

Lab 1

Linux Application Development on Ultra96

Tools:	2018.2
Training Version:	v1
Date:	17 September 2018

© 2018 Avnet. All rights reserved. All trademarks and registered trademarks are the property of their respective owners. All specifications are subject to change without notice.

NOTICE OF DISCLAIMER: Avnet is providing this design, code, or information "as is." By providing the design, code, or information as one possible implementation of this feature, application, or standard, Avnet makes no representation that this implementation is free from any claims of infringement. You are responsible for obtaining any rights you may require for your implementation. Avnet expressly disclaims any warranty whatsoever with respect to the adequacy of the implementation, including but not limited to any warranties or representations that this implementation is free from claims of infringement and any implied warranties of merchantability or fitness for a particular purpose.

Overview

This lab is a short demonstration of the Linux environment on the Zynq UltraScale+ MPSoC illustrating how to debug a Linux software application.

In this lab, you will use the Xilinx Software Development Kit (SDK) to run and debug a simple Linux software application project. The target Ultra96 development board will automatically boot Linux from an SD card.

There are four experiments to be completed in this lab exercise:

1. Create the SDK Workspace and import the application source code.
2. Run the application and examine its behavior. Does it seem like there is a bug?
3. Debug the application. Find and fix the software bug.
4. Profile the application to examine where in the software is the CPU spending the most time.

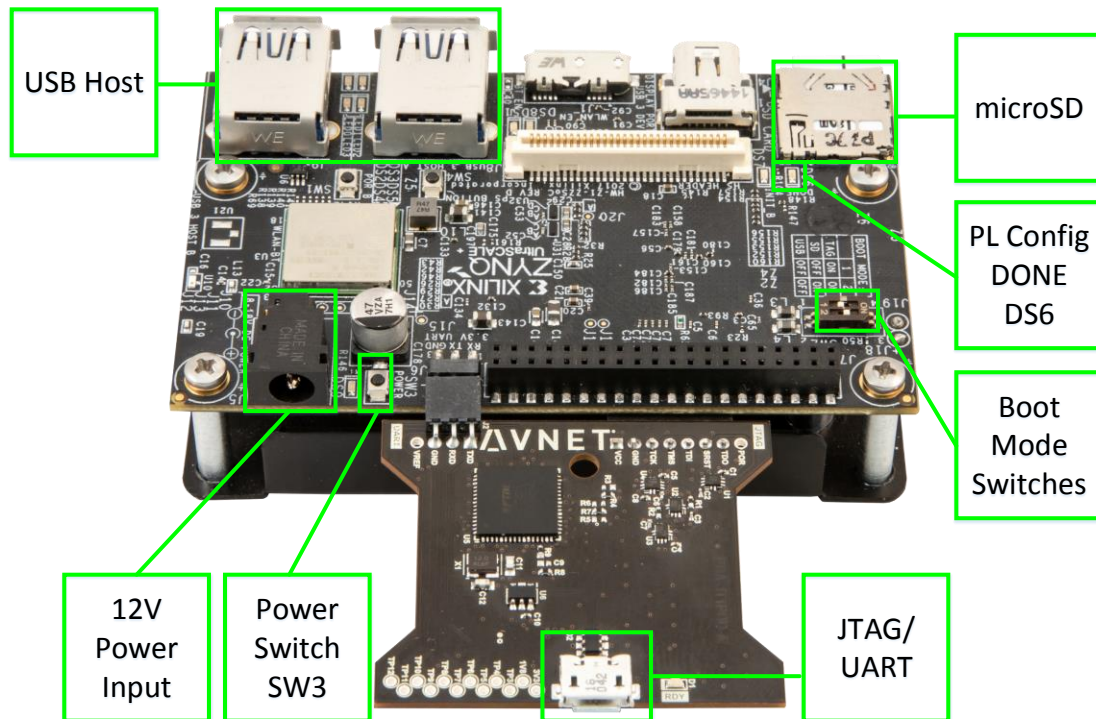
Objectives

When you have completed this lab you will know how to do the following:

- Set up an SDK software application project for debugging
- Use the many features of the SDK debugger

Experiment Setup

1. Connect the 12V power supply to the Ultra96 barrel jack (J5).



2. Connect the USB JTAG/UART port of the Ultra96 JTAG/UART adapter to a PC using a MicroUSB cable.
3. Insert the supplied USB Ethernet dongle into either USB host port on the Ultra96 board and connect the Ethernet port of the dongle directly to the Ethernet port on the host PC.
4. Insert the microSD card into the SD card slot (J4).

Experiment 0: Things to do First


Before we get started with the lab experiments, there are a few things we need to do first

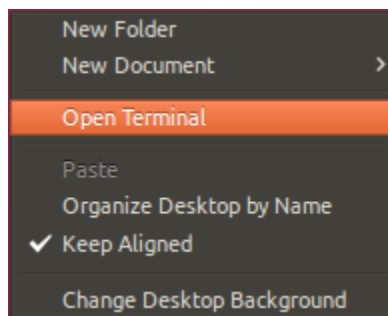
1. Verify the Ultra96 board is assembled and connected to the host laptop as shown in [Experiment Setup](#) and that the boot mode switches are set for SD card boot. The silkscreen on the PCB describes these switch settings.
2. Turn on the Ultra96 board by pressing the small **SW3** power switch near the power input barrel jack. You should see the DS6 (DONE) LED turn green and a short time later the DS8 (WiFi) and DS1 (Bluetooth) LEDs will turn on. Once the board has booted the LED0 LED will blink in a heartbeat pattern.
3. Log into the Ubuntu laptop.

User: <tb>

Password: <tb>



4. Open a command terminal window by clicking the  icon on the Ubuntu taskbar. Alternatively you may right-click on an empty area of the Ubuntu desktop and select **Open Terminal**.



5. Before we can use the Xilinx SDK we need to run a script to set some required environment variables. Do not close this command window. It will be used throughout this lab.

```
$ source /opt/Xilinx/SDK/2018.2/settings64.sh
```

6. Set the IP address of the Ubuntu laptop to 192.168.11.12. The sudo password is <tb>.

```
$ sudo ifconfig eth0 192.168.11.12
```

7. Determine the COM port being used by the USB UART.

```
$ ls /dev |grep USB
```

8. Open a GtkTerm UART console window in a background shell for the COM port used by the Ultra96 board. Substitute <x> for the COM port number determined in the previous step.

```
$ sudo sh -c 'gtkterm -p /dev/ttyUSB<x> -s 115200 &'
```

9. Go to the GtkTerm window and log into the Ubuntu laptop.

User: root
Password: root

10. Set the IP address of the Ultra96 board to 192.168.11.11

```
# ifconfig eth0 192.168.11.11
```

11. Verify the Ethernet link between the laptop and Ultra96 is active.

```
# ping 192.168.11.12
```

Experiment 1: Create the Workspace

The Xilinx SDK uses a 'workspace' to store the source files and metadata for the software application we are developing. A workspace may include more than one software application for the same underlying hardware.

General Instruction:

Launch Xilinx Software Development Kit (SDK) to create the workspace, create a new Linux application, import the software application source, and build an executable that can be run on the Ultra96 target.

Step-by-Step Instructions:

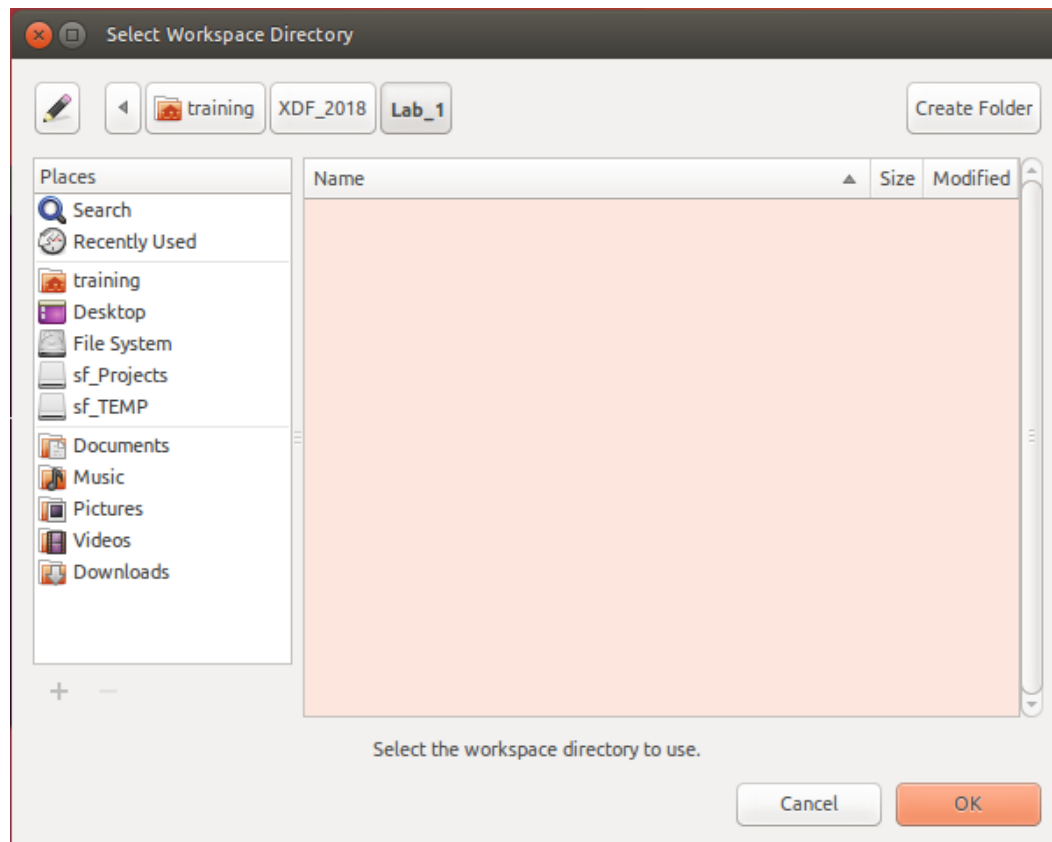
Follow the instructions below to create the workspace, create a new Linux application, import the application source code, and build the executable to run on the Ultra96 board.

1. Launch Xilinx Software Development Kit (SDK) if not already open. Use the '&' operator to open the Xilinx SDK in the background so we can continue to use this command shell.

