Using GDB with Nios® V

For Quartus® Prime 24.1

1 Introduction

This tutorial provides instructions for using the *GNU Project Debugger* (GDB) with the *Nios® V* processor, to develop and debug software programs that are written in assembly-language or C code.

This document is intended for a reader who is using a computer system with Nios V on one of the DE-series boards that are described in the `Teaching and Projects Boards` section of the FPGAcademy.org website. We also assume that the reader is using a computer running a recent version of the *Windows®* Operating system.

Required Software and Hardware:

- Quartus Prime Programmer
- GDB Server and Client for Nios V
- Nios V computer system and DE-series FPGA board
- Nios V software development tools

Contents:

- Installing the required software and hardware
- Setting up the Nios V development environment
- Programming your DE-series board
- Developing and debugging Nios V assembly-language programs
- Developing and debugging C programs
- GDB command reference

2 Installing the Required Software and Hardware

The main software tools that are needed for this tutorial are the FPGA *Programmer Tools* that accompany the Quartus Prime CAD system, the GDB server and client for Nios V, and various Nios V software development tools. The required hardware is a Nios V computer system that can be programmed into a DE-series FPGA board. The procedures for downloading and installing each of these components on your own computer are described below. A reader who is using a computer that already has the required software and hardware can skip ahead to Section 3.

2.1 Installing the Quartus Prime Programmer Tools

The Nios V processor is implemented as part of a computer system in an Altera FPGA device. In this tutorial we assume that the reader is using the *DE1-SoC Computer with Nios V*. A complete description of this system, describing all of the peripherals that are connected to Nios V, is available as part of the `Computer Organization` course on FPGAcademy.org. If a different computer system is being used, then some features of the hardware may not match those described in this tutorial.

We assume that the user has access to a *DE1-SoC* board, and that this board has *not* already been programmed with the *DE1-SoC Computer with Nios V*. Hence, programming of the board has to be performed, by using the *Quartus Prime Programmer* tools. This software can be obtained from the *Internet*, as discussed below.

Three different Quartus Prime software *packages* are available, called *Pro*, *Standard*, and *Lite*. For each of these packages a number of *versions* are released over time. For any Quartus package, the *Programmer* tools can be obtained as a separate, *stand-alone*, program or as a part of a complete Quartus system. When used as a stand-alone program, the *Programmer* tools do not not require any license and are free to use. But if a complete Quartus Prime system were to be obtained and installed, then the *Pro* and *Standard* versions would require paid licenses, and only the *Lite* version could be used without a license. For this tutorial we use the *Programmer* as a stand-alone program. Perform the following steps to download and install this program:

1. Search on the *Internet* for the `Intel (Altera) FPGA Software Download Center`.
2. As illustrated in Figure 1, select a `Software Package` and `Version`. For this tutorial we have selected `Quartus Prime Pro` and `Version 24.1`.
3. Scroll down on the web page to see the types of `Downloads` that are available. Click on the `Additional Software` category. Then, scroll further down on the web page to reveal the `Stand-Alone Software` programs. As illustrated in Figure 2, click on the button that is displayed to `Download` the `Quartus Prime Programmer and Tools`. You will need to accept an `Agreement`, after which a file will be downloaded to your computer.

   The file downloaded above is an executable program (*.exe*). Open the folder on your computer where this executable file has been downloaded and run the program. This action opens the installer dialogue depicted in Figure 3. Click `Next` to see the license agreement, which must be accepted to install the software. Clicking `Next` again allows you to select an installation folder. We recommend that you accept the default location which should be similar to the one shown in Figure 4. Select `Next` to advance to the summary screen of the installation dialogue. On this screen you may see a message about obtaining a license to use the software, but **do not** click on the provided link, because no license is required for the stand-alone *Programmer Tools*. Click `Next` on the summary screen to install the *Programmer Tools*.

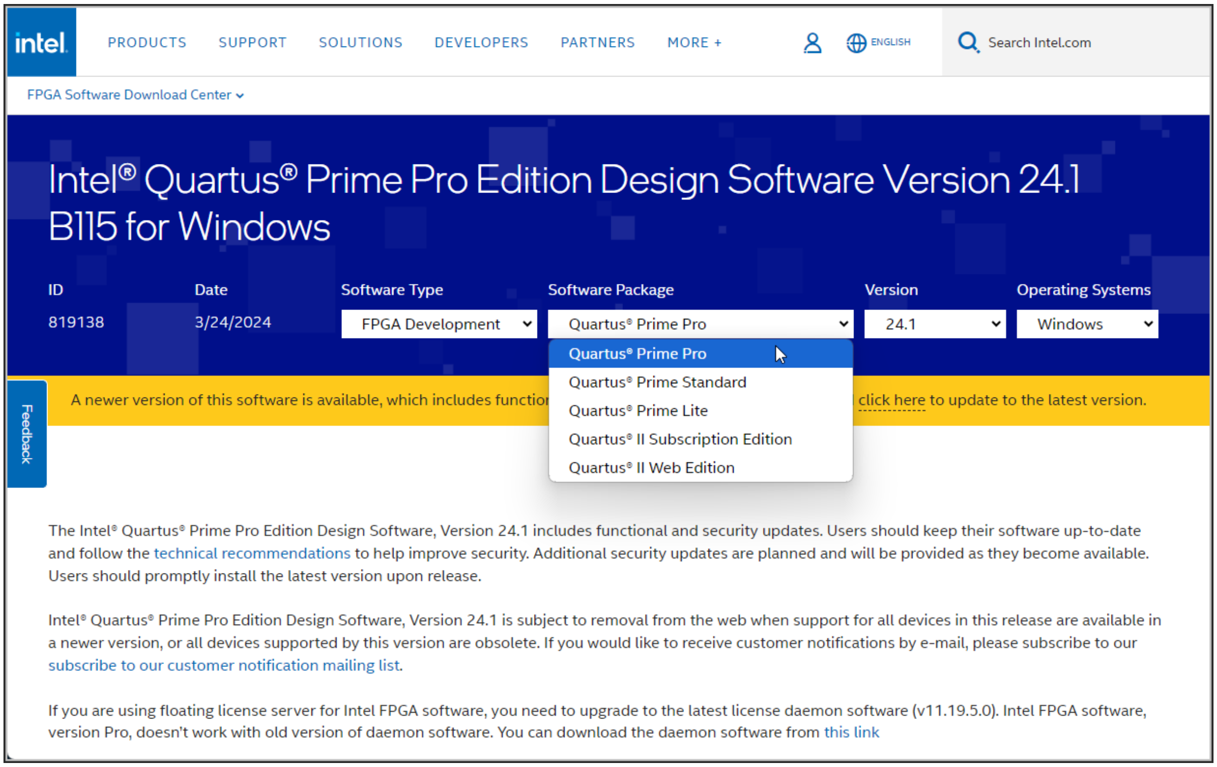


Figure 1. The Intel (Altera) FPGA Software Download Center.

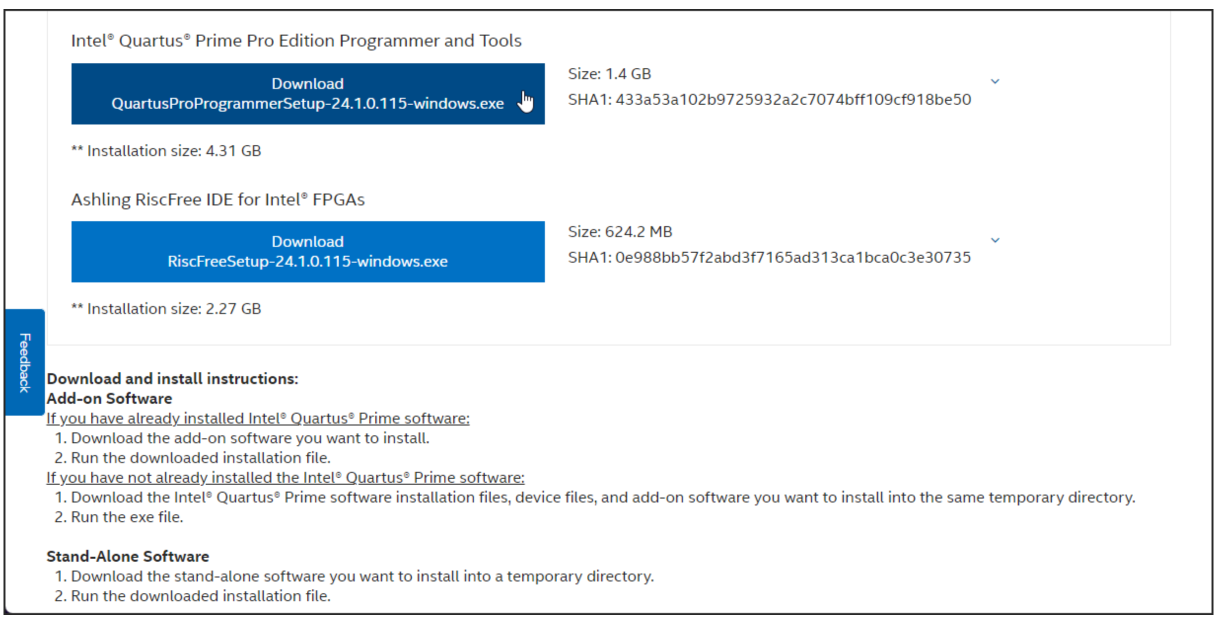


Figure 2. Downloading the Quartus Prime Programmer.

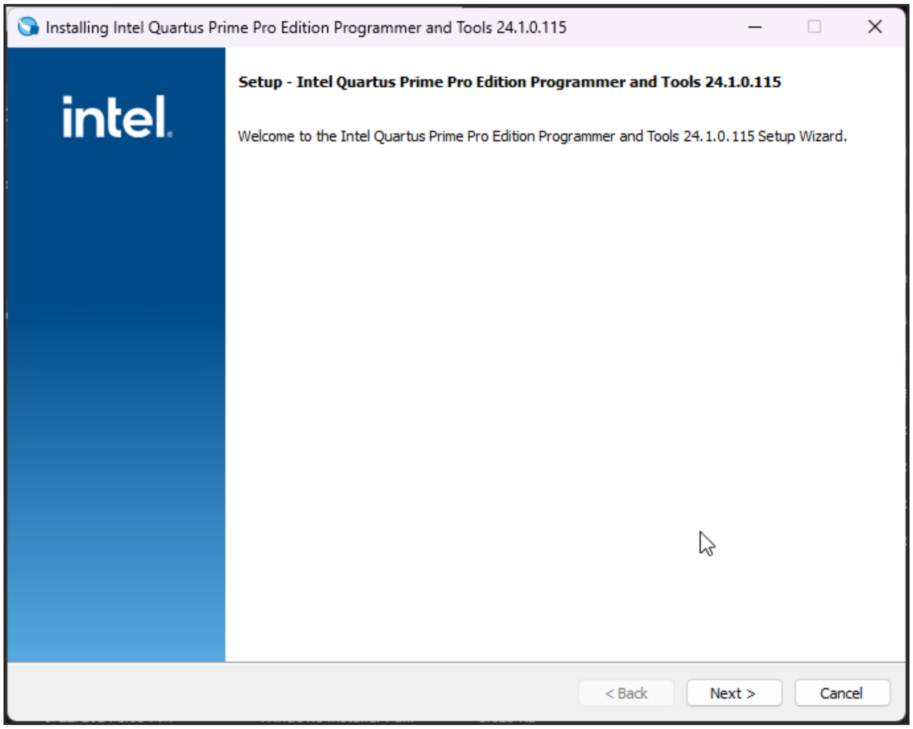


Figure 3. The *Programmer Tools* installation dialogue.

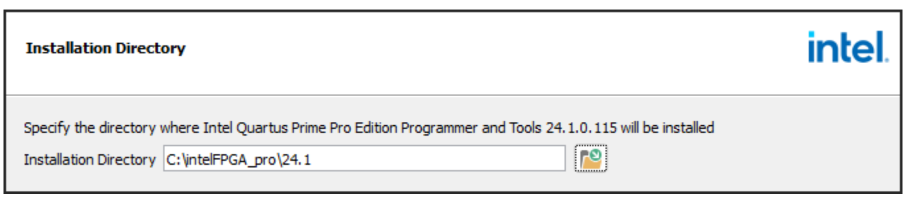


Figure 4. Choosing the installation folder.

4. After completing the *Programmer Tools* installation, the window displayed in Figure 5 may appear. As the figure indicates you should make the selections for installing the `USB Blaster II` driver and the `JTAG Server`. Click *Finish*. Depending on what device drivers are already installed on your computer you may be presented with additional installation dialogues. If so, follow the presented steps to install the necessary drivers. These drivers allow for communication between your computer and an FPGA board that is connected to the computer via a USB cable.

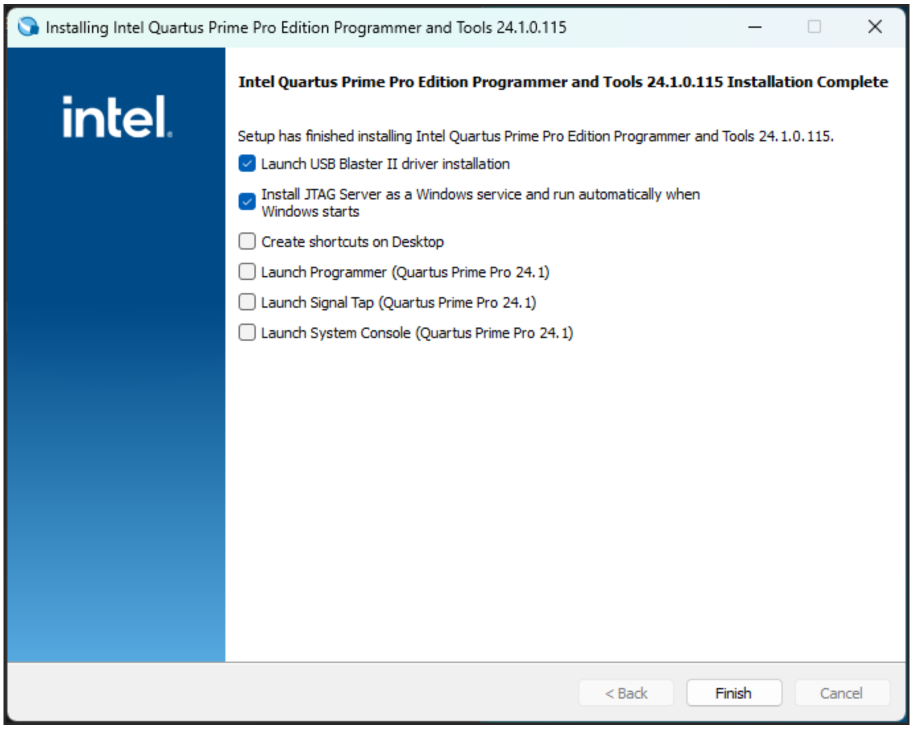


Figure 5. Post-installation selections.

2.2 Installing the GDB Server and Client for Nios V

The GDB software tools for Nios V that are needed for this tutorial are available from the same `Intel (Altera) FPGA Software Download Center` used above to obtain the Quartus *Programmer Tools*. Navigate again to the website that is depicted in Figure 1. Again, scroll down on the website to see the types of `Downloads` that are available, click on the `Additional Software` category, and scroll further down to display the `Stand-Alone Software` programs. The GDB software tools for Nios V are part of the package called `Ashling RiscFree IDE for Intel FPGAs`. Click to download this package as indicated in Figure 6, after which a file will be downloaded to your computer.

The file downloaded above is an executable program (*.exe*). Open the folder on your computer where this executable file has been downloaded and run the program. This action opens the installer dialogue depicted in Figure 7. Click `Next` to see the license agreement, which must be accepted to install the software. Clicking `Next` again allows you to select an installation folder. This folder must be the *same* as the one that you selected for the Quartus Programmer Tools, as mentioned previously for Figure 4. Select `Next` to advance to the summary screen of the installation dialogue. On this screen you may see a message about obtaining a license to use the software, but **do not** click on the provided link, because no license is required for the this software package. After the software has been installed, click `Finish` to close the installation executable.

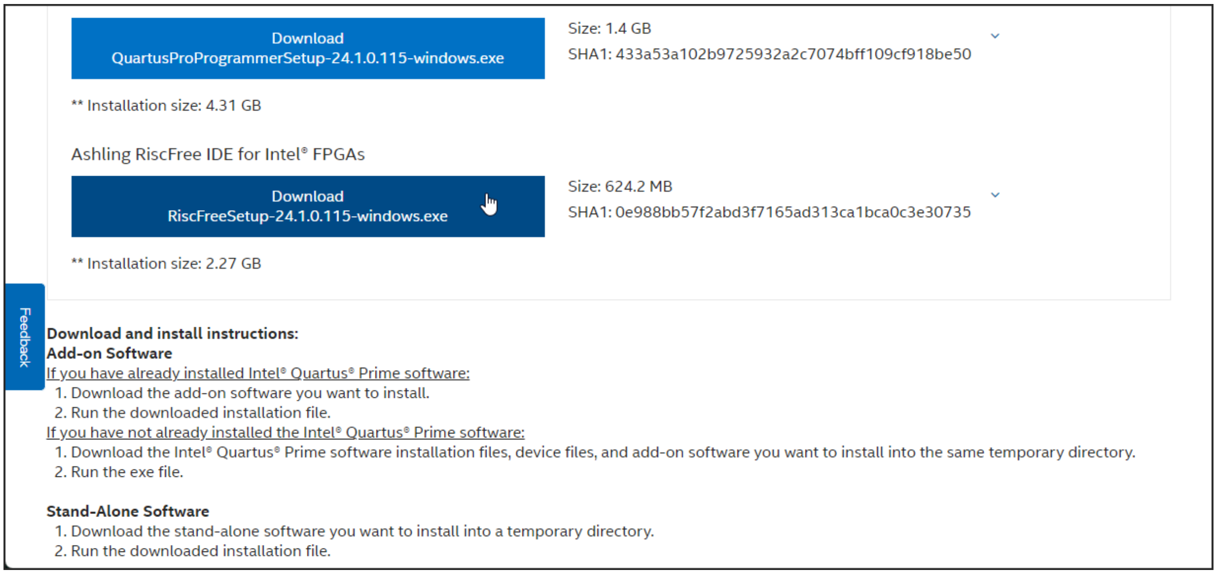


Figure 6. Downloading the GDB tools.

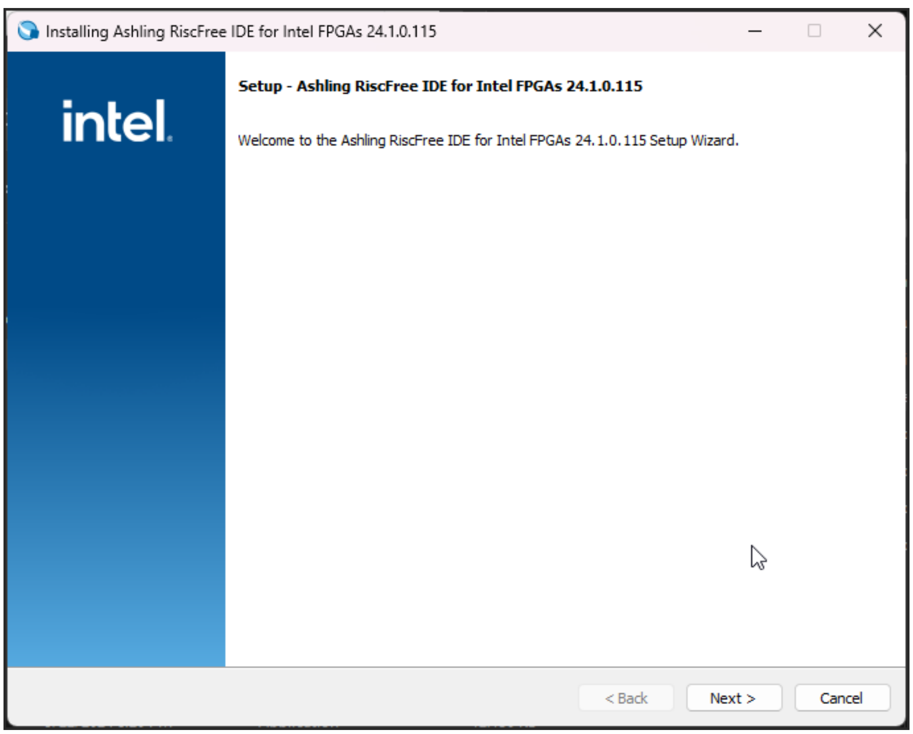


Figure 7. The *Ashling RiscFree Software* installation dialogue.

2.3 Installing the Nios V Hardware and Software Development Tools

This tutorial requires a hardware system containing the Nios V processor, as well as Nios V software development tools for that system. These hardware and software components can be obtained from *GitHub*, at the `URL` below:

https://github.com/fpgacademy/Design_Examples/releases/tag/v1.0.

From the GitHub repository download to your computer the file named *fpgacademy.zip*, which is listed under `Assets`. Next, you need to uncompress this *ZIP* archive file and store its contents into the *same* folder where you installed the Quartus *Programmer Tools* and *Ashling RiscFree Software*.

As illustrated in Figure 8, your installation folder should now contain: the Quartus *Programmer Tools* (in the `qprogrammer` folder), the *Ashling RiscFree Software* (in the `riscfree` folder), and the Nios V hardware system and software development tools (in the `fpgacademy` folder).

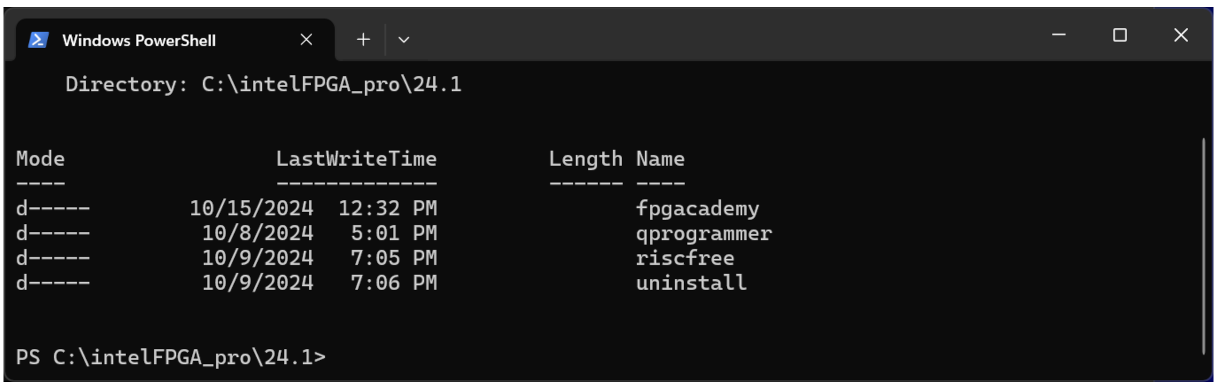


Figure 8. The final contents of your installation folder.

3 Developing and Debugging Nios V Assembly-Language Programs

After the necessary software and hardware components have been installed onto your computer, as discussed in Section 2, you can begin working with the *GDB* debugger to develop Nios V programs. We will first present a few examples that show how to use GDB to develop and debug Nios V assembly code. Later in the tutorial, in Section 4, we will give some examples that use C code.

3.1 Installing the Tutorial Design Examples

On the FPGAcademy.org website, this tutorial is accompanied by *Design Files* that are used to illustrate various features of the *GDB* software for developing and debugging Nios V programs. Download to your computer the provided *Using_GDB_Nios_V_design_files.zip* file. Then, uncompress this archive into any folder of your choice. We will refer to the examples of code and other files in this folder throughout the tutorial. Figure 9 shows the folders included in the *design files*, assuming that they have been installed into a folder named `GDB_tutorial`.

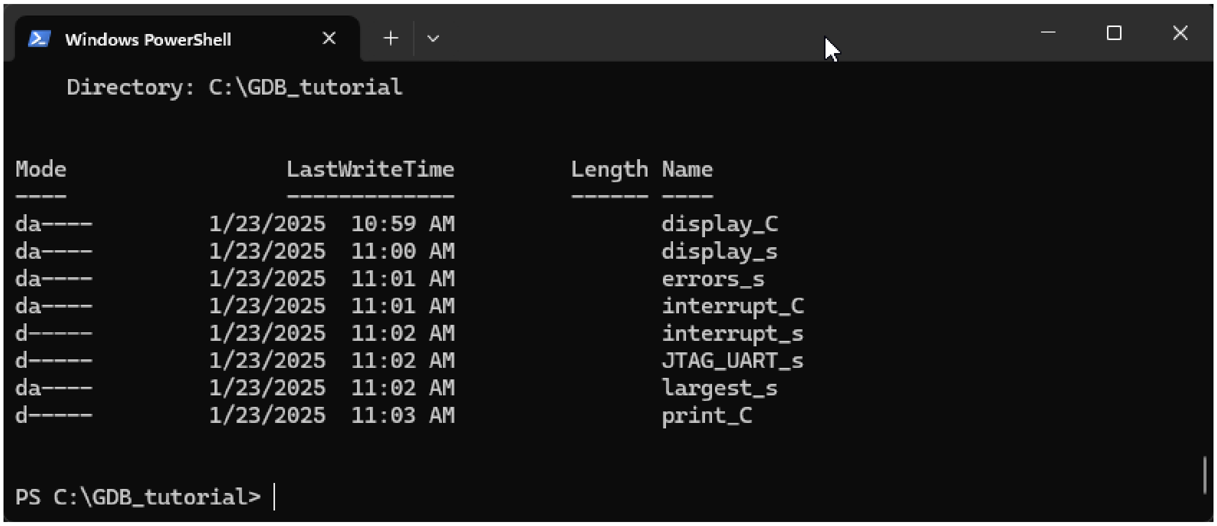


Figure 9. The *Design Files* folders.

3.2 Using the GNU Make Program

In this tutorial all tools are executed by using the command-line environment provided by the *Windows PowerShell*. Open a *PowerShell* terminal using a method of your choosing. Then, navigate to the *design files* folder called `C:\GDB_tutorial\largest_s`. As illustrated in Figure 10, this folder contains an example of a Nios V assembly-language program, *largest.s*, a *Makefile*, and an executable batch file *gmake.bat*.

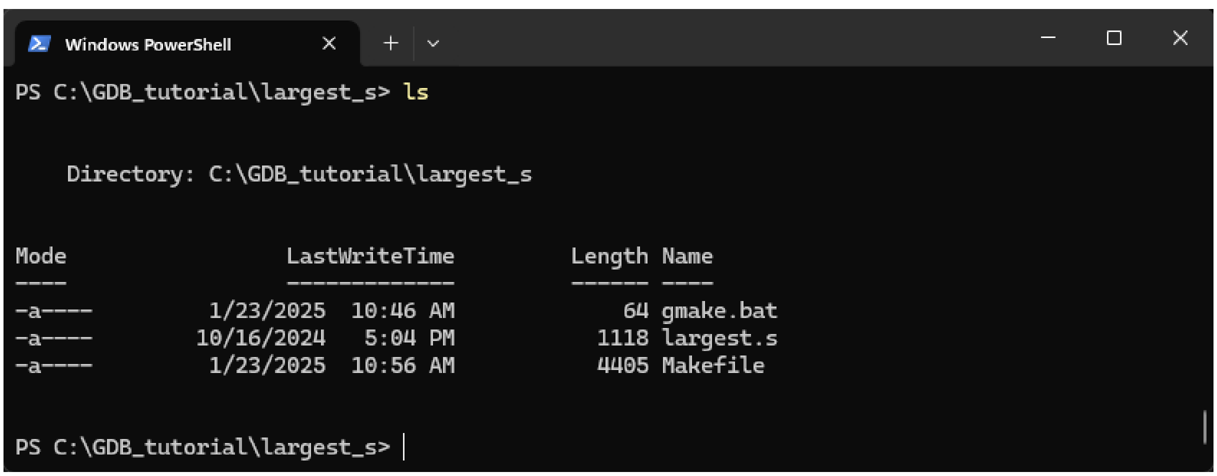


Figure 10. The contents of the largest_s design files folder.

In this tutorial the tools that we use for developing and debugging Nios V programs are executed via the *GNU make* program. A copy of *GNU make* is included as part of the installed software discussed in Section 2, in the location:

`C:/intelFPGA_pro/24.1/fpgacademy/AMP/bin/make.exe`

In the examples presented below we assume that you have added the folder containing this *make.exe* program to your `Path` *Windows Environment Variable*, so that it is easy to run the *make* program. Alternatively, you could execute the make program via the *gmake.bat* file listed in Figure 10. This batch file can be set up to have the full path name to the required *make.exe* and then executed by entering the command `./gmake`.

In the folder of Figure 10 open the *Makefile* in any text editor of your choice. The first few lines of this file are displayed in Figure 11. Line 1 defines a variable called `INSTALL` that specifies the folder in which the software needed for this tutorial has been installed. The setting given in the figure matches the installation folder that we used in Section 2, as shown in Figure 4. If a different installation folder is used on your computer, then change the value of the `INSTALL` variable accordingly (type forward slashes (/) as separators to specify the path to your folder, as done in the Figure 11, as opposed to backward slashes (\)).

```
1 INSTALL := C:/intelFPGA_pro/24.1
2
3 MAIN    := largest.s
4 HDRS    :=
5 SRCS    := $(MAIN)
6
7 # DE1-SoC
8 JTAG_INDEX_SoC  := 2
9
```

Figure 11. The first few lines of the *Makefile*.

3.3 Configuring the DE1-SoC Board

Connect a DE1-SoC board to your computer. The board should be connected by plugging a cable that has a *Type-A USB* connector into the *USB Blaster* port on the board and connecting the other end of this cable to any *USB* port on your computer. Ensure that the DE1-SoC board is properly powered on. To configure your DE1-SoC board with the desired Nios V computer system, in the terminal window of Figure 10 execute the command:

```
make DE1-SoC
```

This command runs the Quartus *Programmer* and configures the board with the *DE1-SoC Computer with Nios V* system. If the command completes without errors, then you can skip ahead to Section 3.4 and begin using GDB with Nios V. Note that in some cases it can take up to a minute to configure the board.

If the Quartus *Programmer* fails to start, then make sure that you have installed the required software, and that you have properly set up the `INSTALL` variable shown in Figure 11. If the Quartus *Programmer* runs, but fails to configure your DE1-SoC board, then try running the command:

```
make DETECT_DEVICES
```

This command checks which devices are visible on the *USB Blaster* cable that is connected to your computer. Part of the expected output from this command is displayed in Figure 12. It shows two devices being detected: first an *SOCVHPS* device, followed by a Cyclone V *5CSE FPGA* device. If the output produced from your board shows these two devices, but in the opposite order, then you have to modify your *Makefile*. Change the variable `JTAG_INDEX_SoC` shown in Line 8 of Figure 11 from the value `2` to the value `1`. You should now be able to successfully configure your DE1-SoC board by executing the `make DE1-SoC` command. Another possible scenario is that you are using a board other than the DE1-SoC board. If using the DE10-Lite board, then run the command `make DE10-Lite` to configure your board. If you are using some other board, then you will need to read carefully through the *Makefile* to determine how its commands have to be modified to suit your board.

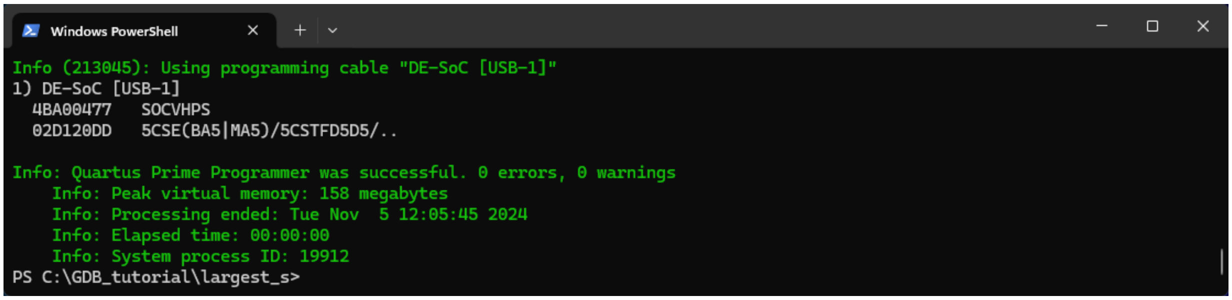


Figure 12. The output from `make DETECT_DEVICES`.

3.4 Using the GDB Server and Client

To develop Nios V programs, you need to use **two** PowerShell terminals: the first one is used to open the *GDB Server*, and the second one is used run the *GDB Client*. To start the GDB Server, in the terminal window of Figure 10 execute the command:

```
make GDB_SERVER
```

The server will then remain running in this window, as indicated in Figure 13. Now, open another PowerShell window (if you are using Microsoft Windows 11, simply click on the **+** symbol located in the title-bar area of Figure 13 to open a new PowerShell *tab*). In this new terminal tab navigate again to the same folder as in Figure 10.

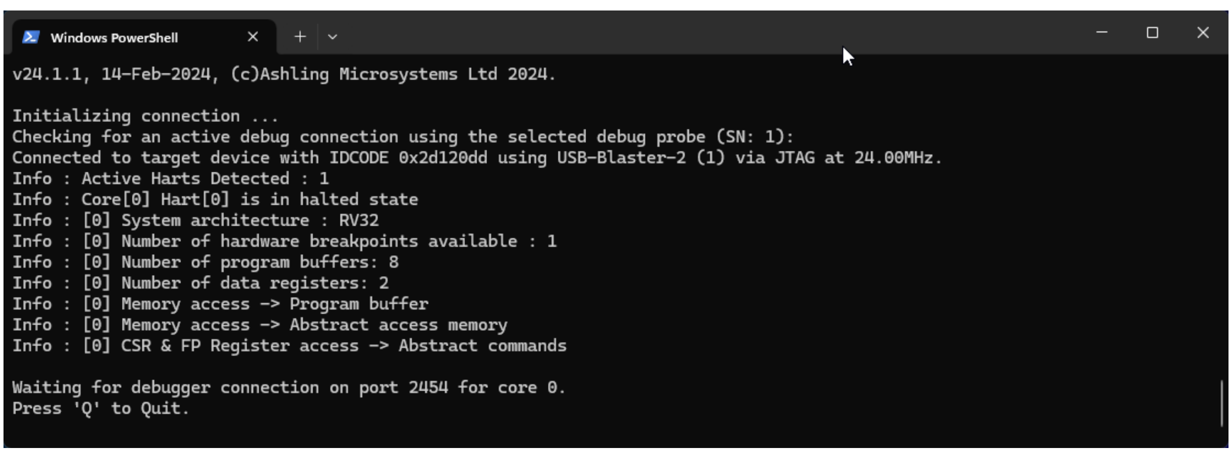


Figure 13. Running the GDB Server.

This part of the tutorial uses an assembly-language program, *largest.s*, which is shown in Figure 14. This program searches through a list of integers that is stored in memory and finds the largest number in the list. Assemble this program by executing the command `make COMPILE`. You could also just type `make`, because `COMPILE` is the first target in the *Makefile*. As illustrated in Figure 15, this command runs the Nios V assembler and linker tools to generate the Nios V executable file *largest.elf*.

```
# Program that finds the largest number in a list of integers
.global _start
_start:     la      t0, result      # t0 = pointer to the result
            lw      t1, 4(t0)       # t1 = counter, initialized with N
            addi    t2, t0, 8       # t2 = pointer to the first number
            lw      t3, (t2)        # t3 = largest found so far
loop:       addi    t1, t1, -1      # decrement counter
            beqz    t1, done        # done when counter is 0
            addi    t2, t2, 4       # point to the next number
            lw      t4, (t2)        # get the next number
            bge     t3, t4, loop    # compare to largest found
            mv      t3, t4          # remember new largest
            j       loop

done:       sw      t3, (t0)        # store result
stop:       j       stop            # wait here

result:     .word   0               # result will be stored here
N:          .word   7               # number of entries in the list
numbers:    .word   4, 5, 3, 6      # numbers in the list
            .word   1, 8, 2         # ...
```

Figure 14. A program that finds the largest number in a list.

```
PS C:\GDB_tutorial\largest_s> make
Assembling
C:/intelFPGA_pro/24.1/fpgacademy/AMP/cygwin64/home/compiler/bin/riscv32-unknown-elf-as.exe -march=rv32im_zicsr --gdwarf2
 largest.s -o largest.s.o
Linking
C:/intelFPGA_pro/24.1/fpgacademy/AMP/cygwin64/home/compiler/bin/riscv32-unknown-elf-ld.exe --section-start .text=0x0 --n
o-relax largest.s.o -o largest.elf

PS C:\GDB_tutorial\largest_s>
```

Figure 15. Making the executable file *largest.elf*.

Now you can run the *GDB Client* by executing the command:

```
make GDB_CLIENT
```

The GDB Client will connect to your DE1-SoC board, load the executable file *largest.elf*, initialize some Nios V control registers, and set the Nios V program counter register, *pc*, to the start of the program. The output produced by this command is displayed in Figure 16.

```
PS C:\GDB_tutorial\largest_s> make GDB_CLIENT
riscv32-unknown-elf-gdb.exe -silent -ex "target remote:2454" -ex "set $mstatus=0" -ex "set $mtvec=0" -ex "set $mie=0" -e
x "load" -ex "set $pc=_start" -ex "info reg $pc" "largest.elf"
Reading symbols from largest.elf...
Remote debugging using :2454
_start () at largest.s:3
3       _start:     la      t0, result      # t0 = pointer to the result
Loading section .text, size 0x5c lma 0x0
Start address 0x00000000, load size 92
Transfer rate: 5 KB/sec, 92 bytes/write.
pc             0x0      0x0 <_start>
(gdb) |
```

Figure 16. Starting the GDB Client.

We will use the *largest.s* program as an example to illustrate some basic GDB commands. A summary of the GDB commands used in this tutorial is provided in Appendix A. Of course, a lot of documentation about GDB commands can also be found on the Internet.

In the GDB Client type the `list` command, as shown in the Figure 17, to see the loaded program.

```
(gdb) l
1       # Program that finds the largest number in a list of integers
2       .global _start
3       _start:     la      t0, result      # t0 = pointer to the result
4                   lw      t1, 4(t0)       # t1 = counter, initialized with N
5                   addi    t2, t0, 8       # t2 = pointer to the first number
6                   lw      t3, (t2)        # t3 = largest found so far
7       loop:       addi    t1, t1, -1      # decrement counter
8                   beqz    t1, done        # done when counter is 0
9                   addi    t2, t2, 4       # point to the next number
10                  lw      t4, (t2)        # get the next number
(gdb)
```

Figure 17. The output of the `list` command.

Next, execute the first two instructions in the program by using the GDB `step` command twice. Then, execute the commands `info reg t0` and `info reg t1` to see that register `t0` holds the address in memory of the `result` label, which is `0x38`, and that register `t1` has the number of elements in the list, which is `7` (this value is specified at the label `N` in the code in Figure 14). The results of these commands are displayed in Figure 18. You can see the contents of memory by using the `x` command. Enter `x/4x result` to see the four words of memory starting at the address of the `result` label (`/4x` designates four `words` displayed in `hexadecimal`).

As illustrated in Figure 19 set a breakpoint at line 7 in the source code, which corresponds to the label `loop`, by using the command `break 7`. Then, run to this breakpoint twice by using the `continue` command. Check the value of register `t3` to see that the largest number found in the list so far is `5`. Now, clear the breakpoint by using the command `clear 7`. The program ends with an infinite loop at the label `stop`, as seen in Figure 14. Set a breakpoint at this label by using the command `break stop`. Enter `continue` to resume the program until it stops at the breakpoint. Finally, use the command `info reg t3` to see that the program found the largest number in the list, which is `8`, and enter `x/4x result` to see that this result has been stored into memory.

We are now finished with the *largest_s* example. As demonstrated in Figure 20, disconnect from your DE1-SoC board by executing the `detach` command. Finally, execute the `quit` command. If you see the prompt `Terminate batch job (Y/N)?` respond with `n`.

In the GDB Server terminal of Figure 13, type `q` to quit. While it is not absolutely necessary to exit the server before starting to work on another Nios V program, it is a good idea to do so. The server occasionally experiences communications failures with the DE1-SoC board and then has to be restarted—hence, leaving the server running for long periods of time may not be a good approach.

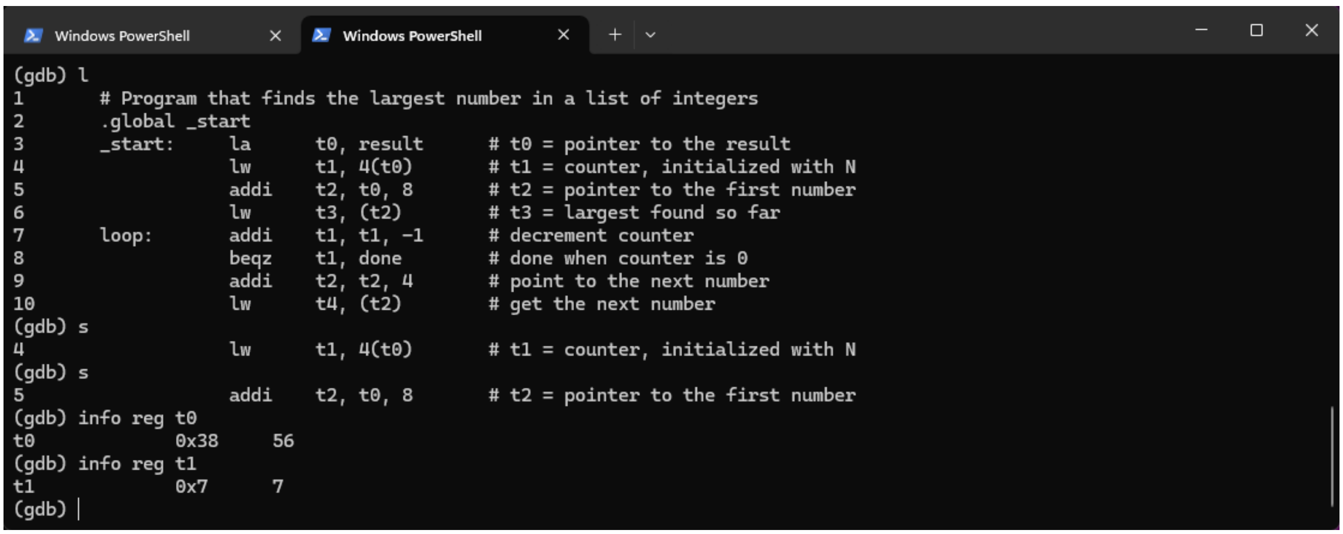


Figure 18. Executing a few GDB commands.

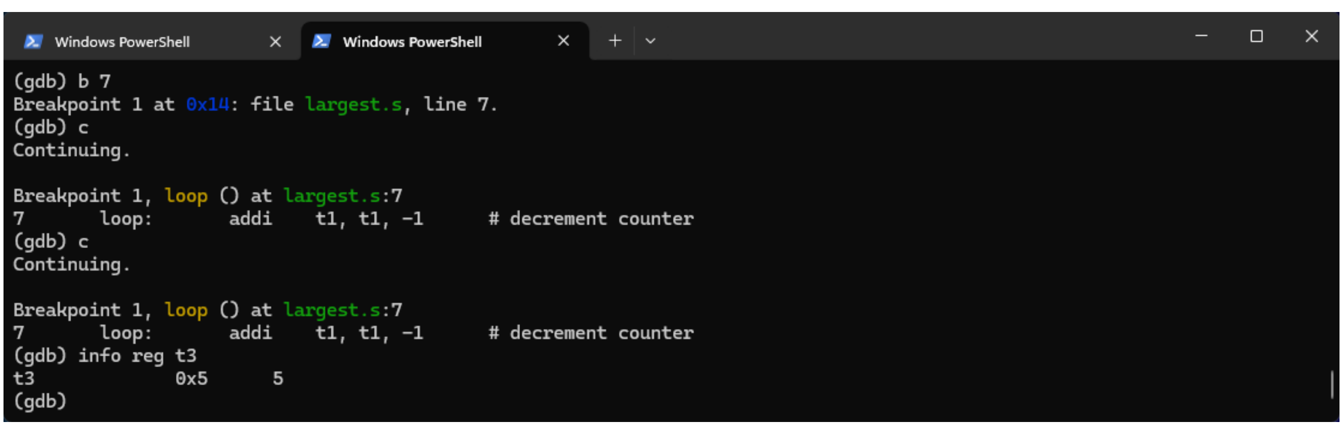


Figure 19. Using a breakpoint.

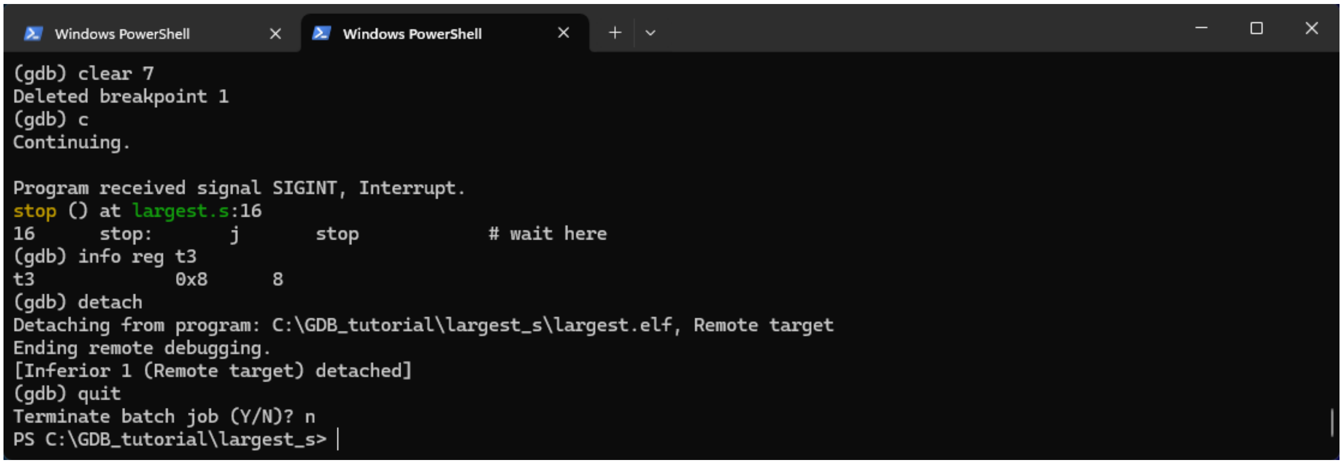


Figure 20. Completing the program and quitting from GDB.

3.5 Setting up Your Own Assembly-Code Makefile

It is easy to customize the *Makefile* shown in Figure 11 so that you can use it with any assembly-language code of your choosing. `Line 3` of the *Makefile* has to specify the name of the assembly-code file that has the `_start` label, which designates the beginning of the program. Any `.s` header files that are used with the program can be listed in `Line 4`. Finally, any additional assembly-language source-code files can be listed in `Line 5`.

Having learned the basics about using GDB with Nios V in Section 3.4, we will now utilize the various design files examples provided with this tutorial to illustrate additional GDB capabilities.

3.6 Using Simple I/O Devices with Assembly Code

When starting to work on a new design example it is a good approach to power your DE1-SoC board `off`, and then `on` again, so that the system is reset. Open a PowerShell terminal and navigate to the folder for the `display_s` design example. In this folder, execute the command `make DE1-SoC` to configure your board. If this programming step fails, refer to the discussion in Section 3.3 for suggestions as to how to fix any issues. Once your board is successfully configured, execute the command `make GDB_SERVER`. Now, open a second PowerShell tab (as described in Section 3.4) and, as illustrated in Figure 21, navigate again to the `display_s` folder and execute the command `make COMPILE`, followed by `make GDB_CLIENT`.

Within the GDB Client, use `list 1,14` to see the source-code of the program. As shown in Figure 22, the program first sets up three pointers to I/O devices in the *DE1-SoC Computer*: register *t0* is initialized to the address of the *LEDR* red light port, register *t1* to the address of the *SW* slide-switch port, and register *t2* to the address of the port connected to 7-segment display *HEX3* to *HEX0*. The program then executes an endless loop in which it loads the current value of the *SW* switch port and stores this value to both the *LEDR* and *HEX0* display ports.

Enter the command `break loop`, and then use `continue` to stop the program at this breakpoint. Now, run through iterations of the loop in the program for a while by executing the command `continue 150`. This com-

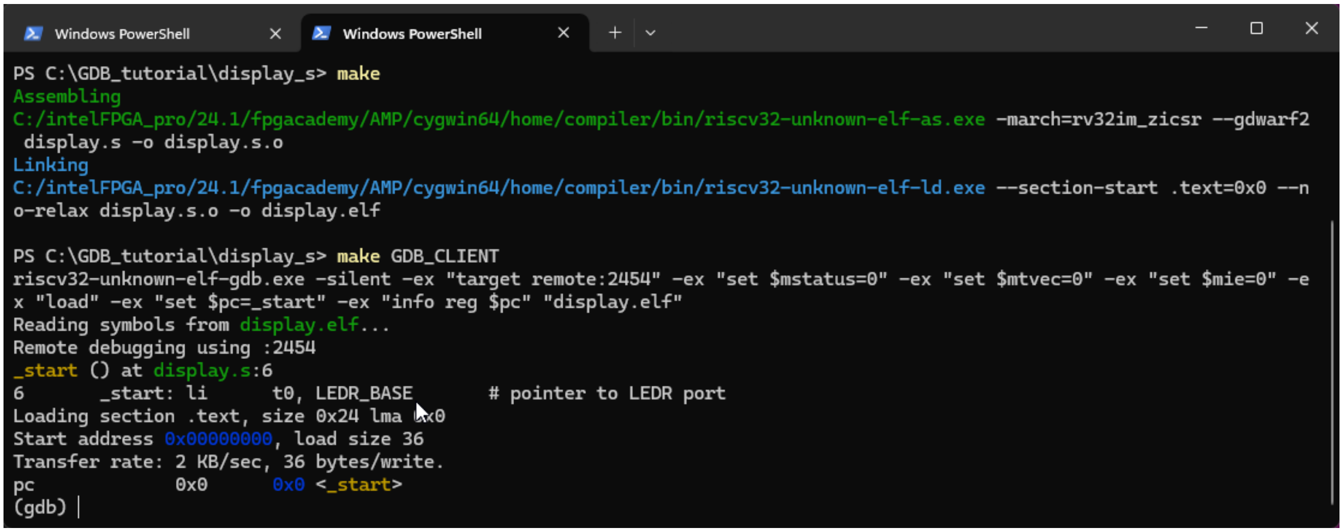


Figure 21. Starting GDB for the `display_s` example.

mand runs the program until the breakpoint at `loop` has been encountered 150 times. While the program is running, try different settings on the SW_{6-0} switches on the DE1-SoC board and observe the *LEDR* lights and *HEX0* display.

Wait until the program has stopped executing and control has been returned to the GDB Client. Now, set a new pattern of your choice on the *SW* switches and then enter the `step` command, followed by `info reg t3` to see the value loaded from the *SW* port. Execute `step` again and observe the *LEDR* lights, then use `step` a third time and observe the *HEX0* display.

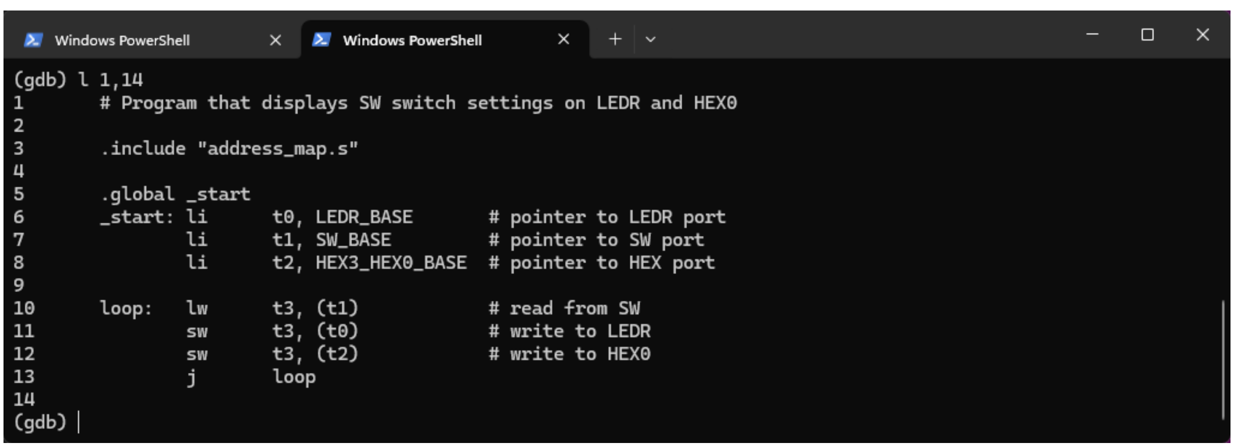


Figure 22. The `list` command.

Use `continue` again to get to the top of the loop, and then `step` to execute the *load* instruction at `Line 10`. Now, use the GDB command `set $t3 = 0x3ff` to overwrite the value loaded into register *t3*. Use `step` again and then observe on the DE1-SoC board that the value you placed into register *t3* turns on all ten *LEDR* lights.

We are now done with this design example, so use the `detach` command to disconnect from the DE1-SoC board. Finally, `quit` from the GDB Client, and then go to the GDB Server PowerShell tab and type `q` to close the server.

3.7 Using Interrupts with Assembly Code

The assembly code example for this part of the tutorial displays a one-second binary counter on the red lights *LEDR*. The speed of the counter is controlled by using interrupts from the Nios V Machine Timer.

Open a PowerShell terminal and navigate to the `interrupt_s` folder. If not already done, configure your DE1-SoC board by running `make DE1-SoC`. Execute `make GDB_SERVER`. In a second PowerShell tab navigate again to the `interrupt_s` folder and run `make COMPILE` and `make GDB_CLIENT`.

In the GDB Client, as illustrated in Figure 23, run `list 1,15`. Use `step` to execute the instruction on `Line 5` that initializes the stack pointer register. Next, enter `x/4x 0xff202100`. This command displays the contents of the memory-mapped 64-bit Nios V `Machine Timer` registers, which are referred to as *mtime*, which has the address `0xff202100`, and *mtimecmp*, which has the address `0xff202108`.

Next, set a breakpoint using `break 9`. Then, execute `continue`, which runs the Nios V program to call the subroutine *set_timer* and then stops, after returning, at the breakpoint on `Line 9`. Again, as illustrated in Figure 24, use `x/4x 0xff202100` to display the Machine Timer registers. As indicated in the figure, the *mtime* register was cleared to 0 (and then continued counting up at its 100 MHz clock rate), and the *mtimecmp* register was set to 100,000,000 (`0x5f5e100`) to provide machine timer timeouts for every one second.

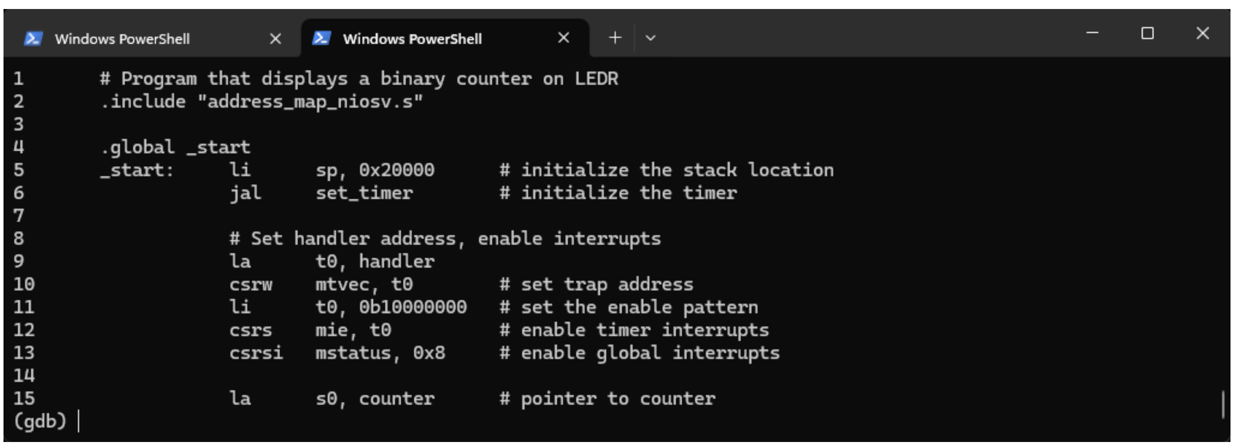


Figure 23. The `interrupt_s` example.

The next five instructions in the program set up Nios V interrupts as needed for this example. As shown in Figure 25, display the contents of the (uninitialized) *mtvec* control register by using `info reg mtvec`. Enter `step 2` to execute two instructions, and then use `info reg mtvec` to see that this register has been initialized to `0x68`. Enter `info symbol 0x68` to see that this is the address in memory of the interrupt *handler* routine. Display the current contents of the *mie* and *mstatus* control registers with `info reg mie status`. Execute three more instructions (`step 3`) and then enter `info reg mie mstatus` again to see that the *mie* register now contains the value `0x80`, which has the interrupt-enable bit corresponding to the Machine Timer set to 1, and *mstatus* shows

```
(gdb) step
6                   jal     set_timer        # initialize the timer
(gdb) x/4x 0xff202100
0xff202100:     0xa16d42aa      0x00000001      0x00000000      0x00000000
(gdb) b 9
Breakpoint 1 at 0x8: file interrupt.s, line 9.
(gdb) cont
Continuing.

Breakpoint 1, _start () at interrupt.s:9
9                   la      t0, handler
(gdb) x/4x 0xff202100
0xff202100:     0x0fc12e4f      0x00000000      0x05f5e100      0x00000000
(gdb) |
```

Figure 24. Examining the Nios V Machine Timer registers.

that the *Machine-mode Interrupt Enable* bit (*MIE*) in this register is now set to 1, meaning that Nios V interrupts are now enabled (*Machine* mode is the only processor mode supported in Nios V).

```
(gdb) info reg mtvec
mtvec          0x0      0x0 <_start>
(gdb) s 2
11                  li      t0, 0b10000000   # set the enable pattern
(gdb) info reg mtvec
mtvec          0x68     0x68 <handler>
(gdb) info symbol 0x68
handler in section .text
(gdb) info reg mie mstatus
mie            0x0      0
mstatus        0x3800   SD:0 VM:00 MXR:0 PUM:0 MPRV:0 XS:0 FS:1 MPP:3 HPP:0 SPP:0 MPIE:0 HPIE:0 SPIE:0 UPIE:0
MIE:0 HIE:0 SIE:0 UIE:0
(gdb) s 3
15                  la      s0, counter      # pointer to counter
(gdb) info reg mie mstatus
mie            0x80     128
mstatus        0x3808   SD:0 VM:00 MXR:0 PUM:0 MPRV:0 XS:0 FS:1 MPP:3 HPP:0 SPP:0 MPIE:0 HPIE:0 SPIE:0 UPIE:0
MIE:1 HIE:0 SIE:0 UIE:0
(gdb) |
```

Figure 25. Examining Nios V control registers.

Clear the breakpoint that was previously set by entering `clear 9`. Enter the command `continue &`. The `&` at the end of this command means "run the program in the *background*," and immediately provide the GDB command prompt so that commands can still be executed. While the Nios V program is running, observe that the *LEDR* lights show a binary counter incrementing once per second. Now, stop the running program by executing the GDB command `interrupt`. GDB will stop the program, in the same manner as when a breakpoint is reached.

Next, enter `break handler` to set a breakpoint at the interrupt handler routine. Note that this code is in a different source-code file, *handler.s*, from the main program. Enter `continue` to run the program until it reaches the breakpoint. Enter `list` to see the first few lines of code in the *handler* routine, as displayed in Figure 26. Execute the five instructions displayed in the figure, using `step 5`. Then, enter `info reg t0` to see the cause of the interrupt (contents of the *mcause* control register). The displayed value `0x80000007`, as seen in Figure 27, shows that a hardware interrupt has occurred (`0x8`) from the device with interrupt number `7`. This is the expected result, as interrupt 7 corresponds to the Machine Timer.

Clear the handler interrupt by entering `clear handler`, and then execute `continue &` to run the program in the *background*. Observe the *LEDR* lights to get an indication of the value of the binary *counter* being displayed. Enter `interrupt` to stop the program and return control to the GDB Client. To see the current value of the *counter* use the command `x counter`. It is possible to change the value of this "variable" by using the `set` command. For example, set the counter to the value `0x3f0` by using `set {int} counter = 0x3f0`. The cast to type `{int}` is required so that GDB knows the *type* of the variable. Enter continue and observe the new value of the counter displayed on the *LEDR* lights. Another way to modify the value of the counter is to first find its address with the command `info address counter`. This command returns the address value `0x64`. Thus, an alternative way to set the counter to the value `0x3f0` is to used the command `set *0x64 = 0x3f0`. Here `*0x64` uses the syntax of the C language to set the *contents* of an address (*pointer*).

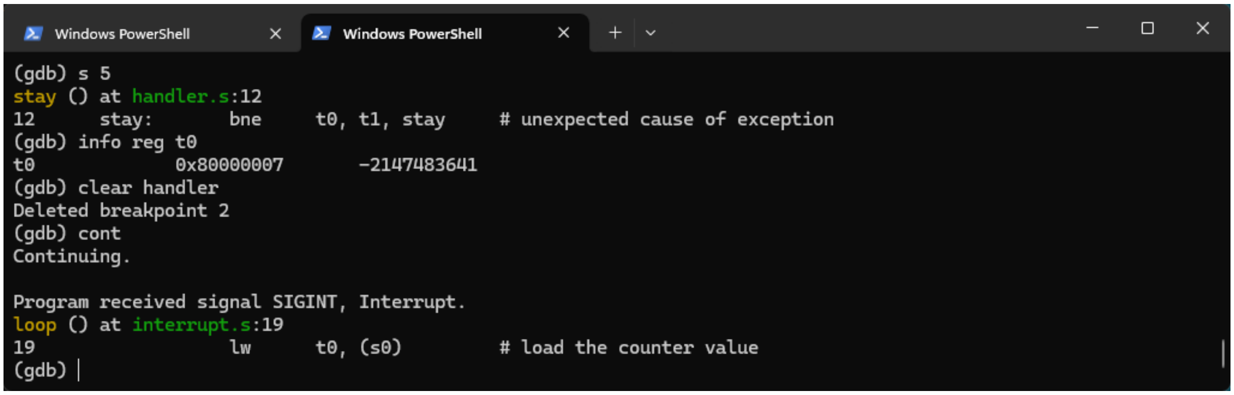
```
Windows PowerShell    Windows PowerShell
(gdb) break handler
Breakpoint 2 at 0x68: file handler.s, line 4.
(gdb) cont
Continuing.

Breakpoint 2, handler () at handler.s:4
4       handler:    addi    sp, sp, -8      # save regs that will be modified
(gdb) l
1       .include "address_map_niosv.s"
2
3       .global handler                     # Trap handler
4       handler:    addi    sp, sp, -8      # save regs that will be modified
5                   sw      t1, 4(sp)
6                   sw      t0, (sp)
7
8                   # check for cause of trap
9                   csrr    t0, mcause      # read mcause register
10                  li      t1, 0x80000007  # should have interrupt bit set (0x8...)
(gdb) |
```

Figure 26. The interrupt *handler* routine.

We are now finished with this example, so enter `detach` to close the connection between the GDB Client and the DE1-SoC board, and then enter `quit`.

3.8 Using a Terminal to Print Text Messages from Assembly Code

In this example we will show how you can use assembly code to "print" text messages to a terminal window. This example can be found in the design files folder `JTAG_UART_s`. As done previously, open two PowerShell terminals and navigate to the proper folder. In one terminal start the GDB Server. In the other terminal, execute `make` to build the program's executable file, and then start the GDB Client. Now, open a *third* PowerShell terminal and navigate again to the `JTAG_UART_s` folder. Execute the command `make TERMINAL`. This command creates a communications link between the PowerShell terminal and the JTAG UART on the DE1-SoC board, which can be used to display text messages.



```
(gdb) s 5
stay () at handler.s:12
12      stay:       bne     t0, t1, stay     # unexpected cause of exception
(gdb) info reg t0
t0             0x80000007       -2147483641
(gdb) clear handler
Deleted breakpoint 2
(gdb) cont
Continuing.

Program received signal SIGINT, Interrupt.
loop () at interrupt.s:19
19                  lw      t0, (s0)         # load the counter value
(gdb)
```

Figure 27. Checking the cause of the interrupt.

In the GDB Client run the assembly program by entering `continue &` to run the program in the *background*. Observe that the message

```
JTAG UART example code
```

appears in the terminal that is connected to the JTAG UART. Also, on a separate line the > prompt is shown. Click on this line, and then type some text with your keyboard. The text is simply echoed back to the terminal window by the assembly program that is running.

In the GDB Client enter `interrupt` to stop the running program. Then, to see how GDB can be used to restart a program enter the command `set $pc = _start`, or (equivalently) `set $pc = 0`. Use `continue &` to restart the program in the *background*, and observe the JTAG terminal window.

Again, enter the `interrupt` command to stop the program. Then, `detach` from the GDB Client and `quit`, and also quit from the GDB Server. Finally, close the connection to the JTAG terminal by typing `^C` in its window (while holding down the `ctrl` keyboard key, press `C`).

3.9 Using a Trap Handler to Catch Exceptions

As a final example using assembly language, we will show how you can handle certain error conditions that may arise in assembly code. This example can be found in the design files folder `errors_s`. As done previously, open two PowerShell terminals and navigate to the proper folder. In one terminal start the GDB Server. In the other terminal, execute `make` to build the program's executable file, and then start the GDB Client.

In case an assembly program causes a Nios V error condition, such as a *misaligned* address, we can include in our assembly-code program a *trap handler* that catches such an error. In the GDB Client enter `list 1,12` as displayed in Figure 28. After first initializing the stack pointer *sp*, the program sets the Nios V *mtvec* control register to the address of a *trap handler* routine. Then, in `Line 6` the program initializes register *t0* to point to a data word in memory, and then in `Line 7` loads that word into register *t1*.

Enter the step command a few times to reach `Line 8`. Enter `step` again to execute this instruction, which increments the value in register *t0*, so that it is no longer a multiple of four (*not* word aligned). Now, enter `step &` to execute, in the *background*, the instruction on `Line 9`. Since the address value in *t0* is not word-aligned, this `lw` instruction causes a Nios V exception and Nios V transfers control to the address in the *mtvec* register, which is (*handler*). The handler code is displayed in Figure 29.

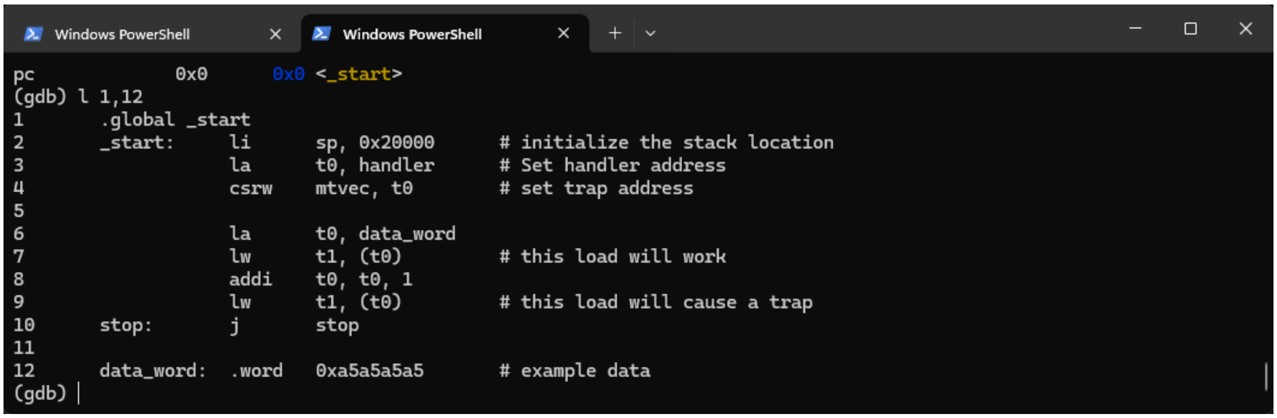


Figure 28. A program that causes an exception.

```
handler:    addi    sp, sp, -4       # save regs that will be modified
            sw      t0, (sp)

            csrr    t0, mcause       # cause of the trap
stay:       bnez    t0, stay         # stay here to allow inspection of exception
                                     # mepc points to the offending instruction
                                     # mtval has the offending address

            lw      t0, (sp)         # restore regs
            addi    sp, sp, 4
            mret
```

Figure 29. The handler routine.

As a result of the exception, the program will be caught in the loop at the label `stay`. Execute the `interrupt` command to return control to the GDB Client. Then, as depicted in Figure 30, execute the command `info reg mcause mepc mtval`. The *mcause* register has the value `4` because this is the exception code that indicates an *address alignment* error. The *mepc* register has the address of the instruction that caused this error, which is `0x20` (`Line 9` in Figure 28), and the *mtval* register shows the value of the offending address, which is `0x29`.

Similar exception-handler code as demonstrated in this example can be included in any assembly-language program, so that inadvertent errors that cause Nios V exceptions can be caught and examined.

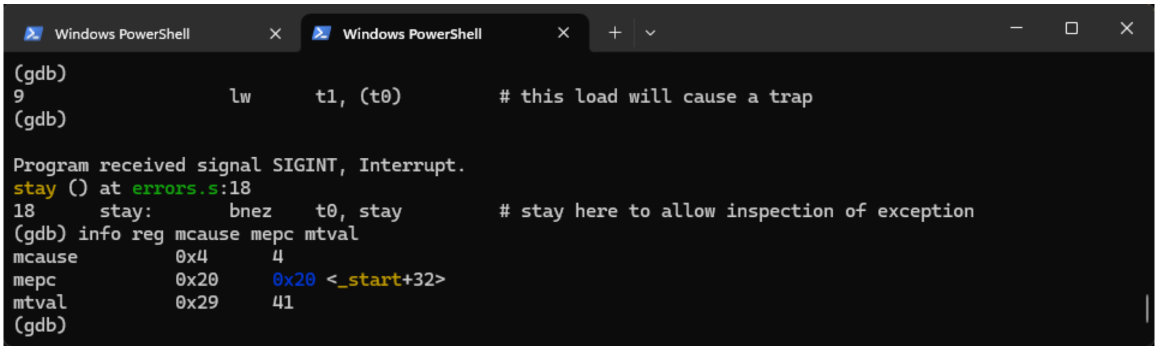


Figure 30. Examining the cause of the exception.

4 Developing and Debugging Nios V C-Language Programs

This section provides examples of using GDB for Nios V with C code. The process is mostly the same as for using assembly-language code, but the *Makefile* and some GDB commands are somewhat different.

4.1 Using I/O Devices with C Code

In this part of the tutorial we will use GDB to run a C program that accesses some simple I/O devices. As in previous examples, open two PowerShell terminals. In each terminal navigate to the folder for this design example, which is `display_C`. Use one terminal to start the GDB server. In the other terminal first execute `make`, which builds the executable program by running the C compiler and linker, and then start the GDB Client, as shown in Figure 31.

Within the GDB Client enter `break main`, and then run the program using `continue`. Type `list` to see the beginning part of the main program for this example, as displayed in Figure 32.

Enter `break 15` to set a breakpoint, and then use `continue` to run to Line 15 in the source code. On the DE1-SoC board, set the *SW* switches to any value of your choosing, for example `0x7f`. Use `step` to execute the C statement on Line 15, and then execute `print /x value` to examine the data that was loaded, as depicted in Figure 33. To see which Nios V register is used to hold this data, execute the `disassemble` command. As illustrated in Figure 34, register *a5* is used to hold this *value*. Display the contents of this register by entering `info reg a5`, as seen in Figure 35. Use `step` to execute the next instruction, and observe that the *LEDR* lights are updated. Then, enter `continue` to run the program until it reaches the breakpoint again at Line 15. Observe that the *HEX0* display is now updated.

```
PS C:\GDB_tutorial\display_C> make
Compiling
C:/intelFPGA_pro/24.1/fpgacademy/AMP/cygwin64/home/compiler/bin/riscv32-unknown-elf-gcc.exe -Wall -c -g -O1 -ffunction-s
ections -fverbose-asm -fno-inline -gdwarf-2  -march=rv32im_zicsr -mabi=ilp32 display.c -o display.c.o
Linking
C:/intelFPGA_pro/24.1/fpgacademy/AMP/cygwin64/home/compiler/bin/riscv32-unknown-elf-gcc.exe -Wl,--defsym=__stack_pointer
$=0x4000000 -Wl,--defsym  -Wl,JTAG_UART_BASE=0xff201000 -lm -march=rv32im_zicsr -mabi=ilp32  display.c.o -o display.elf

PS C:\GDB_tutorial\display_C> make GDB_CLIENT
riscv32-unknown-elf-gdb.exe -silent -ex "target remote:2454" -ex "set $mstatus=0" -ex "set $mtvec=0" -ex "set $mie=0" -e
x "load" -ex "set $pc=_start" -ex "info reg $pc" "display.elf"
Reading symbols from display.elf...
Remote debugging using :2454
main () at display.c:15
15              value = *SW_ptr;    // read SW values
Loading section .text, size 0x1f44 lma 0x10094
Loading section .eh_frame, size 0x4 lma 0x12000
Loading section .init_array, size 0x4 lma 0x12004
Loading section .fini_array, size 0x4 lma 0x12008
Loading section .data, size 0x548 lma 0x12010
Loading section .sdata, size 0x14 lma 0x12558
Start address 0x00010094, load size 9388
Transfer rate: 82 KB/sec, 1564 bytes/write.
pc             0x10094  0x10094 <_start>
(gdb)
```

Figure 31. Starting GDB for the `display_C` example.

```
(gdb) b main
Breakpoint 1 at 0x101ac: file display.c, line 12.
(gdb) cont
Continuing.

Breakpoint 1, main () at display.c:12
12          volatile unsigned int *HEX3_HEX0_ptr = (unsigned int *) HEX3_HEX0_BASE;
(gdb) list
7       int main(void)
8       {
9           int value;
10          volatile unsigned int *SW_ptr = (unsigned int *) SW_BASE;
11          volatile unsigned int *LEDR_ptr = (unsigned int *) LEDR_BASE;
12          volatile unsigned int *HEX3_HEX0_ptr = (unsigned int *) HEX3_HEX0_BASE;
13
14          while ( 1 ){
15              value = *SW_ptr;    // read SW values
16              *LEDR_ptr = value;  // display on LEDR
(gdb)
```

Figure 32. Listing the *main* function.

```
(gdb) b 15
Breakpoint 2 at 0x101c8: file display.c, line 15.
(gdb) cont
Continuing.

Breakpoint 2, main () at display.c:15
15              value = *SW_ptr;    // read SW values
(gdb) s
16              *LEDR_ptr = value;  // display on LEDR
(gdb) print /x value
$1 = 0x7f
(gdb)
```

Figure 33. Loading a variable from an I/O device.

Enter the command `info break` to see the two breakpoints that are currently set, at Line 12 (*main*) and Line 15. Use the `delete` command to clear these breakpoints.

As we are finished with this example, use `detach` to disconnect from the DE1-SoC board and then `quit` from the GDB client. In the GDB Server terminal type `q` to quit.

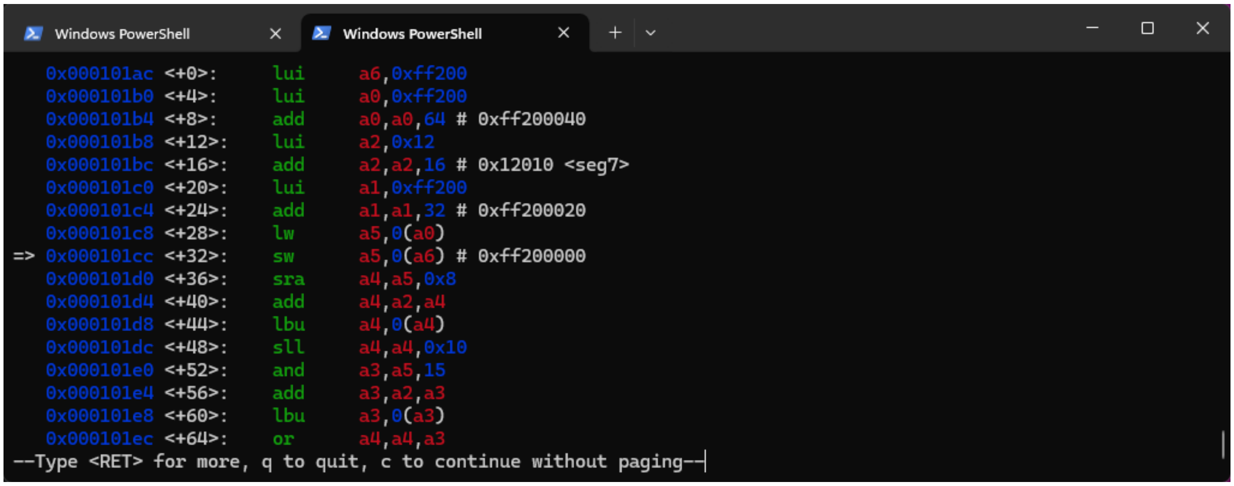


Figure 34. Disassembling C code into assembly code.

4.2 Setting up Your Own C-Code Makefile

It is easy to customize the *Makefile* for the `display_C` example so that you can use it with any C code of your choosing. `Line 3` of the *Makefile* has to specify the name of the C-code file that has the `main` function, which designates the beginning of the program. Any `.h` header files that are used with the program can be listed in `Line 4`. Finally, any additional C source-code files can be listed in `Line 5`.

```
Windows PowerShell    Windows PowerShell
(gdb) info reg a5
a5             0x7f     127
(gdb) s
19              *HEX3_HEX0_ptr = seg7[value & 0xF] |
(gdb) cont
Continuing.

Breakpoint 2, main () at display.c:15
15              value = *SW_ptr;    // read SW values
(gdb) info break
Num     Type           Disp Enb Address    What
1       breakpoint     keep y   0x000101ac in main at display.c:12
        breakpoint already hit 1 time
2       breakpoint     keep y   0x000101c8 in main at display.c:15
        breakpoint already hit 2 times
(gdb) delete
Delete all breakpoints? (y or n) y
(gdb)
```

Figure 35. Stepping through C code.

4.3 Using Interrupts with C Code

This part of the tutorial uses Nios V interrupts with C code. As in previous examples, open two PowerShell terminals. In each terminal navigate to the folder for this design example, which is `interrupt_C`. Use one terminal to start the GDB server. In the other terminal first execute `make`, which builds the executable program by running the C compiler and linker, and then start the GDB Client.

Within the GDB Client enter `break main`, and then run the program using `continue`. Enter `list 37,46` to see the beginning part of the main program for this example, as displayed in Figure 36.

```
Windows PowerShell    Windows PowerShell
(gdb) b main
Breakpoint 1 at 0x10238: file interrupt.c, line 37.
(gdb) cont
Continuing.

Breakpoint 1, main () at interrupt.c:37
37      int main(void) {
(gdb) l 37,46
37      int main(void) {
38          /* Declare volatile pointers to I/O registers (volatile means that the
39           * accesses will always go to the memory (I/O) address */
40          volatile int *mtime_ptr = (int *) MTIME_BASE;
41          volatile int *LEDR_ptr = (int *) LEDR_BASE;
42          volatile int *HEX3_HEX0_ptr = (int *) HEX3_HEX0_BASE;
43
44          set_mtimer();
45          set_itimer();
46          set_KEY();
(gdb)
```

Figure 36. The *main* function.

Execute the `break 44` command and then use `continue` to run to Line 44 of the source code. Before executing the subroutine `set_mtimer()` in the program, use the command `x/4x mtime_ptr`, as depicted in Figure 37, to observe the current contents of the Nios V Machine Timer registers. These registers are referred to as *mtime* and *mtimecmp*, as described in Section 3.6. Now, execute the GDB command `next`, which causes Nios V to execute the `set_mtimer()` subroutine in the C program and then return. The `set_mtimer()` subroutine sets up the Machine Timer for one-second timeouts by reading the value of the *mtime* register, adding 100,000,000 to it, and then storing this result into *mtimecmp*. The *mtime* register then continues to increment at its 100 MHz clock rate. Rerun the command `x/4x mtime_ptr` to see the updated values of the Machine Timer registers.

```
Windows PowerShell    Windows PowerShell
(gdb) b 44
Breakpoint 2 at 0x10240: file interrupt.c, line 44.
(gdb) cont
Continuing.

Breakpoint 2, main () at interrupt.c:44
44          set_mtimer();
(gdb) x/4x mtime_ptr
0xff202100:     0xd4c0598c      0x000001d4      0xa294feff      0x000001c8
(gdb) next
45          set_itimer();
(gdb) x/4x mtime_ptr
0xff202100:     0x7d342c07      0x000001d5      0x596ed6f9      0x000001d5
(gdb) b 48
Breakpoint 3 at 0x1024c: file interrupt.c, line 48.
(gdb) c
Continuing.

Breakpoint 3, main () at interrupt.c:51
51          __asm__ volatile ("csrc mstatus, %0" :: "r"(mstatus_value));
(gdb) |
```

Figure 37. Setting the Machine Timer registers.

Enter `break 48` and then `continue` to Line 48. Then, use `list 50,63` to show the lines of code displayed in Figure 38. The *inline assembly* code shown in the figure sets up Nios V interrupts as needed for the program. Use the command `info reg mstatus mtvec mie` to display the current values of the pertinent registers. Run the program up to Line 63 and then rerun the command `info reg mstatus mtvec mie` to see the updated register values. The *mstatus* register shows that Nios V interrupts are now enabled for machine mode, and the *mtvec* register contains the address of the interrupt `handler` routine. The *mie* register has bits set that enable several sources of interrupts: Nios V software interrupts, machine timer, FPGA interval timer, and *KEY* push-buttons.

Use the command `list 63, 71` to see the remainder of the *main* program. As shown in Figure 40, it contains a loop that reads two variables from memory, called *counter* and *digit*. The *counter* variable is displayed in binary on the *LEDR* lights, and a decimal number is displayed on *HEX0* based on the value of the *digit* variable. Both of these variables are updated by interrupt service routines that respond to hardware timers.

The interrupt *handler* for this program is given in Figure 41. It uses assembly code to read the contents of the Nios V *mcause* control register, and then uses this value to call the appropriate interrupt service routine. Assume that we wish to trace the execution of the program when an *interval timer* interrupt occurs. In the GDB Client enter the command `break itimer_ISR`. Type `continue`. When the breakpoint has been reached, enter `list`, as displayed in Figure 42. Type *step* to execute the next line of code, as depicted in Figure 43. To see the value of the

```
(gdb) b 48
Breakpoint 1 at 0x1024c: file interrupt.c, line 48.
(gdb) cont
Continuing.

Breakpoint 1, main () at interrupt.c:51
51          __asm__ volatile ("csrc mstatus, %0" :: "r"(mstatus_value));
(gdb) l 50,63
50          // disable interrupts
51          __asm__ volatile ("csrc mstatus, %0" :: "r"(mstatus_value));
52          mtvec_value  = (int) &handler; // set trap address
53          __asm__ volatile ("csrw mtvec, %0" :: "r"(mtvec_value));
54          // disable all interrupts that are currently enabled
55          __asm__ volatile ("csrr %0, mie" : "=r"(mie_value));
56          __asm__ volatile ("csrc mie, %0" :: "r"(mie_value));
57          mie_value = 0x50088; // KEY, itimer, mtimer, SW interrupts
58          // set interrupt enables
59          __asm__ volatile ("csrs mie, %0" :: "r"(mie_value));
60          // enable Nios V interrupts
61          __asm__ volatile ("csrs mstatus, %0" :: "r"(mstatus_value));
62
63          *(mtime_ptr + 4) = 1;   // cause a software interrupt
(gdb)
```

Figure 38. Setting up interrupts.

```
(gdb) info reg mstatus mtvec mie
mstatus        0x3800   SD:0 VM:00 MXR:0 PUM:0 MPRV:0 XS:0 FS:1 MPP:3 HPP:0 SPP:0 MPIE:0 HPIE:0 SPIE:0 UPIE:0 MIE:0 HIE:
0 SIE:0 UIE:0
mtvec          0x0      0x0
mie            0x0      0
(gdb) break 63
Breakpoint 2 at 0x10278: file interrupt.c, line 63.
(gdb) cont
Continuing.

Breakpoint 2, main () at interrupt.c:63
63          *(mtime_ptr + 4) = 1;   // cause a software interrupt
(gdb) info reg mstatus mtvec mie
mstatus        0x3888   SD:0 VM:00 MXR:0 PUM:0 MPRV:0 XS:0 FS:1 MPP:3 HPP:0 SPP:0 MPIE:1 HPIE:0 SPIE:0 UPIE:0 MIE:1 HIE:
0 SIE:0 UIE:0
mtvec          0x10350  0x10350 <handler>
mie            0x50088  327816
(gdb) |
```

Figure 39. Nios V control registers.

KEY_dir variable loaded by the program enter `print KEY_dir`. Type `step` to update the `digit` variable. Enter the `step` command until the program returns from this interrupt service routine to the interrupted program.

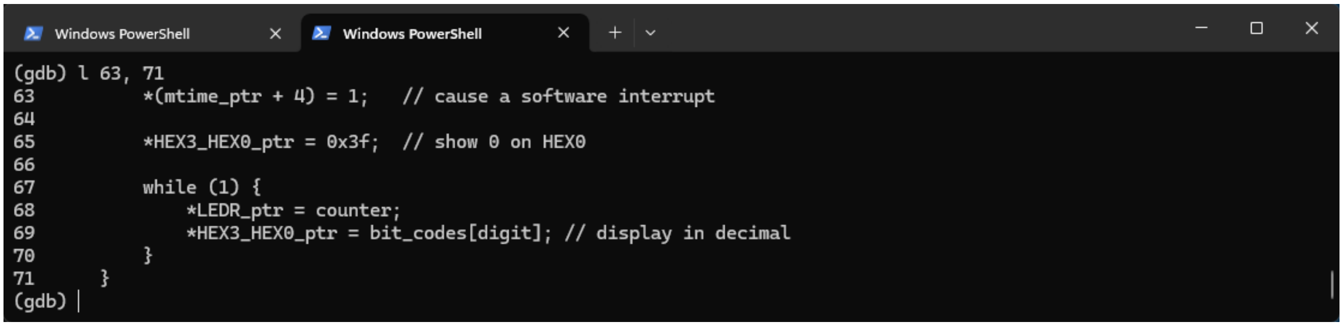


Figure 40. The main program loop.

```
/*********************************************************************
 * Trap handler: determine what caused the interrupt and call the
 *  appropriate subroutine.
 ********************************************************************/
void handler (void){
    int mcause_value;
    __asm__ volatile ("csrr %0, mcause" : "=r"(mcause_value));
    if (mcause_value == 0x80000003) // software interrupt
        SWI_ISR();
    else if (mcause_value == 0x80000007) // machine timer
        mtimer_ISR();
    else if (mcause_value == 0x80000010) // interval timer
        itimer_ISR();
    else if (mcause_value == 0x80000012) // KEY port
        KEY_ISR();
    // else, ignore the trap
}
```

Figure 41. The interrupt handler.

Enter the `delete` command to clear all breakpoints. Then, use `continue &` to run the program in the *background*. Observe on the DE1-SoC board that a binary counter is displayed on the *LEDR* lights, and a digit counter appears on *HEX0*. If you press any *KEY* push-button, then the direction of counting for the digit is reversed.

Stop the running program by executing the `interrupt` command. We are now finished with this example, so use `detach` to disconnect from the DE1-SoC board and then `quit` from the GDB client. Also, quit from the GDB Server in its terminal window.

4.4 Using a Terminal to Print from C Code

As a final example we will show how you can use *printf* in C code from within the GDB Client. This example can be found in the `print_C` design files folder. As in previous examples, open two PowerShell terminals. In each terminal navigate to the folder for this design example. Use one terminal to start the GDB server. In the other

```
(gdb) break itimer_ISR
Breakpoint 4 at 0x10310: file handlers.c, line 57.
(gdb) cont
Continuing.

Breakpoint 4, itimer_ISR () at handlers.c:57
57          *timer_ptr = 0; // clear the interrupt
(gdb) l
52
53      // FPGA interval timer interrupt service routine
54      void itimer_ISR(void){
55          int new_digit;
56          volatile int * timer_ptr = (int *) TIMER_BASE;
57          *timer_ptr = 0; // clear the interrupt
58          new_digit = digit + KEY_dir;    // inc/dec the digit
59          if (new_digit < 10 && new_digit > -1)
60              digit = new_digit;  // decimal (0 to 9)
61      }
(gdb)
```

Figure 42. A breakpoint at an interrupt service routine.

```
(gdb) s
58          new_digit = digit + KEY_dir;    // inc/dec the digit
(gdb) print KEY_dir
$1 = 1
(gdb) s
59          if (new_digit < 10 && new_digit > -1)
(gdb) s
60              digit = new_digit;  // decimal (0 to 9)
(gdb) s
main () at interrupt.c:65
65          *HEX3_HEX0_ptr = 0x3f;  // show 0 on HEX0
(gdb)
```

Figure 43. Servicing an interval timer interrupt.

terminal first execute `make` to build the executable program, and then start the GDB Client. Now, open a *third* PowerShell terminal and navigate again to the `print_C` folder. Execute the command `make TERMINAL`. This command opens a program called the *nios2-terminal*, which allows for text-based communication between the GDB Client and the DE1-SoC board via its *JTAG UART*.

Within the GDB Client enter `break main`, and then run the program using `continue`. Enter `list 4,20`, as displayed in Figure 44. The program uses an endless loop to read from the *SW* switches and *KEY* push-buttons. Whenever any KEY is pressed, the value read from the SW switches at that time is displayed as a hexadecimal value by calling the *printf* library routine. The output from *printf* appears on the *nios2-terminal* window.

Use `continue &` to run the program in the background. Try different settings of the *SW* switches and press any *KEY* push-button to see the corresponding value displayed on the *nios2-terminal*.

```
(gdb) l 4,20
4       int main(void)
5       {
6           int SW_value, KEY_value;
7           volatile unsigned int *SW_ptr = (unsigned int *) SW_BASE;
8           volatile unsigned int *KEY_ptr = (unsigned int *) KEY_BASE;
9           volatile unsigned int *LEDR_ptr = (unsigned int *) LEDR_BASE;
10
11          while ( 1 ){
12              SW_value = *SW_ptr;    // read SW values
13              *LEDR_ptr = SW_value;  // display on LEDR
14
15              // display on Terminal
16              if ((KEY_value = *(KEY_ptr+3)) != 0) {
17                  *(KEY_ptr+3) = KEY_value;               // clear KEY port
18                  printf ("SW: 0x%0x\n", SW_value);       // print in hexadecimal
19              }
20          }
(gdb)
```

Figure 44. The *printf* example.

In the GDB Client enter the `interrupt` command to stop the running program. Then, use `detach` to disconnect from the DE1-SoC board and then `quit` from the GDB client. Also, quit from the GDB Server in its terminal window. Finally, close the *nios2-terminal* by typing `^C` in its window.

Appendix A: GDB Command Reference

Examples of GDB commands are summarized below. You can often execute a command by typing only part of its name: for example **s** executes the *step* command, **b** executes *break*, and **cont** executes *continue*.

step	execute a single source-code line
`<CR>`	simply pressing the `ENTER` key (*Carriage Return*) repeats the *last command*
step &	execute a single instruction, in the *background*
step *n*	execute *n* source-code lines
stepi	execute a single Nios V machine instruction (not used in the tutorial)
continue	run the program from its current location
continue &	run the program from its current location, in the *background*
interrupt	stop the execution of a program that is running in the *background*
next	execute the next source-code line, stepping over a subroutine call
break *k*	set a breakpoint at Line k in source code
clear *k*	clear the breakpoint at Line k
info reg [*name*, ...]	show the current value of Nios V register(s) *name*, ...
info symbol address	give the address of a symbol
info address symbol	give the symbol at an address
info break	list all active breakpoints
set $name = *value*	Set the contents of Nios V register *name* to *value*
set name = *value*	Set the contents of *name* (for example, a variable) to *value*
delete	clear all active breakpoints
x *A*	display the word in memory at address *A*
x/x *A*	display the word in memory in hexadecimal at address *A*
x/4x *A*	display the four words in memory in hexadecimal starting at address *A*
x/xb *A*	display the byte in memory in hexadecimal at address *A*
print *expr*	print the value of an expression (for example, a variable)
print /x *expr*	print the value of an expression in hexadecimal
detach	disconnect the GDB Client from the target
quit	close the GDB client, or GDB Server
load	load the executable program into memory (used in Makefiles)
target remote *port*	connect to remote debugging *port* (used in Makefiles)