE:       stw       r2, (r8)        # store the largest number into RESULT

STOP:       br        STOP            # remain here when done

RESULT:     .skip     4               # space for the largest number found
N:          .word     7               # number of entries in the list
NUMBERS:    .word     4, 5, 3, 6      # numbers in the list . . .
            .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 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 **movia r8, RESULT** from your source code has been implemented by using the two instructions **orhi r8, zero, 0x0** and **addi r8, r8, 0x38**. These instructions load the 32-bit address of the label RESULT, which is 0x00000038, into register r8.

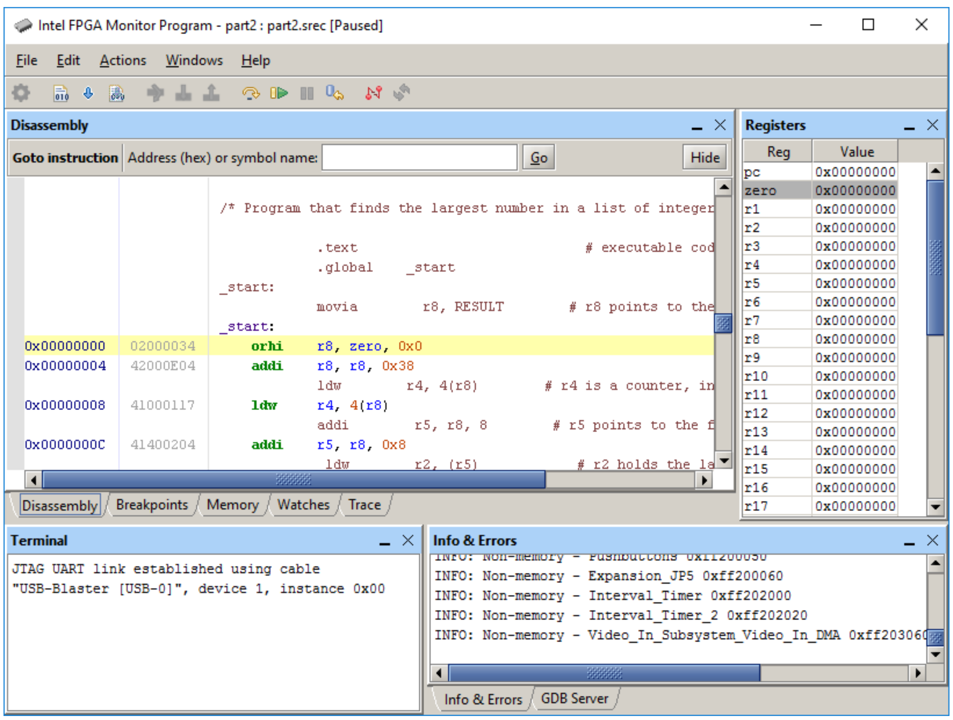

Figure 10: The disassembled view of the program in Figure 9.

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 computer 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 **STOP: br STOP**. Note that the largest number found in the sample list is 8 as indicated by the content of register r2. 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 11, 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.

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 .

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 r2 each time the breakpoint is reached.

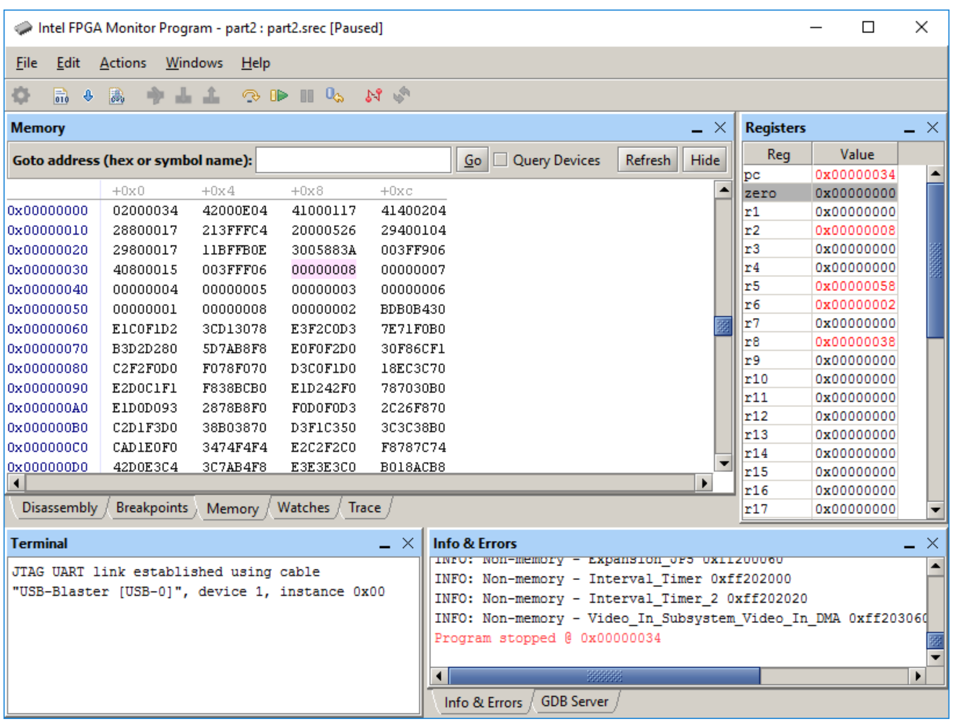

Figure 11: Displaying the result in the memory tab.

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 r4 and r5. The subroutine returns the value of the largest number to the calling program via register r2. A suitable main program is given in Figure 12.

Create a new folder and a new Monitor Program project to compile and download your program. Run your program to verify its correctness.

```
/* Program that finds the largest number in a list of integers */
            .text
            .global  _start
_start:
            movia    r8, RESULT      # r8 points to the result location
            ldw      r4, 4(r8)       # r4 holds number of elements in the list
            addi     r5, r8, 8       # r5 points to the start of the list
            call     LARGE
            stw      r2, (r8)        # r2 holds the subroutine return value

STOP:       br       STOP

LARGE:      . . .
            . . .
RESULT:     .skip    4               # space for the largest number found
N:          .word    7               # number of entries in the list
NUMBERS:    .word    4, 5, 3, 6      # the data . . .
            .word    1, 8, 2         . . .

            .end
```

Figure 12: Main program for Part III.

Part IV

The program shown in Figure 13 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 13 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 r5.

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:
            movia     r4, N
            addi      r8, r4, 4          # r8 points to the decimal digits storage location
            ldw       r4, (r4)           # r4 holds N

            call      DIVIDE             # parameter for DIVIDE is in r4
            /* Tens digit is now in r3, ones digit is in r2 */
            stb       r3, 1(r8)
            stb       r2, (r8)

END:        br        END

/* Subroutine to perform the integer division r4 / 10.
 * Returns: quotient in r3, and remainder in r2
 */
DIVIDE:     mov       r2, r4             # r2 will be the remainder
            movi      r5, 10             # divisor
            movi      r3, 0              # r3 will be the quotient
CONT:       blt       r2, r5, DIV_END
            sub       r2, r2, r5         # subtract the divisor, and ...
            addi      r3, r3, 1          # increment the quotient
            br        CONT
DIV_END:    ret                          # quotient is in r3, remainder in r2

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

            .end
```

Figure 13: A program that converts a binary number to two decimal digits.