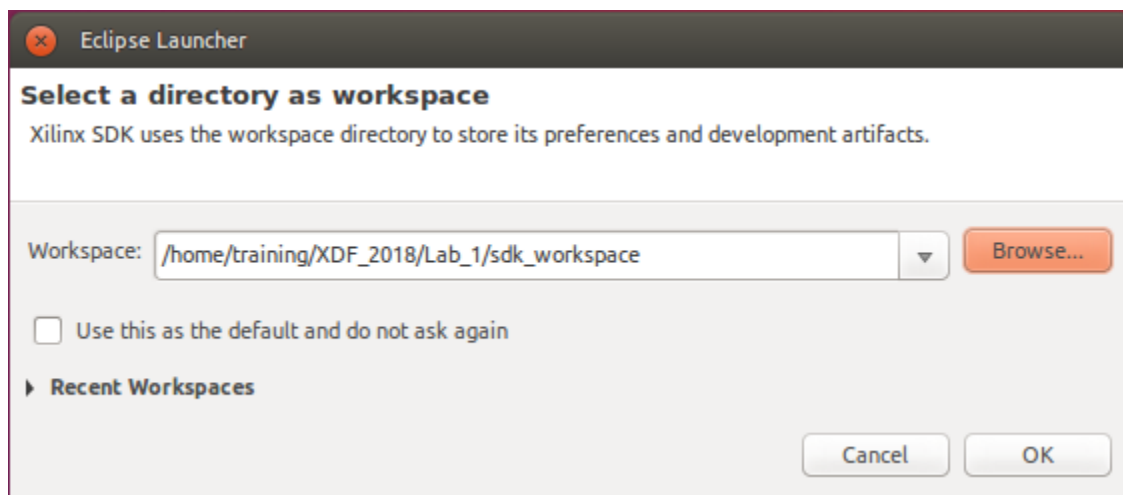
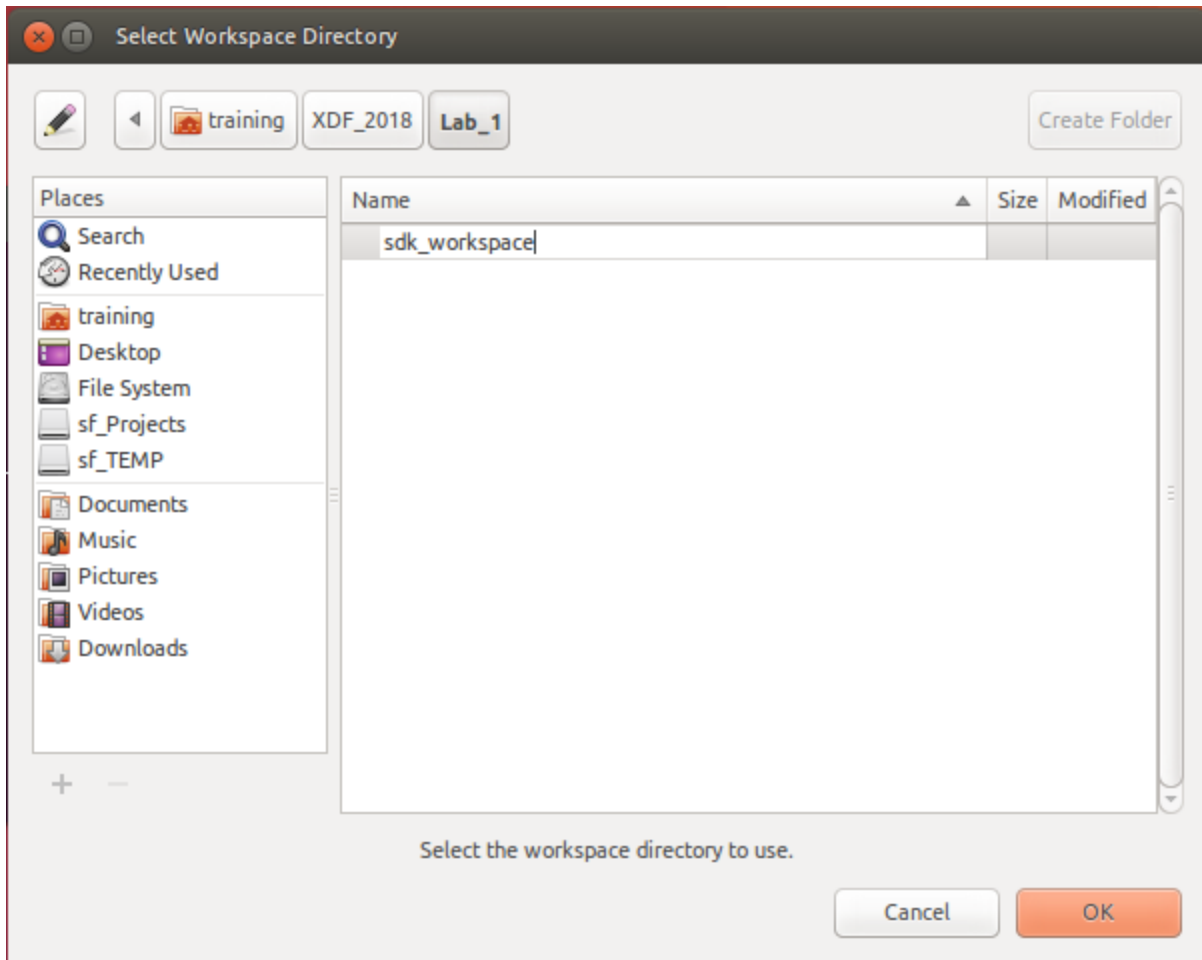
```
$ xsdk &
```

SDK creates a workspace environment consisting of project files, tool settings, and the software application. Once set, you cannot change the location of this workspace. If it is necessary to move a software application to another location or computer, use the Import and Export facilities built into SDK.

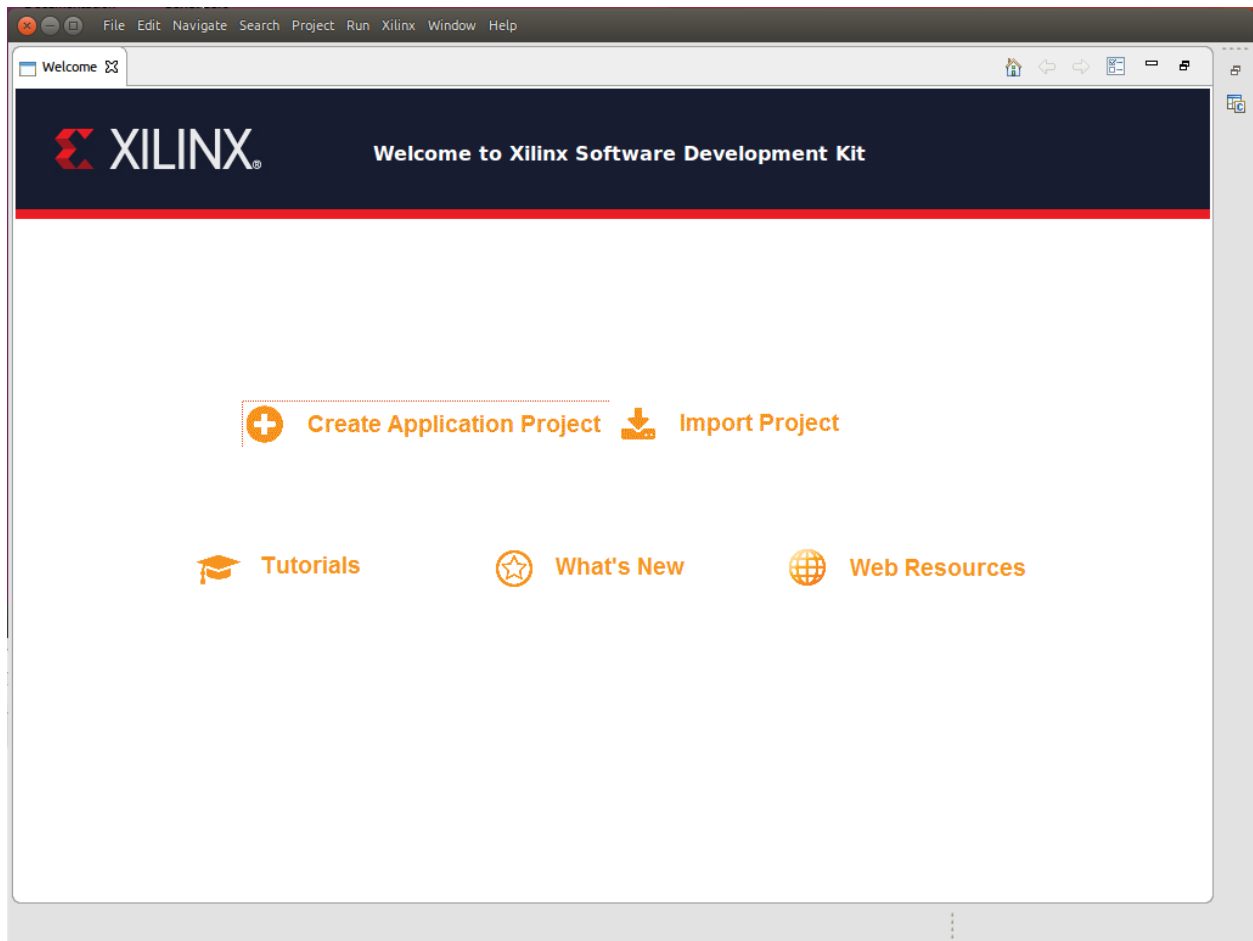
2. Navigate to the `/home/<user>/XDF_2018/Lab_1` folder and select **Create Folder**.



3. Name the folder **sdk_workspace** and press enter, then click **OK** to continue. Click **OK** again to launch the Xilinx SDK in the selected folder.



4. Click on **Create Application Project**.



5. Set the following options and click **Next**:

- **Project name:** matrix_mult
- **OS Platform:** linux
- **Processor Type:** psu_cortexa53

New Project

Application Project
Create a managed make application project.

Project name:

☒ Use default location

Location:

Choose file system:

OS Platform:

Target Hardware

Processor Type:

Endianness: ☒ Little-endian ☐ Big-endian

Target Software

Language: ☒ C ☐ C++

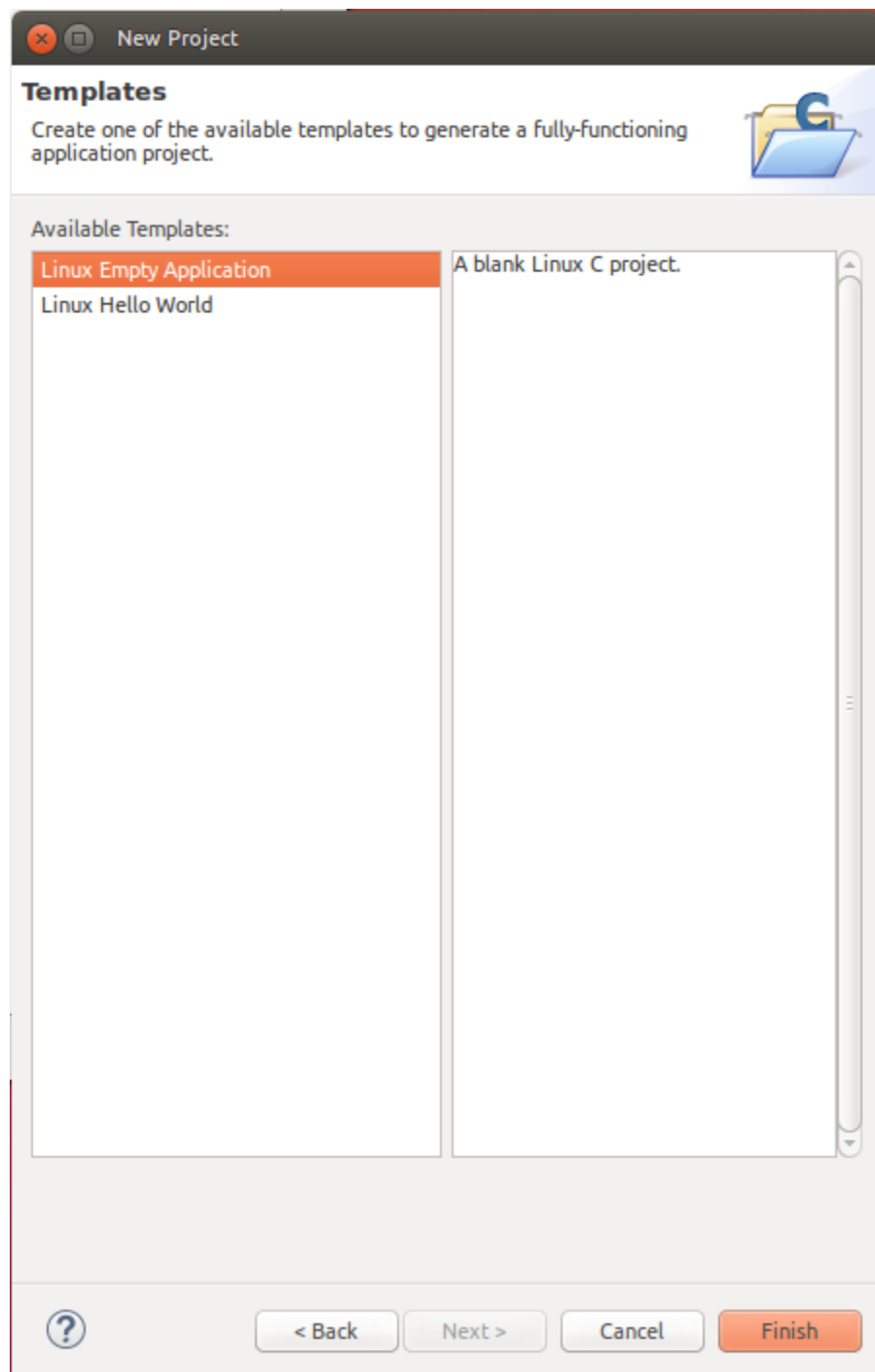
Compiler:

Hypervisor Guest:

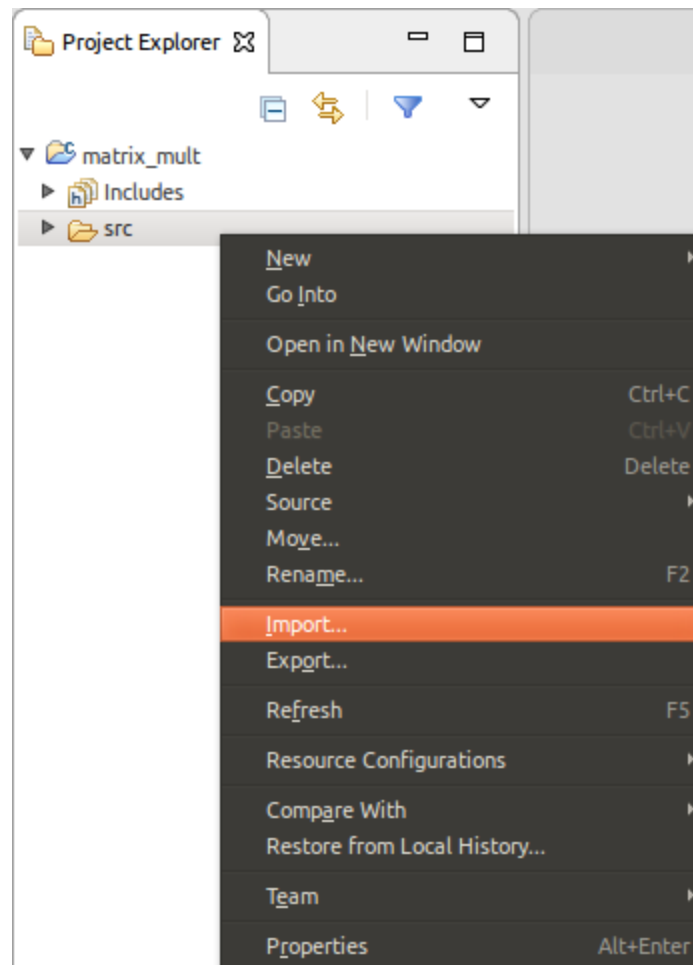
☐ Linux System Root:

☐ Linux Toolchain:

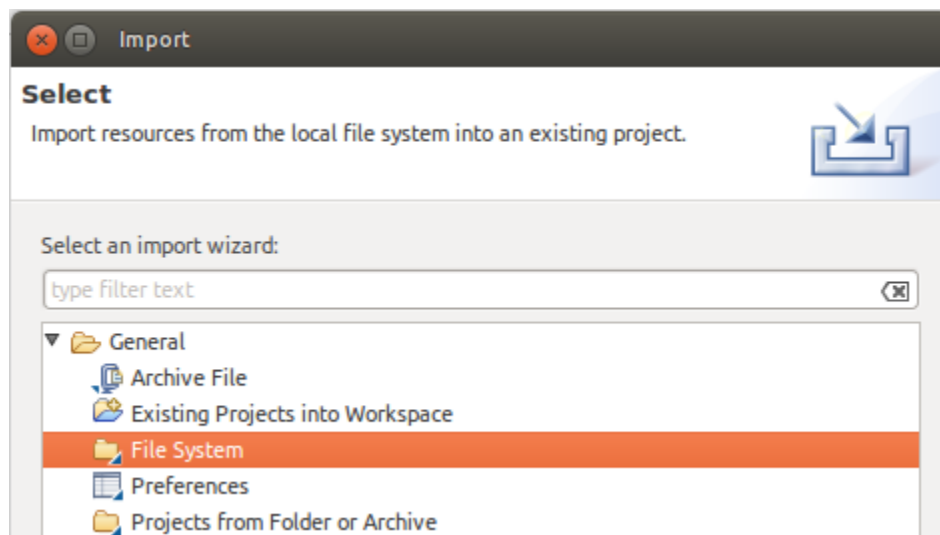
6. Select the **Linux Empty Application** template and click **Finish**:



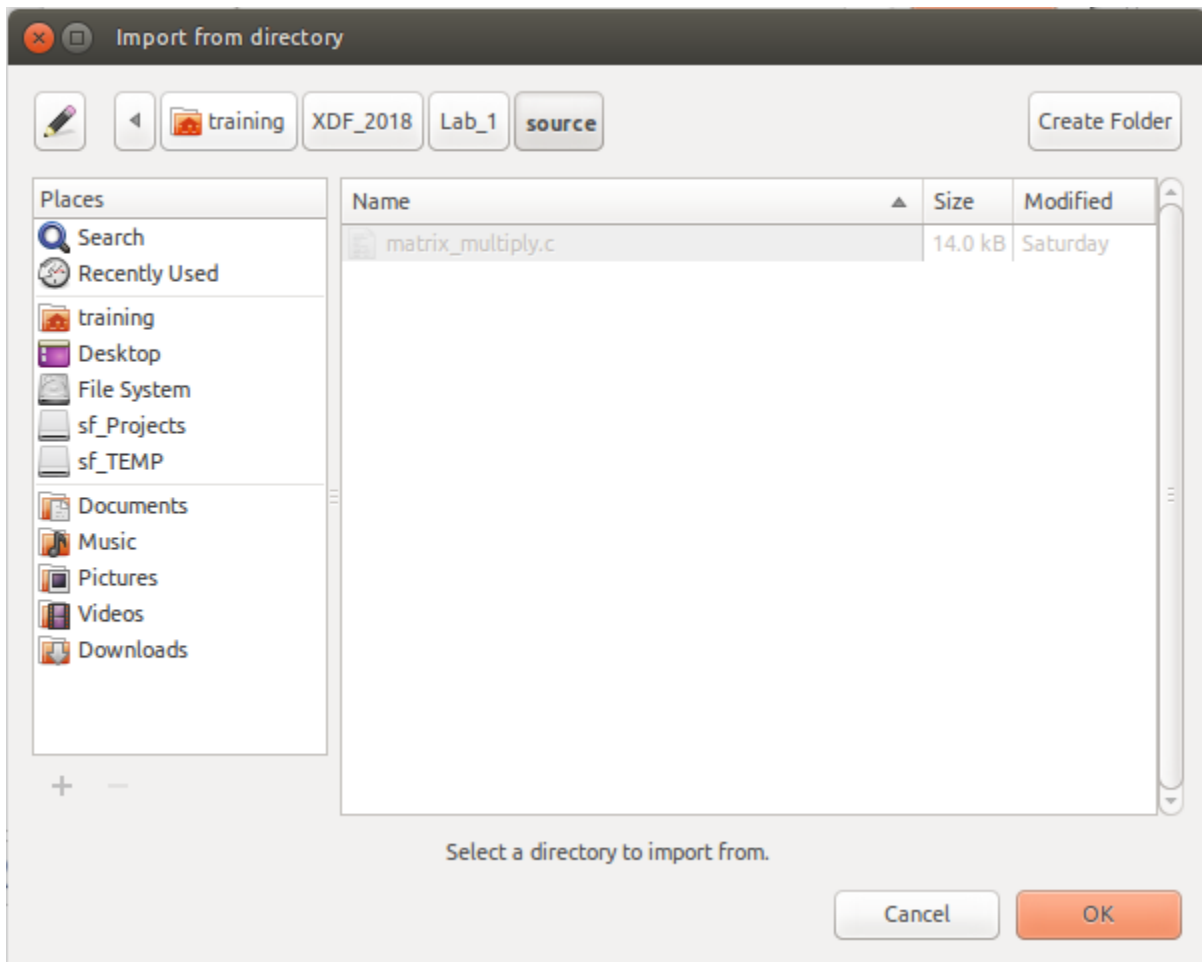
- Expand the new **matrix_mult** application in the **Project Explorer** pane and right-click on the **src** folder and select **Import...**



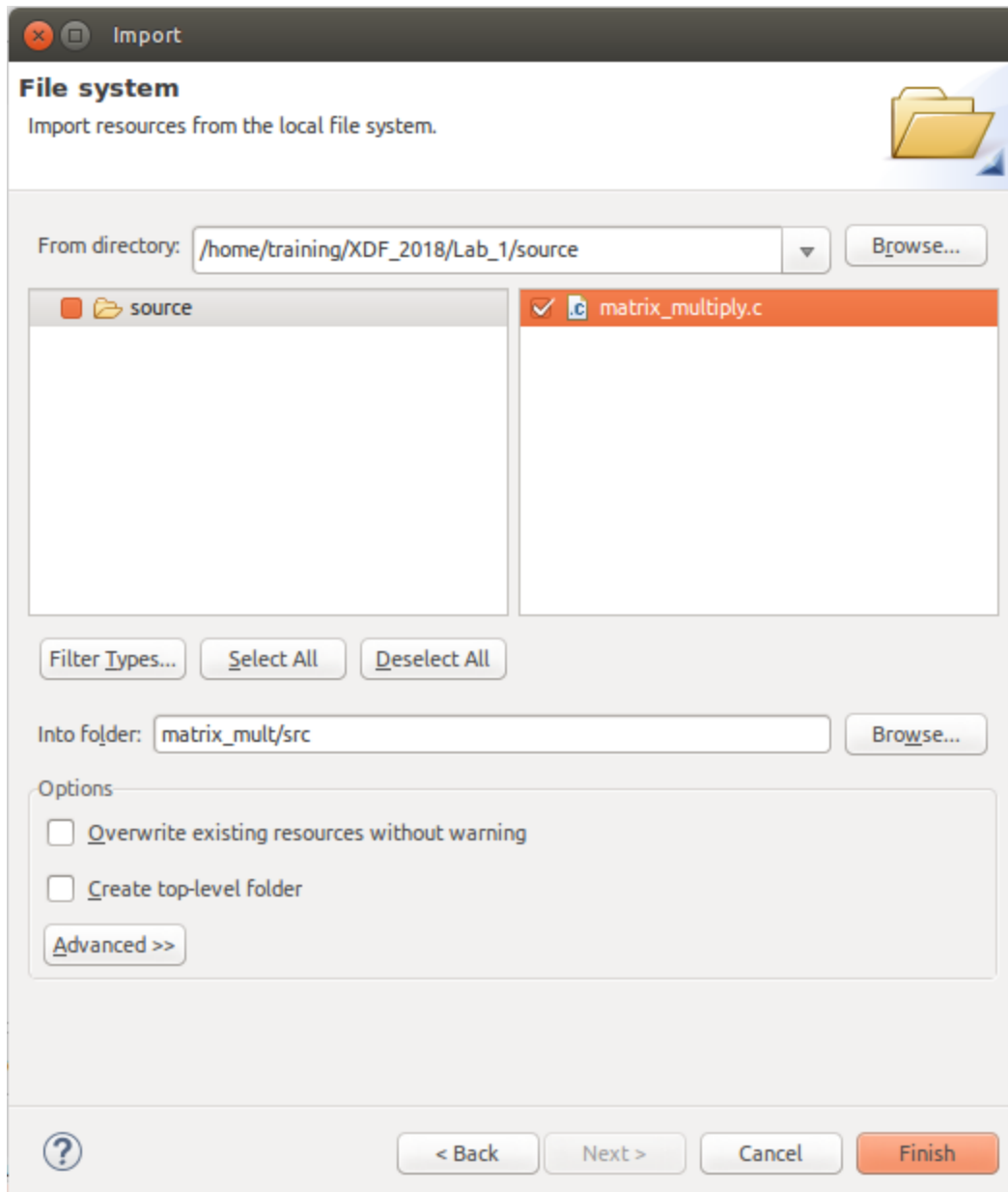
- Expand the **General** category folder and select **File System** then click **Next**



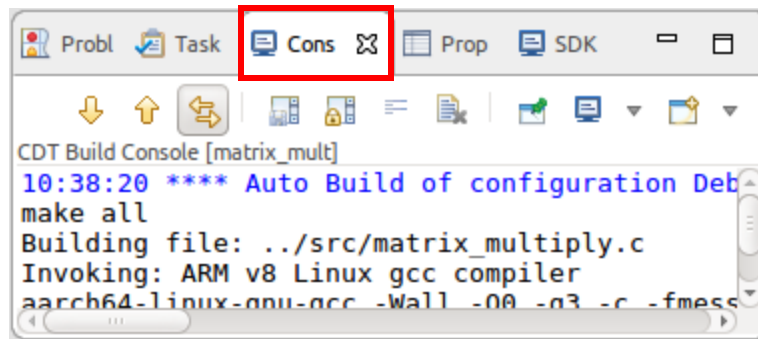
9. Navigate to the **/home/<user>/XDF_2018/Lab_1/source** folder and click **OK**.



10. Select the **matrix_multiply.c** source file and click **Finish**. The source will be imported and the app will build.



11. Verify the app built successfully by double-clicking on the **Console** tab. Double-click the tab again to return the console to its normal position.

A screenshot of the IDE's Console window. The 'Console' tab is selected. The output shows the build process for 'matrix_mult'. It starts with a timestamp and a message: '10:38:20 **** Auto Build of configuration Debug for project matrix_mult ****'. This is followed by 'make all', 'Building file: ../src/matrix_multiply.c', and 'Invoking: ARM v8 Linux gcc compiler'. The compiler command is shown: 'aarch64-linux-gnu-gcc -Wall -O0 -g3 -c -fmessage-length=0 -MT"src/matrix_multiply.o" -MMD -MP -MF"src/matrix_multiply.o.d" -o "src/matrix_multiply.o" ../src/matrix_multiply.c'. The output then shows 'Finished building: ../src/matrix_multiply.c', 'Building target: matrix_mult.elf', 'Invoking: ARM v8 Linux gcc linker', and the linker command: 'aarch64-linux-gnu-gcc -o "matrix_mult.elf" ../src/matrix_multiply.o'. This is followed by 'Finished building target: matrix_mult.elf', 'Invoking: ARM v8 Linux Print Size', and the size command: 'aarch64-linux-gnu-size matrix_mult.elf |tee "matrix_mult.elf.size"'. The size output is shown in a table format: 'text data bss dec hex filename', '6494 696 8 7198 1c1e matrix_mult.elf'. The final line is 'Finished building: matrix_mult.elf.size'.

```
CDT Build Console [matrix_mult]
10:38:20 **** Auto Build of configuration Debug for project matrix_mult ****
make all
Building file: ../src/matrix_multiply.c
Invoking: ARM v8 Linux gcc compiler
aarch64-linux-gnu-gcc -Wall -O0 -g3 -c -fmessage-length=0 -MT"src/matrix_multiply.o" -MMD -MP -MF"src/matrix_multiply.o.d" -o "src/matrix_multiply.o" ../src/matrix_multiply.c
Finished building: ../src/matrix_multiply.c

Building target: matrix_mult.elf
Invoking: ARM v8 Linux gcc linker
aarch64-linux-gnu-gcc -o "matrix_mult.elf" ../src/matrix_multiply.o
Finished building target: matrix_mult.elf

Invoking: ARM v8 Linux Print Size
aarch64-linux-gnu-size matrix_mult.elf |tee "matrix_mult.elf.size"
text data bss dec hex filename
6494 696 8 7198 1c1e matrix_mult.elf
Finished building: matrix_mult.elf.size
```

Experiment 2: Run the Application

Create a run configuration to run the application remotely on Ultra96 board over the Ethernet connection. The matrix multiply application performs two loops to perform the matrix multiplications. There is an outer loop of 20 iterations that runs an inner loop of 1024 32x32 matrix multiplies. The outer while() loop evaluates the exit conditions for a button press on the Ultra96 board, <ctrl>-c press on the host PC, or reaching the maximum loop count. After each completed loop of 1024 matrix multiply the user LEDs on the Ultra96 board will rotate in a clockwise pattern. This is a visual indication on the board that the program is running. The outer loop count in the software counts 20 iterations of the inner loop, so each LED should be toggled on/off 5 times.

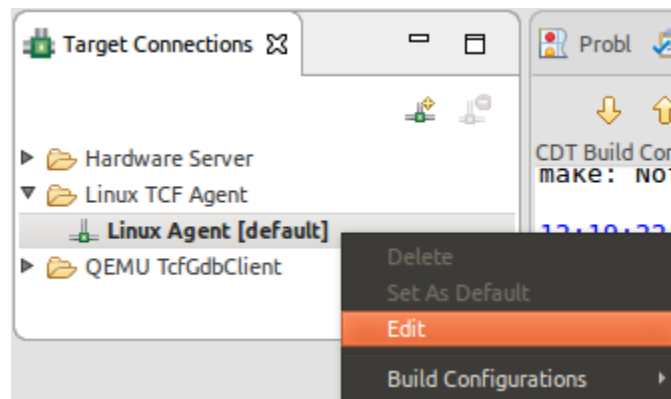
General Instruction:

Use the Xilinx SDK to create a new run configuration and deploy the application code remotely on the target Ultra96 board hardware over an Ethernet connection.

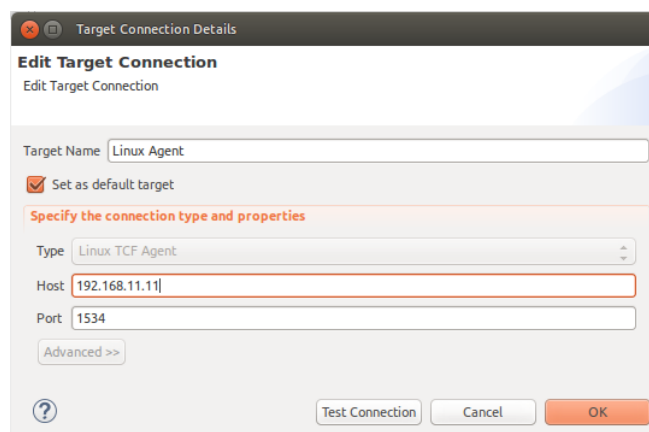
Step-by-Step Instructions:

Follow the instructions below to run the application on the Ultra96 board.

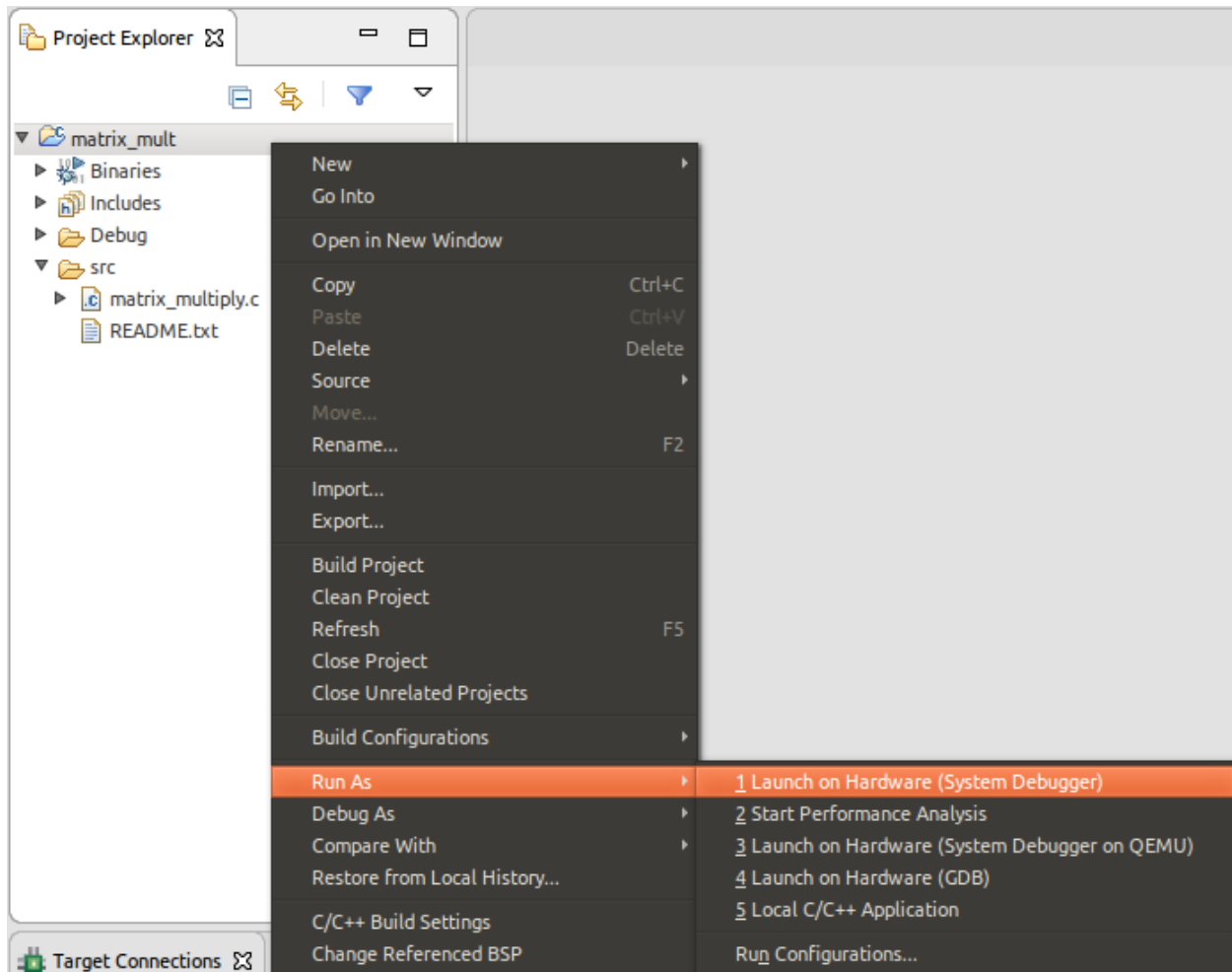
1. Before we can run the matrix multiply application on the Ultra96 board, we need to first configure the Linux TCF Agent with the Ethernet IP address of the Ultra96 board. Expand the **Linux TCF Agent** in the **Target Connections** pane, right-click on **Linux Agent [default]** and select **Edit**.



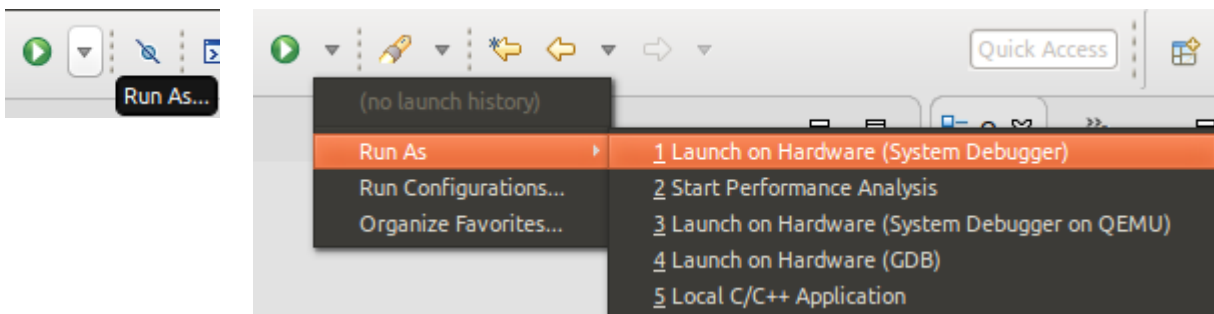
2. Recall from [Experiment 0](#) that we set the IP address of the Ultra96 board to 192.168.11.11. Click in the Host box and change the IP address to this IP address and click OK.



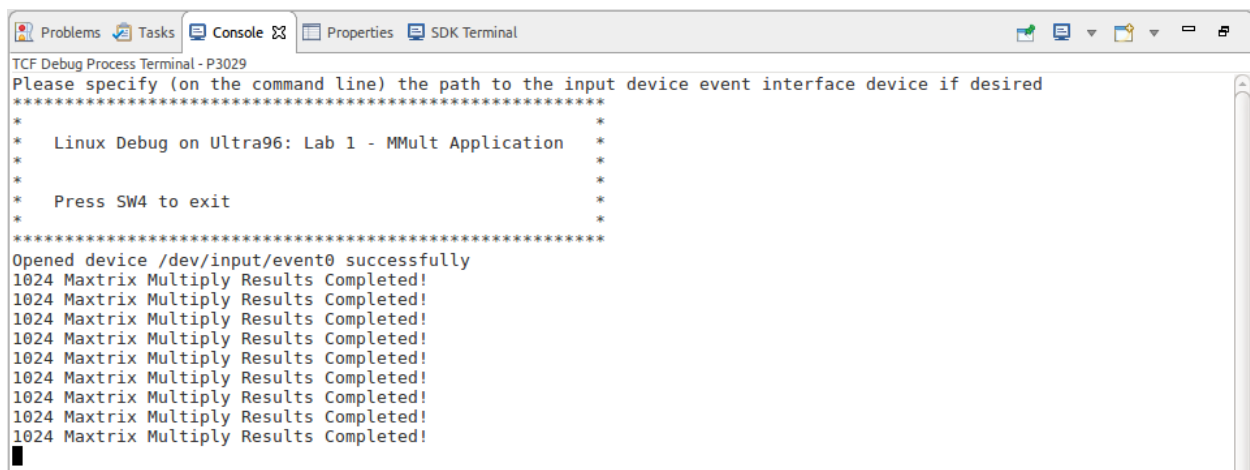
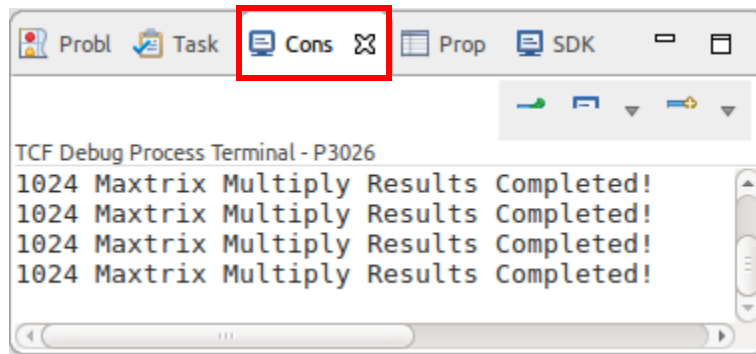
3. Right-click on the `matrix_mult` application in the **Project Explorer** pane and select **Run As → Launch on Hardware (System Debugger)**



Alternatively you may also expand the **Run** icon on the toolbar and select **Run As → Launch on Hardware (System Debugger)**.

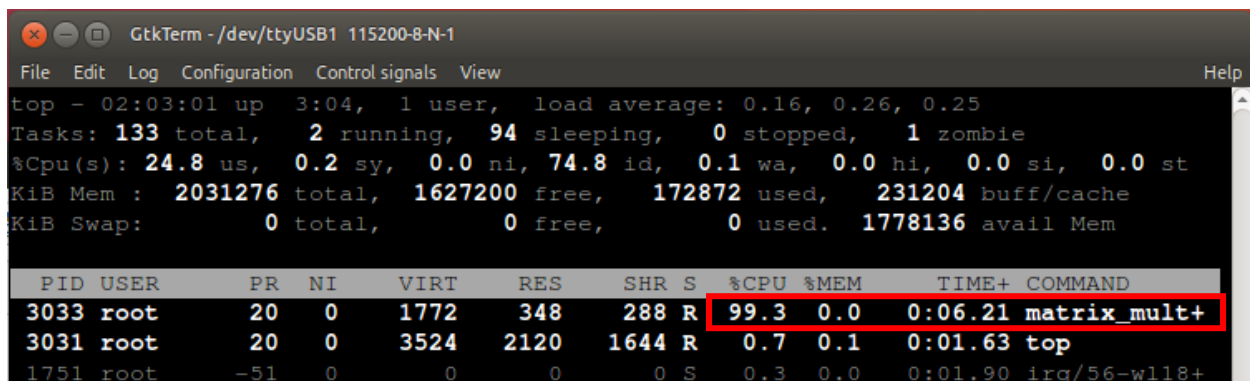


- The app will download to the Ultra96 board and launch automatically. Double-click on the **Console** tab to see it run. Double-click the tab again to return the console to its normal position.



- As you can probably imagine, the matrix multiply is a CPU-intensive task. We can use the Linux `top` command to examine just how busy the processing system is running this application. Go to the GtkTerm window and run this command.

`top`



- Notice that the processing system - all four ARM Cortex-A53 CPU cores - are nearly 100% busy performing the matrix multiplies! This will be interesting to remember for Lab 2 later. Press "q" to stop the `top` application and return to the Linux command line.

7. Also notice that the application never stops. It appears to loop forever - that could be a sign there is a bug in the program. We will check for bugs in [Experiment 3](#). Return to the Xilinx SDK and press <ctrl>-c in the console window where the application results are being displayed to stop the matrix multiply application.

```
1024 Maxtrix Multiply Results Completed!  
1024 Maxtrix Multiply Results Completed!  
1024 Maxtrix Multiply Results Completed!  
1024 Maxtrix Multiply Results Completed!  
^C1024 Maxtrix Multiply Results Completed!  
  
<ctrl>-c Pressed! Exit gracefully.  
Exiting application...
```

Experiment 3: Debug the Application

Running the program like we did in [Experiment 2](#) is a good way to test the program and make sure it works as expected, but if there are obscure bugs in the software then you will likely want to use a debugger to examine the code as it runs on the processor. There is an error in the matrix multiply application that we will use the SDK debugger to find and fix. Recall from Experiment 2 that the matrix multiply application performs an outer loop of 20 iterations that runs an inner loop of 1024 32x32 matrix multiplies. There is a main while() loop that evaluates the exit conditions for a button press on the Ultra96 board, <ctrl>-c press on the host PC, or reaching the maximum loop count.

General Instruction:

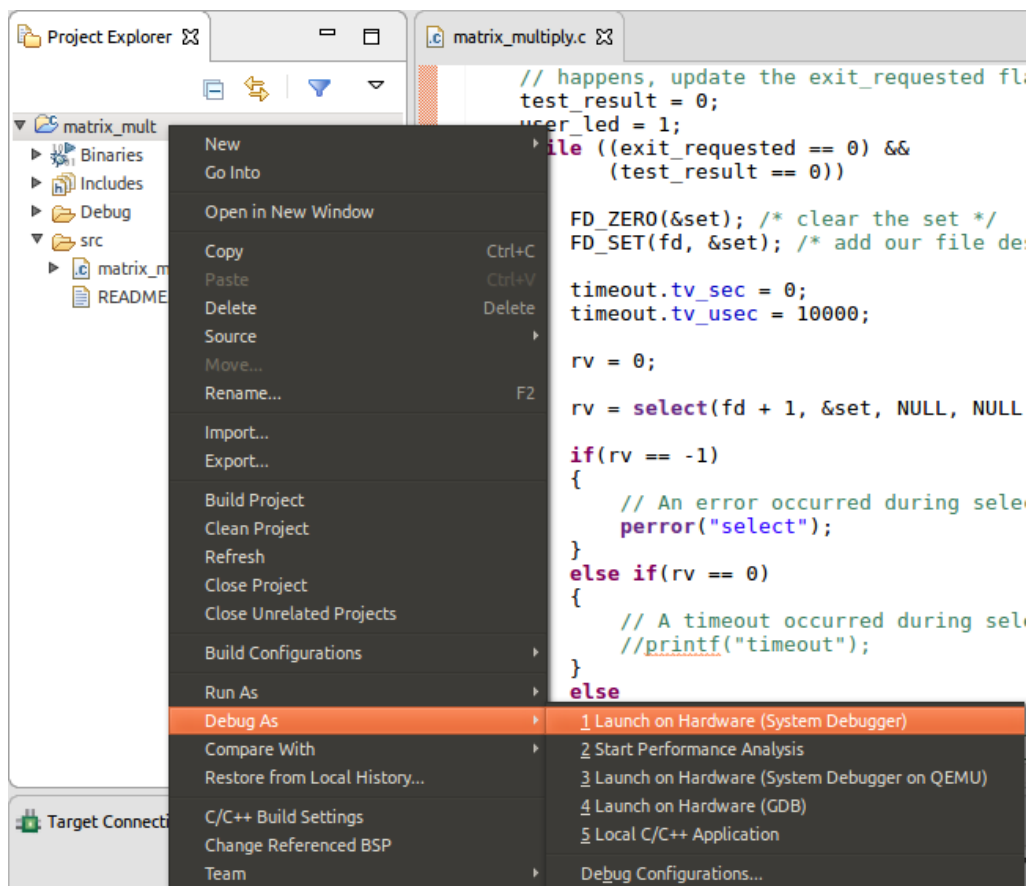
Use the Xilinx SDK to debug the application as it is running on the Ultra96 board. Find and fix the error in the source code.

Step-by-Step Instructions:

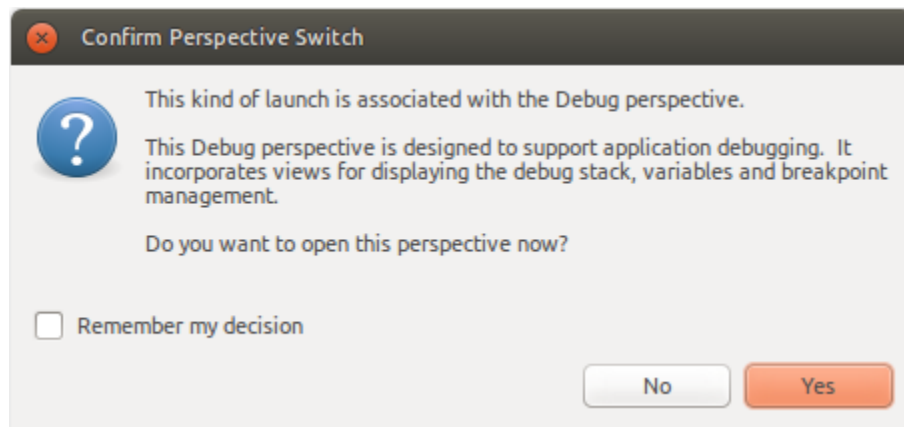
Follow the instructions below to debug the application running on the Ultra96 board.

1. We will reuse the Run configuration from [Experiment 2](#) as our application Debug configuration. Debug configurations simply associate an ELF object file to a target for execution. In this case, the target is a hardware board accessed over a network TCP/IP connection.

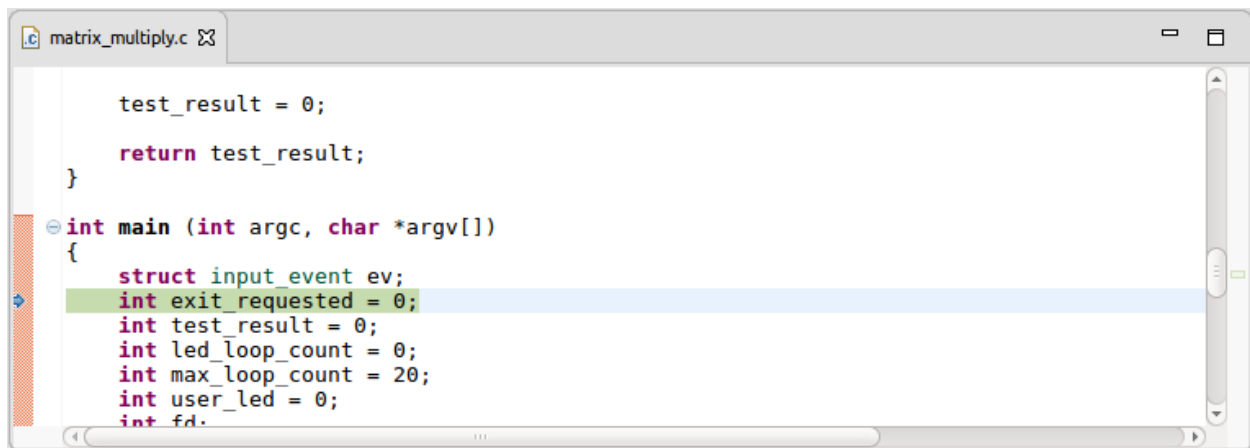
In the **Project Explorer** pane, right-click the **matrix_mult** application project and select the **Debug As → Launch on Hardware (System Debugger)** menu item.



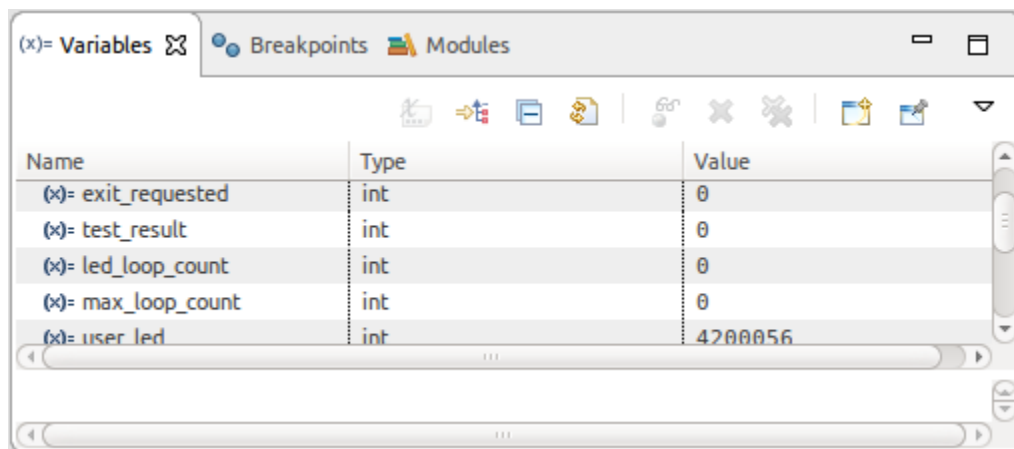
2. The SDK may ask if you want to change to the Debug perspective. Click **Yes**.



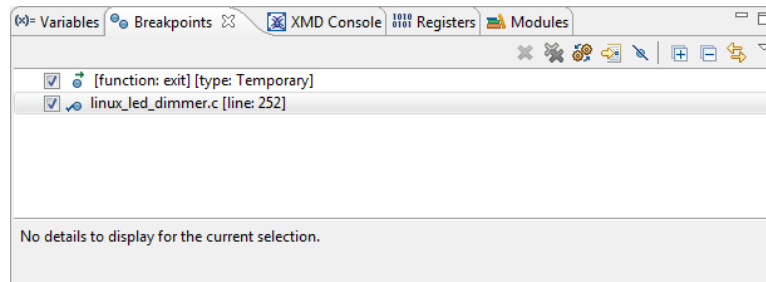
3. After the application debug configuration launches, observe that program operation is suspended at the first executable statement in **main()** (not running).



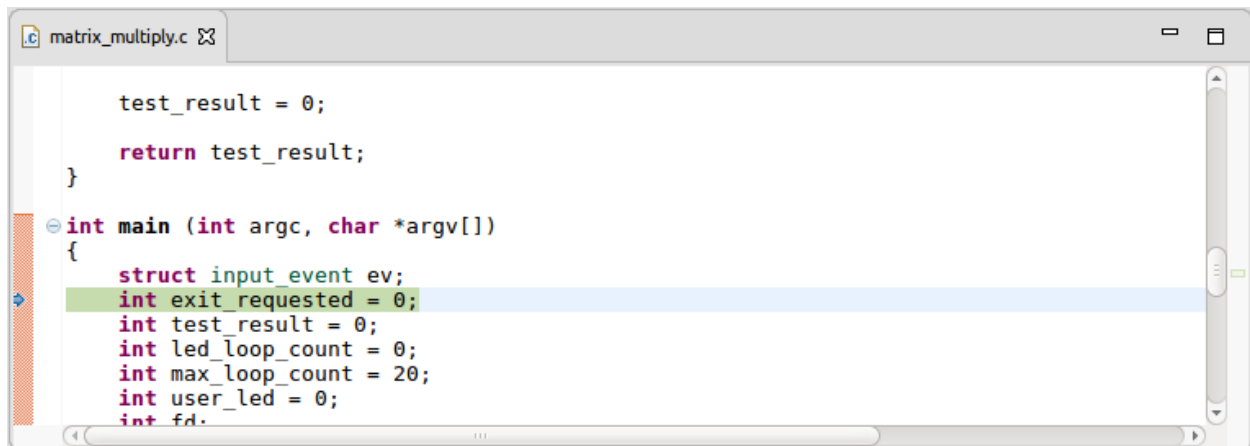
4. Note that local variables for the current function are shown in the **Variables** tab.



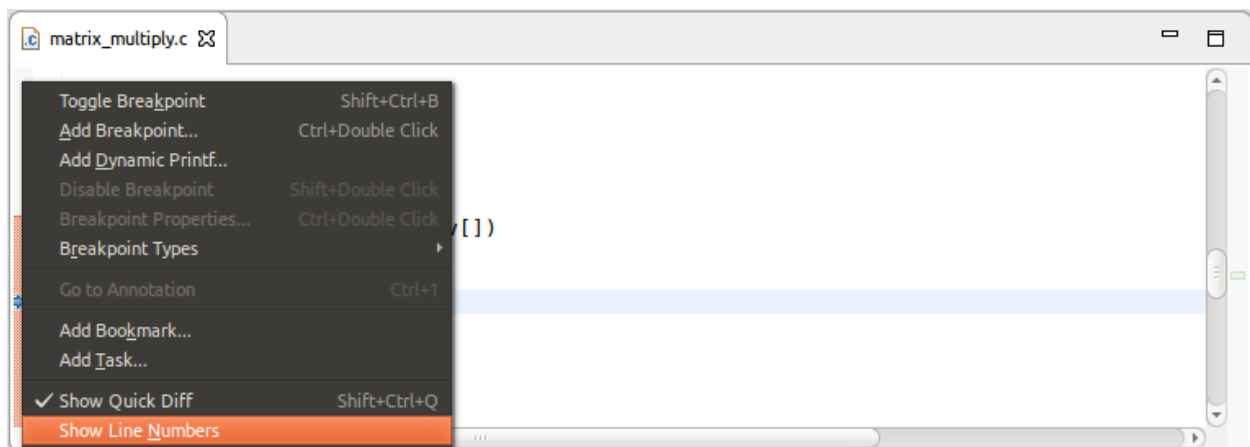
5. Use the debugger to verify the switch toggling by using breakpoints and identify how to run and halt a program thread. Begin by selecting the **Breakpoints** tab. If this tab is not open, you may open the tab by selecting the **Window → Show View→ Breakpoints** menu item.



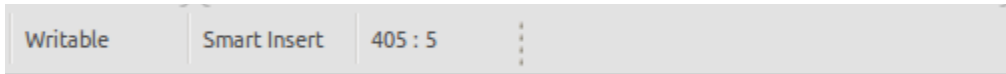
6. The **matrix_multiply.c** source file should be visible in the sources panel. Click within the source code panel to make it active.



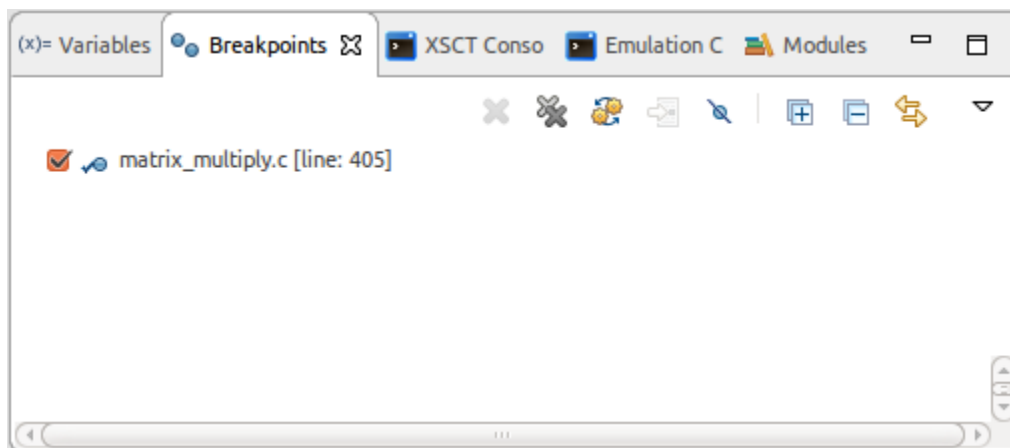
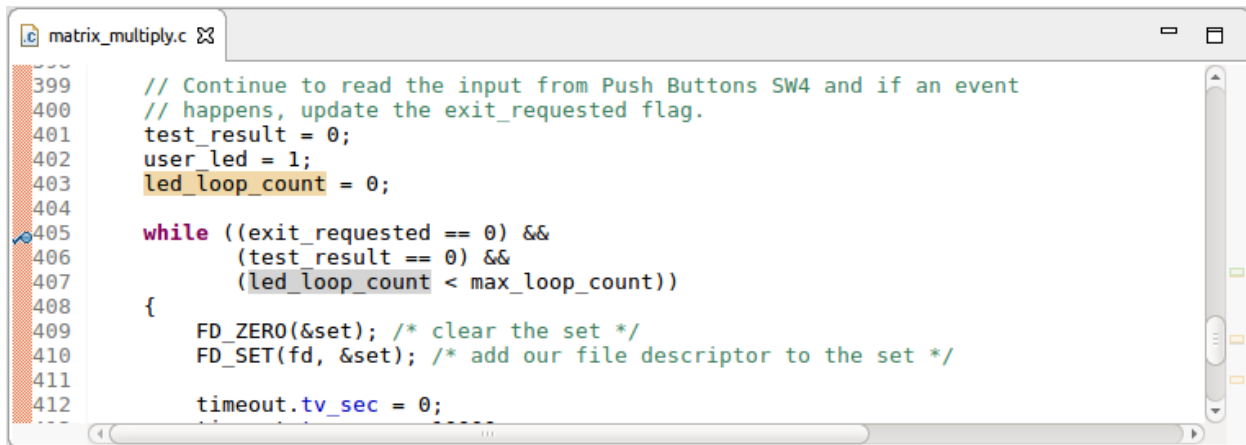
7. The editor window panel has a line numbers feature that is turned off by default. To make it easier to locate lines of code referenced in this lab turn on line numbers in the editor panel. In the editor panel, right-click in the gray space to the right of the pane and select the **Show Line Numbers** option.



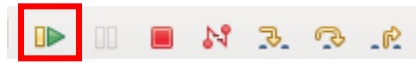
8. Set a breakpoint to check the loop count incrementing. Locate the line of control code that falls within the conditional statement which checks for the loop count to be less than the maximum loop count on line 405. The line number can also be seen in the lower toolbar of SDK. The first set of digits represents the text line number and the second set of digits represents the text column number.



Double-click on line number 405 within the shaded orange strip on the left of the source editor to set a breakpoint there (a check mark becomes visible). Note that the breakpoint that has just been set now appears in the **Breakpoints** panel.



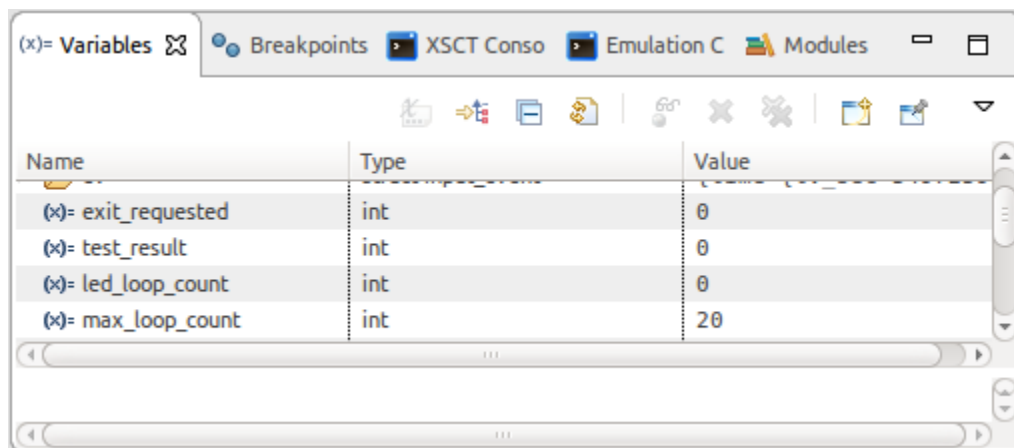
9. Run the program by clicking the **Play/Resume** button (green triangle in the top toolbar) to run the program. If you encounter a series of launch errors, simply terminate and relaunch the debug session to recover.



The program will stop at the breakpoint to evaluate the loop conditions. The program suspends execution at line number 405.

```
matrix_multiply.c
399 // Continue to read the input from Push Buttons SW4 and if an event
400 // happens, update the exit_requested flag.
401 test_result = 0;
402 user_led = 1;
403 led_loop_count = 0;
404
405 while ((exit_requested == 0) &&
406        (test_result == 0) &&
407        (led_loop_count < max_loop_count))
408 {
409     FD_ZERO(&set); /* clear the set */
410     FD_SET(fd, &set); /* add our file descriptor to the set */
411
412     timeout.tv_sec = 0;
```

10. Take some time to scroll through and evaluate each of the variables in the Variables panel. The **led_loop_count** variable is getting assigned to a value of 0 as would be expected since this is the first time entering this while() loop.

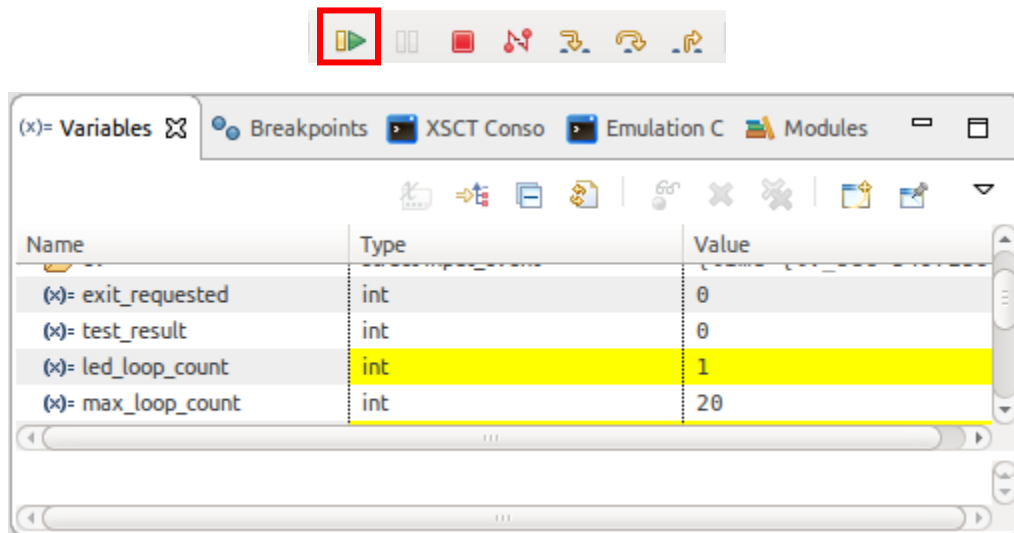


11. The **led_loop_count** variable should increment each time the while() loop is executed. Click on the **Step Over** button a few times to advance code execution ahead so we enter the while() loop.



12. Application execution advances to the next line of code each time the Step Over button is pressed and execution is again suspended. Notice now that the **led_loop_count** variable has not been updated yet.

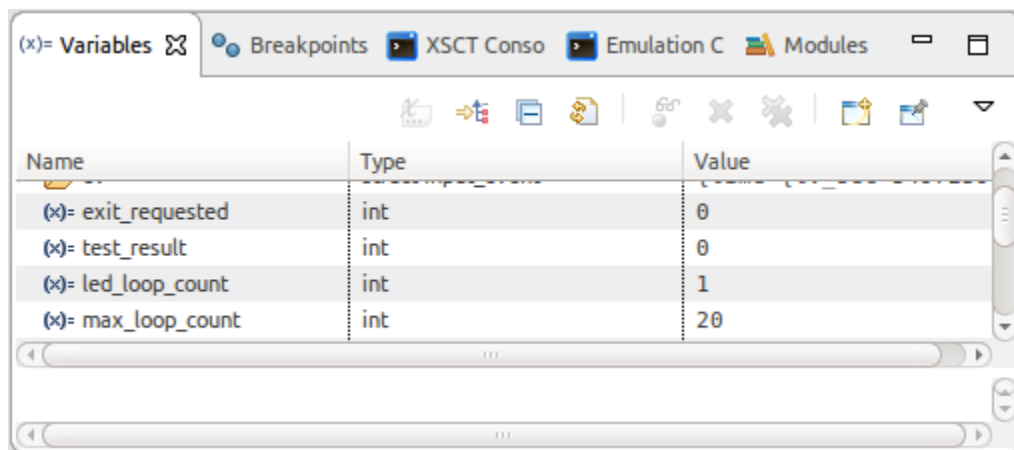
13. Continue executing the program by clicking the **Play/Resume** button (green triangle) to resume the program execution again. Notice that the `led_loop_count` variable has incremented as expected since we just completed an iteration of the `while()` loop.



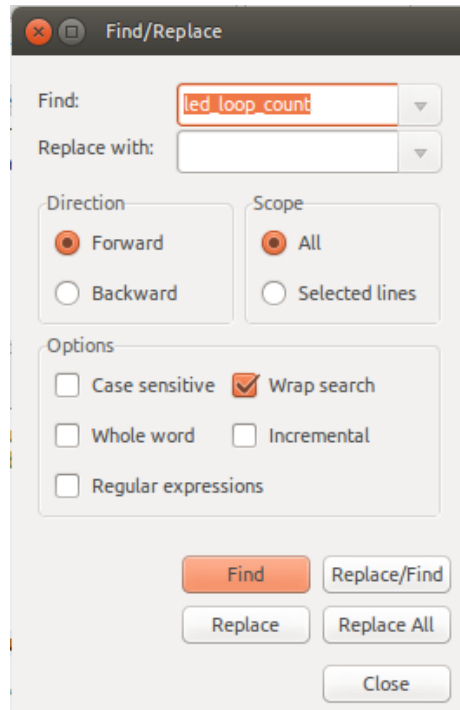
14. Continue executing the program by clicking the **Play/Resume** button (green triangle) to resume the program execution again.



Notice the `led_loop_count` variable has not incremented as expected. This is a clue to the nature of the bug that is being tracked down.



15. Manually looking through the source code main control loop for other instances of the **led_loop_count** variable could take up valuable developer time. Click within the source panel and use the **<ctrl>-f** keyboard shortcut to open the **Find/Replace** tool and enter **led_loop_count** in the **Find** field. Click the **Find** button to locate other instances where this variable is used.



16. We find that the only other code locations where the **led_loop_count** variable is accessed is on line 468 where it is initialized to one before returning to the beginning of the main control loop and again on line 405 where it is evaluated as part of the control loop. Here we see something that is likely the cause of the bug that we have observed. Rather than incrementing the variable, the developer made a mistake by omitting the plus sign and inadvertently made this into an assignment statement. This can be a tricky type of bug to track down because the syntax is correct and the code is operational, but the program does not behave correctly. Now that it has been located, let us correct the increment statement and see if that resolves the errant behavior we have observed.

```
matrix_multiply.c
461     signal(SIGINT, sigint_handler);
462     if(flag)
463     {
464         printf("\r\n<ctrl>-c Pressed! Exit gracefully.\r\n");
465         exit_requested = 1;
466     }
467
468     led_loop_count = 1;
469 }
470
471 printf("Exiting application...");
472 return 0;
473 }
474
```

17. Stop debugger execution by clicking on the **Terminate** button in the Debug control panel.



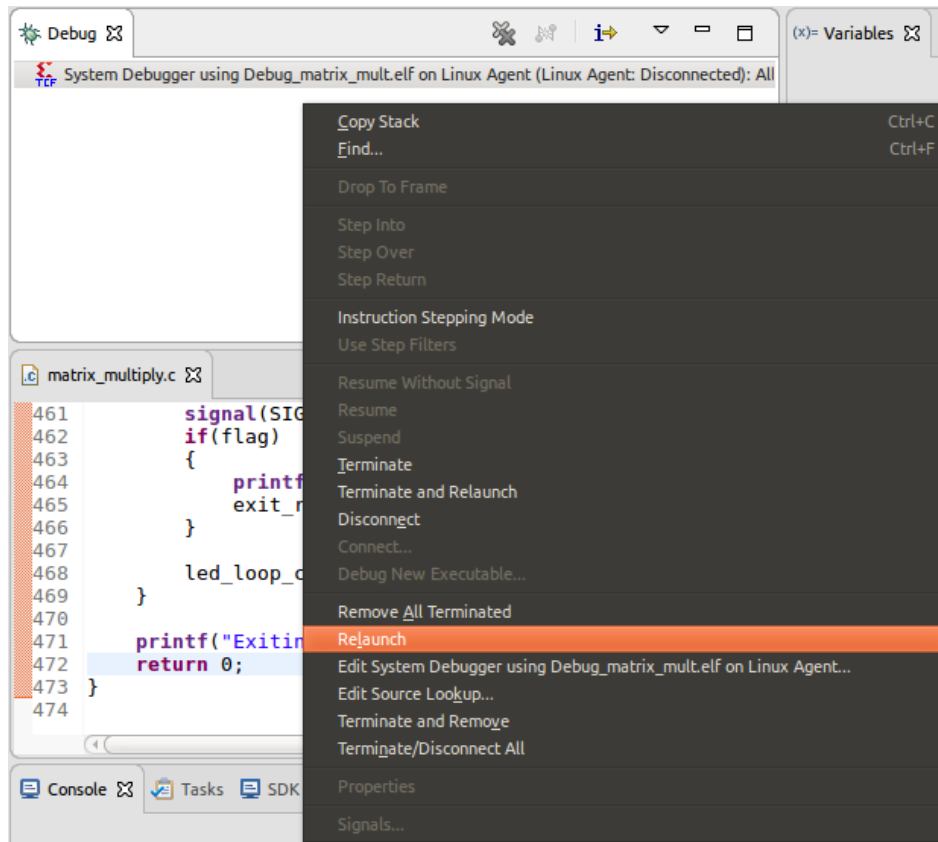
18. Edit the source code to change the assignment statement to an increment statement. After saving the changes to the source file, notice that it is automatically rebuilt in the background.

A screenshot of a code editor window titled 'matrix_multiply.c'. The code is as follows:

```
461     signal(SIGINT, sigint_handler);
462     if(flag)
463     {
464         printf("\r\n<ctrl>-c Pressed! Exit gracefully.\r\n");
465         exit_requested = 1;
466     }
467
468     led_loop_count += 1;
469 }
470
471 printf("Exiting application...");
472 return 0;
473 }
474
```

The code is highlighted in blue, and the line numbers are visible on the left.

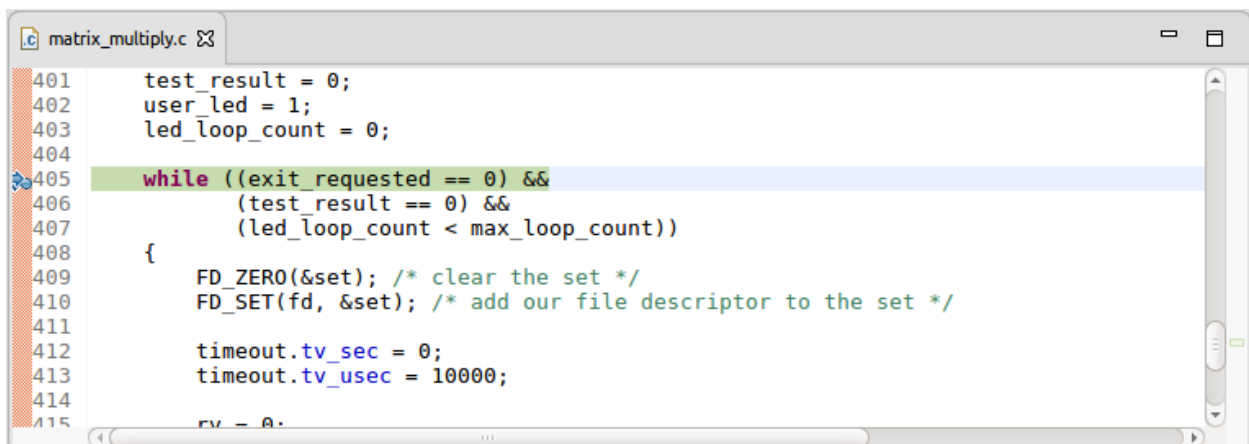
19. Re-launch the Debug configuration so that we can verify that the bug has been corrected. In the **Debug** panel, right-click and select the **Relaunch** menu item. The debug configuration loads the updated target executable to the target platform and halts execution at the main control routine.



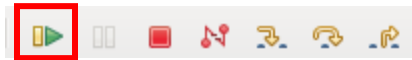
20. Use the debugger to verify the `led_loop_count` incrementing by using breakpoints to evaluate whether the code changes were effective in resolving the bug. Run the program by clicking the **Play/Resume** button (green triangle) to run the program.



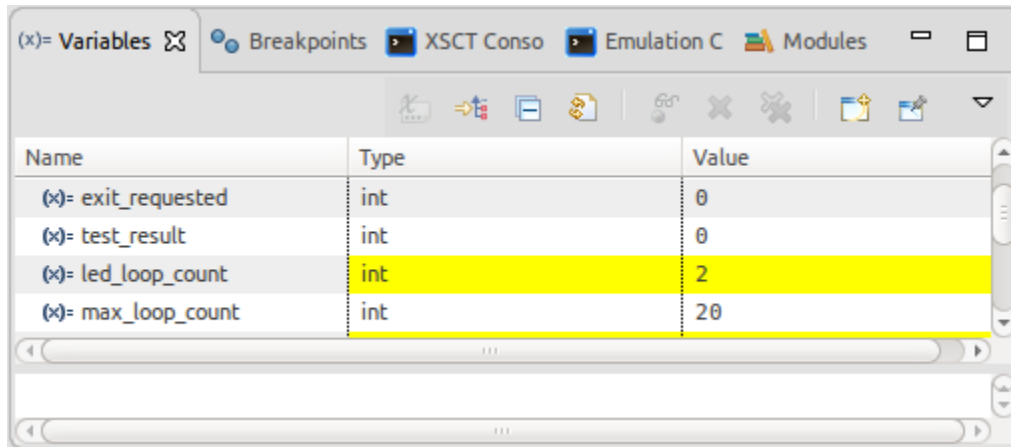
The program again suspends execution at line number 405.



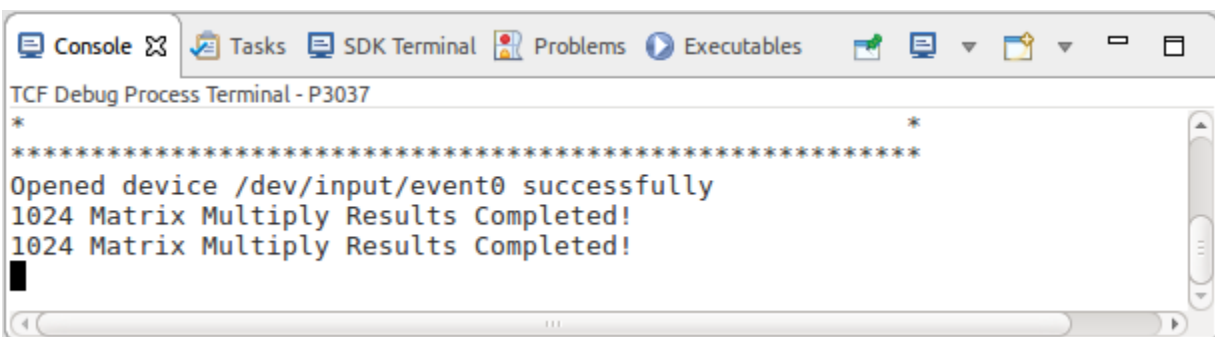
21. Continue executing the program by clicking the **Play/Resume** button (green triangle) to resume the program execution again.



Notice how the `led_loop_count` variable now increments as it is supposed to. The bug has been resolved by correcting the increment statement as performed in step 18.



Name	Type	Value
(x)= exit_requested	int	0
(x)= test_result	int	0
(x)= led_loop_count	int	2
(x)= max_loop_count	int	20



```
TCF Debug Process Terminal - P3037
*
*****
Opened device /dev/input/event0 successfully
1024 Matrix Multiply Results Completed!
1024 Matrix Multiply Results Completed!
█
```

22. Do not close the debugger. We will continue to use this in [Experiment 4](#).

Questions:

Answer the following questions:

- Why did the assignment statement prevent the application from operating properly?

Experiment 4: Profile the Application

Recall in [Experiment 2](#) we identified that the matrix multiply application consumes nearly all of the CPU resources when it is running on the Ultra96 board. The Linux 'top' utility told us this, but that is actually a bit vague since it doesn't tell us *what part* of the running application is consuming so much of the CPU resources. To identify that we can use the SDK TCF Profiler to examine what software application task(s) are using the CPU resources the most.

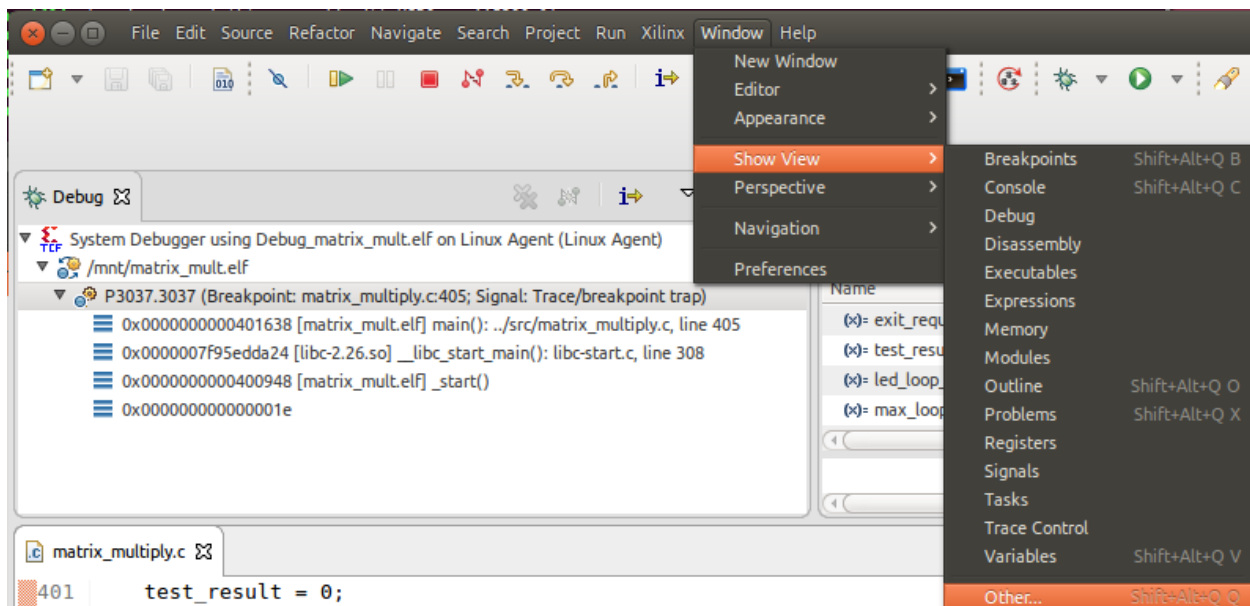
General Instruction:

Use the Xilinx SDK profiler to examine the performance of the application as it is running on the Ultra96 board.

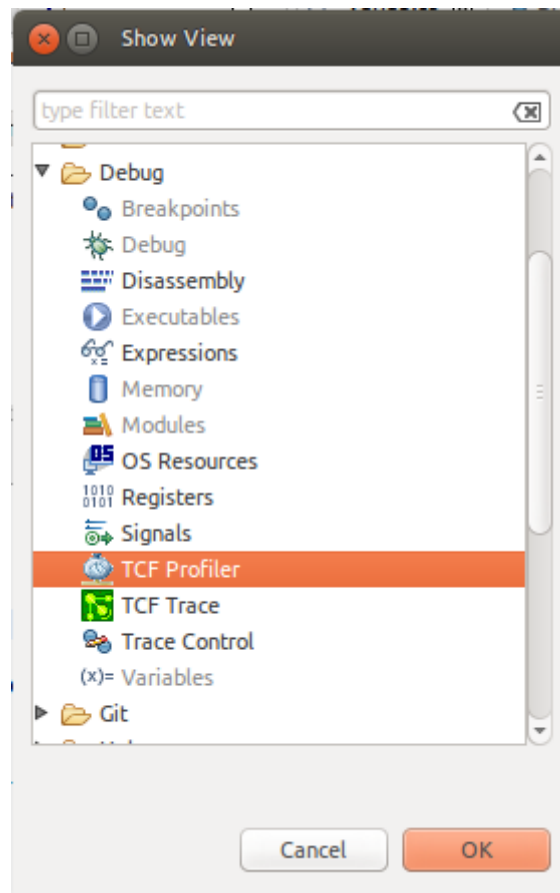
Step-by-Step Instructions:

Follow the instructions below to run the profiler on the application running on the Ultra96 board.

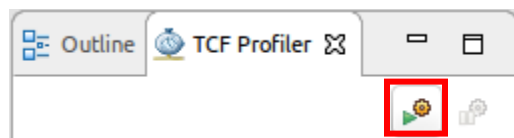
1. With the debugger still running from [Experiment 3](#), launch the TCF Profiler view by navigating the SDK menu bar to **Window** → **Show View** → **Other**.



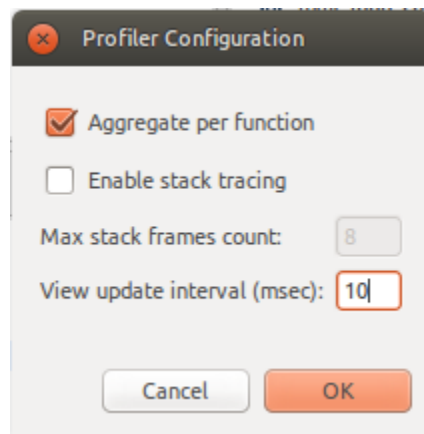
2. In the menu that opens select **Debug** → **TCF Profiler** and click **OK**.



3. Before we launch the profiler we need to change the rate at which it samples the running software. Click on the **Start** icon in the TCF Profiler pane.



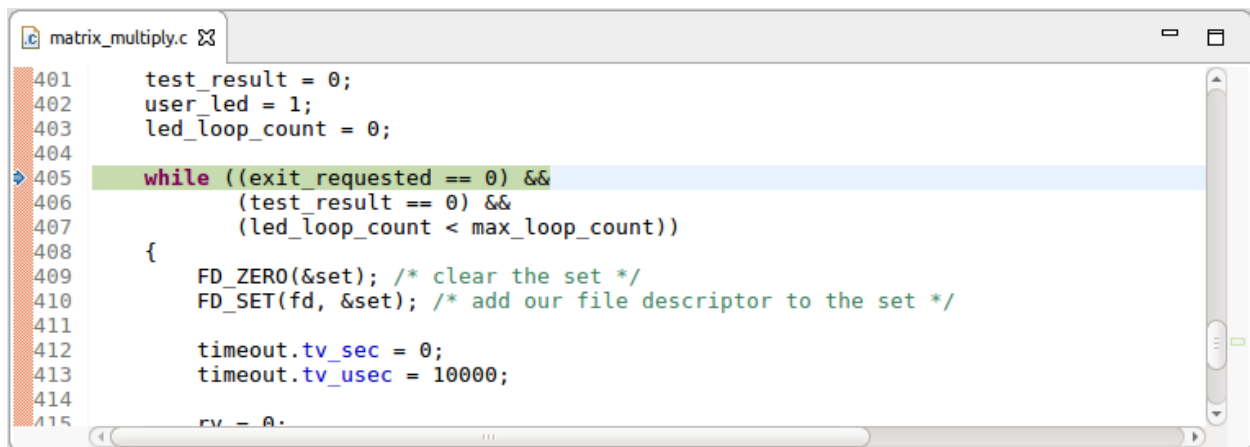
4. