

## 1 Introduction

This tutorial presents an introduction to the Intel FPGA Monitor Program, which can be used to compile, assemble, download and debug programs for the Intel Nios<sup>®</sup> II processor. The tutorial is intended for a user who wishes to use a Nios II based system on an Intel Development and Education board. It gives step-by-step instructions that illustrate the features of the Monitor Program.

The Monitor Program is a software application which runs on a host PC, and communicates with a Nios II hardware system on an FPGA board. It can be used to compile/assemble a Nios II software application, download the application onto the FPGA board, and then debug the running application. It provides features that allow a user to:

- Set up a Nios II project that specifies a desired hardware system and software program
- Download the hardware system onto an FPGA board
- Compile software programs, specified in assembly language or C, and download the resulting machine code into the hardware system
- Display the Nios II machine code stored in memory
- Run the Nios II processor, either continuously or by single-stepping instructions
- Examine and modify the contents of processor registers
- Examine and modify the contents of memory, as well as memory-mapped registers in I/O devices
- Set breakpoints that stop the execution of a program at a specified address, or when certain conditions are met
- Develop Nios II programs that make use of device driver functions provided through Intel's Hardware Abstraction Layer (HAL)

The process of downloading and debugging a Nios II program requires an FPGA board to implement the Nios II hardware system. In this tutorial it is assumed that the reader has access to the Intel DE1-SoC Development and Education board, connected to a computer that has Quartus Prime and Nios II Embedded Design Suite (EDS) software installed. Although a reader who does not have access to an FPGA board will not be able to execute the Monitor Program commands described in the tutorial, it should still be possible to follow the discussion.

## 1.1 Who should use the Monitor Program

The Monitor Program is intended to be used in an educational environment by professors and students. It is not intended for commercial use.

## 2 Installing the Monitor Program

The Monitor Program is released as part of the University Program Design Suite (UPDS). Before the UPDS can be installed on a computer, it is necessary to first install Intel's Quartus® Prime CAD software (either the Lite, Standard, or Pro edition) and the Nios II Embedded Design Suite (EDS). A particular release of the Monitor Program can be used only with a corresponding version of the Quartus Prime software and Nios II EDS. This software can be obtained from the on Intel's website at *university.altera.com*.

Once the Quartus Prime software and Nios II EDS are installed, the UPDS can be installed.

Note that if the Quartus Prime software is re-installed at some future time, then it will be necessary to re-install the Monitor Program at that time.

### 2.1 Using a Windows Operating System

When using a Windows operating system, perform the following:


1. Install the Intel UPDS from the University Program section of Intel's website. It can be found by going to *university.altera.com* and choosing *MATERIALS* followed by *Software* and then *Intel FPGA Monitor Program*. Specify the installed version of Quartus Prime software. Then click on the *EXE* item in the displayed table, which links to an installation program called *altera\_upds\_setup.exe*. When prompted to **Run** or **Save** this file, select **Run**.
2. The first screen of the installer is shown in Figure 1. Click on the **Next** button.
3. The installer will display the License Agreement; if you accept the terms of this agreement, then click **I Agree** to continue.
4. The installer now displays the root directory where the FPGA University Program Design Suite will be installed. Click **Next**.
5. The next screen, shown in Figure 2, lists the components that will be installed, which include the Monitor Program software and University Program IP Cores. These IP Cores provide a number of I/O device circuits that can be used in hardware systems to be implemented on the FPGA board.
6. The installer is now ready to begin copying files. Click **Install** to proceed and then click **Next** after the installation has been completed. If you answered **Yes** when prompted about placing a shortcut on your Windows Desktop, then an icon  is provided on the Desktop that can be used to start the Monitor Program.



Figure 1. Intel UPDS Setup Program.

7. Now, the FPGA University Program Design Suite is successfully installed on your computer, so click **Finish** to finish the installation.
8. Should an error occur during the installation procedure, a pop-up window will suggest the appropriate action. Possible errors include:
  - Quartus Prime software is not installed or the Quartus Prime version is incorrect.
  - Nios II EDS software is not installed or the version is incorrect.

## 2.2 Using a Linux\* Operating System

When using a Linux\* operating system, perform the following:

1. Install the UPDS from the University Program section of Intel's website. It can be found by going to *university.altera.com* and choosing *SUPPORT* followed by *Training* and *University Program*. Then, select *software tools > Intel FPGA Monitor Program*. Specify the installed version of Quartus Prime software. Then click on the *TAR* item in the displayed table, which links to an installation tarball called *altera\_upds\_setup.tar*. Save this file to a directory of your choosing.

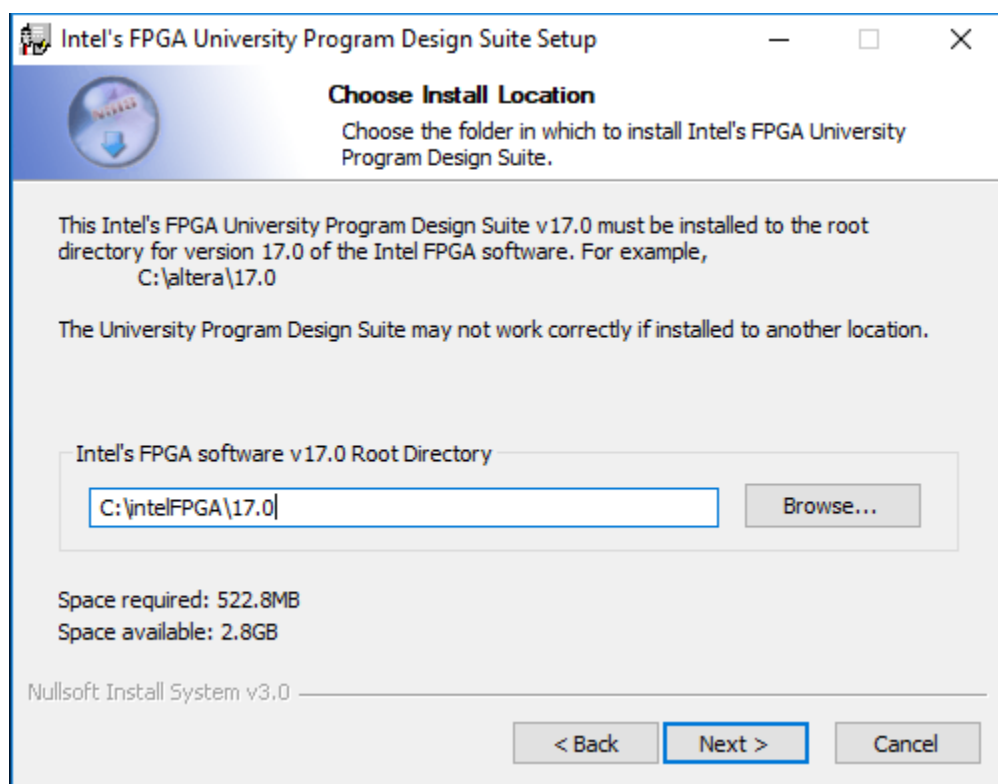


Figure 2. The components that will be installed.

2. Using a console, navigate to the directory to which the file was saved. Extract the contents of *altera\_upds\_setup.tar* using the following command: **tar -xf altera\_upds\_setup.tar**.
3. Among the extracted files is a shell script named *install\_altera\_upds* which will be used to install the UPDS. Ensure that the script is executable by using the following command: **chmod +x install\_altera\_upds**.
4. Run the installation script with superuser privileges by using the following command: **sudo ./install\_altera\_upds**.
5. Follow the instructions displayed by the script to complete the installation.

### 3 Main Features of the Monitor Program

Each Nios II software application that is developed with the Intel FPGA Monitor Program is called a *project*. The Monitor Program works on one project at a time and keeps all information for that project in a single directory in the file system. The first step is to create a directory to hold the project's files. To store the design files for this tutorial, we will use a directory named *Monitor\_Tutorial*. The running example for this tutorial is a simple assembly-language program that controls some lights on a DE1-SoC board.

If you are using a Windows\* operating system, then start the Monitor Program software either by double-clicking its icon on the Windows Desktop or by accessing the program in the Windows Start menu under Intel > University Program > Intel FPGA Monitor Program. You should see a display similar to the one in Figure 3.

If you are using a Linux operating system, then start the Monitor Program software by running the *altera-monitor-program* shell script located in *<path to Intel software>/University Program/Monitor Program/bin*. You should see a display similar to the one in Figure 3.

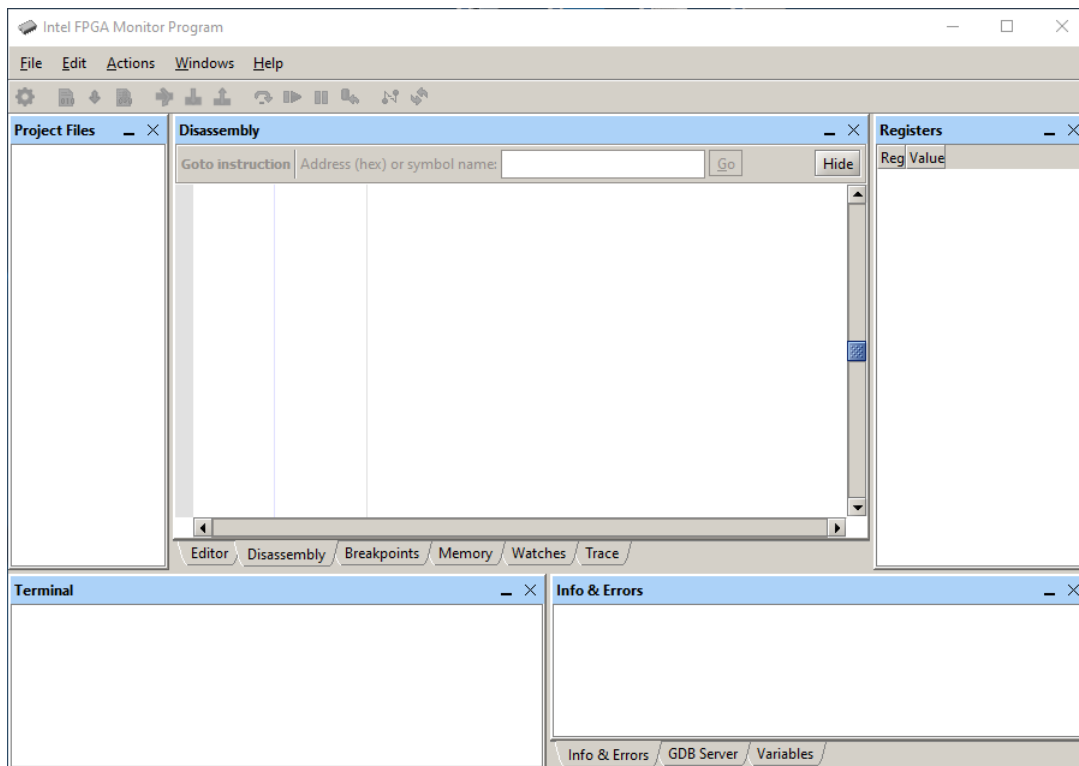


Figure 3. The main Monitor Program display.

This display consists of several windows that provide access to all of the features of the Monitor Program, which the user selects with the computer mouse. Most of the commands provided by the Monitor Program can be accessed by using a set of menus that are located below the title bar. For example, in Figure 3 clicking the left mouse button on the File command opens the menu shown in Figure 4. Clicking the left mouse button on the entry Exit exits from the Monitor Program. In most cases, whenever the mouse is used to select something, the left button is used. Hence we will not normally specify which button to press.

For some commands it is necessary to access two or more menus in sequence. We use the convention **Menu1 > Menu2 > Item** to indicate that to select the desired command the user should first click the mouse button on **Menu1**, then within this menu click on **Menu2**, and then within **Menu2** click on **Item**. For example, **File > Exit** uses the mouse to exit from the Monitor Program. Many commands can alternatively be invoked by clicking on an icon displayed in the Monitor Program window. To see the command associated with an icon, position the mouse over the icon and a tooltip will appear that displays the command name.

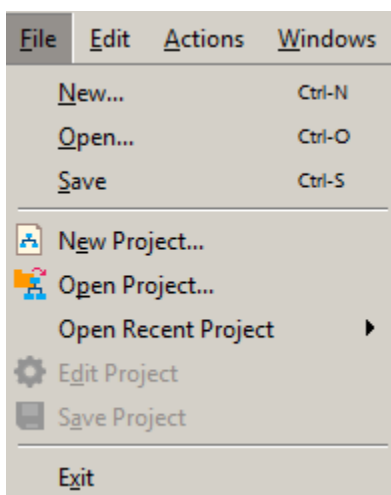


Figure 4. An example of the File menu.

It is possible to modify the organization of the Monitor Program display in Figure 3 in many ways. Section 9 shows how to move, resize, close, and open windows within the Monitor Program display.

### 3.1 Creating a Project

To start working on a Nios II software application we first have to create a new project, as follows:

1. Select **File > New Project** to open the *New Project Wizard*, which leads to the screen in Figure 5. The Wizard presents a sequence of screens for defining a new project. Each screen includes a number of dialogs, as well as a message area at the bottom of the window. The message area is used to display error and information messages associated with the dialogs in the window. Double-clicking the mouse on an error message moves the cursor into the dialog box that contains the source of the error.

In Figure 5 we have specified the file system directory *D:\Monitor\_Tutorial* and the project name *Monitor\_Tutorial*. For simplicity, we have used a project name that matches the directory name, but this is not required.

If the file system directory specified for the project does not already exist, a message will be displayed indicating that this new directory will be created. To select an existing directory by browsing through the file system, click on the **BROWSE** button. Note that a given directory may contain at most one project.

The Monitor Program can be used with either an ARM\* based system or a Nios II-based system. The choice of a processor is made in the window in Figure 5 in the box labeled Architecture. We have chosen the Nios II architecture for this tutorial.

2. Click **Next** to advance to the window shown in Figure 6, which is used to specify a particular system. A hardware system to be implemented on the FPGA board is usually generated by using Quartus's Platform Designer tool. Information about creating systems using Platform Designer can be found in the *Introduction to the Intel Platform Designer Tool* tutorial, which is available in the University Program section of Intel's website.

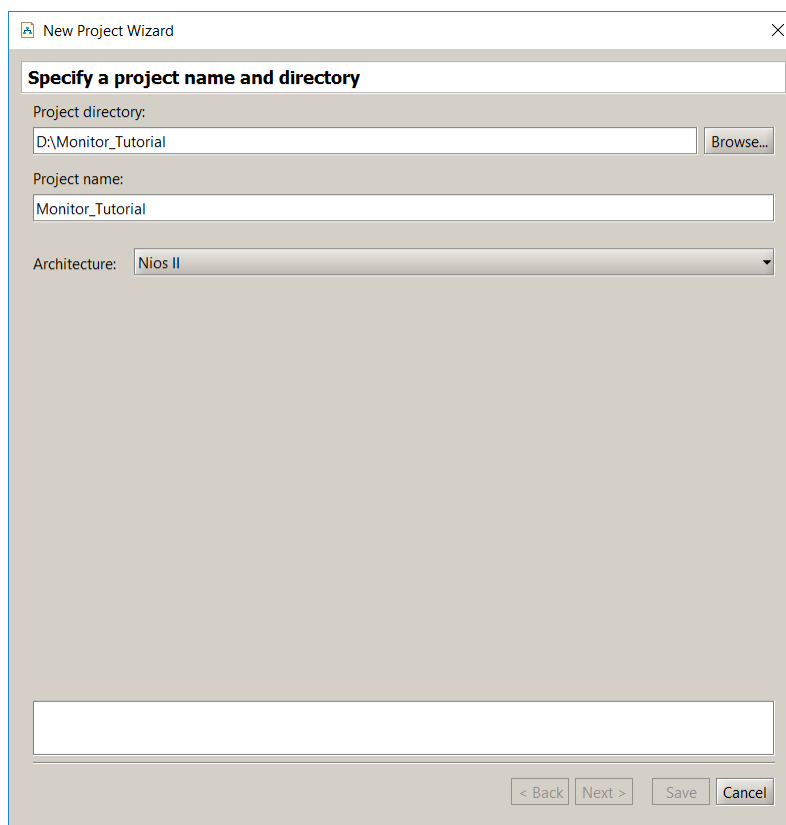


Figure 5. Specifying the project directory and name.

