

Altera Debug Client

This tutorial presents an introduction to the Altera Debug Client, which can be used to compile, assemble, download and debug programs for Altera's Nios II processor. The tutorial gives step-by-step instructions that illustrate the features of the Altera Debug Client.

The process of downloading and debugging a Nios II application requires the presence of an actual FPGA device to implement the Nios II processor. For the purposes of this tutorial, it is assumed that the user has access to the Altera DE2 Development and Education board connected to a computer that has Quartus II (version 5.1 or higher) and Nios II software installed.

The screen captures in the tutorial were obtained using version 2.2 of the Altera Debug Client; if other versions of the software are used, some of the images may be slightly different.

Who should use the Altera Debug Client

The Altera Debug Client is intended to be used in an educational environment by professors and students. For commercial system and application development, Altera's *Nios II Integrated Development Environment* should be used.

Contents

1	Installing the Altera Debug Client	3
2	Starting the Altera Debug Client	6
3	Configuring a Nios II System	7
4	Configuring a Nios II Program	9
5	Compiling and Loading the Program	10
6	Running the Program	12
7	Using the Disassembly Window	12
8	Single step	14
9	Using Breakpoints	15
10	Examining and Changing Register Values	17
11	Examining and Changing Memory Contents	19
12	Setting a Watch Expression	23
13	Examining the Instruction Trace	24
14	Using Configuration Files	26
15	Using the Terminal	27

Introduction

The Altera Debug Client is a software application that runs on a host PC connected to a Nios II System. It allows the user to compile or assemble Nios II applications, download the application to the Nios II system and then debug the running application. The Debug Client provides functionality that allows the user to:

- Examine and modify register and memory contents.
- Disassemble the machine code present in any memory region.
- Single step through each assembly language instruction in the program.
- Set breakpoints that stop the execution of a program when certain instructions are reached or when certain data addresses are accessed.
- Set watch expressions and watch their values at different points in the execution of the program.
- Examine a graphical view of an *instruction trace* that records the set of recently executed instructions.
- Perform terminal input/output via the JTAG UART component.

1 Installing the Altera Debug Client

To install the Altera Debug Client, proceed as follows:

1. Use **Microsoft Windows Explorer** to open the folder *Altera_Debug_Client* on the DE2 CD-ROM. As shown in Figure 1, the folder will contain a single executable file named *setup.exe*.

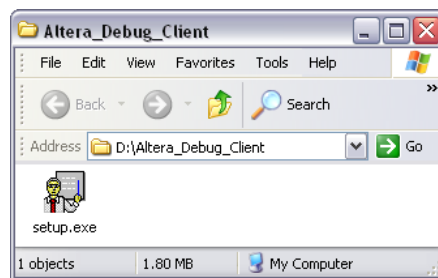


Figure 1. Altera Debug Client installer on DE2 System CD.

2. Double-click on this *setup.exe* executable file. This will bring up the first screen of the installer as illustrated in Figure 2. Click on the **Next** button and proceed to the next step.
3. The installer will display the License Agreement; click **I Agree** to continue.
4. On the next screen, you can choose which components to install, as shown in Figure 3. Leave both components checked to install the Debug Client program files along with the tutorial and sample files needed for this tutorial. Click **Next** to continue.
5. The next screen will allow you to select a destination folder into which the installer will copy the Debug Client tutorial and its sample files. Click **Browse** to select a directory in your system. The directory that you specify in this step will be referred to as *<TUTORIAL_FILES>* throughout the remainder of this tutorial. For example, in Figure 4, the installation directory is *d:\Debug_Client_Tutorial*. Note that you do not have the option of specifying the location where the Debug Client program files are stored; this location is displayed in the window of Figure 4, and is determined by the installation setup for the *Nios II Embedded Design Suite*. Click **Next** to proceed to the next step.

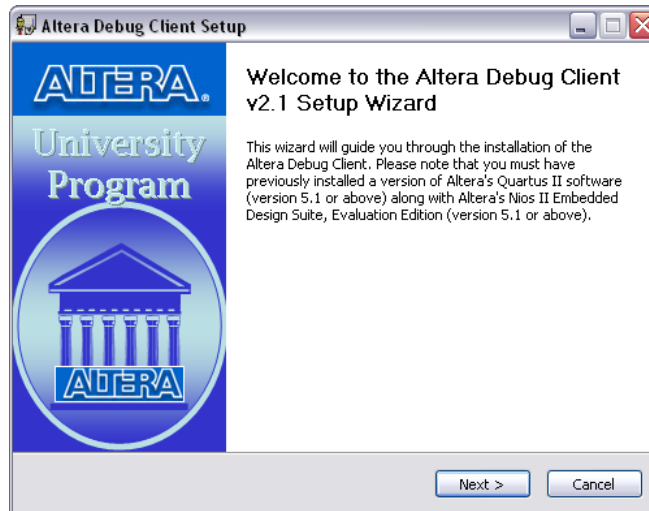


Figure 2. Altera Debug Client Install Wizard.

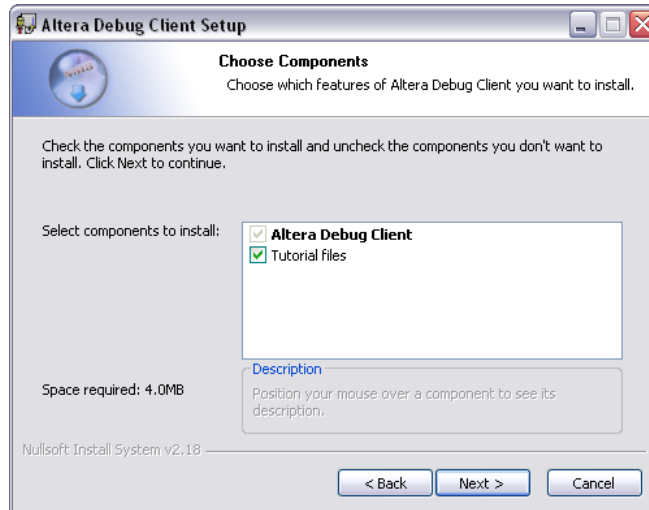


Figure 3. Choosing which components to install.

6. The installer is now ready to begin copying files. Click **Install** to install the Debug Client. During the installation process, you will be asked if you would like a shortcut to the Altera Debug Client to be placed on your **Windows Desktop**. Answering yes will install an icon similar to the one shown in Figure 5 on your desktop.
7. Assuming that the install was successful, the screen shown in Figure 6 will be displayed. Click on the **Finish** button to complete the installation. Should an error occur, a window will suggest the appropriate action. Errors include:
 - Quartus II Software is not installed or the Quartus II version is too old.
 - Nios II SDK Software is not installed or the version is too old.

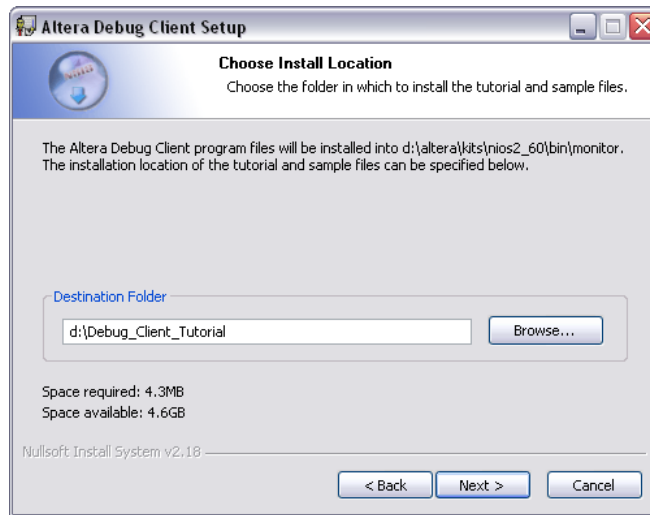


Figure 4. Specifying the installation location of the tutorial and sample files.



Figure 5. Altera Debug Client desktop icon.

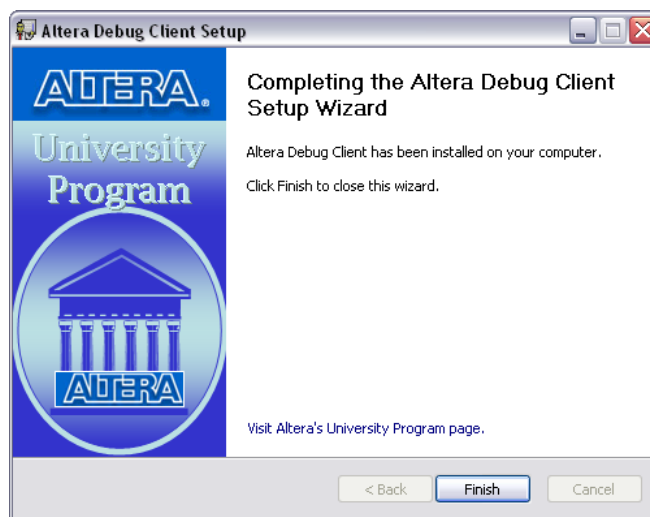


Figure 6. Altera Debug Client installation finished.

2 Starting the Altera Debug Client

Before starting the Altera Debug Client, a Nios II system has to be downloaded onto the DE2 Development and Education board using the Quartus II software. This tutorial assumes that the Nios II system located at `<TUTORIAL_FILES>\example\hw\example.sof` is downloaded.

The tutorial called *Introduction to the Quartus II Software* explains how to download a circuit onto the FPGA on the DE2 board and the tutorial called *Introduction to the SOPC Builder* tool shows you how to create Nios II systems. These tutorials are provided on the DE2 System CD and in the University Program section of Altera's web site and must be completed before using this tutorial.

If you have chosen to install a Windows Desktop Shortcut, you can start the Altera Debug Client by double clicking on the desktop icon. The Debug Client can also be started from the Windows Start Menu by following the links from **Altera > Altera Debug Client > Altera Debug Client**.

Another method to start the Debug Client is to use the *Nios II Command Shell*. Start the *Nios II Command Shell* from the Windows Start Menu by following the links from **Altera > Nios II EDS 6.0 > Nios II Command Shell** and enter the command `altera-debug-client`.

After startup, the Debug Client will appear as shown in Figure 7.

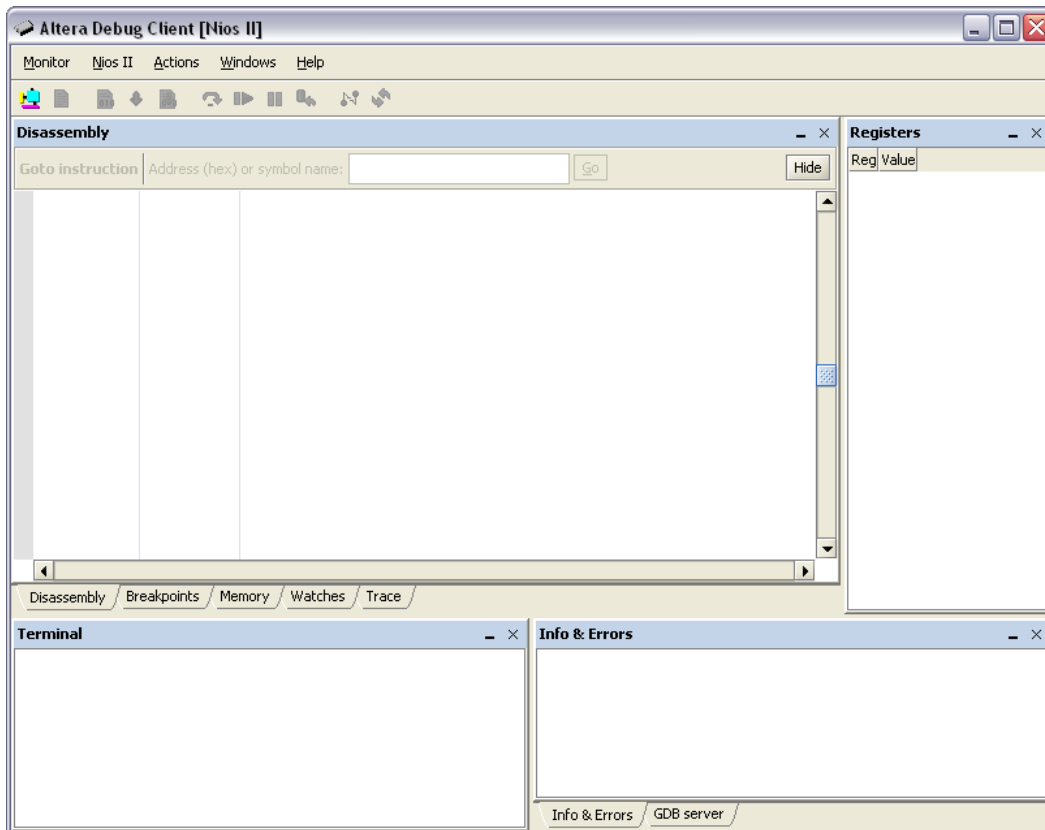



Figure 7. The Altera Debug Client at startup.

3 Configuring a Nios II System

Nios II systems have a user-configurable architecture. The designer may choose from a variety of peripherals and memory options in Altera's SOPC Builder. The Altera Debug Client needs information describing the Nios II system that is being targeted in order to compile and load programs for the system. Systems created by Altera's SOPC Builder are described by a *.ptf* file located in the Quartus II project's main directory. This file contains information about all the peripherals connected to the Nios II processor, including the system memory map.

To describe the Nios II system to the Debug Client, click the **Nios II > Configure system...** menu item or click the  toolbar button. The **Nios II System Configuration** window will appear, similar to Figure 8. Proceed with the steps below to configure the example system.

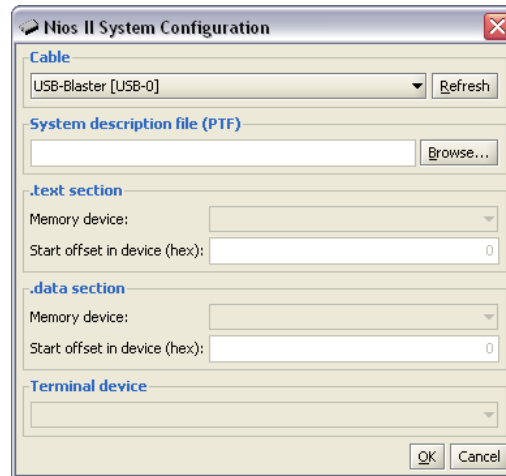


Figure 8. Initial Nios II System Configuration window.

1. Select the cable from the **Cable** drop-down list that corresponds to the DE2 board. The DE2 board is connected via a USB-Blaster cable.
2. Click **Browse...** to display a window similar to Figure 9. Navigate to `<TUTORIAL_FILES>\example\hw\system.ptf`, which is the system description file for the example system. Select *system.ptf* and click **Load**.

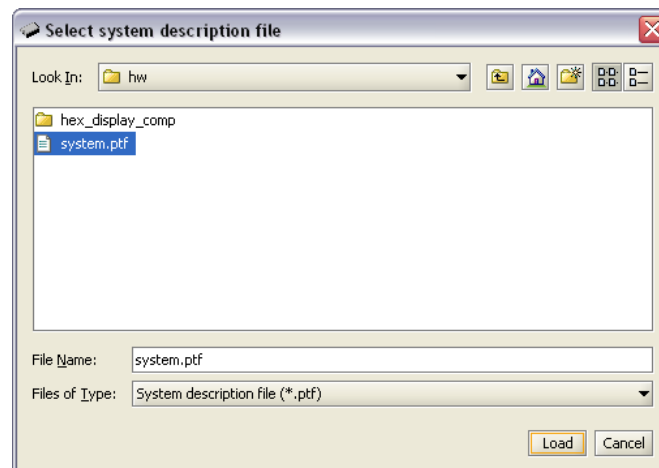


Figure 9. Select system description file window.

- In the System Configuration window in Figure 8, click **Load**. The fields in the **.text section** and **.data section** will now be enabled.
- Nios II programs are compiled into an *Executable and Linking Format* (ELF) file. This format supports *sections*, which can be used to divide a program into multiple parts, such as an executable code section and a data section. Each section has its own set of attributes, including memory location, alignment, and size. The partitioning of a program into different sections is performed by the linker, which receives this information via either a linker script or linker invocation arguments. The Altera Debug Client allows the user to specify the linker arguments for two sections of the program, as described below. Each section will be placed at the address value of *base address of memory device + start offset*. If a start offset is not specified, its default value will be zero.

The `.text` section contains the program machine code. The **.text section** of the System Configuration window is used to place the machine code at different memory locations. Use the following settings:

- Memory device: `memory/s1 (0h - 7ffffh)`, corresponding to the on-chip memory of the Nios II system (this is the default and only choice in the example `system.ptf` that we are using)
- Start offset in device: `1000`, corresponding to a 4096-byte offset into the on-chip memory to start the `.text` section

The `.data` section contains program data, such as variables or constants. The **.data section** of the System Configuration window is used to place the data at different memory locations. Use the following settings:

- Memory device: `memory/s1 (0h - 7ffffh)`, corresponding to the on-chip memory of the Nios II system (this is the default and only choice in the example `system.ptf` that we are using)
- Start offset in device: `1000`, corresponding to a 4096-byte offset into the on-chip memory to start the `.data` section. Note that the `.text` and `.data` offsets are the same, which would lead to an overlap of the two sections in memory. However, because the two offsets are exactly the same, the linker script used to produce the program executable will instead automatically place the `.data` section immediately after the `.text` section.

- If there is more than one terminal device available in the system, one can be chosen from the **Terminal device** list to be connected to the terminal in the Debug Client. In this example, `jtag_uart` is the only terminal device present.
- Your system configuration should appear as shown in Figure 10. Click **OK** to save the system configuration.

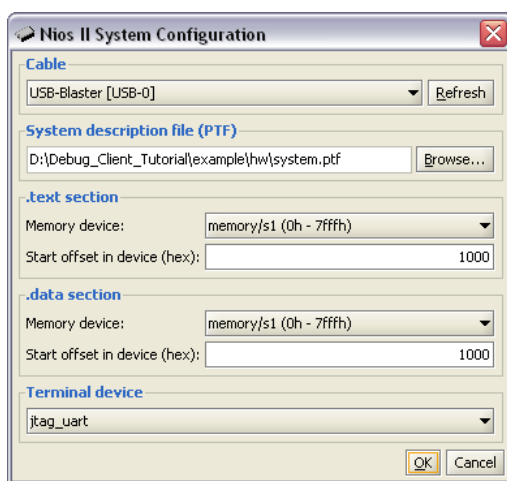



Figure 10. Nios II System Configuration window configured for the example system.

4 Configuring a Nios II Program

Before compiling and loading a program with the Debug Client, the desired source files must be specified. To configure the program source files for this part of the tutorial, click the **Nios II > Configure program...** menu item or click the  toolbar button. The **Nios II Program Configuration** window will appear, similar to Figure 11.

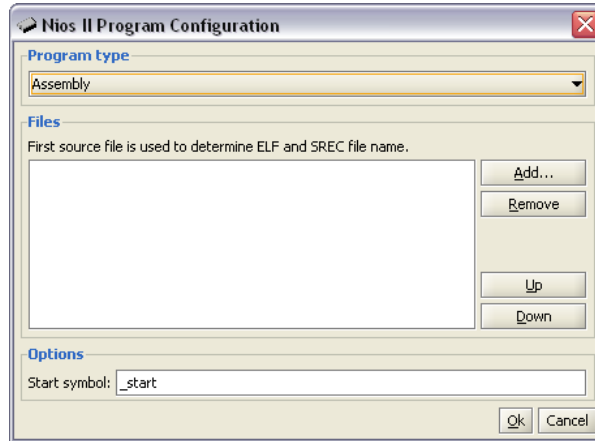


Figure 11. Initial Nios II Program Configuration window.

Proceed with the following steps to configure the program:

1. For this example, the **Program type** will be *Assembly*, so the field can be left at its default value. The Debug Client also supports C programs and programs already compiled in ELF or SREC formats.
2. Click **Add...** to display a window similar to Figure 12. The source file for this part of the tutorial is located at `<TUTORIAL_FILES>\example\sw\main_tutorial_src\main_tutorial.s`. Choose that file and click **Select**.

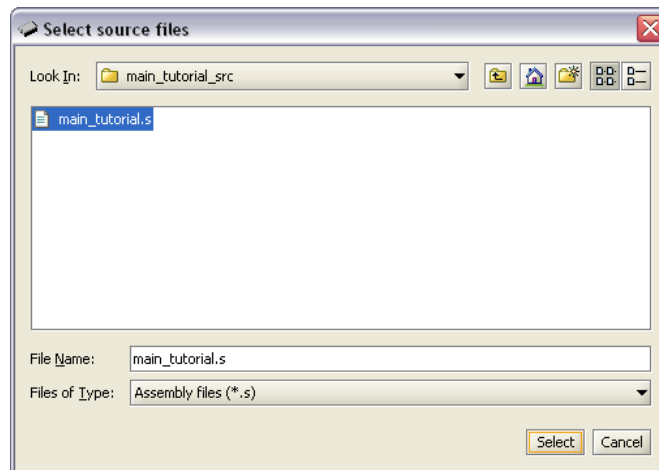


Figure 12. Select source files window.

3. The *start symbol* of an assembly-language program identifies the label that corresponds to the first instruction of the program. The default start symbol is `_start` and this is the symbol that is used in `main_tutorial.s`.
4. Your program configuration should appear as depicted in Figure 13. Click **OK** to save the program configuration.

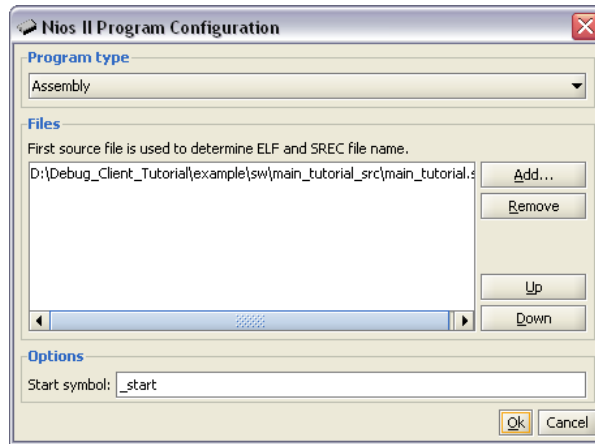






Figure 13. Nios II Program Configuration window configured for the example program.

5 Compiling and Loading the Program

After successfully configuring the system and program, the program can be compiled and downloaded onto the DE2 board. There are three different commands that can be used to compile and/or load a program:

- **Actions > Compile** menu item or  toolbar button:
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 same directory as the first source file.
- **Actions > Load** menu item or  toolbar button:
Loads the compiled SREC file onto the board and begins a debugging session in the Debug Client. Loading progress messages are displayed in the **Info & Errors** window.
- **Actions > Compile & Load** menu item or  toolbar button:
Performs the operations of both compilation and loading.

In this example, the program has not yet been compiled, so it cannot be loaded (the **Load** option is disabled). Click the **Actions > Compile & Load** menu item or click the  toolbar button 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 15.

After successfully completing this step, your Debug Client display should look similar to Figure 16. At this point, the program is paused at its first instruction.

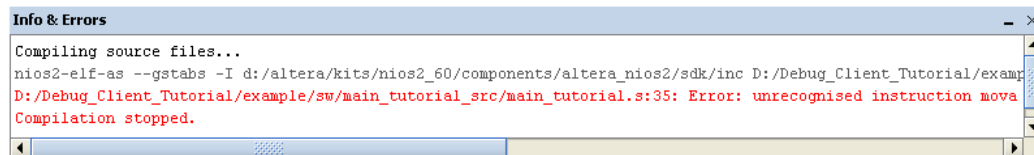


Figure 14. A compiler error message.

Compilation errors

During the process of developing software, you will likely encounter compilation errors. Error messages from the Nios II assembler or from the C compiler are displayed in the **Info & Errors** window. An example of a compiler error message is shown in Figure 14. The file name and the line number corresponding to the source of the error are displayed, in addition to an indication of the cause of the error. You may be able to deduce the real cause of the error from the message or you may need to do some additional searching.

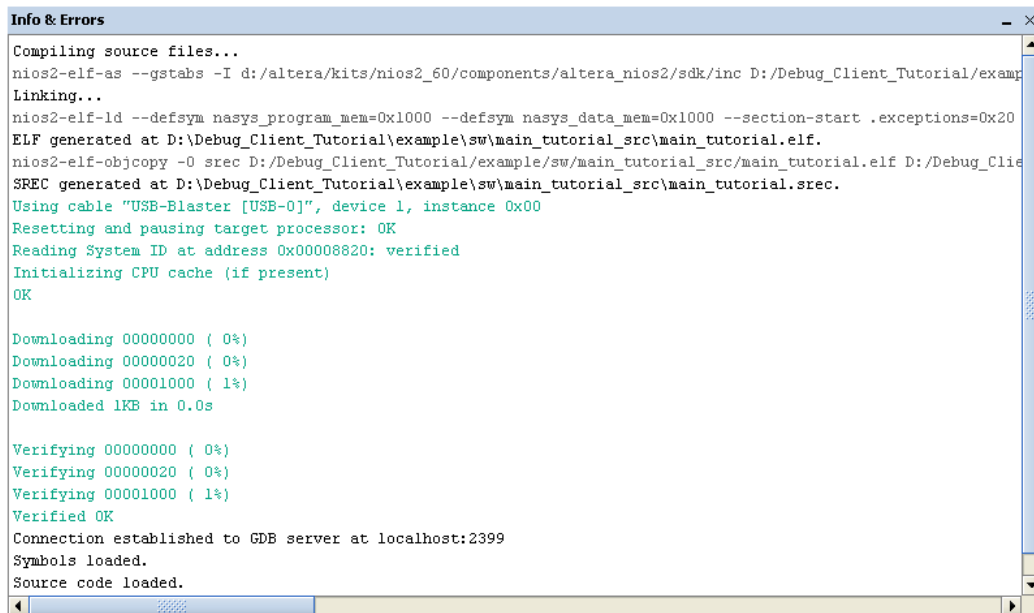


Figure 15. Compilation and loading messages (the **Info & Errors** window has been maximized).

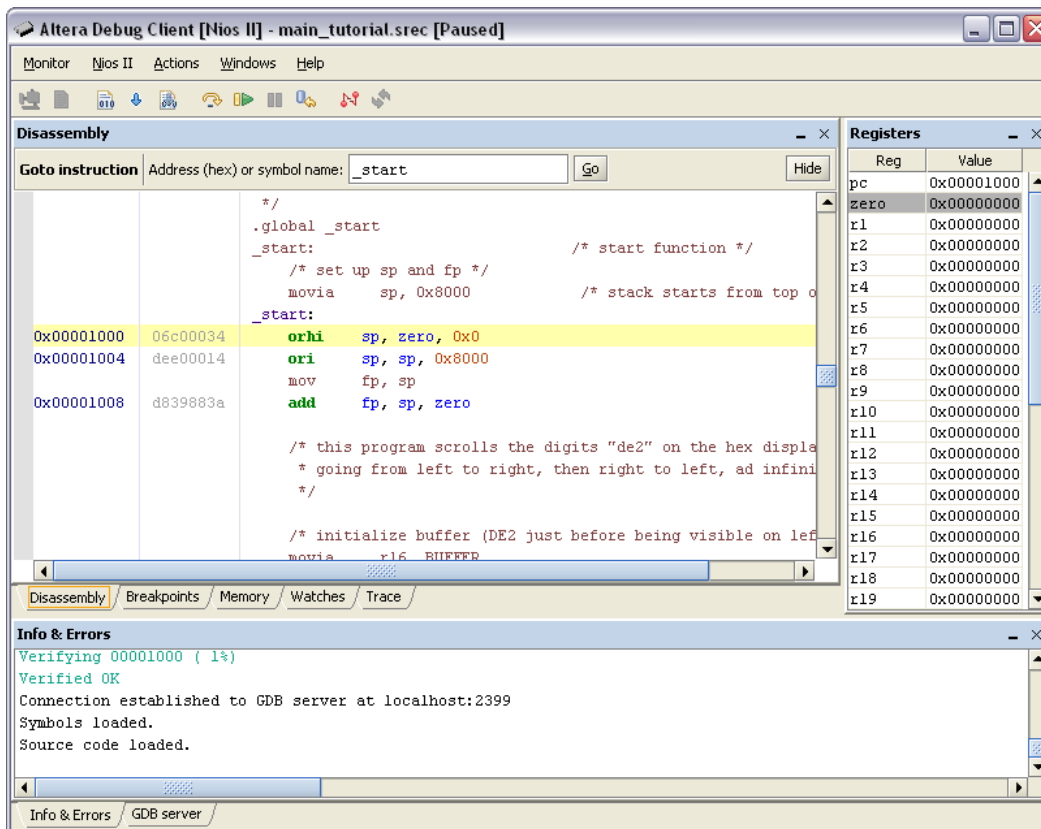




Figure 16. The Altera Debug Client window after loading the example program.

6 Running the Program

As mentioned at the end of the previous section, the program is paused at its first instruction after it has been loaded. To run the program, click the **Actions > Continue** menu item or click the  toolbar button. The sample program will continuously scroll the digits dE2 across the 7-segment displays on the DE2 board.

The **Continue** command runs the program until something halts the processor's execution, such as a breakpoint or a forced user halt. To force the program to halt, click the **Actions > Stop** menu item or click the  toolbar button; the processor will stop at the instruction to be executed next.

When the program is stopped, all debugging windows are updated with new data. As seen in Figure 17, the **Disassembly** window highlights the next instruction to be executed in yellow and the **Registers** window highlights register values that have changed since the last program stoppage in red. The other windows in the Debug Client are also updated, which will be shown in later parts of this tutorial.

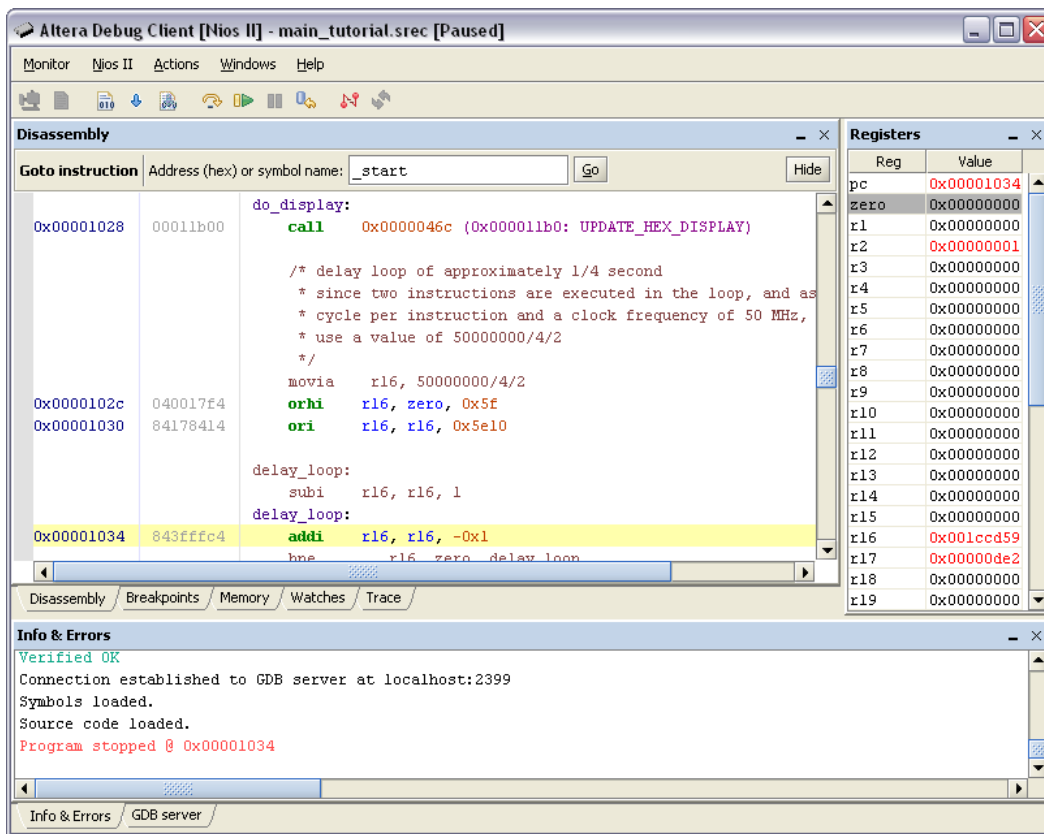


Figure 17. Disassembly window after the program has been stopped.

7 Using the Disassembly Window

The **Disassembly** window displays human-readable machine code by interpreting the memory values as encoded instructions. As shown in Figure 17, there are three columns in the window. The left-most column gives the memory address corresponding to the instruction displayed on that line. The middle column displays the 32-bit instruction word corresponding to the machine encoding of the instruction. The right-most column displays the human-readable instruction along with the corresponding source code. For example, in Figure 17, the four instructions located at memory addresses 0x00001028, 0x0000102c, 0x00001030, and 0x00001034 have been disassembled.

The Disassembly window can be configured to display less information on the screen, such as not showing the source code from the .s assembly language file or not showing the machine encoding of the instructions. These

settings can be changed by right-clicking on the window and selecting the appropriate menu item, as shown in Figure 18. The display in the window also uses a color-coded scheme, as detailed in Table 1.

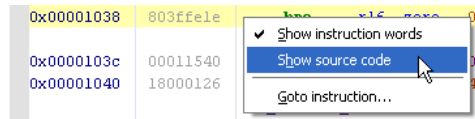


Figure 18. Pop-up menu to configure the display of the Disassembly window.

Color	Description
Brown	Source code
Green	Disassembled instruction name
Blue	Registers
Orange	Immediate & offset values
Dark blue	Address values & labels
Purple	Clickable link
Gray	Machine encoding of the instruction

Table 1. **Disassembly** window color-coded scheme.

By scrolling using the vertical scrollbar on the right side of the **Disassembly** window or by using a mouse scroll wheel, different regions of memory can be disassembled and displayed. It is also possible to scroll to a memory address or an instruction symbol directly by using the **Goto instruction** panel in the Disassembly window. Access this panel through the **Actions > Goto instruction...** menu item and enter a symbol name or an instruction address in hexadecimal format. The instruction address must be a multiple of 4 because every instruction address is aligned on a 32-bit word boundary. For example, enter `_start` or `1000` and press **Go**. The Disassembly window will show the `0x00001000` address as its first instruction, as shown in Figure 19, which also corresponds to the `_start` symbol. Also note that the instruction is highlighted with a pink background.

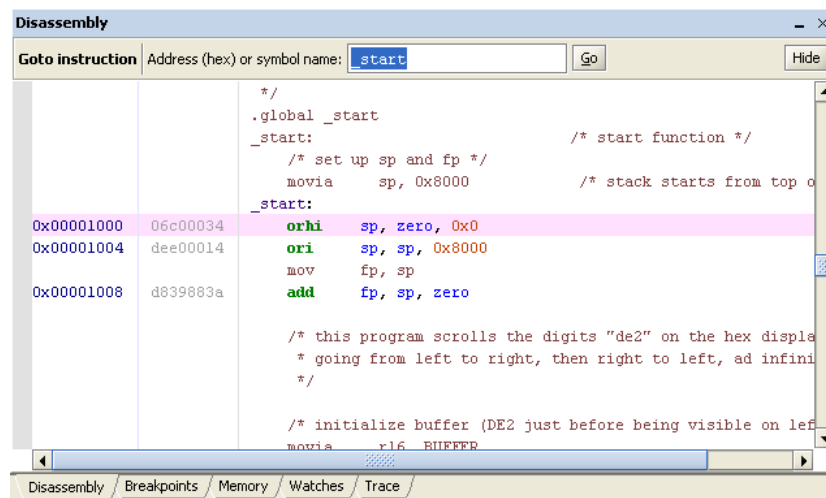


Figure 19. **Goto instruction** panel in the **Disassembly** window.

Register and memory values can be examined in the Disassembly window while the program is in a *Paused* state. This is done by hovering your mouse over a *register* or a *register + offset* in the window, as shown in Figure 20.

The Disassembly window also produces special clickable links in its display of branch instructions. Clicking one of these links will display the instruction that the processor would jump to if the branch was taken. Figure 21 shows one example of a link associated with a `call` instruction.

0x00001018	84400015	stw r17, 0(r16)
		stw zero, 4(r16)
0x0000101c	80000115	stw zero, 4(r16)

Figure 20. Examining a memory value in the **Disassembly** window.


0x00001028	00011b00	display: call UPDATE_HEX_DISPLAY
		display: call 0x0000046c (0x000011b0: UPDATE_HEX_DISPLAY)
		# delay loop of app. Goto instruction label 'UPDATE_HEX_DISPLAY' (0x000011b0)

Figure 21. A clickable link in the **Disassembly** window.


Assembly Language and Machine Instructions

The Disassembly window is a good place to examine what machine instructions are produced by the compiler from your assembly-language instructions or C code. The translation from assembly-language instructions to machine instructions is handled by the Nios II assembler and it is a transparent process to the programmer. However, it is beneficial to examine the disassembled code and compare it with the source code. This is readily done because the Debug Client displays both the source statements and the disassembled code in different colors. Observe that *pseudoinstructions* are implemented as different machine instructions. For example, the `movia` pseudoinstruction is implemented by the two instructions `orhi` and `ori`, as shown at the address values 0000102c and 00001030, respectively, in Figure 17.

8 Single step

Before discussing the single step action, it is convenient (for demonstration purposes) to restart execution of the program from the beginning. Click the **Actions > Restart** menu item or click the  toolbar button to restart the program. Notice that the *pc* register value displayed in the **Registers** window is 0x00001000 and the Disassembly window is highlighting that instruction.

The Debug Client has the ability to perform single step actions. Each single step consists of executing a single machine instruction and returning control to the Debug Client. Note that if the program being debugged was written in C, each individual single step will still correspond to one assembly language instruction generated from the C code. The ability to step through statements in the high-level source code is not supported by the Debug Client; however, Altera's *Nios II Integrated Development Environment* supports this advanced feature.

The single step action is invoked by clicking on the **Actions > Single step** menu item or by clicking on the  toolbar button. The instruction that is executed by the processor is the one highlighted by the **Disassembly** window before the single step.

Since the first step in this section was to restart the program, the first single step will execute the instruction at 0x1000, which will zero out the upper-16-bits of the *sp* register. Subsequent single steps will continue to execute one instruction at a time, in sequential order. Single stepping at a branch instruction may jump to a non-sequential instruction address if the branch is taken. You can observe this behavior by single stepping to 0x00001028, which is a `call` instruction. Single stepping at this instruction will set the *pc* value to 0x000011b0, which is the location of the `UPDATE_HEX_DISPLAY` label.

9 Using Breakpoints

Breakpoints are special conditions that are checked by dedicated hardware in the Nios II processor as the application is running in real-time. Breakpoints can be triggered in four different ways:

1. Program execution has reached a particular address
2. A read operation has been performed on a particular address
3. A write operation has been performed on a particular address
4. The processor has accessed the memory at a particular address

This section of the tutorial will cover the process of setting an instruction breakpoint (trigger type 1). There are two ways to set an instruction breakpoint. The first method can be used to set a simple instruction breakpoint as follows:

1. Switch to the **Disassembly** window.
2. Navigate to the instruction address that will have the breakpoint. For this example, display the `check_shift` instruction label.
3. Click on the gray bar to the left of the address `0000103c` (address value of the `check_shift` label) to set an instruction breakpoint at this location. See Figure 22 for an illustration of what a breakpoint in the Disassembly window looks like. Clicking the same location again will remove the breakpoint.

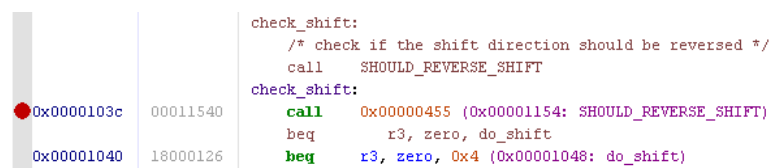


Figure 22. Setting a simple instruction breakpoint in the **Disassembly** window.

Once the instruction breakpoint has been set, run the program and the breakpoint should trigger when the `pc` register value equals `0x0000103c`. The Debug Client will look similar to Figure 23; notice the message in the **Info & Errors** window indicating that an instruction breakpoint has been triggered.

The second method of setting a breakpoint can be used for all four trigger types as follows:

1. Switch to the **Breakpoints** window, which is shown in Figure 24.
2. The breakpoint that was set earlier in the Disassembly window also appears in this window. The check mark beside the breakpoint can be used to enable or disable it. In this case, leave the check mark as it is.
3. Right-click on the header corresponding to the breakpoint type you want to add. For example, to set a breakpoint that triggers when the processor writes to a particular address, right-click on the write watchpoints table, as shown in Figure 25.
4. Click **Add**. A new entry will appear in the corresponding table. Enter the desired breakpoint address.

The Debug Client also supports a more advanced form of a breakpoint, called a *conditional breakpoint*. A conditional breakpoint is an instruction breakpoint that only triggers when the usual instruction breakpoint condition is met and an additional user-specified condition is met. For this example, you will use the same breakpoint from before but with the condition `r2 == 0`, which in this program's context is when the scroll direction is to the left. The process to set this conditional breakpoint is as follows:

1. Switch to the **Breakpoints** window.
2. For the breakpoint at `0000103c`, double-click on the table cell under the **Condition** column.

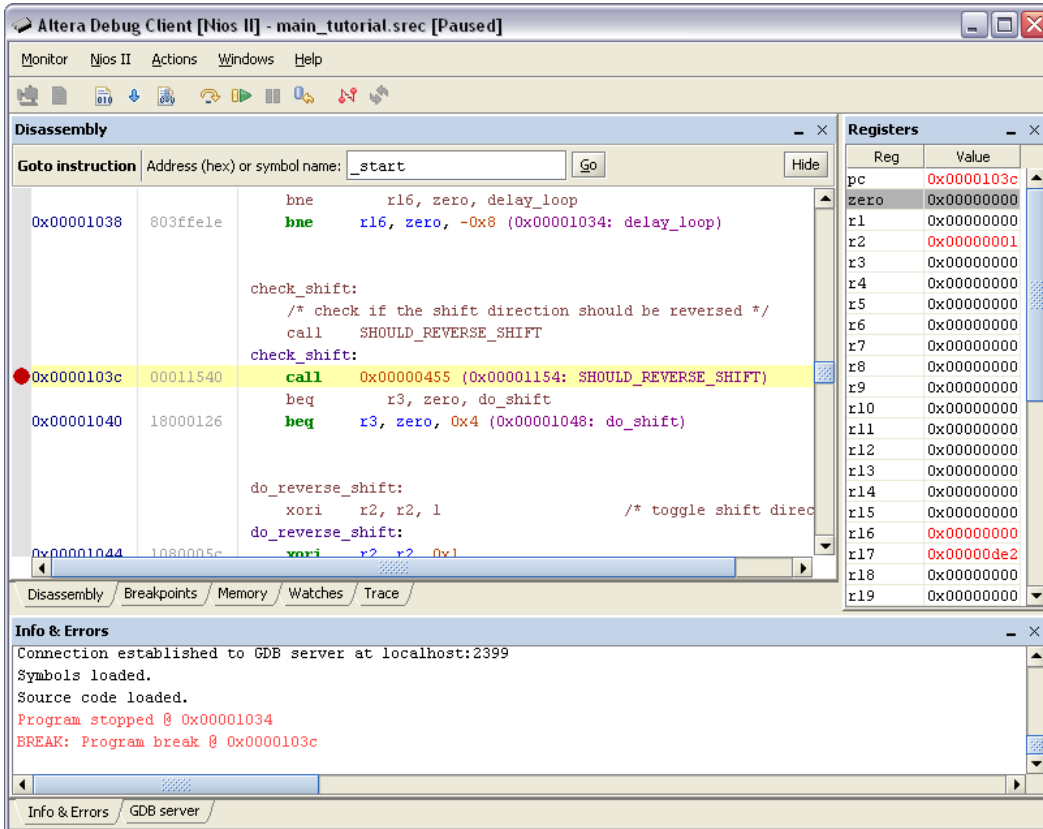


Figure 23. The Debug Client after the breakpoint has been triggered.

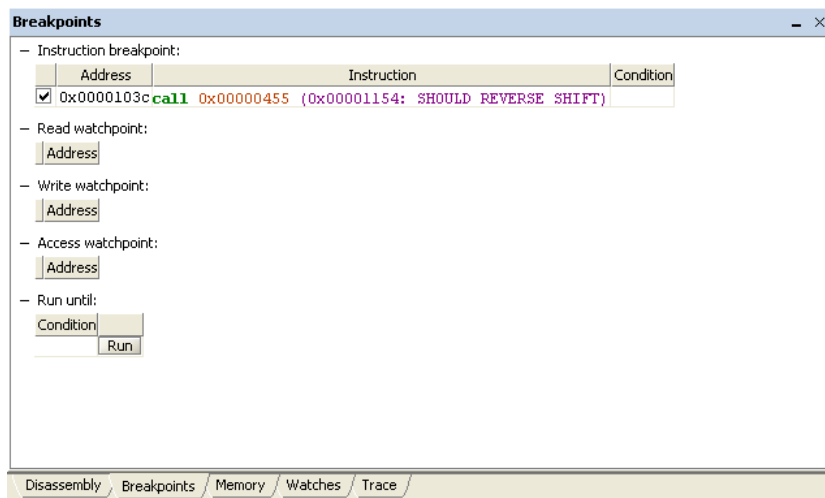


Figure 24. Breakpoints window.

3. The window in Figure 26 will appear. This window contains information about the syntax used to describe a condition. For this example type `r2 == 0`.
4. Press **Ok**. The **Condition** field for the breakpoint will now show the condition you entered.

The conditional breakpoint is now set. Run the program and as the `dE2` digits on the hexadecimal display disappear and the program begins to shift the digits to the left, the breakpoint will trigger. The **Info & Errors**

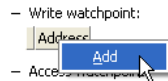


Figure 25. Adding a breakpoint, in this case a write watchpoint, in the **Breakpoints** window.

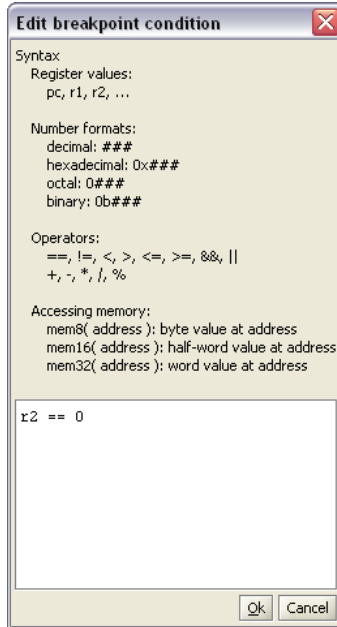


Figure 26. **Edit breakpoint condition** window.

window will again have a message about the cause of the breakpoint, including the trigger condition that was satisfied, as shown in Figure 27.

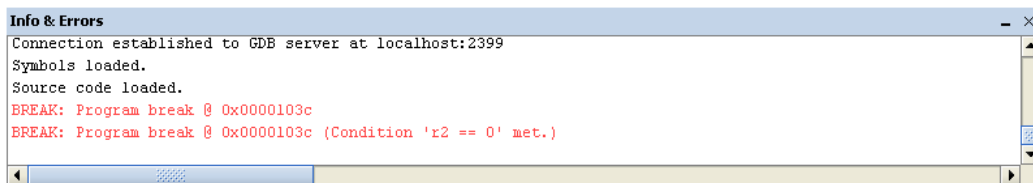


Figure 27. Message displayed in the **Info & Errors** window due to a triggered conditional breakpoint.

10 Examining and Changing Register Values

The **Registers** window displays the value of each register in the Nios II processor and allows the user to change most of the register values. The number format of the register values can be changed by right-clicking in the **Registers** window, as shown in Figure 28. You can choose among binary, octal, decimal, and hexadecimal representations in both signed and unsigned versions.

Every time program execution is stopped, the debugger updates all of the register values and highlights any changes in red. The user can also change the register values while the program is stopped.

As a demonstration of changing a register value, this section of the tutorial will set a breakpoint to halt the program when the hexadecimal display is showing . . . dE2 . (. represents a blank) and the scroll direction is to the right. When the breakpoint is triggered, you will toggle the shift direction via the **Registers** window and then resume program execution. The detailed steps are as follows:

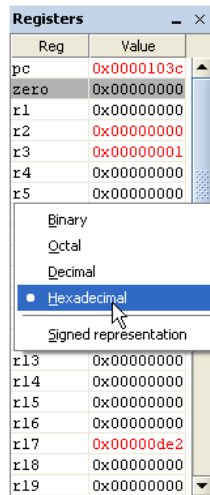


Figure 28. Changing the number format of the register values in the **Registers** window.

1. Switch to the **Breakpoints** window.
2. For the breakpoint at 0000103c, change the condition string to `mem32(0x11fc) == 0x0000de20 && r2 == 1`. The `mem32` syntax is used to read a 32-bit value from memory at the specified address. In this program, the 32-bit value at the address 0x11fc contains the value to be displayed that will be transferred to the hexadecimal display.
3. Resume program execution and wait for the breakpoint to trigger.
4. The **Registers** window should look similar to the left image of Figure 29. To edit the value of the `r2` register, which controls the scroll direction, double click on its value in the window. This will bring up a text box, as shown in the right image of Figure 29, and you can put in its new value of 0.

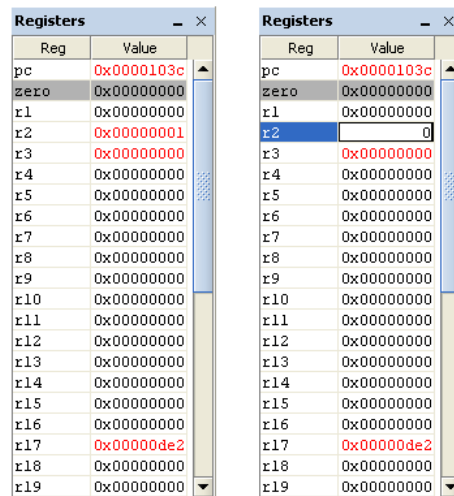


Figure 29. Register values after the breakpoint trigger; editing the value of the `r2` register.

5. Press **Enter** or click away from the text box to apply the change.
6. Resume the execution of the program and you should see that the value on the hexadecimal display is scrolling left now.

- Eventually the conditional breakpoint that was set in step 2 will trigger again. Continue on to the next section of the tutorial when this occurs.

11 Examining and Changing Memory Contents

The **Memory** window displays the contents of the system's memory space and allows the user to edit its values. The memory display will look similar to Figure 30, with hexadecimal addresses in the left-most column and consecutive values displayed horizontally. The numbers at the top of the window represent hexadecimal address offsets from the corresponding address in the left-most column. For example, referring to Figure 30, the address of the last word in the second row is $0x00000010 + 0xc = 0x0000001c$.

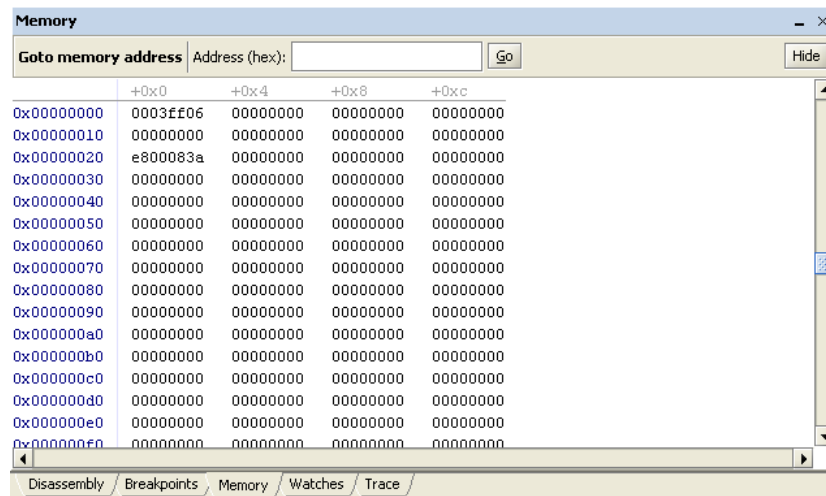


Figure 30. Example **Memory** window.

The display is configurable by a number of parameters:

- Memory element size: the display can format the memory contents as bytes, half-words (2-bytes), or words (4-bytes). This can be configured from the context menu accessible by right-clicking on the **Memory** window, as shown in Figure 31.

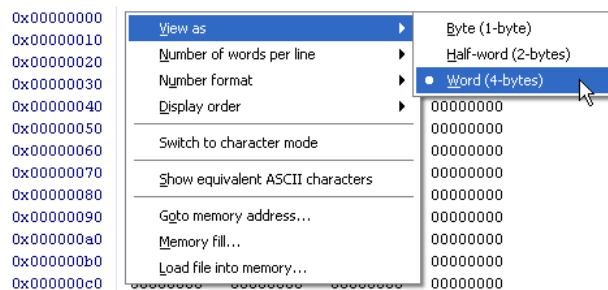


Figure 31. **View as** menu used to select the memory element size in the **Memory** window.

- Number of words per line: the number of words per line can be configured to make it easier to find memory addresses. This can be configured from the context menu accessible by right-clicking on the **Memory** window, as shown in Figure 32.
- Number format: this is similar to the number format option in the **Register** window. This can also be configured from the context menu accessible by right-clicking on the **Memory** window.

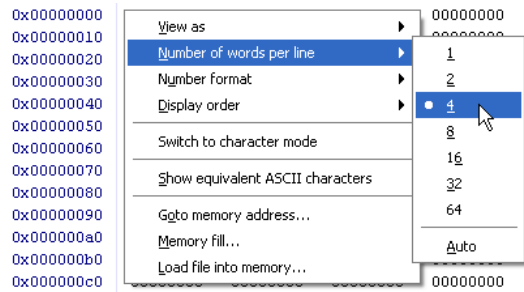


Figure 32. **Number of words per line** menu in the **Memory** window.

- Display order: the display can display addresses increasing from left-to-right or right-to-left. Configure this option by right-clicking on the **Memory** window, as shown in Figure 33.

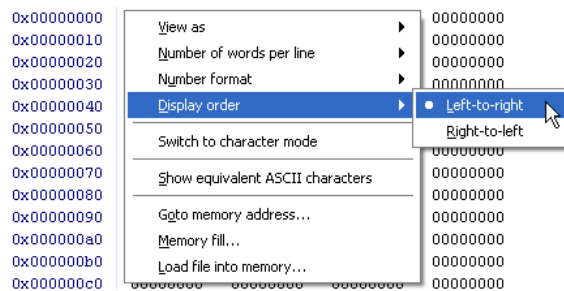


Figure 33. **Display order** menu in the **Memory** window.

Similar to the **Disassembly** window, you can 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 memory address** panel in the **Memory** window analogous to the **Goto instruction** window. Click the **Actions > Goto memory address...** menu item to display the **Goto memory address** panel. As shown in Figure 34, you can enter any address in hexadecimal, press **Go**, and the Memory window will display that address. In this example, display the `11f8` address, which is where the buffer used by the program is stored.

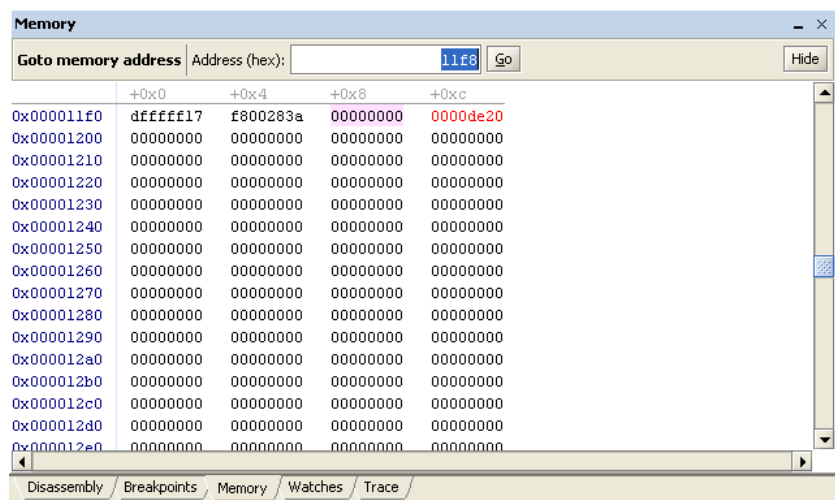


Figure 34. **Goto Memory Address** window.

Since the program reads from this buffer and passes the value to the hexadecimal display, the value shown on the hexadecimal display can be changed by changing the memory value. Proceed as follows:

1. In the row starting at the address 000011f0, double-click the word under the +0xc column. This will bring up a text box to edit the word value at the address 000011fc.
2. Type in abcd, as in Figure 35.

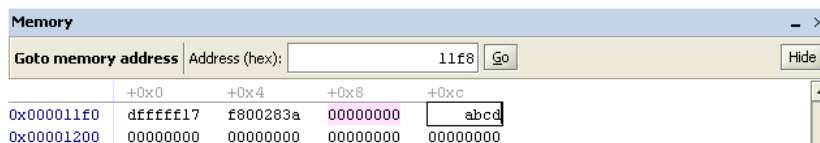


Figure 35. Editing the value at the 000011fc memory address.

3. Press **Enter** or click away from the text box to apply the memory change.

Upon resuming program execution, you will see that the hexadecimal display is now scrolling abCd.

Character display

The **Memory** window can also be configured to interpret memory byte values as ASCII characters. This can be done by checking the **Show equivalent ASCII characters** menu item, accessible by right-clicking on the **Memory** window, as shown in Figure 36.

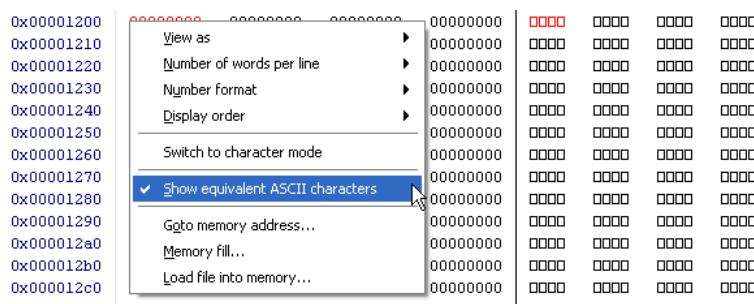


Figure 36. Checking the **Show equivalent ASCII characters** menu item.

The right side of the figure shows a sample ASCII character display. Usually, it is more 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, which can be seen in Figure 36. A sample character display in the character mode is shown in Figure 37.

You can return to the previous memory view mode by right-clicking and clicking the **Revert to previous mode** menu item.

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 click the **Memory fill...** menu item. The **Memory fill** panel will appear on the left-side of the **Memory** window. Simply fill in the desired values and click **Fill**. An example memory fill is shown in Figure 38, which starts at address 0x2000 and ends at 0x2020. The fill value is 2-bytes in length and has a value of 0xabcd.

Load file data into memory

File data can also be loaded into the memory using the Debug Client. This can be useful for providing different data sets to a program. To use this functionality, click the **Actions > Load file into memory...** menu item or right-click on the **Memory** window and click the **Load file into memory...** menu item. The **Load file** panel will appear on the left-side of the **Memory** window. Click **Browse** to select the file to load. There are three types of files that are supported:

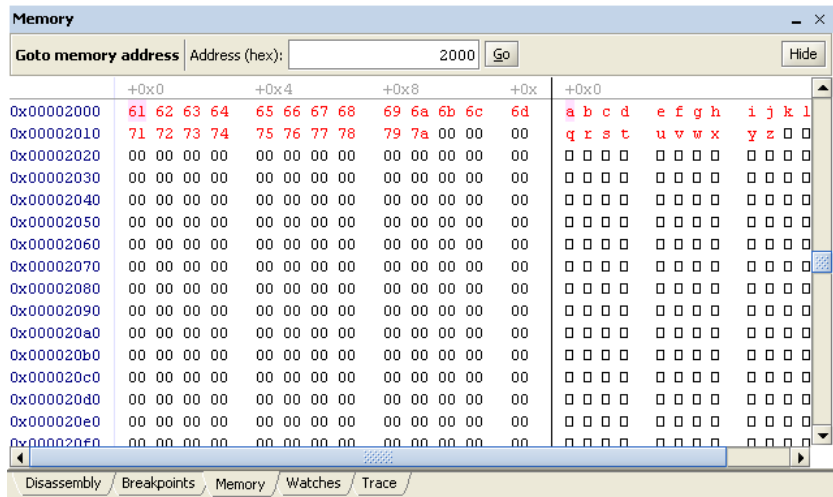


Figure 37. Character mode display.

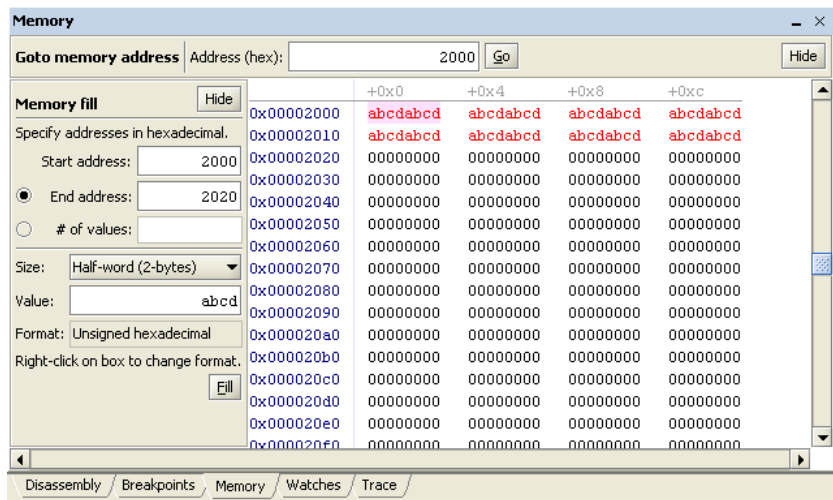


Figure 38. **Memory fill** panel being used to perform a memory fill.

1. *Delimited hexadecimal value files*: These files are plain ASCII files aimed at loading numerical data into memory. The file format is defined as follows:
 - All memory values are specified as hexadecimal values, with an optional sign in front of the value.
 - Individual memory values are separated by a *delimiter character*.
 - There can be multiple lines of delimited values in the file.

For the purpose of this tutorial, select the `<TUTORIAL_FILES>\sample.csv` file. This file is a *comma-delimited hexadecimal value file*.

2. *Intel HEX-format files*: These files are in another special format aimed at loading numerical data into memory. You can use the Quartus II software to create Intel HEX-format files. When doing so, ensure that the word size is set to 8-bits.
3. *Binary files*: These files are loaded byte-by-byte without any interpretation. This is useful for loading binary data, such as audio or image files.

After selecting a file, a start address needs to be specified to indicate where to start loading the file data. In this example, specify 2000. In the case of delimited hexadecimal value files, two additional parameters need to be specified:

1. *Delimiter character*: This is the character that separate consecutive values in the file. In this example, a comma (,) separates the values in the file.
2. *Value byte size*: The byte size of each value element. The Debug Client will truncate values to be within the specified byte size. Specify a value of 4.

After all the parameters have been specified, click **Load** and you should see the loaded values in the **Memory** window, as shown in Figure 39.

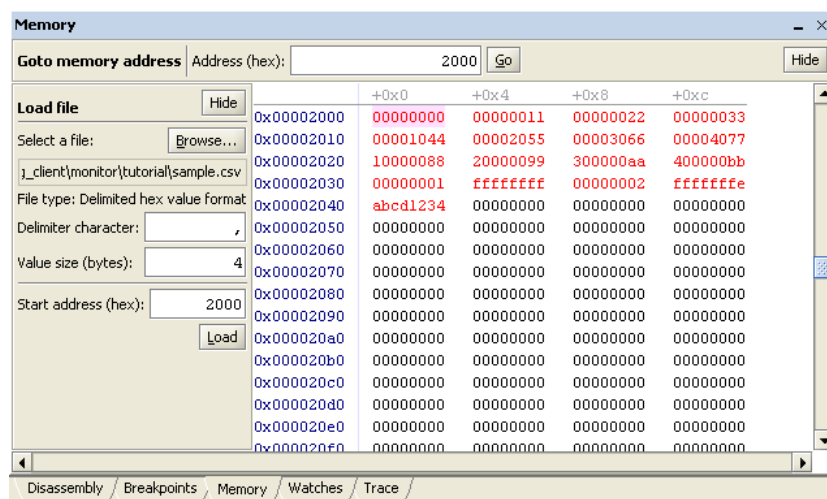


Figure 39. **Load file** panel being used to load a delimited hexadecimal value file into memory.

12 Setting a Watch Expression

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

1. Switch to the **Watches** window.
2. Right-click on the gray bar, like in Figure 40, and click **Add**.

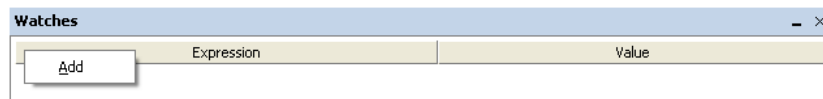


Figure 40. Adding a watch expression via the **Add** menu item.

3. The **Edit Watch Expression** window will appear, like in Figure 41. Enter the desired watch expression, such as `mem32(sp)`, which will display the full-word value at the current stack pointer address.
4. Click **Ok**. The watch expression and its current value will show up in the table.
5. The number format of the displayed value can be changed by right-clicking on the row for that value, as shown in Figure 42.
6. As you repeatedly run the program and stop it at various points, the watch expression will be re-evaluated each time and its value shown in the table of watch values.

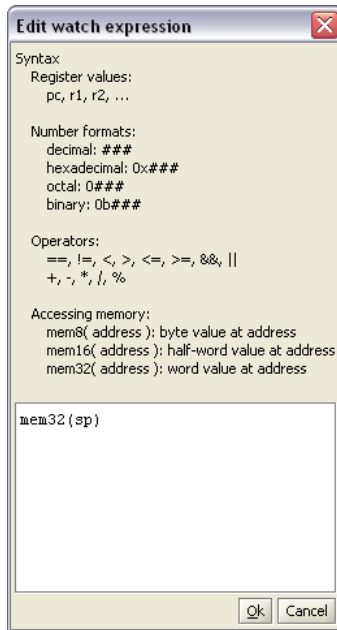


Figure 41. The **Edit Watch Expression** window.

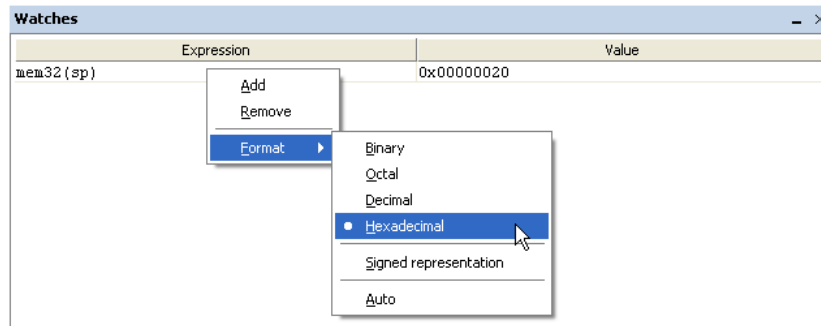


Figure 42. Changing the number format of a watch value.

13 Examining the Instruction Trace

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* configuration of the JTAG Debug Module¹. If the required JTAG Debug Module is not present, a warning will be shown in the **Info & Errors** window after loading the program. The warning will say *WARNING: Could not reset trace. Trace is disabled*. This warning can be safely ignored if there is no trace support in the Nios II system.

The example system used in this tutorial has a Level 3 JTAG Debug Module and so it supports the instruction trace. To demonstrate the **Trace** window in the Debug Client, follow the steps below:

1. Switch to the **Trace** window. There is probably already something there because the instruction trace has been running since the program was loaded.
2. For this part of the tutorial, first clear the trace by right-clicking in the **Trace** window and clicking **Clear sequences**, as shown in Figure 43.

¹See chapter 4 in the *Nios II Processor Reference Handbook* for more information about the configuration settings of the JTAG Debug Module. The Handbook can be found in the Nios II Literature section of Altera's web site.

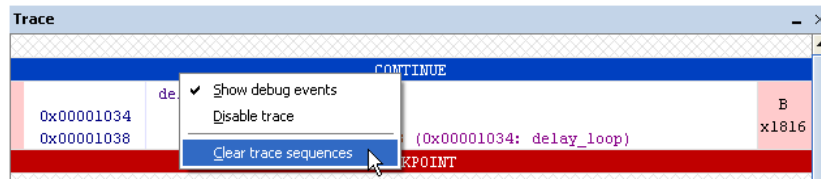


Figure 43. Clear the trace sequences.

- Remove all existing breakpoints and add two new instruction breakpoints at 1030 and 103c. These two breakpoints will be used to skip the `delay_loop` section of the code, which does not produce very interesting results for the instruction trace.
- Run the program until the breakpoint at 103c is triggered. Your **Disassembly** window should look similar to Figure 44 with the breakpoints set and the instruction at 103c highlighted.

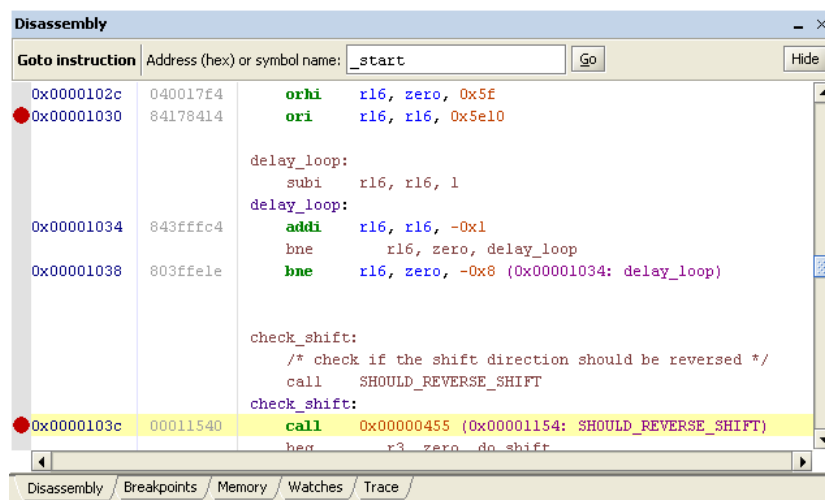


Figure 44. Disassembly window display after setting breakpoints.

- Continue the program's execution; it will execute the code to do some checks on the buffer as well as actually shift the contents of the buffer by one hexadecimal digit. The hexadecimal display will also be updated by a call to `UPDATE_HEX_DISPLAY` before the breakpoint at 1030 is triggered.
- Switch back to the **Trace** window. The window will now be displaying all the instructions executed starting from the first breakpoint to the second breakpoint, as indicated in Figure 45. As seen in the figure, the trace is divided into *instruction blocks*. Although it is not evident in this trace, the Debug Client will try to find repeated instruction blocks and common sequences to reduce the length of the trace on the screen.

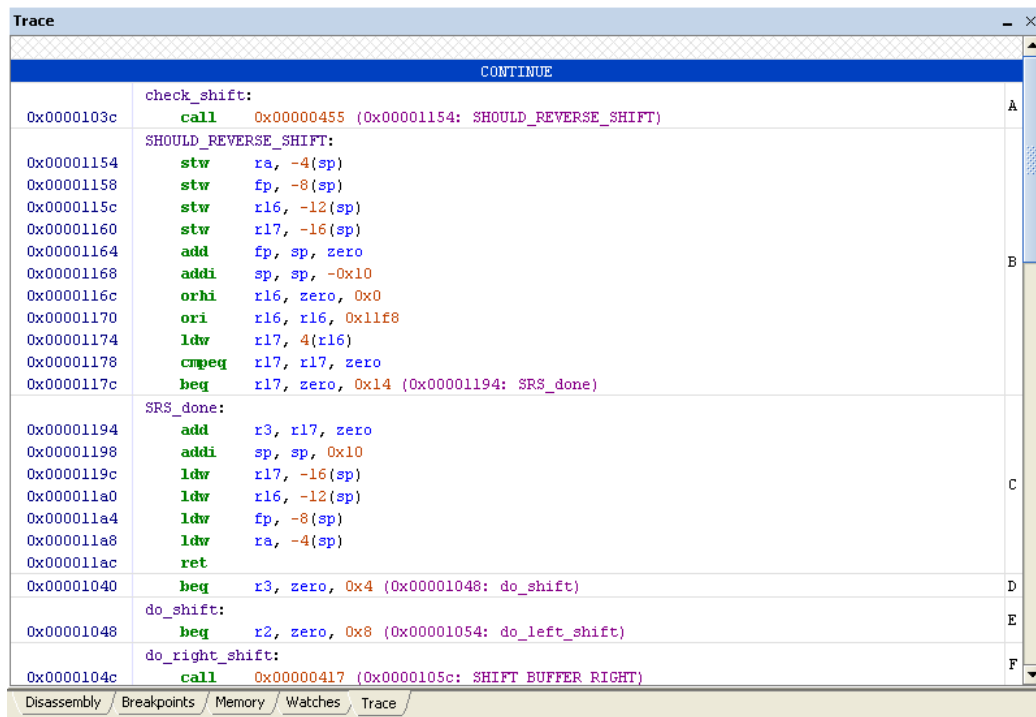


Figure 45. A partial view of the trace between the two breakpoints.

14 Using Configuration Files

Configuration files store the settings used to configure the system (see Section 3) and to configure the program (see Section 4). This allows you to easily reload programs that you were working on previously without having to reconfigure everything each time the Debug Client is started. There are four configuration commands available under the **Monitor** menu, as shown in Figure 46:

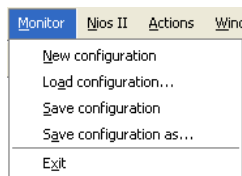



Figure 46. Menu options related to configuration files.

1. **New configuration:** Clears all the settings in the current configuration.
2. **Load configuration:** Displays a window to select a Nios II Configuration File (NCF) to load.
3. **Save configuration:** Saves the current configuration with the same file name that the configuration was last saved as. If the configuration has not been saved before, then this is equivalent to **Save configuration as**.
4. **Save configuration as:** Displays a window to select the name of the Nios II Configuration File, then saves the current configuration into that file.

The next section of this tutorial demonstrates the functionality of the **Terminal** window. The section uses a different source file and thus the program needs to be reconfigured. A configuration file is provided to automatically load the correct configuration settings. To load the configuration file, proceed as follows:

1. Click the **Actions > Disconnect** menu item or click the  toolbar button to end the current GDB debugging session.
2. Click the **Monitor > Load configuration...** menu item.
3. Select the file located at `<TUTORIAL_FILES>\example\sw\terminal_tutorial.ncf` and click **Select**.
4. This step is not required but it is useful as a demonstration to verify that the correct configuration was loaded. Open the **Nios II Program Configuration** window and you should see one source file, similar to Figure 47. Note that the correct source file paths are listed, no matter where the tutorial files were copied. As long as the source file paths remain the same relative to the NCF configuration file, the Debug Client will be able to locate it correctly.

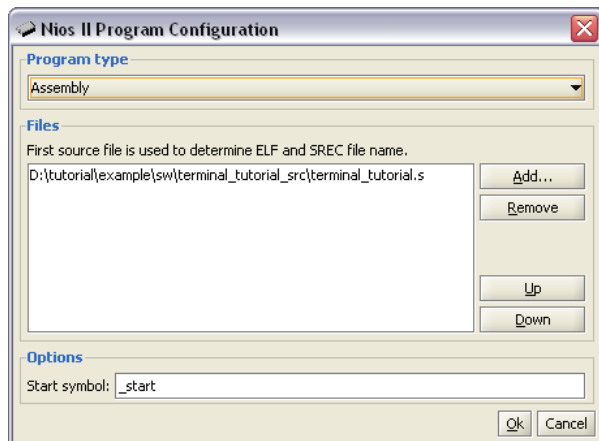


Figure 47. **Nios II Program Configuration** window for the tutorial on the **Terminal** window.

15 Using the Terminal

Assuming that the system and program configuration settings have been set correctly (refer to the previous section of the tutorial for instructions on loading the correct configuration file), this section of the tutorial will demonstrate the functionality of the **Terminal** window in the Debug Client.

1. **Compile & Load** the program.
2. Once the program has been successfully loaded, the **Terminal** window will indicate that a connection has been established, similar to Figure 48.

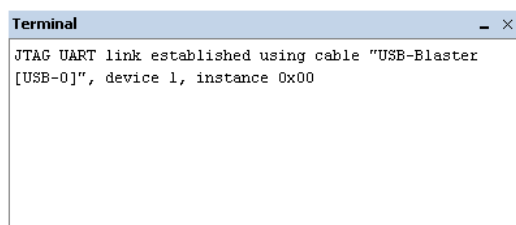


Figure 48. The **Terminal** window after a connection has been established to the JTAG UART on the board.

3. Run the program. The program will clear the Terminal window and write the string `Hello Altera Debug Client`, as shown in Figure 49.

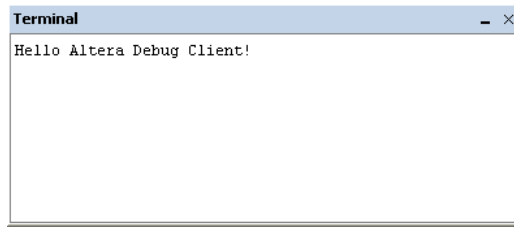


Figure 49. Output from the terminal tutorial program.

4. At this point of the program's execution, the program is ready to accept terminal input and echo it back. Click on the **Terminal** window and type something; you will see that what you type shows up in the window. This is done by the code in the program; the Terminal window, by default, does *not* automatically echo what is typed.

As mentioned in step 3, the program clears the Terminal window. This is accomplished through a special character sequence that is interpreted by the Debug Client to be a *terminal command*. The Debug Client's Terminal window supports a subset of the VT100 terminal commands. The supported commands are listed in Table 2. Note that <ESC> is actually one character, with the ASCII value 0x1B.

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;#2f	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> [2J	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 2. VT100 commands supported by the **Terminal** window in the Debug Client.

Copyright ©2006 Altera Corporation. All rights reserved. Altera, The Programmable Solutions Company, the stylized Altera logo, specific device designations, and all other words and logos that are identified as trademarks and/or service marks are, unless noted otherwise, the trademarks and service marks of Altera Corporation in the U.S. and other countries. All other product or service names are the property of their respective holders. Altera products are protected under numerous U.S. and foreign patents and pending applications, mask work rights, and copyrights. Altera warrants performance of its semiconductor products to current specifications in accordance with Altera's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Altera 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 Altera Corporation. Altera 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.

This document is being provided on an "as-is" basis and as an accommodation and therefore all warranties, representations or guarantees of any kind (whether express, implied or statutory) including, without limitation, warranties of merchantability, non-infringement, or fitness for a particular purpose, are specifically disclaimed.