A system designed and generated by using Quartus Prime and its Platform Designer tool is described in *SOPCInfo* and *SOFF* files. The former gives a high-level description of the system. The latter represents the FPGA circuit that implements the designed system; this file can be downloaded into the FPGA chip on the board that is being used.

Any system which contains a *Hard Processor System* (HPS) component must also specify the preloader to be run immediately following the circuit being downloaded. This preloader is used to configure the components within the HPS with the setting required for the specific board.

The drop-down list on the **Select a system** pane can be used to choose the system to be used in the project. The Monitor Program includes a number of prebuilt computer systems for Intel's Development and Education boards. Since in this tutorial we assume that the user has access to a DE1-SoC board, we will use a system called the DE1-SoC Computer. This computer includes a number of interfaces to input/output devices implemented in the FPGA fabric of the chip. It was created using Quartus Prime and its Platform Designer tool. It is represented by *.sopcinfo* and *.sof* files which are automatically included when this computer is selected. The DE1-SoC preloader is also automatically selected.

The user may also design and implement a custom system. If the custom system is selected, then the user must manually specify the *.sopcinfo* and *.sof* files that define the required system in the **System details** pane. If the custom system contains an HPS, the user must select their board from the preloader dropdown menu.

In the top right corner of Figure 6 there is a Documentation button. Clicking on this button opens a user guide that provides all information needed for developing Nios II programs for the DE1-SoC Computer, such as the memory map for addressing all of the I/O devices in the system. This file can also be accessed at a later time by using the command Settings > System Settings and then clicking on the Documentation button.

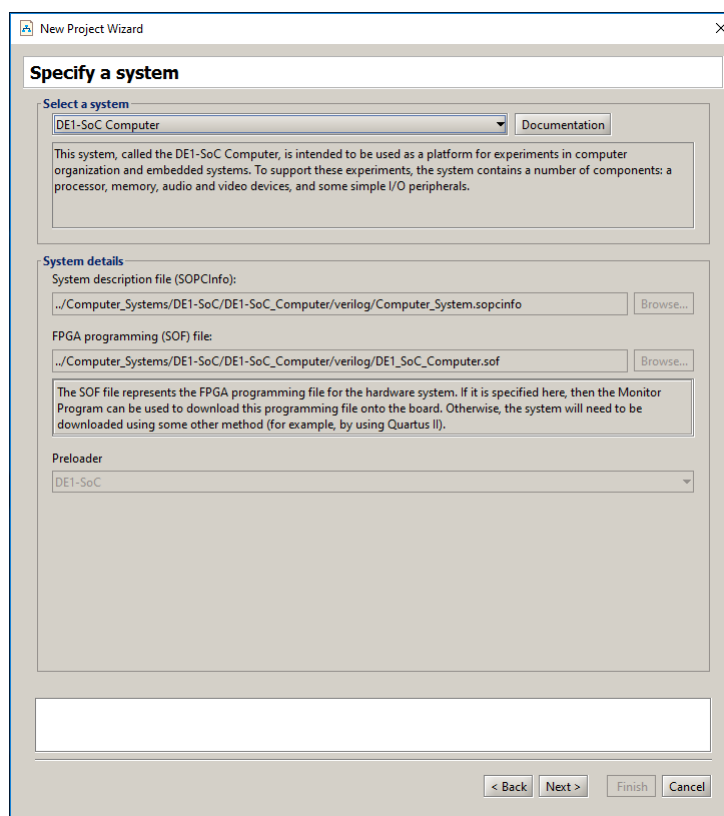


Figure 6. Specifying the desired hardware system.

3. Click **Next** to advance to the screen in Figure 7, which is used to specify the program source files that are associated with the project. The **Program Type** drop-down list can be used to select one of the following program types:
  - **Assembly Program:** allows the Monitor Program to be used with Nios II assembly-language code
  - **C Program:** allows the Monitor Program to be used with C code
  - **Program with Device Driver Support:** this is an advanced option, which can be used to develop programs that make use of device driver software for the I/O devices in the Nios II hardware system. Programs that use this option can be written in either assembly, C, or C++ language (or any combination). More information about writing programs that use device drivers can be found in Appendix B.
  - **ELF or SREC File:** allows the Monitor Program to be used with a precompiled program, in ELF or SREC format



- **No Program:** allows the Monitor Program to connect to the Nios II hardware system without first loading a program; this can be useful if one wants to examine the current state of some I/O devices without running an actual program.

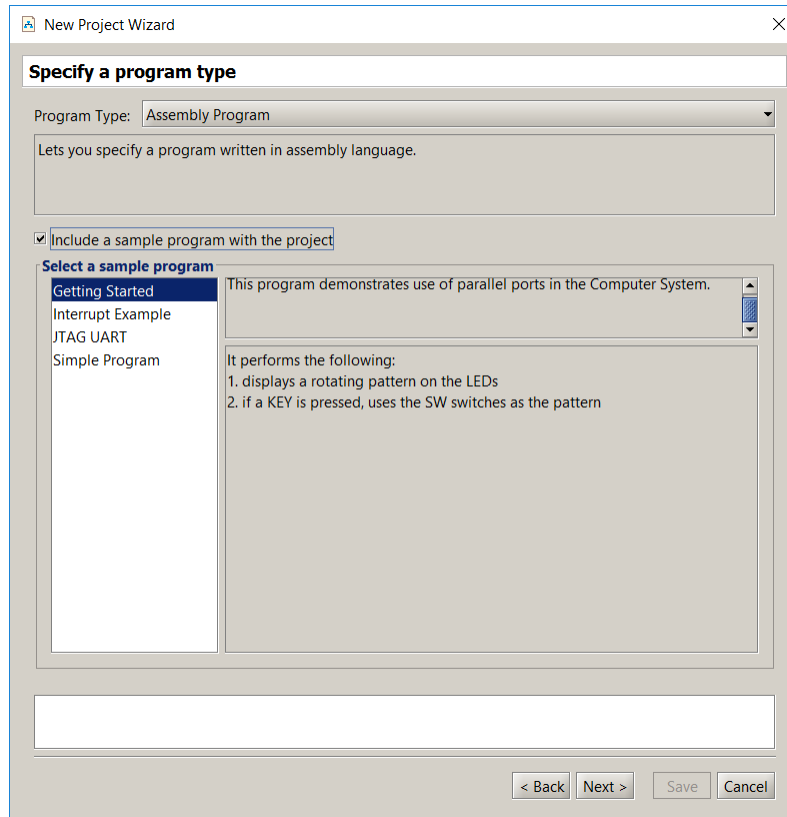


Figure 7. Selecting a program type and sample program.

For our example, set the program type to **Assembly Program**. When the DE1-SoC computer has been selected for the project, it is possible to click on the selection **Include a sample program with the project**.

As illustrated in [Figure 7](#), several sample assembly-language programs are available for this prebuilt computer. For our tutorial select the program named *Simple Program*. This is a very simple program which continuously reads the state of the slider switches on the DE1-SoC board and displays their state on the red LEDs. The source code for the program is:

```

.text
.equ    LEDs, 0xFF200000
.equ    SWITCHES, 0xFF200040
.global _start
_start:
    movia    r2, LEDs          /* Address of red LEDs. */
    movia    r3, SWITCHES     /* Address of switches. */
LOOP:   ldwio    r4, (r3)      /* Read the state of switches. */
        stwio    r4, (r2)      /* Display the state on LEDs. */
        br      LOOP
.end

```

Click Next to advance to the screen in Figure 8.

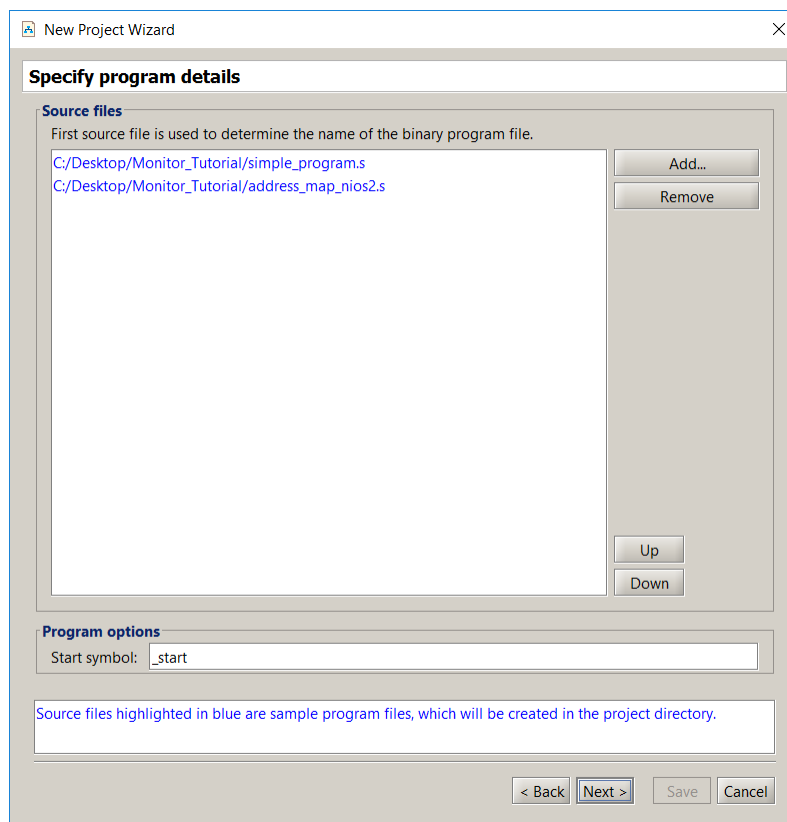


Figure 8. Specifying source code files.

When a sample program has been selected, the source code file(s) associated with this program is listed in the Source files box. In this case, the source file is named *simple\_program.s*; this file will be copied into the directory used for the project by the Monitor Program. If a sample program is not used, then it is necessary to click the Add button and browse to select the desired source file(s).

Figure 8 shows how it is possible to specify a label that identifies the first instruction to be executed. In the *simple\_program.s* file, this label is called `_start`, as indicated in the figure.

4. Click **Next** to advance to the window in Figure 9. This window is used to specify the connection to the FPGA board, the processor that should be used (some hardware systems may contain multiple processors), and the terminal device. The **Host connection** drop-down list contains the physical connection links (such as cables) that exist between the host computer and any FPGA boards connected to it. The processors available in the system are found in the **Processor** drop-down list, and all terminal devices connected to the selected processor are displayed in the **Terminal device** drop-down list. We discuss terminal devices in Section 6.

Accept the default choices that are displayed in Figure 9. If the Host Connection box is blank, make sure that the DE1-SoC board is connected to the host by a USB cable and that its power is turned on. Then, press the **Refresh** button and select the USB Blaster as the desired choice. For the DE1-SoC board the required choice is DE-SoC.

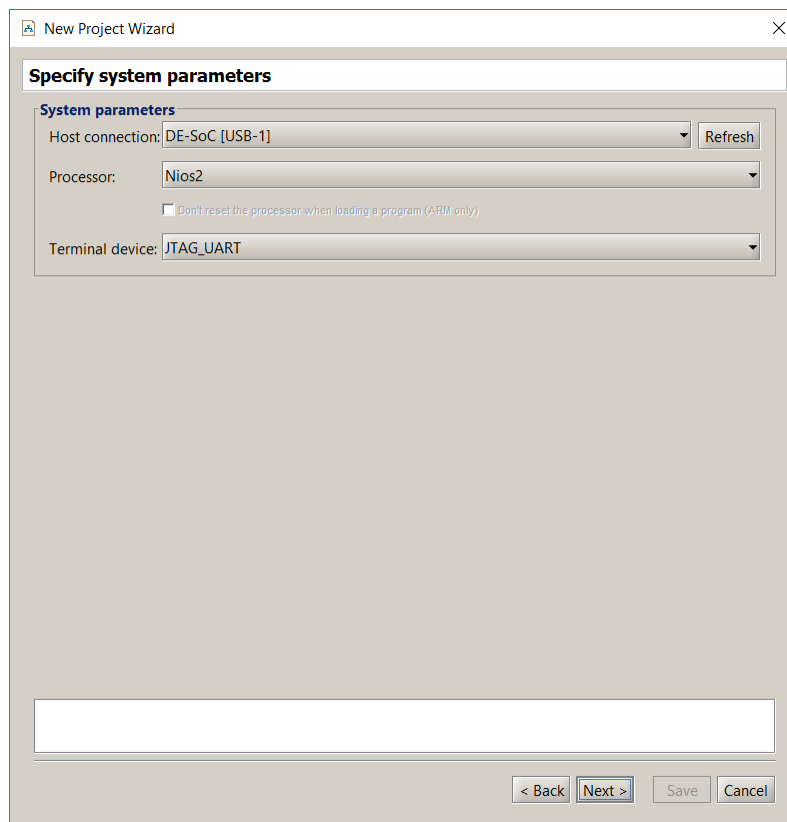


Figure 9. Specifying system settings.

5. Click **Next** to reach the final screen for creating the new project, shown in Figure 10. This screen is used to specify memory settings that are needed for compiling and linking the program.

There are two modes that can be selected. In the **Basic** mode, which does not provide explicitly for the use of interrupts, the application program starts at memory address `0x00000000` as shown in the figure. A more general alternative is to use the **Interrupts** mode, which is discussed in Section 8. The program in the *.text*

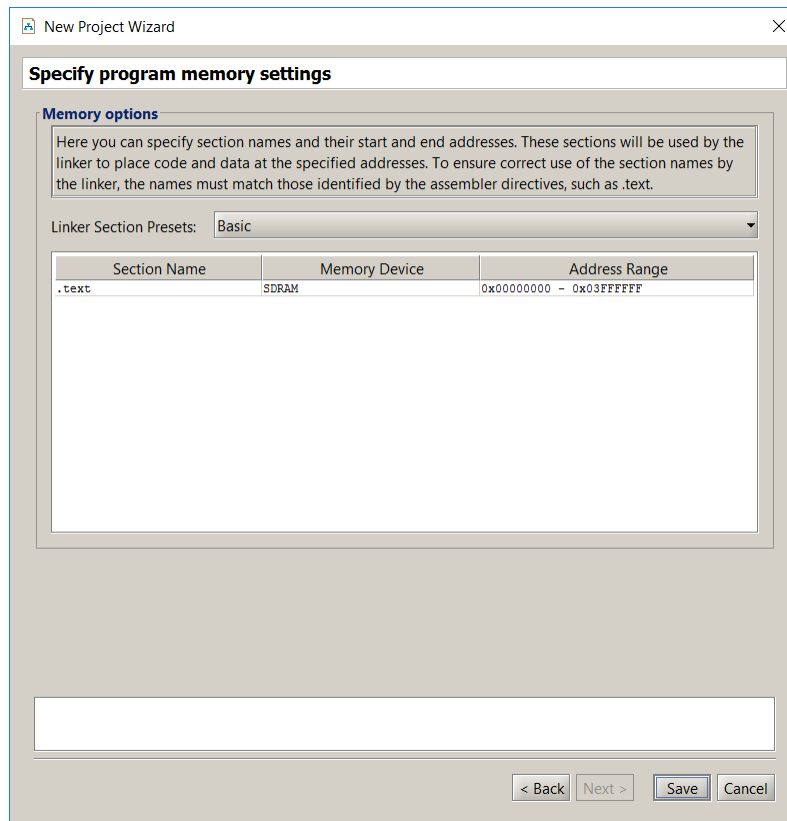


Figure 10. Specifying memory settings.

section can start at some other address, as may be specified by the user. To change the address, double-click on the `.text` entry and change the address in the pop-up box that appears.

Click **Finish** to complete the creation of the new project. At this point, the Monitor Program displays the prompt shown in Figure 11. Clicking **Yes** instructs the Monitor Program to download the hardware system associated with the project onto the FPGA board. It is also possible to download the system at a later time by using the Monitor Program command **Actions > Download System**. If the downloaded system contains more than one processor, the Monitor Program will prompt you to halt the processors other than the one being used for the current project. It is generally recommended to halt the other processors because they can execute without you knowing, resulting in unexpected behavior.

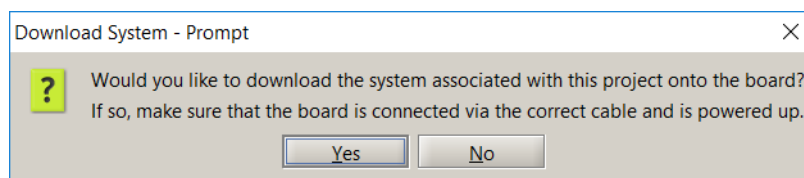





Figure 11. Download the hardware system.


### 3.1.1 Downloading a Nios II Hardware System

When downloading a Nios II hardware system onto an FPGA board, it is important to consider the type of license that is included in the hardware system for the processor. The Nios II processor uses a licensing scheme that provides two modes of operation: 1. an evaluation mode that allows the processor to be used with some restrictions when no license is present, and 2. a normal mode that allows unrestricted use when a license is present. Nios II licenses can be purchased from Intel, and are also available on a donated basis through the University Program. The prebuilt computer systems provided with the Monitor Program, such as the DE1-SoC Computer, include a Nios II processor that has a license. However, if other systems are being used with the Monitor Program, then it is possible that a license is not present, and the Nios II processor may be used in the evaluation mode. In this case it is necessary to use a different scheme, which is described in Section 5, to download the Nios II hardware system onto the FPGA board and activate the evaluation mode.

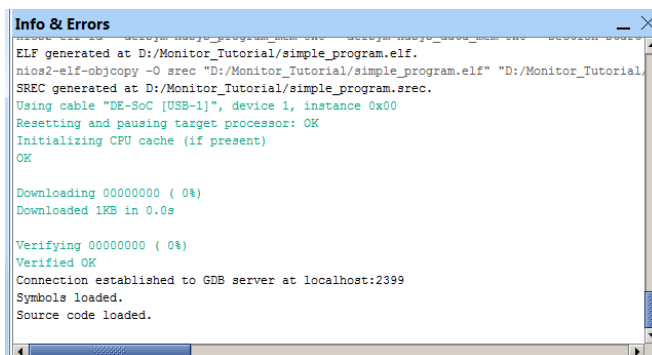
## 3.2 Compiling and Loading the Program

After successfully creating a project, its software files can be compiled/assembled and downloaded onto the FPGA board using the following commands:

- Actions > Compile menu item or  icon: compiles the source files into an ELF and SREC file. Build warnings and errors will show up in the Info & Errors window. The generated ELF and SREC files are placed in the project's directory.
- Actions > Load menu item or  icon: loads the compiled SREC file onto the board and begins a debugging session in the Monitor Program. Loading progress messages are displayed in the Info & Errors window.
- Actions > Compile & Load menu item or  icon: performs the operations of both compilation and loading.

Our example project has not yet been compiled, so it cannot be loaded (the Load option is disabled). Select the Actions > Compile & Load menu item or click the  icon to begin the compilation and loading process. Throughout the process, messages are displayed in the Info & Errors window. The messages should resemble those shown in Figure 12.

After successfully completing this step, the Monitor Program display should look similar to Figure 13. At this point the processor is halted at the first instruction of the program that has to be executed, which is highlighted in yellow shading. The main part of the display in Figure 13 is called the *Disassembly* window. It shows the machine code for the assembled program, as well as the addresses of memory locations in which the instructions are loaded. It also shows the assembly-language version of the assembled instructions.



```

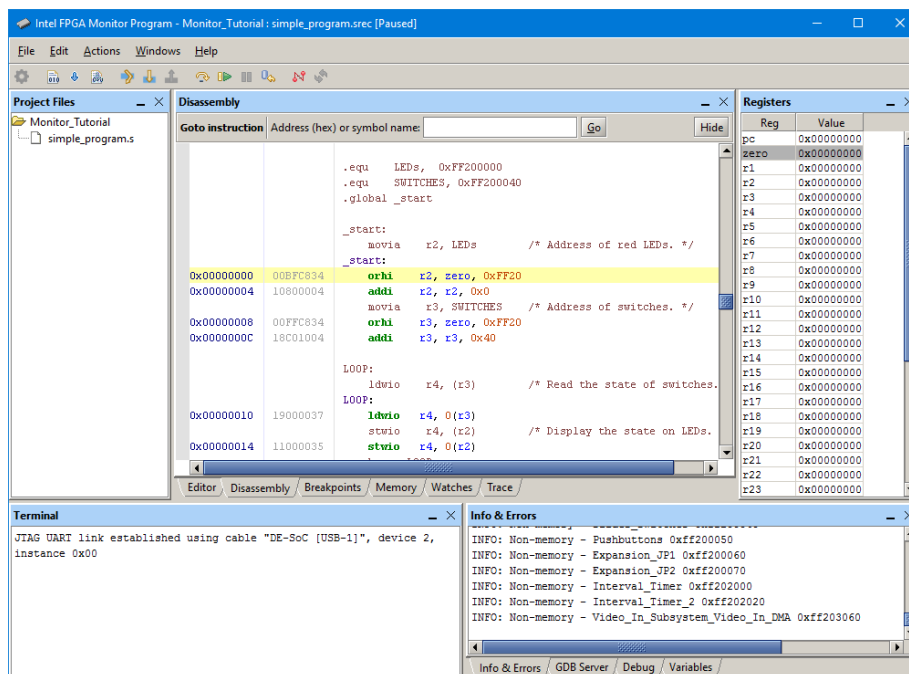
Info & Errors
-----
ELF generated at D:/Monitor_Tutorial/simple_program.elf.
nios2-elf-objcopy -O srec "D:/Monitor_Tutorial/simple_program.elf" "D:/Monitor_Tutorial/
SREC generated at D:/Monitor_Tutorial/simple_program.srec.
Using cable "DE-SoC [USB-1]", device 1, instance 0x00
Resetting and pausing target processor: OK
Initializing CPU cache (if present)
OK

Downloading 00000000 ( 0%)
Downloaded 1KB in 0.0s

Verifying 00000000 ( 0%)
Verified OK
Connection established to GDB server at localhost:2399
Symbols loaded.
Source code loaded.

```

Figure 12. Compilation and loading messages.



```

Intel FPGA Monitor Program - Monitor_Tutorial : simple_program.srec [Paused]
-----
Project Files
-----
Monitor_Tutorial
  simple_program.s

Disassembly
-----
Goto instruction Address (hex) or symbol name: [ ] Go Hide
-----
.equ LEDs, 0xFF200000
.equ SWITCHES, 0xFF200040
.global _start

_start:
  movia r2, LEDs /* Address of red LEDs. */
_start:
0x00000000 00BFC834 orhi r2, zero, 0xFF20
0x00000004 10800004 addi r2, r2, 0x0
0x00000008 00FFC834 movia r3, SWITCHES /* Address of switches. */
0x0000000C 18C01004 orhi r3, zero, 0xFF20
                                addi r3, r3, 0x40

LOOP:
  ldwio r4, (r3) /* Read the state of switches. */
LOOP:
0x00000010 19000037 ldwio r4, 0(r3)
                                stwio r4, (r2) /* Display the state on LEDs. */
0x00000014 11000035 stwio r4, 0(r2)

Registers
-----
Reg Value
pc 0x00000000
zero 0x00000000
r1 0x00000000
r2 0x00000000
r3 0x00000000
r4 0x00000000
r5 0x00000000
r6 0x00000000
r7 0x00000000
r8 0x00000000
r9 0x00000000
r10 0x00000000
r11 0x00000000
r12 0x00000000
r13 0x00000000
r14 0x00000000
r15 0x00000000
r16 0x00000000
r17 0x00000000
r18 0x00000000
r19 0x00000000
r20 0x00000000
r21 0x00000000
r22 0x00000000
r23 0x00000000

Terminal
-----
JTAG UART link established using cable "DE-SoC [USB-1]", device 2,
instance 0x00

Info & Errors
-----
INFO: Non-memory - Pushbuttons 0xff200050
INFO: Non-memory - Expansion_JP1 0xff200060
INFO: Non-memory - Expansion_JP2 0xff200070
INFO: Non-memory - Interval_Timer 0xff202000
INFO: Non-memory - Interval_Timer_2 0xff202020
INFO: Non-memory - Video_In_Subsystem_Video_In_DMA 0xff203060

```

Figure 13. The Monitor Program window after loading the program.

Most instructions in a Nios II assembly-language source program are assembled into directly-corresponding machine instructions in the object code that is loaded into the memory for execution. But, this is not the case with all instructions. The Nios II assembly language provides numerous *pseudo-instructions*, which are often replaced by actual instructions that look quite different but have the same effect when executed. For instance, the pseudo-instruction

```
movia r3, SWITCHES
```

loads into processor register r3 the memory address of the I/O data register that is connected to the slider switches on the board. The required address is 32 bits long. However, immediate operands in Nios II Load instructions can be at most 16 bits long. Therefore, as seen in Figure 13, the second `movia` instruction is replaced with two instructions. The instruction

```
orhi r3, zero, 0xFF20
```

places the immediate operand 0xFF20 into the high-order 16 bits of register r3 and leaves the low-order 16 bits equal to zero. The instruction

```
addi r3, r3, 0x40
```

changes the low-order 16 bits into 0x40, thus completing in register r3 the required address 0xFF200040. Information about Nios II instructions and pseudo-instructions can be found in the tutorial *Introduction to the Intel Nios II Soft Processor*, available in the University Program section of Intel's website.

### 3.2.1 Compilation Errors

During the process of developing software, it is likely that compilation errors will be encountered. Error messages from the Nios II assembler or from the C compiler are displayed in the Info & Errors window. To see an example of a compiler error message, edit the file `simple_program.s`, which is in the project's directory, and replace the Branch instruction mnemonic `br` with `b`. Recompile the project to see the error shown in Figure 14. The error message indicates the type of error and it gives the line number in the file where the error was detected. Fix the error, and then compile and load the program again.

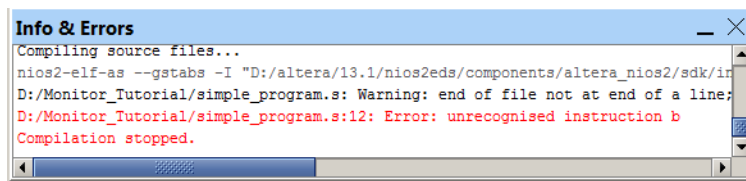


Figure 14. An example of a compiler error message.

## 3.3 Running the Program



As mentioned in the previous section, the processor is halted at the first instruction after the program has been loaded. To run the program, select the **Actions > Continue** menu item or click the  icon. The `simple_program` displays the current values of DE1-SoC board's slider switches on the red LEDs. The **Continue** command runs the program indefinitely. To force the program to halt, select the **Actions > Stop** command, or click the  icon. This command causes the processor to halt at the instruction to be executed next, and returns control to the Monitor Program.

Figure 15 shows an example of what the display may look like when the program is halted by using the **Stop** command. The display highlights in yellow the next program instruction to be executed, which is at address

0x00000014, and highlights in red the values in the processor registers that have changed since the last program stoppage. Other screens in the Monitor Program are also updated, which will be described in later parts of this tutorial.

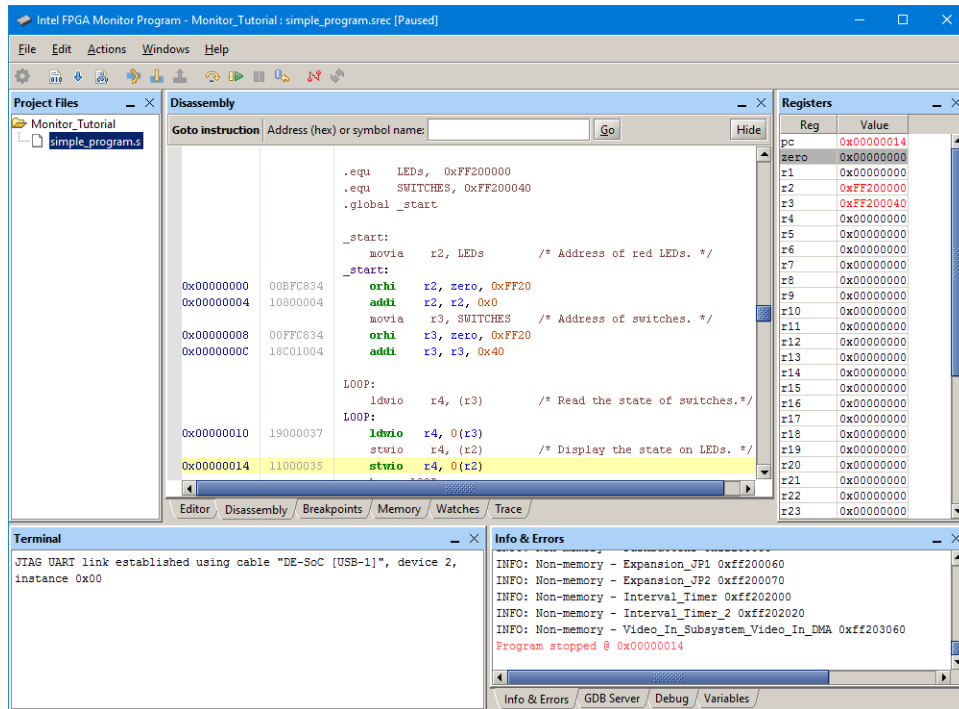


Figure 15. The Monitor Program display after the program has been stopped.

### 3.4 Using the Disassembly Window

In Figure 15, the Disassembly window shows the machine instructions for our program. The leftmost column in the window gives the memory addresses, the middle column displays the machine code at these addresses, and the rightmost column shows both the original source code for the instruction, in a brown color, and the disassembled view of the machine code that is stored in memory, in a green color.

The Disassembly window can be configured to display less information on the screen, such as not showing the assembly-language instructions or not showing the machine encoding of the instructions. These choices can be made by right-clicking on the Disassembly window and selecting the appropriate menu item, as indicated in Figure 16.

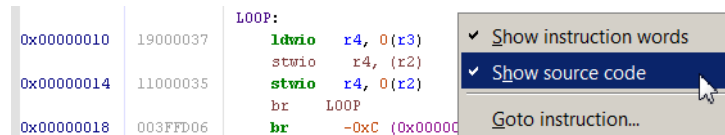




Figure 16. Display options for the Disassembly window.



Different parts of memory can be displayed by scrolling, using either the vertical scrollbar on the right side of the Disassembly window or a mouse scroll wheel. It is also possible to go to a different region of memory by using the **Goto** instruction panel at the top of the Disassembly window, or by using the command **Actions > Goto** instruction. The instruction address provided for the **Goto** command must be a multiple of four, because Nios II instructions are word-aligned.

### 3.5 Single Stepping Program Instructions

When debugging a program, it is often very useful to be able to single step through the program and observe the effect of executing each instruction. The Monitor Program has the ability to perform single-step operations. Each single step consists of executing a single machine instruction and then returning control to the Monitor Program. If the source code of the program being debugged is written in the C language, then each individual single step will still correspond to one assembly-language (machine) instruction generated from the C code.

The single-step operation is invoked by selecting the **Actions > Single step** menu item or by clicking on the  icon. The instruction that is executed by the processor is the one highlighted in yellow in the Disassembly window. Consider our *simple\_program* example. You can go to the first instruction of the program, which has the label *\_start*, by selecting **Actions > Restart** menu item or by clicking the  icon. If the program is running, it must first be halted before the restart command can be performed. The restart command loads into the Program Counter the address of the first instruction, thus causing the execution to start at this point in the program. Now, single step through the program and observe the displayed changes. Note that the register values are indicated in red when they change as a result of executing the last instruction.

In a program that contains subroutines it is possible to step over an entire subroutine by using the **Step Over Subroutine** command in the **Actions** menu. This command performs a normal single step, unless the current instruction is a **Call** instruction, in which case the program will run until the called subroutine is completed.

### 3.6 Using Breakpoints

An *instruction breakpoint* provides a means of stopping the execution of a program when it reaches an instruction at a specific address. The procedure for setting a breakpoint is:

1. In the Disassembly window, scroll to display the instruction that will have the breakpoint. For example, in the window in Figure 15 scroll to the Branch instruction at address 0x00000018.
2. Click on the gray bar to the left of the address 0x00000018. As illustrated in Figure 17, the Monitor Program displays a red dot next to the address to show that a breakpoint has been set. Clicking the same location again removes the breakpoint.

Once the instruction breakpoint has been set, run the program. The breakpoint will trigger when the Program Counter value equals 0x00000018. Control then returns to the Monitor Program, and the Disassembly window highlights in a yellow color the instruction at the breakpoint. A corresponding message is shown in the **Info & Errors** pane.

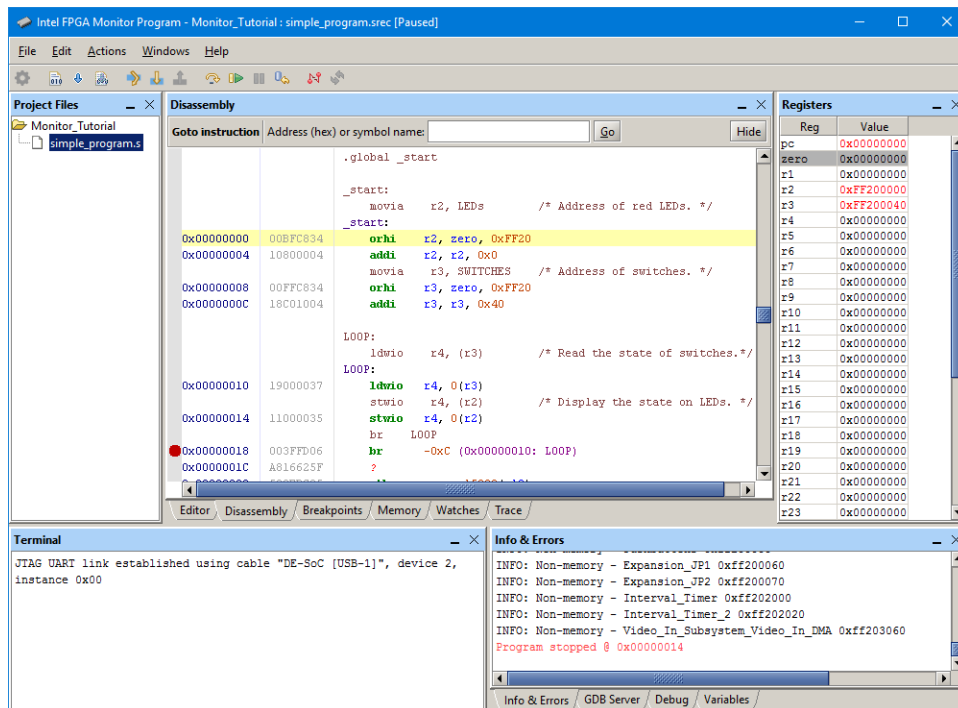


Figure 17. Setting a breakpoint.

Some versions of the Nios II processor support other types of breakpoints in addition to instruction breakpoints. Other types of breakpoints are described in Appendix A.

### 3.7 Examining and Changing Register Values

The Registers window on the right-hand side of the Monitor Program display shows the values of processor registers. It also allows the user to edit most of the register values. The number format in which the register values are displayed can be changed by right-clicking in the Registers window and selecting the desired format, as illustrated in Figure 18.

Each time program execution is halted, the Monitor Program updates the register values and highlights any changes in red. The user can edit the register values while the program is halted. Any edits made are visible to the processor when the program's execution is resumed.

As an example of editing a register value, set the slider switches on the DE1-SoC board to some pattern of 0s and 1s. Run the *simple\_program* and observe that the LEDs display the selected pattern. Next, stop the execution of the program and set a breakpoint at the Store instruction at address 0x00000014. Run the program and after the execution stops at the breakpoint, observe that the value in register r4 corresponds to the current setting of the slider switches. Now, as indicated in Figure 19, double-click on the contents of register r4 and change them to the value FFF. Press Enter on the computer keyboard, or click away from the register value to apply the edit. Then, single-step the program to see that all LEDs will be turned on.

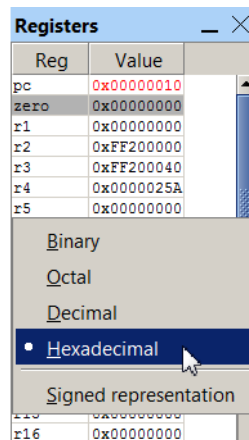


Figure 18. Setting the number format for displaying register values.

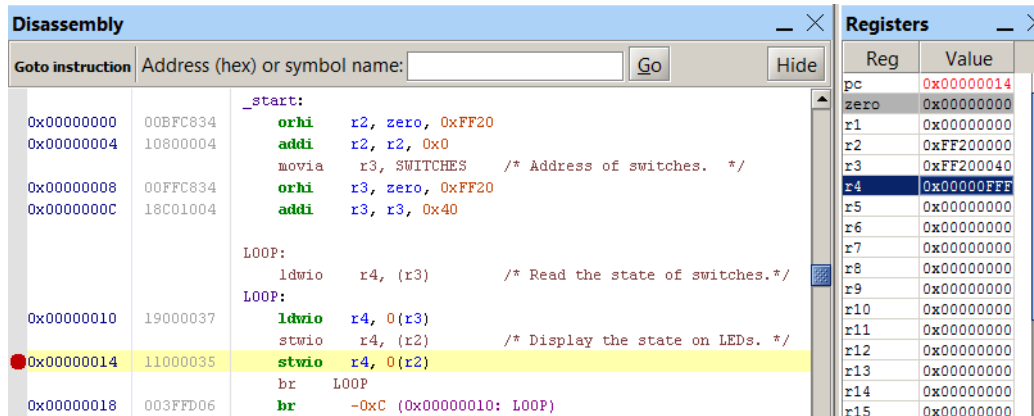


Figure 19. Editing a register value.

### 3.8 Examining and Changing Memory Contents

The Memory window, depicted in Figure 20, displays the contents of the system's memory space and allows the user to edit memory values. The leftmost column in the window gives a memory address, and the numbers at the top of the window represent hexadecimal address offsets from that corresponding address.

In this figure, the address of the second word in the second row is  $0x00000010 + 0x4 = 0x00000014$ . The displayed contents of this memory location are  $0x11000035$ , which is the machine code for the instruction

```
stwio r4, 0(r2)
```

If a program is running, the data values displayed in the Memory window are not updated. But, when the program is stopped, the data values are automatically updated. They can also be updated by pressing the Refresh button. By default, the Memory window shows only the contents of memory devices, and does not display any values from

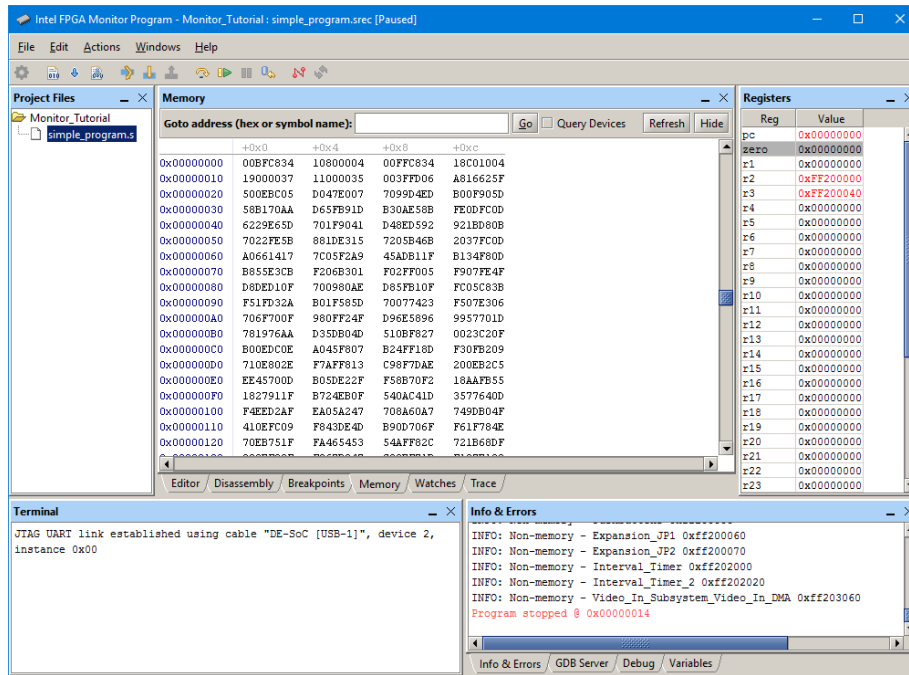


Figure 20. The Memory window.

memory-mapped I/O devices. To cause the window to display memory-mapped I/O locations, click on the check mark beside Query Devices, and then click Refresh. For example, set the slider switches to some pattern, press Refresh, enter the address `0xFF200040` into the *goto address* box and then press Go. Figure 21 shows the display we obtained when choosing the pattern `0x30F`.

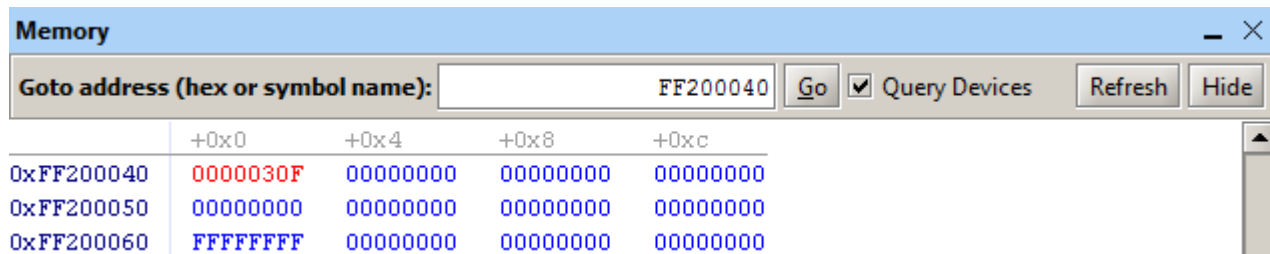


Figure 21. Displaying the I/O locations.

The color of a memory word displayed depends on whether that location corresponds to an actual memory device, a memory-mapped I/O device, or is not mapped at all in the system. A memory location that corresponds to a memory device will be colored black, as in Figure 20. Memory-mapped I/O is shown in blue color, and a non-mapped address is shown in grey. If a memory location changed value since it was previously displayed, then that memory location is shown in a red color.

Similar to the Disassembly window, it is possible to view different memory regions by scrolling using the vertical scroll bar on the right, or by using a mouse scroll wheel. There is also a **Goto address** panel, which is analogous to the **Goto instruction** panel discussed in Section 3.4. Note that in Figure 21 we reached the I/O device by typing the address FF200040 in this panel.

As an example of editing a memory value, go to address FF200000 which is the address of LEDs. Double-click on the memory word at this address and type the data value FFF. Press **Enter** on the computer keyboard, or click away from the memory word to apply the edit. This should cause all LEDs to be turned on.

When accessing an I/O device, some reads may be destructive. Namely, after some register in the I/O interface is read, its contents may no longer be valid. Therefore, it is not appropriate to read all I/O registers when refreshing the information in the Memory window. Instead, it is prudent to read only the registers that are of specific interest. This can be accomplished by left-clicking on the address of interest, then right-clicking and selecting **Read Selected Address Range** to update the displayed contents. Several consecutive addresses can be selected by clicking on the first address and dragging across the other addresses.

It is possible to change the appearance of the Memory window in a number of ways, such as displaying data as bytes, half-words or words. The Memory window provides additional features that are described in more detail in Appendix A of this document.

## 4 Working with Project Files


Project files store the settings for a particular project, such as the specification of a hardware system and program source files. A project file, which has the filename extension *.amp*, is stored into a project's directory when the project is created.

The Monitor Program provides the following commands, under the **File** menu, for working with project files:

1. **New Project**: Presents a series of screens that are used to create a new project.
2. **Open Project**: Displays a dialog to select an existing project file and loads the project.
3. **Open Recent Project**: Displays the five most recently used project files, and allows these projects to be reopened.
4. **Save Project**: Saves the current project's settings after they have been modified by using the **Settings** command.

### 4.1 Modifying the Settings of an Existing Project

After a project has been created, it is possible to modify many of its settings, if needed. This can be done by clicking on the menu item **File > Edit Project > System Settings** in the Monitor Program. This action will display the existing system settings for the project, and allow them to be changed. Similarly, the program settings for the project can be displayed and modified by using the command **File > Edit Project > Program Settings**. To change these

settings, the Monitor Program has to first be disconnected from the system being debugged. This can be done by using the command **Actions > Disconnect**, or by clicking the  icon.

## 5 Using the Monitor Program with a Nios® II Evaluation License

In our discussion of Figure 11 in Section 3.1, we showed how the Monitor Program can be used to download a prebuilt Nios II hardware system onto an FPGA board, when the Nios II processor has a license. It is also possible to use the Monitor Program to debug hardware systems in which the Nios II processor includes only an evaluation license. In this case it is necessary to download the hardware system onto the FPGA board by using the *Programmer* tool provided in the Quartus Prime software, rather than using the Monitor Program for this purpose. The Quartus Prime Programmer tool provides a pop-up window, shown in Figure 22, which indicates activation of the evaluation license for the Nios II processor. This pop-up window has to remain open in order to maintain the evaluation license for Nios II. As long as the pop-up window remains open, the Monitor Program can be used to compile and download software programs into the hardware system.

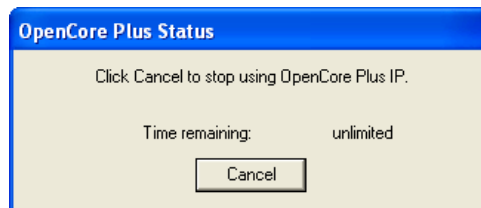


Figure 22. The Quartus Prime Programmer pop-up window.

## 6 Using the Terminal Window

Monitor Program's *Terminal* window supports text-based input and output. To see its operation, create a new Monitor Program project, called *Monitor\_Terminal*. When creating the project, follow the same steps shown for the *Monitor\_Tutorial* project, which were described in Section 3.1. For the screen shown in Figure 7 set the program type to **Assembly Program**, and select the sample program named *JTAG\* UART*. The source code file for that program is called *JTAG\_UART.s*. It communicates using memory-mapped I/O with the JTAG UART in the DE1-SoC Computer that is selected as the **Terminal device** in the screen of Figure 9.

Compile, load and run the program. The Monitor Program window should appear as shown in Figure 23. Click the mouse inside the Terminal window. Now, any characters typed on the computer keyboard are sent by the Monitor Program to the JTAG UART. These characters are shown in the Terminal window as they are typed, because the *JTAG\_UART.s* program echoes the characters back to the Terminal window.

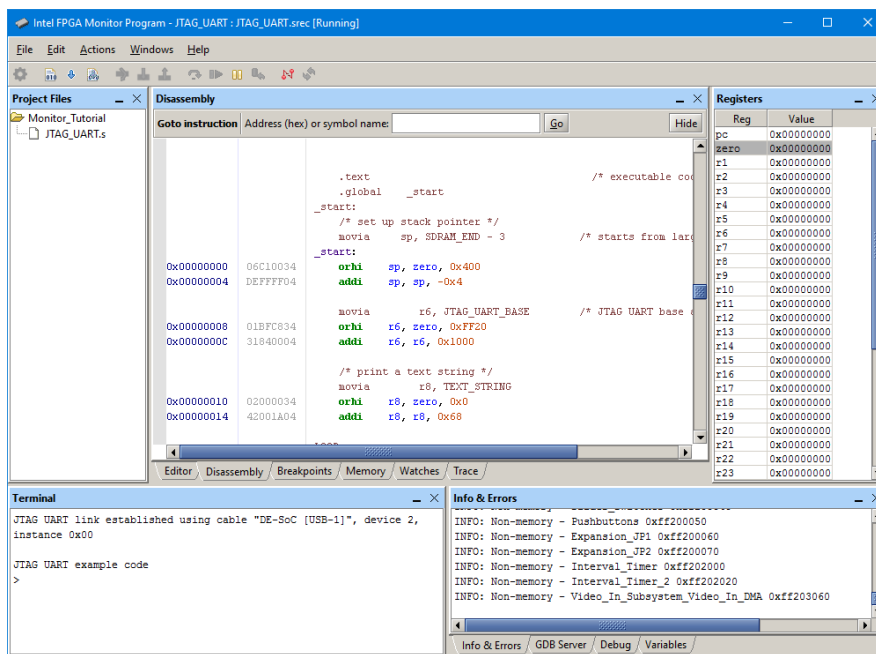


Figure 23. Using the Terminal window.

The Terminal window supports a subset of the control character commands used for a de facto standard terminal, called the *VT100\**. The supported commands are listed in Table 1. In this table <ESC> represents the ASCII character with the code 0x1B.

## 7 Using C Programs

C programs are used with the Monitor Program in a similar way as assembly-language programs. To see an example of a C program, create a new Monitor Program project called *Monitor\_Terminal\_C*. Use the same settings as for the *Monitor\_Terminal* example, but set the program type for this project to **C Program**. Select the C sample program called *JTAG UART*. As illustrated in Figure 24, this program includes a C source file named *JTAG\_UART.c*; it has the same functionality as the assembly-language code used in the previous example. Compile and run the program to observe its behavior.

The C code in *JTAG\_UART.c* uses memory-mapped I/O to communicate with the JTAG UART. Alternatively, it is possible to use functions from the standard C library *stdio.h*, such as *putchar*, *printf*, *getchar*, and *scanf* for this purpose. Using these library functions impacts the size of the Nios II executable code that is produced when the C program is compiled, by about 30 to 64 KBytes, depending on which functions are needed. It is possible to minimize the size of the code generated for this library by checking the box labeled **Use small C library** in Figure 24. When this option is used the library has reduced functionality. Some limitations of the small C library include: no floating-point support in the output routines, such as *printf*, and no support for input routines, such as *scanf* and *getchar*.

Character Sequence	Description
<ESC> [ 2J	Erases everything in the Terminal window
<ESC> [ 7h	Enable line wrap mode
<ESC> [ 7l	Disable line wrap mode
<ESC> [ #A	Move cursor up by # rows or by one row if # is not specified
<ESC> [ #B	Move cursor down by # rows or by one row if # is not specified
<ESC> [ #C	Move cursor right by # columns or by one column if # is not specified
<ESC> [ #D	Move cursor left by # columns or by one column if # is not specified
<ESC> [ #1 ; #2 f	Move the cursor to row #1 and column #2
<ESC> [ H	Move the cursor to the home position (row 0 and column 0)
<ESC> [ s	Save the current cursor position
<ESC> [ u	Restore the cursor to the previously saved position
<ESC> [ 7	Same as <ESC> [ s
<ESC> [ 8	Same as <ESC> [ u
<ESC> [ K	Erase from current cursor position to the end of the line
<ESC> [ 1K	Erase from current cursor position to the start of the line
<ESC> [ 2K	Erase entire line
<ESC> [ J	Erase from current line to the bottom of the screen
<ESC> [ 1J	Erase from current cursor position to the top of the screen
<ESC> [ 6n	Queries the cursor position. A reply is sent back in the format <ESC> [ #1 ; #2R, corresponding to row #1 and column #2.

Table 1. VT100 commands supported by the Terminal window.

In Figure 24 the option *Emulate unimplemented instructions* is checked. This option causes the C compiler to include code for emulating any operations that are needed to execute the C program but which are not supported by the processor. For example, the Nios II Economy version does not include a *multiply* instruction, but the C program may need to perform this operation. By checking this option, a multiply instruction will be implemented in software (by using addition and shift operations).



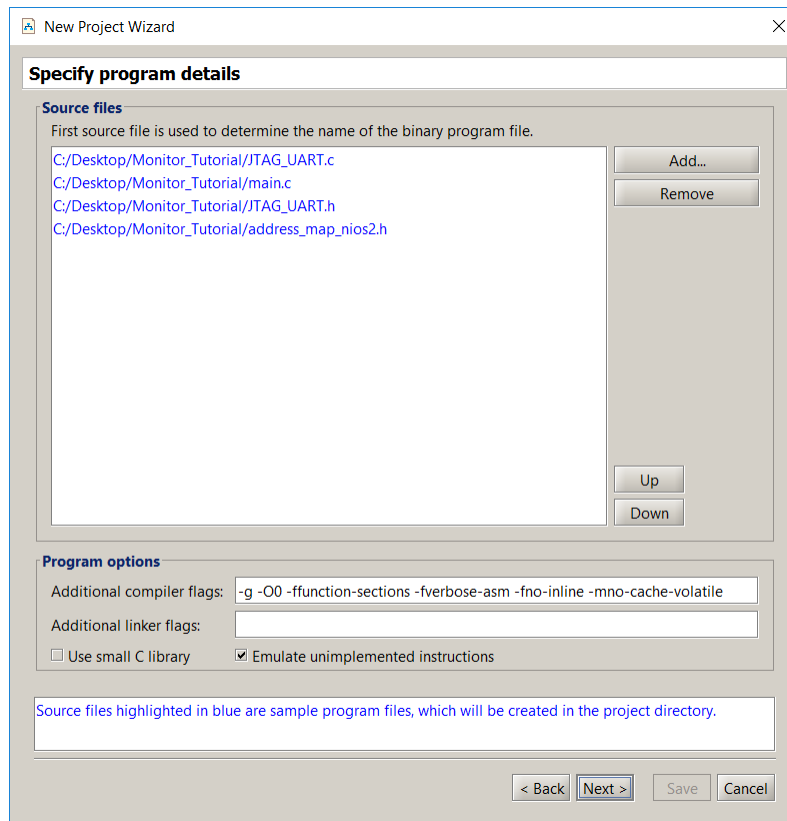


Figure 24. Settings for a C program.

## 7.1 Source Level Debugging

The Monitor program supports common source level debugging features such as step over, step into, step out, and visualizing variables. Using the JTAG UART sample program project you created in the previous section, go to the project settings (File > Edit Project ) and navigate to the *Program Settings* tab. In the *Compiler Flags* input box, ensure that the optimization level is set to 0, by replacing *-O*, *-O1*, *-O2*, or *-O3* flag with *-O0*. An optimization level of 0 allows the Monitor Program to read and display variables from memory. Figure 25 shows the Monitor Program's text editor. The editor will be disabled during the debug session, and re-enabled when the debug session is exited. Now save the project (File > Save Project), and compile and load the program (Actions > Compile & load).

### 7.1.1 Using Breakpoints

Once the program is loaded, navigate to the Editor window of the Monitor Program. Go to the File menu and select File > Open... to open the C source file which contains the *main* function of your program (most likely *main.c*).

Once the program is loaded, toggle the breakpoint at a line of source code by clicking on the numbers to the left of the source code text. If a breakpoint does not show up on the line similar to Figure 26, the line of source code likely does not correspond to an instruction. If this happens, try choosing a different line.

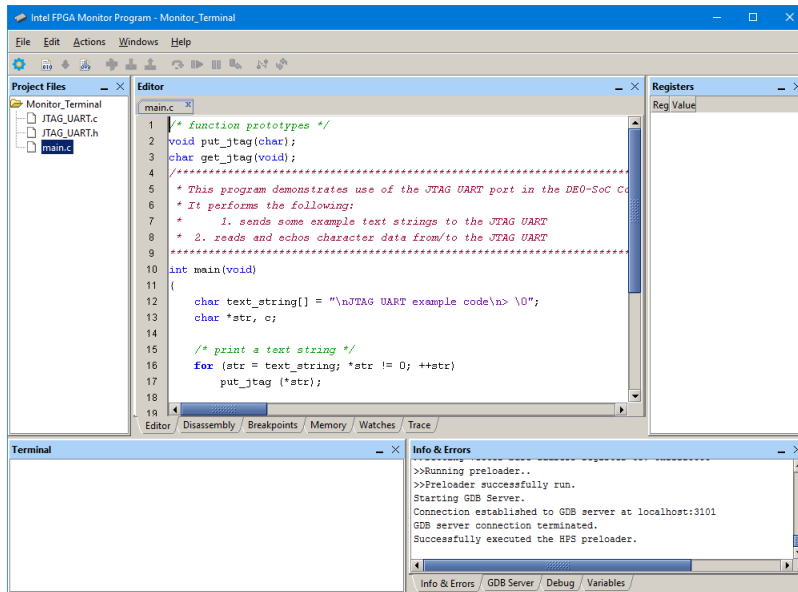


Figure 25. The Monitor Program with a source file open in editor view.

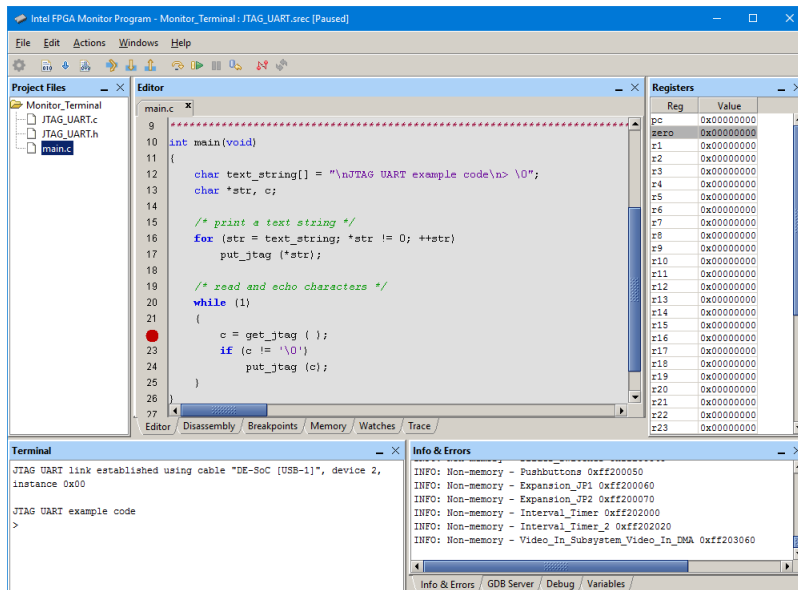


Figure 26. Setting a breakpoint in the editor view.

Once the breakpoint is set, continue the program by clicking the green arrow on the toolbar, or **Actions > Continue**. Once the program halts, the Monitor Program should look similar to Figure 27. In the Disassembly view the source level breakpoint is marked with a red square as in Figure 28. This differentiates source level breakpoints from instruction level breakpoints.

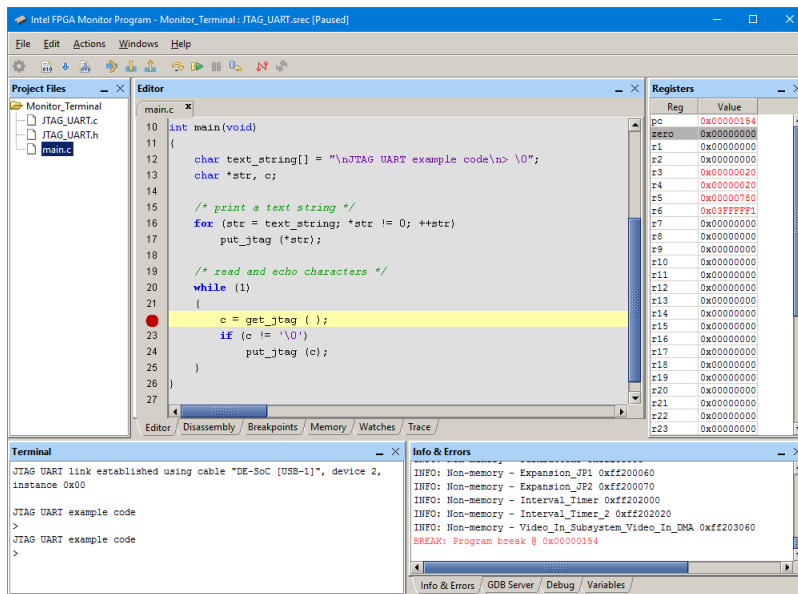


Figure 27. Hitting a breakpoint in the editor view.

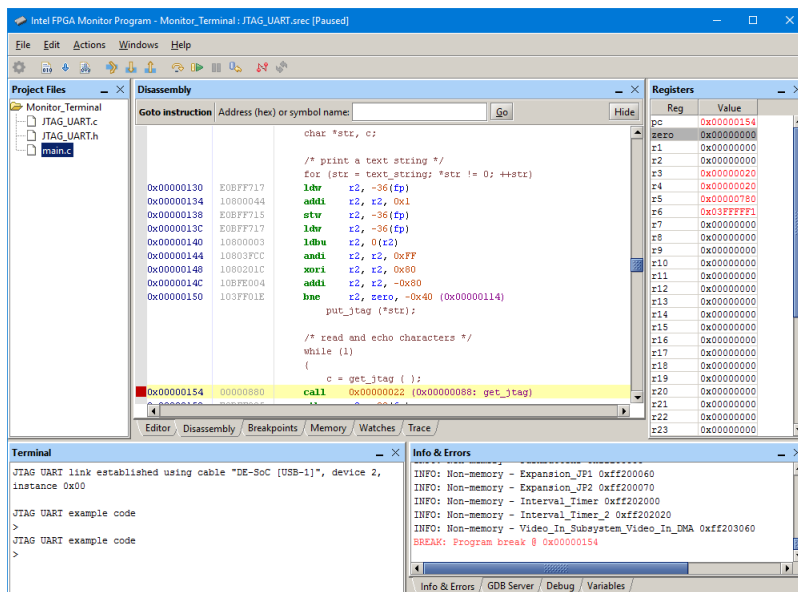


Figure 28. Source level breakpoint in the disassembly view.

## 7.1.2 Source Level Debugging Actions

Navigate back the editor view and perform a Step Into action by selecting **Actions > Step Into**, or by using the main toolbar. This will step to the next line of source code to be executed. If the program steps into a function in another file, the Monitor Program will open the file in a new tab and highlight the line.



Figure 29. Step Over, Step Into, Step Out toolbar icons

Next, perform a Step Out action by selecting **Actions > Step Out**, or by using the main toolbar. This will step out of the current function by executing until the first line of source code after returning from the current function. The Monitor Program will print an error to the **Info & Errors** window if it cannot step out of the current function. This may occur if the program is currently in the *main* function, or if the function does not return. The step out function is only available for C programs, it is not available for assembly programs.

The Step Over action (**Actions > Step Over**) moves to the next line of source code without stepping into functions. Execution will continue to the next line of source code inside the current function.

### 7.1.3 Variable Values

Variables		
Name	Type	Value
[-] Locals		
c	char	y
str	char *	0x3FFFFFF
[-] text_string	char[28]	JTAG UART example co...

Figure 30. Monitor Program Variable View.

The Monitor Program's **Variables** view displays the value of C program variables when the program is halted. Some variable types such as Arrays, Typedefs, Structures and Unions will be expandable in the view. Use the + button to expand and view the variables contents. Right clicking on a variable presents the options to jump to the declaration of the variable, and the display format of the variable.

**Go To Declaration** will open the file the variable is declared in and scroll to the declaration line number. **Display As...** will change the format in which the variable is displayed.

Variable values are only available with an optimization level of **0** (gcc command line argument *-O0*). For instructions on how to change the programs optimization level, see the first paragraph of this section.

### 7.1.4 Enabling and Disabling Source Level Debugging

The source level debugging feature of the Monitor Program is a beta feature in the current release. The feature can be enabled and disabled at any point by going to the **Edit** menu and selecting **Edit > Enable Source Level Debugging**, or **Edit > Disable Source Level Debugging**, depending on whether the feature is currently disabled or enabled respectively.

### 7.1.5 Setting the Optimization Level in Programs with Driver Support.

To set the optimization level for a *Program with Driver Support* (or BSP), first create a TCL script in the base directory of the project (the same directory as your AMP project file). The TCL file should have a *.tcl* file extension, for example *config.tcl*. Open this file in a text editor and add the single line:

```
set_setting hal.make.bsp_cflags_optimization -O0
```

Where the argument *-O0* above is the desired optimization level. Now open the project settings in the Monitor Program and navigate to the *Program Settings* tab. In the *BSP settings Tcl script* input box (shown in Figure 31) enter the path to the TCL script you just created, or use the *Browse* button to search for it.

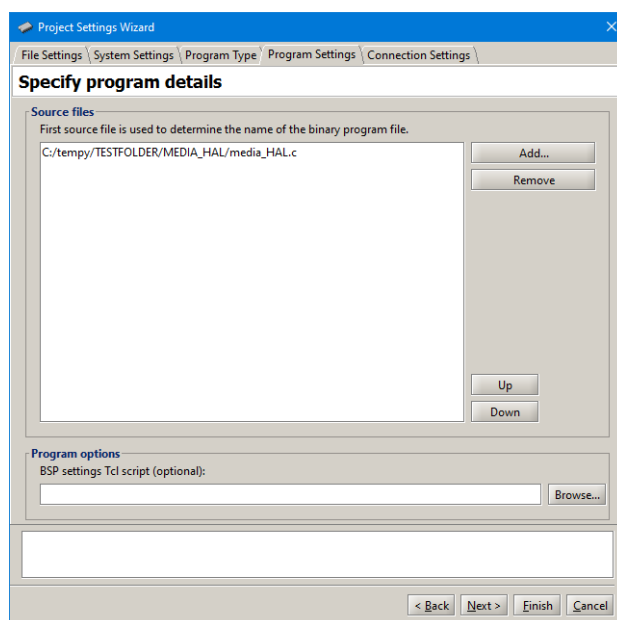


Figure 31. Adding a TCL script to a Program with Driver Support.

Click the *Finish* button to close the dialog and save and compile the project. The optimization level should be set for both the generated (BSP) files, as well as your project files.

## 8 Using the Monitor Program with Interrupts

The Monitor Program supports the use of interrupts in Nios II programs. Two examples of interrupts are illustrated below, using assembly-language code and using C code.

### 8.1 Interrupts with Assembly-Language Programs

To see an example using interrupts with assembly-language code, create a new Monitor Program project called *Monitor\_Interrupts*. When creating the new project set the program type to assembly language and select the sample program named *Interrupt Example*. Figure 32 lists the source files for this sample program. The main program for

the example is the file *interrupt\_example.s*, which initializes some I/O devices and enables Nios II interrupts. The other source files provide the reset and exception handling for the program, and two interrupt service routines.

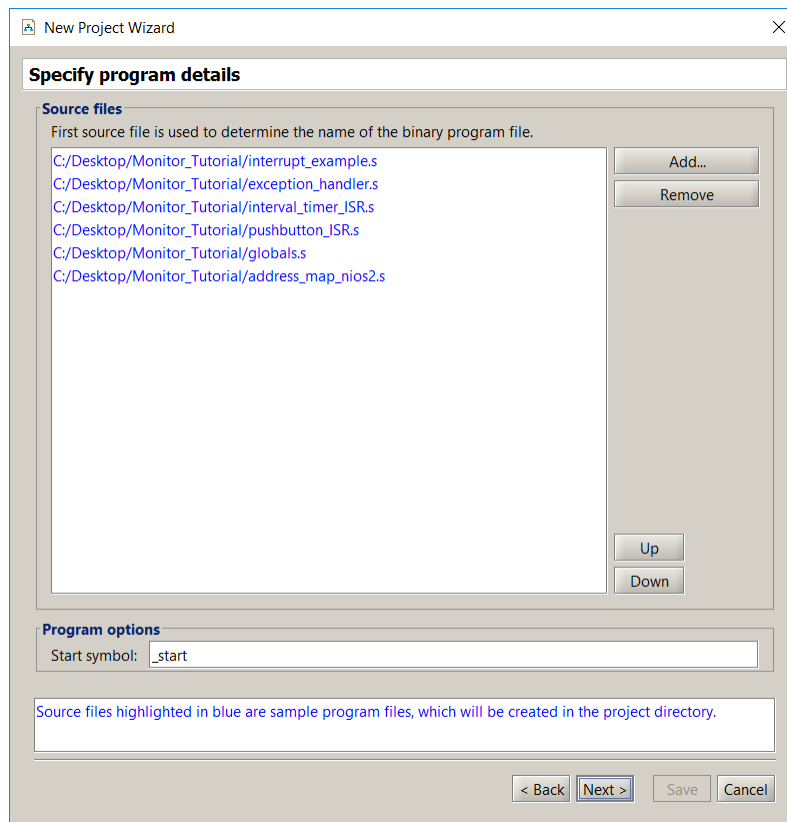


Figure 32. The source files for the interrupt example.

Figure 33 shows the memory settings for this program. The reset vector of the Nios II processor is at address  $0 \times 0$  and the exceptions vector is at address  $0 \times 20$ . Enough space has to be left between the exceptions vector location and the text section of the program to accommodate the exceptions processing code, which corresponds to the assembly language code in the file *exception\_handler.s*. Starting the main program at address  $0 \times 200$ , as shown in the figure, leaves enough space to accommodate the exception processing code for this example.

Compile and load the program. Then, scroll the Disassembly window to the label *EXCEPTION\_HANDLER*, which is at address  $0 \times 00000020$ . As illustrated in Figure 34, set a breakpoint at this address. Run the program. When the breakpoint is reached, single step the program a few more instructions to determine the cause of the interrupt. The source of the interrupt is a circuit in the DE1-SoC Computer called the *interval timer*. This circuit provides the ability to generate an interrupt whenever a specified time period elapses. Single step the program until the processor enters the interrupt-service routine for the interval timer. This routine first clears the timer register that caused the interrupt, so that an interrupt request will not be raised immediately again, and then performs other functions needed for the program.

Finally, remove the breakpoint that was set earlier, at address  $0 \times 00000020$ , and then select the **Continue** command to run the program. Observe that the program displays a rotating pattern across the HEX displays on the DE1-SoC

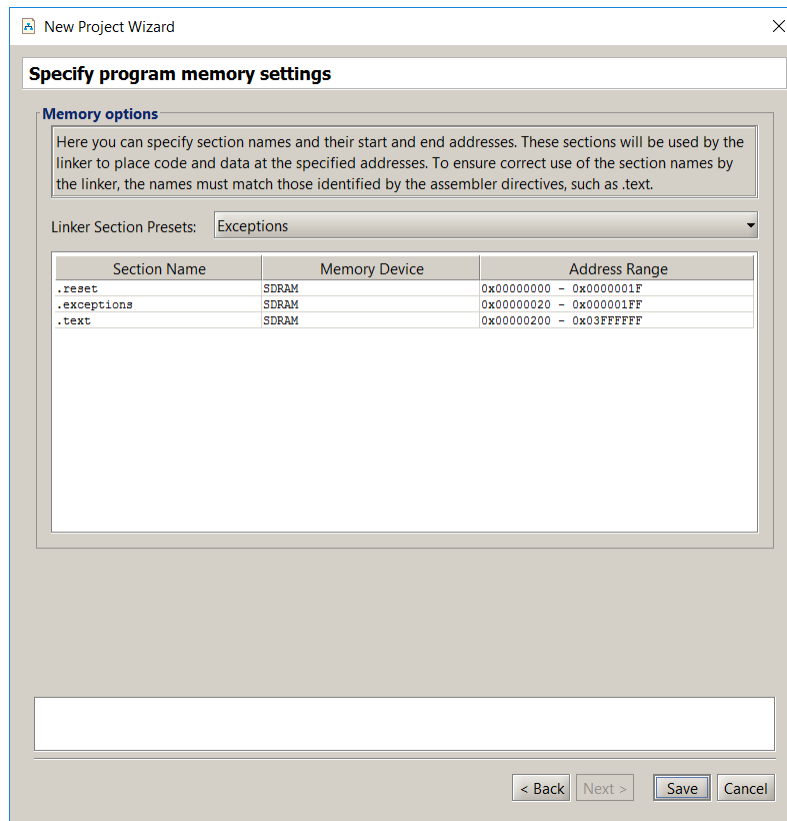


Figure 33. Memory offset settings for the interrupt example.

board. The direction of rotation can be changed by pressing the pushbuttons KEY<sub>1</sub> or KEY<sub>2</sub> on the DE1-SoC board, and the pattern can be changed to correspond to the values of the slider switches by pressing KEY<sub>3</sub>.

## 8.2 Interrupts with C Programs

To see an example of a C program that uses interrupts, create a new project called *Monitor\_Interrupts\_C*. When creating this project, set the program type to **C Program** and select the sample program named *Interrupt Example*; this program gives C code that performs the same operations as the assembly-language code in the previous example. The source files for the C code are listed in Figure 35. The main program is given in the file *interrupt\_example.c*, and the other source files provide the reset and exception handling for the C program, as well as two interrupt-service routines. Complete the steps for creating the project, and then compile and load it.

Set a breakpoint at the address 0x00000020, which is the exception vector address for the Nios II processor. Also, scroll the Disassembly window to the function called *interrupt\_handler*. As illustrated in Figure 36, set another breakpoint at this address. Now, run the program to reach the first breakpoint, at address 0x00000020. The code at this address, which is found in the file *exception\_handler.c*, reads the contents of a control register in the Nios II processor to determine if the interrupt is caused by an external device, then saves registers on the stack, and then calls the *interrupt\_handler* function.

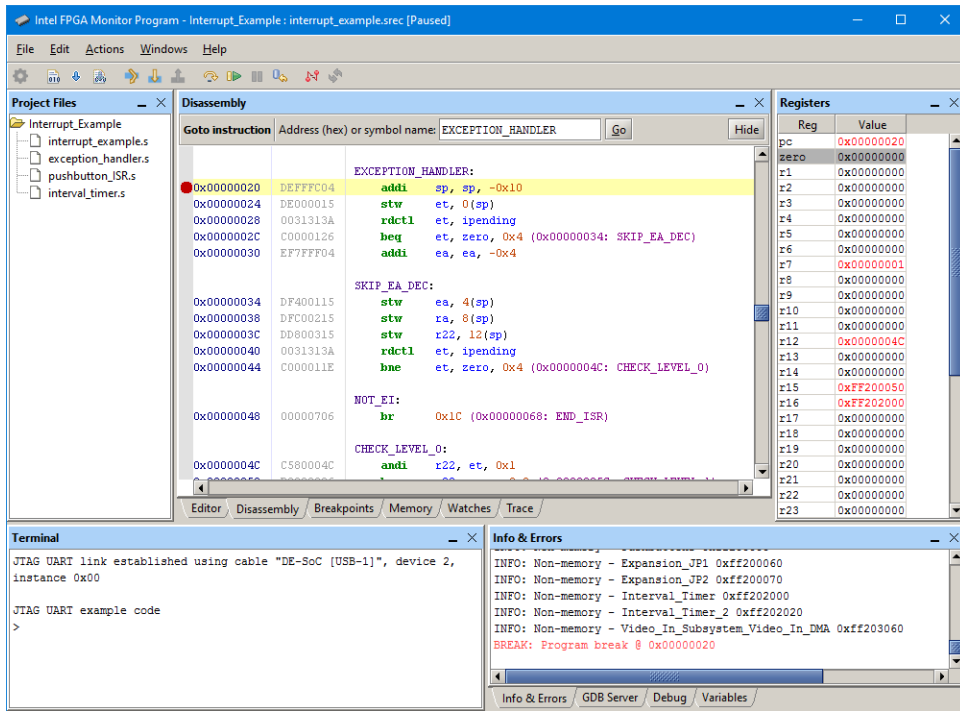


Figure 34. The exception handler.



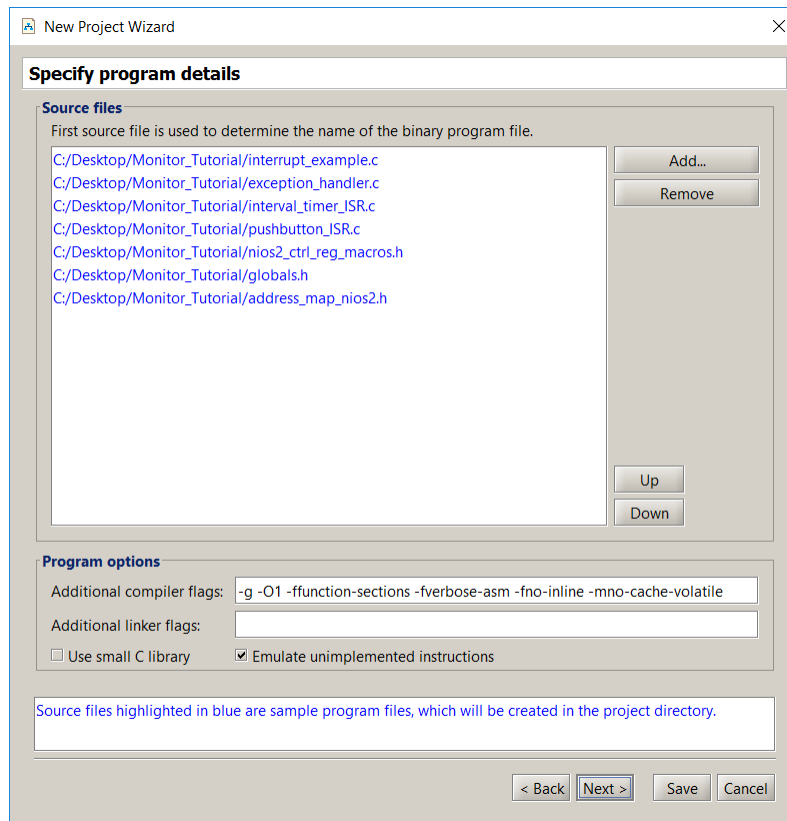


Figure 35. The source files for the C code interrupt example.

<pre> 0x000005A0 0x000005A4 0x000005A8 0x000005AC 0x000005B0 0x000005B4 0x000005B8 </pre>	<pre> DEFFFE04 DFC00115 DC000015 0021313A 8080004C 10000126 00005D80 </pre>	<pre> void interrupt_handler(void) {     interrupt_handler:     addi    sp, sp, -0x8     stw    ra, 4(sp)     stw    r16, 0(sp)     int ipending;     NIOS2_READ_IPENDING(ipending);     rdctl  r16, ipending     if ( ipending &amp; 0x1 )         // interval timer is interrupt level 0     andi   r2, r16, 0x1     beq    r2, zero, 0x4 (0x000005BC)     {         interval_timer_isr( );     }     call   0x00000176 (0x000005D8: interval_timer_isr) } </pre>
---	---	---

Figure 36. The interrupt handler.

Press **Actions > Continue** in the Monitor Program to reach the second breakpoint. Single stepping the program a few more instructions shows that the interrupt is caused by the interval timer in the DE1-SoC Computer, as discussed in the previous example. Additional single stepping causes the processor to enter the interrupt-service routine for the interval timer, as depicted in Figure 37. This routine first clears the timer register that caused the interrupt, and then performs other functions needed for the program. Finally, clear both breakpoints that were set earlier, at address

0x00000020 and *interrupt\_handler*, and then run the program; it displays a rotating pattern on the HEX displays of the DE1-SoC board, as discussed in the previous example.

<pre> 0x000005D8  00BFC834 0x000005DC  10880004 0x000005E0  10000035 </pre>	<pre> volatile int * interval_timer_ptr = (int *) INTERVAL_TIMER_BASE; volatile int * HEX3_HEX0_ptr      = (int *) HEX3_HEX0_BASE; // HEX3_HEX0 address volatile int * HEX7_HEX4_ptr      = (int *) HEX7_HEX4_BASE; // HEX7_HEX4 address  *(interval_timer_ptr) = 0; // Clear the interrupt  interval_timer_isr: orhi  r2, zero, 0xFF20 addi  r2, r2, 0x2000 stwi  zero, 0(r2) </pre>
---	---

Figure 37. The interrupt service routine for the interval timer.

## 9 Working with Windows and Tabs

It is possible to rearrange the Monitor Program workspace by moving, resizing, or closing the internal windows inside the main Monitor Program window.

To move a particular window to a different location, click on the window title or the tab associated with the window, and drag the mouse to the new location. As the mouse is moved across the main window, the dragged window will snap to different locations. To detach the dragged window from the main window, drag it beyond the boundaries of the main window. To re-attach a window to the main window, drag the tab associated with the window onto the main window.

To resize a window, hover the mouse over one of its borders, and then drag the mouse. Resizing a window that is attached to the main window will cause any adjacent attached windows to also change in size accordingly.

To hide or display a particular window, use the **Windows** menu. To revert to the default window arrangement, simply exit and then restart the Monitor Program. Figure 38 shows an example of a rearranged workspace.

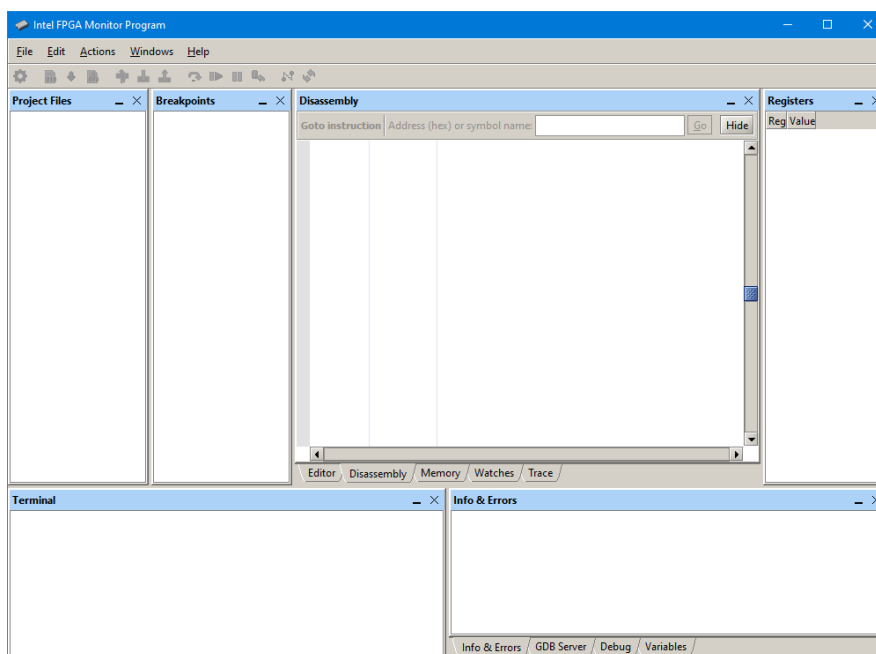


Figure 38. The Intel FPGA Monitor Program with a rearranged workspace.

## 10 Appendix A

This appendix describes a number of Monitor Program features that are useful for advanced debugging or other purposes.

### 10.1 Using the Breakpoints Window

In Section 3.6 we introduced instruction breakpoints and showed how they can be set using the Disassembly window. Another way to set breakpoints is to use the *Breakpoints* window, which is depicted in Figure 39. This window supports three types of breakpoints in addition to the instruction breakpoint: *read watchpoint*, *write watchpoint*, and *access watchpoint*, as follows:

- Read watchpoint - the processor is halted when a read operation is performed on a specific address.
- Write watchpoint - the processor is halted when a write operation is performed on a specific address.
- Access watchpoint - the processor is halted when a read or write operation is performed on a specific address.

In Figure 39 an instruction breakpoint is shown for the address `0x00000018`. This corresponds to an address in *simple\_program.s*. In Section 3.6 we showed how to create such an instruction breakpoint by using the Disassembly window. But we could alternatively have created this breakpoint by right-clicking in a grey box under the label Instruction breakpoint in Figure 39 and then selecting Add. A breakpoint can be deleted by unchecking the box beside its address.

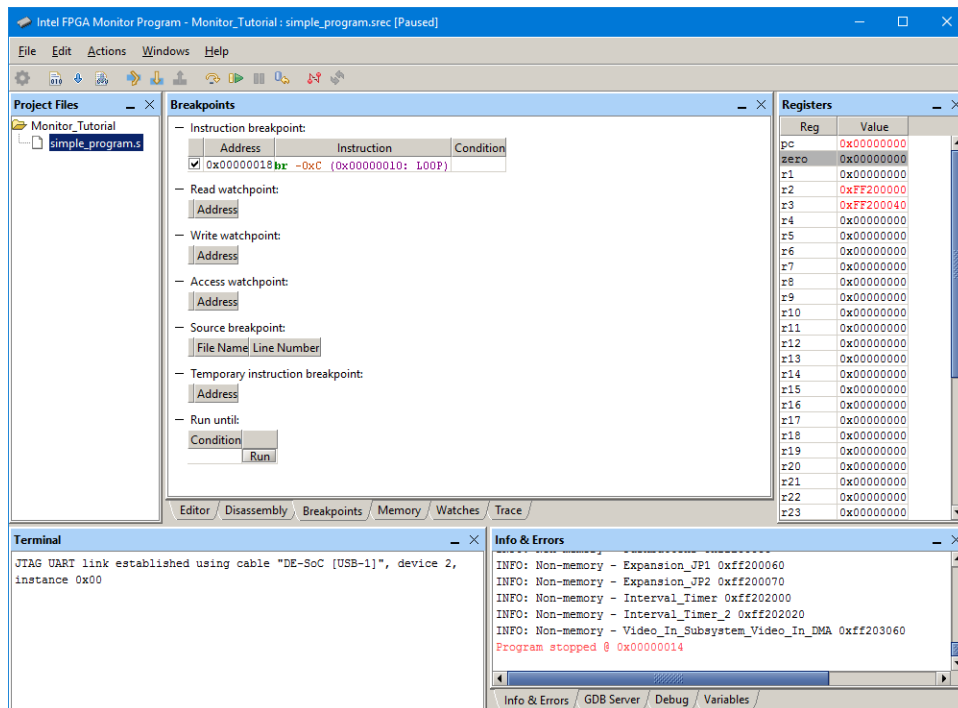


Figure 39. The Breakpoints window.

Setting a read, write, or access watchpoint is done by right-clicking on the appropriate box in Figure 39 and specifying the desired address.

The Monitor Program also supports a type of breakpoint called a *conditional breakpoint*, which triggers only when a user-specified condition is met. This type of breakpoint is specified by double-clicking in the empty box *under* the label **Condition** in Figure 39 to open the dialog shown in Figure 40. The condition can be associated with an instruction breakpoint, or it can be a stand-alone condition if entered in the **Run until** box in the Breakpoints window. As an example, we compiled and loaded the *simple\_program* project. Then, we entered the condition `r4 == 5`. The condition causes the breakpoint to trigger only if register `r4` contains the value 5. Thus, running this program causes the LEDs to display the current state of the slider switches as these switches are set to different patterns. But, when the selected pattern is `0x005`, the conditional breakpoint will stop the execution of the program.

Note that if a stand-alone condition is entered in the **Run until** box, then the **Run** button associated with this box must be used to run the program, rather than the normal **Actions > Continue** command. The processor runs much more slowly than in its normal execution mode when a conditional breakpoint is being used.

## 10.2 Working with the Memory Window

The Memory window was shown in Figure 20. This window is configurable in a variety of ways:

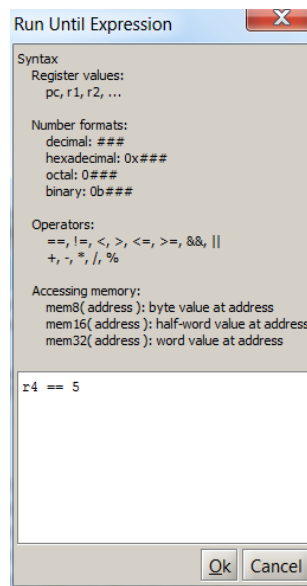


Figure 40. The Conditional Breakpoint dialog.

- Memory element size - the display can format the memory contents as bytes, half-words (2-bytes), or words (4-bytes). This setting can be configured by right-clicking on the Memory window, as illustrated in Figure 35.

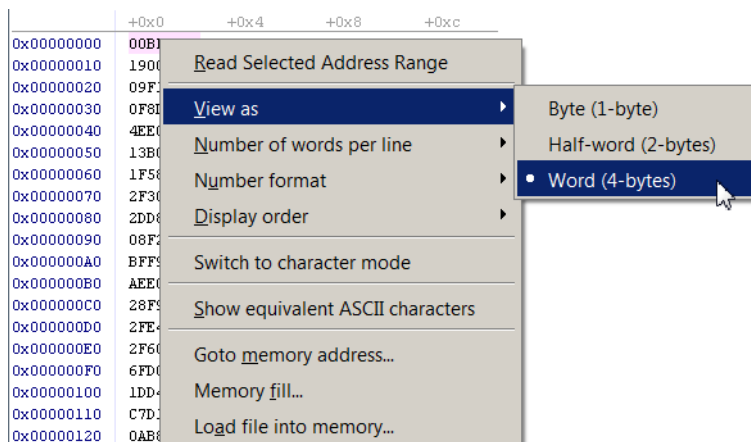


Figure 41. Setting the memory element size.

- Number of words per line - the number of words per line can be configured to make it easier to find memory addresses, as depicted in Figure 42.
- Number format - this is similar to the number format option in the Register window described in Section 3.7, and can be configured by right-clicking on the Memory window.
- Display order - the Memory window can display addresses increasing from left-to-right or right-to-left.

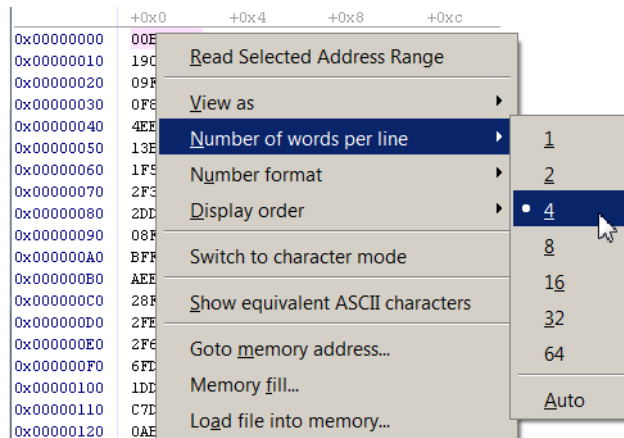


Figure 42. Setting the number of words per line.

### 10.2.1 Character Display

The Memory window can also be configured to interpret memory byte values as ASCII characters. This is useful if one wishes to examine character strings that are stored in the memory. For this purpose it is convenient to view the memory in bytes and characters simultaneously so that the characters appear in the correct sequence. This can be accomplished by clicking the **Switch to character mode** menu item, as illustrated in Figure 43. A sample display in the character mode is shown in Figure 44.

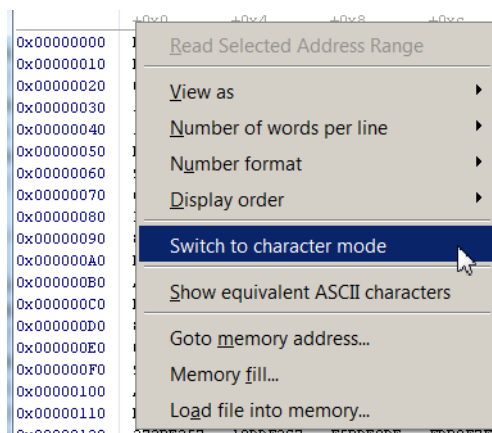


Figure 43. Switching to the character mode.

It is possible to return to the previous memory view mode by right-clicking and selecting the **Revert to previous mode** menu item.

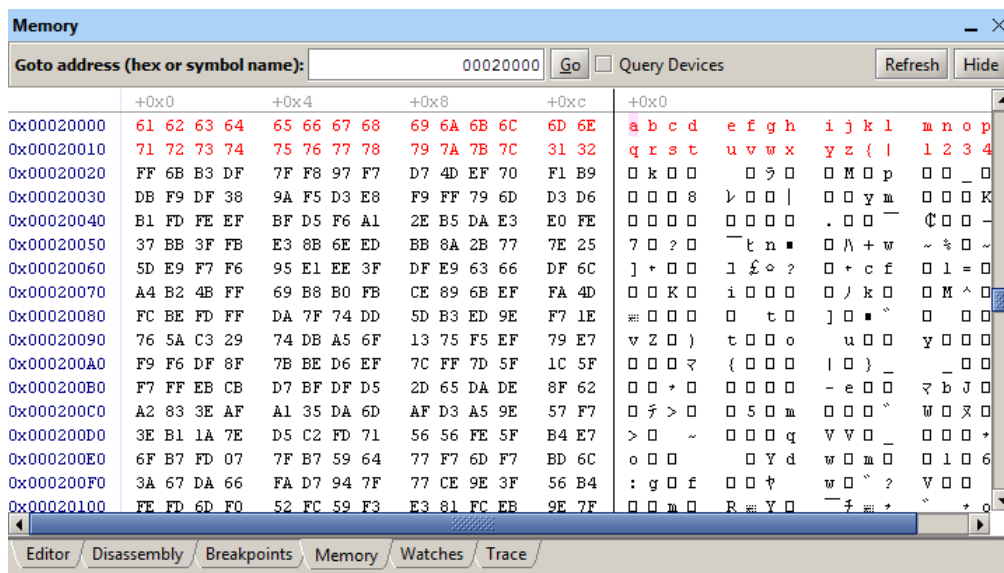


Figure 44. Character mode display.

## 10.2.2 Memory Fill

Memory fills can be performed in the Memory window. Click the **Actions > Memory fill** menu item or right-click on the Memory window and select **Memory fill**. A Memory fill panel will appear on the left side of the Memory window. Simply fill in the desired values and click **Fill**.

## 10.2.3 Load File Data into Memory

Data stored in a file can be loaded into the memory by using the Memory window. This feature is accessed by selecting the command **Actions > Load file into memory** or by right-clicking on the Memory window. The **Load file** panel will appear on the left side of the Memory window, as illustrated in Figure 45, to allow the user to browse and select a data file. The user provides a base address in memory where the data should be stored.

The format of these files is illustrated in Figure 46. The file consists of any number of lines, where each line comprises a comma-separated list of data values. Each data value is expressed as a hexadecimal number with an optional **-** sign. Two additional parameters can be specified: the value of the delimiter character (comma is the default), and size in bytes of each data value (1 is the default).

## 10.3 Setting a Watch Expression

Watch expressions provide a convenient means of keeping track of the value of multiple expressions of interest. These expressions are re-evaluated each time program execution is stopped. To add a watch expression:

1. Switch to the *Watches* window.
2. Right-click on the gray bar and click **Add**, as illustrated in Figure 47.

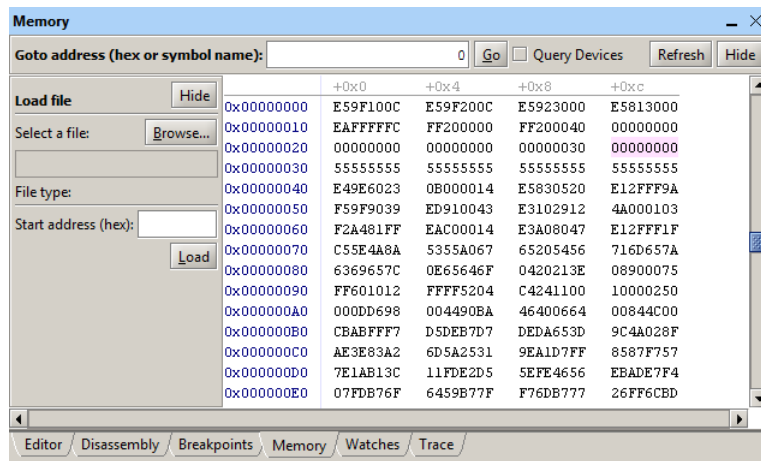


Figure 45. The Load file panel.

```
00,11,22,33
1044,2055,3066,4077
10000088,20000099,300000aa,400000bb
1,-1,2,-2
```

Figure 46. A Delimited hexadecimal value file.

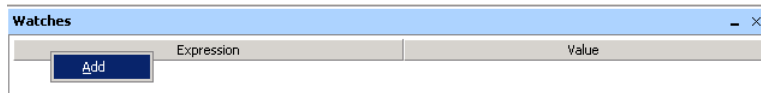


Figure 47. The Watches window.

3. The *Edit Watch Expression* window will appear, as shown in Figure 48. The desired watch expression can then be entered, using the syntax indicated in the window. In the figure, the expression `mem32(sp)` is entered, which will display the value of the data word at the current stack pointer address.
4. Click Ok. The watch expression and its current value will appear in the table. The number format of a value displayed in the watch expression window can be changed by right-clicking on the row for that value. As the program being debugged is repeatedly run, the watch expression will be re-evaluated each time and its value will be shown in the table of watch values.

## 10.4 The GDB Server Panel (Advanced)

To see this panel, select the GDB Server panel of the Monitor Program. This window will display the low-level commands being sent to the GDB Server, used to interact with the HPS system on the DE1-SoC board. It will also show the responses that GDB sends back. The Monitor Program provides the option of typing GDB commands and sending them to the debugger. Consult online resources for the GDB program to learn what commands are available.



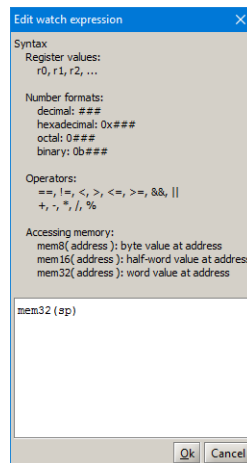


Figure 48. The Edit Watch Expression window.

## 11 Appendix B - Using Device Drivers (Advanced)

Intel's development environment for Nios II programs provides a facility for using device driver functions for the I/O devices in a hardware system. This facility, which is called the *hardware abstraction layer* (HAL), is supported by the Monitor Program. Using device driver functions is not recommended for beginning students, and is intended for more advanced users.

To see an example of code that uses device driver functions create a project called *Monitor\_HAL*. Select the DE1-SoC Computer system. Set the program type to Program with Device Driver Support, check Include a sample program with the project, and select the sample program named *Media*. The source file for this sample program is called *media.c*. When creating this project, the New Project Wizard does not display the screen for choosing memory settings, such as the one in Figure 33. This is because the HAL automatically chooses the necessary memory settings for projects that make use of device drivers.

The *media.c* program communicates with I/O devices by making calls to device driver functions, rather than using memory-mapped I/O as has been done in previous examples in this tutorial. To see some examples of such function-calls, examine the source code in the file *media.c*. It calls device driver functions for the audio devices in the DE1-SoC Computer, the VGA output port, the PS/2 port, and parallel ports. The device driver functions for each of these devices are defined in *include files* that are specified at the top of the *media.c* file. The set of device driver functions provided for an IP core is specified as part of the documentation for that IP core.

Compile and load the program by using the command **Actions > Compile & Load**. The Monitor Program automatically compiles both the *media.c* program and all device drivers that it uses. In subsequent compilations of the program, only the *media.c* code is compiled.

Run the program. It performs the following:

- Records audio for about 10 seconds when KEY[1] is pressed. LEDR[0] is lit while recording.

- Plays the recorded audio when KEY[2] is pressed. LEDR[1] is lit while playing.
- Draws a blue box on the VGA display, and places a text string inside the box.
- Shows on the HEX displays the last three bytes of data received from a device connected to the PS/2 port.

More details about developing programs with the Monitor Program that use HAL device drivers can be found in the tutorial *Using HAL Device Drivers with the Intel FPGA Monitor Program*, which is available in the University Program section of Intel's website. More information about HAL can be found in the *Nios II Software Developer's Handbook*.

## 12 Appendix C - Running Multiple Instances of the Monitor Program (Advanced)

In some cases it may be useful to run more than one instance of the Monitor Program on the same computer. For example, the selected system may contain more than one processor. An instance of the Monitor Program is required to run and debug programs on each available processor. As described in Section 3.1, it is possible to select a particular processor in a system via the **PROCESSOR** drop-down list in the *New Project Wizard* and *Project Settings* windows.

The Monitor Program uses the *GDB Server* to interact with the HPS system, and connects to the GDB Server using TCP ports. By default, the Monitor Program uses port 2399 as the base port, and to connect to each processor in a system the Monitor Program will attempt to use a port located at a fixed offset from this base port. For example, a single system consisting of four processors corresponds to ports 2399-2402.

However, the Monitor Program does not detect any ports that may already be in use by other applications. If the Monitor Program fails to connect to the GDB Server due to a port conflict, then the base port number can be changed by creating an environment variable called `ALTERA_MONITOR_DEBUGGER_BASE_PORT` and specifying a different number.

It is also possible to have more than one board connected to the host computer. As described in Section 3.1, a particular board can be selected via the **Host connection** drop-down list in the *New Project Wizard* and *Project Settings* windows. In this case, a separate instance of the Monitor Program is needed to interact with each processor on each physical board. By default, the Monitor Program assumes a maximum of ten Nios II processors per board. This means that ports 2399-2408 are used by the Monitor Program for the first board connected to the computer, and the first processor on the second board will use port 2409.

It is possible to specify a different value for the maximum number of processors per Nios II hardware system by creating an environment variable called `ALTERA_MONITOR_DEBUGGER_MAX_PORTS_PER_CABLE` and specifying a different number. This is useful if a system contains more than ten Nios II processors. It is also useful if a port conflict exists and none of the systems contain ten or more processors. In this case, decreasing this number (in conjunction with changing the base port number) may provide a solution.

## 13 Appendix D - Examining the Instruction Trace (Advanced)

An instruction trace is a hardware-level mechanism to record a log of all recently executed instructions. The *Nios II JTAG Debug Module* has the instruction trace capability, but only if a Level 3 or higher debugging level is selected in the *SOPC Builder* or *Platform Designer* configuration of the JTAG Debug Module (See the *Nios II Processor Reference Handbook*, available from Intel, for more information about the configuration settings of the JTAG Debug Module). If the required JTAG Debug Module is not present, a message will be shown in the Info & Errors window of the Monitor Program after loading a program, to indicate that instruction trace is not available.

The *Trace* feature is disabled by default. To enable the trace feature, go to the Trace window, right click inside the window, then select **Enable Trace**. To view the instruction trace of a program, go to the Trace window after pausing the program during execution. As shown in Figure 49, the instructions are grouped into different colored blocks and labeled alphabetically. The number of times each instruction block is executed is shown beneath its alphabetical label.

Address	Instruction	Label
0x000004c8	stw r15, 0(r16)	E
0x000004cc	ldw r14, 0(r16)	
0x000004d0	bne r14, r15, 0x80 (0x00000554: SHOW_ERROR)	F
0x000004d4	addi r16, r16, 0x4	
0x000004d8	lge r17, r16, -0x28 (0x000004b4: MEM_LOOP)	G
MEM_LOOP:		
0x000004b4	beq r17, zero, 0x4 (0x000004bc: SKIP_NOP)	H
0x000004b8	add zero, zero, zero	
0x000004bc	call 0x0000015e (0x00000578: UPDATE_HEX_DISPLAY)	I
UPDATE_HEX_DISPLAY:		
0x00000578	addi sp, sp, -0x24	
0x0000057c	stw ra, 0(sp)	
0x00000580	stw fp, 4(sp)	
0x00000584	stw r15, 8(sp)	
0x00000588	stw r16, 12(sp)	
0x0000058c	stw r17, 16(sp)	
0x00000590	stw r18, 20(sp)	
0x00000594	stw r19, 24(sp)	
0x00000598	stw r20, 28(sp)	
0x0000059c	stw r21, 32(sp)	
0x000005a0	addi fp, sp, 0x24	
0x000005a4	orhi r15, zero, 0x0	
0x000005a8	addi r15, r15, 0xaa4	
0x000005ac	ldw r16, 4(r15)	
0x000005b0	orhi r17, zero, 0x0	
0x000005b4	addi r17, r17, 0x7	
0x000005b8	orhi r15, zero, 0x0	
0x000005bc	addi r15, r15, 0xac4	
0x000005c0	orhi r19, zero, 0x0	

Figure 49. The Trace window.

Right-clicking anywhere in the Trace window brings up several options, as shown in Figure 50. The Trace feature can be turned on or off by selecting the **Enable trace** or **Disable trace** options. It is also possible to toggle the *debug events* in the trace on or off by selecting **Show debug events**, or clear current trace sequences by selecting **Clear trace sequences**.

Running the program using the **Actions > Continue** or **Actions > Single Step** commands will show up in the trace sequence as *debug events* after each time the program pauses execution, as shown in Figure 51.

If the *pc* value is changed before the program continues to run, the Monitor Program will insert a gap sequence in the trace, as shown in Figure 52. The **Actions > Restart** command will set the *pc* value back to the initial starting

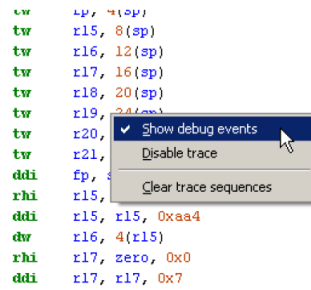


Figure 50. Right-click options in the Trace window.

Trace				P	M	B	Q
0x000006f4	andhi	r18, r17, 0xffff					
0x000006f8	beg	r18, zero, 0x24 (0x00000720: NO_CHAR)					x21
NO_CHAR:							
0x00000720	ldw	ra, 0(sp)					
0x00000724	ldw	fp, 4(sp)					AF
0x00000728	ldw	r15, 8(sp)					
0x0000072c	ldw	r16, 12(sp)					
FORCED HALT							
SINGLE-STEP							
0x00000730	ldw	r17, 16(sp)					AG
SINGLE-STEP							
0x00000734	ldw	r18, 20(sp)					AH
SINGLE-STEP							
0x00000738	ldw	r19, 24(sp)					AI
SINGLE-STEP							
0x0000073c	addi	sp, sp, 0x1c					AJ
SINGLE-STEP							
0x00000740	ret						AK
CONTINUE							
0x000004c8	stw	r15, 0(r16)					
0x000004cc	ldw	r14, 0(r16)					D
0x000004d0	bne	r14, r15, 0x80 (0x00000554: SHOW_ERROR)					
0x000004d4	addi	r16, r16, 0x4					E
0x000004d8	bge	r17, r16, -0x28 (0x000004b4: MEM_LOOP)					
MEM_LOOP:							
0x000004b4	beg	et, zero, 0x4 (0x000004bc: SKIP_NOP)					F
0x000004b8	add	zero, zero, zero					
0x000004bc	call	0x0000015e (0x00000578: UPDATE_HEX_DISPLAY)					G
UPDATE_HEX_DISPLAY:							
0x00000578	addi	sp, sp, -0x24					H

Figure 51. The Trace window with various debug events.

address. The *pc* value can also be arbitrarily set by double clicking its value in the Registers window and editing its hexadecimal value.

Breakpoints in the program will also show up in the trace sequence as a *debug event* each time the breakpoint condition is met, as illustrated in Figure 53.

### 13.0.1 Note About Tracing Interrupt Sequences

It is possible that interrupt sequences are happening in the program, yet do not show up in the Trace window in the Monitor Program. This is because the instruction blocks shown in the trace sequence are actually sampled from a window of time over the entire program execution. As a result, the interrupt sequences may not be included in the

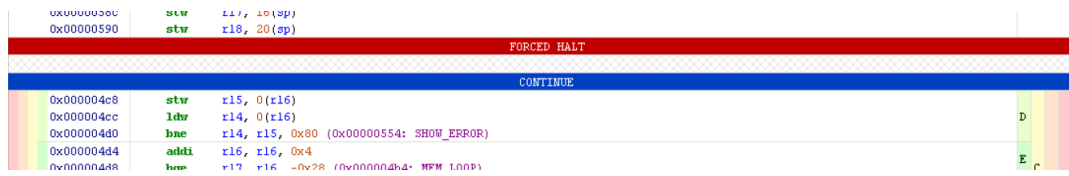


Figure 52. A gap sequence in the instruction trace.

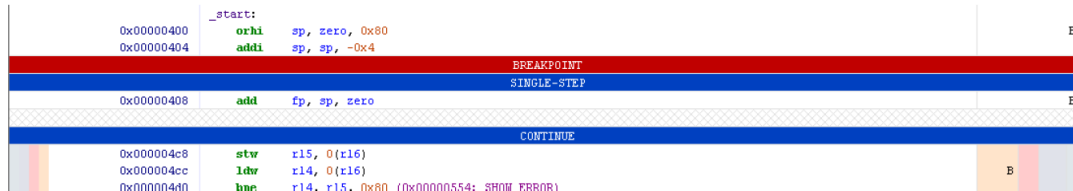


Figure 53. A breakpoint in the instruction trace.

sample of instruction blocks displayed in the Monitor Program. One way to deal with this problem is to trigger a breakpoint after an interrupt finishes executing.

## 14 Appendix D - Configuration File

The Monitor Program configuration file allows default values to be set for project creation. The monitor program searches \$(UniversityProgramRoot)/amp.config for the configuration file, where UniversityProgramRoot is the path to the University Program directory in the Quartus installation.

For example C:/intelFPGA/16.1/University\_Program/amp.config.

To change the default path to the configuration file, add the following command line argument when running the Monitor Program: `--config-file=<Path to File>`

Table 2 summarizes the configuration options available in the Monitor Program.

The configuration file uses white space or an equal sign as a delimiter, for example: `flag option` or `flag=option`. Where *flag* is one of the values in the first column of Table 2 and *option* is the default value for that flag. Number signs (#) can be used to add comments to the configuration file. Lines starting with the symbol will not be processed with the configuration file. Boolean values can use integers or case insensitive strings. Options of 'false', 'no' and '0' will all produce a false Boolean, any other values will produce a true Boolean.

Flag	Explanation
project_name	The project name.
project_path	The new project directory path.
architecture	The architecture.
system	The default sample system to be used (ex. DE1-SoC Computer)
c_compiler_flags	C Compiler flags
c_linker_flags	C Linker flags
use_small_c_lib	Boolean to use the small C Library (Nios II)
emulate_instr	Boolean to emulate unimplemented instructions
include_system_info_file	Boolean whether to include the system info header by default.
answer_for_reload_file	<i>yes</i> or <i>no</i> option to bypass the file reload dialog when files are edited outside the program. If undefined, the dialog will be shown.

Table 2. Configuration Flags and Default Options.

Copyright © Intel Corporation. All rights reserved. Intel, the Intel logo, Altera, Arria, Avalon, Cyclone, Enpirion, MAX, Nios, Quartus and Stratix words and logos are trademarks of Intel Corporation or its subsidiaries in the U.S. and/or other countries. Intel warrants performance of its FPGA and semiconductor products to current specifications in accordance with Intel's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Intel assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Intel. Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.

\*Other names and brands may be claimed as the property of others.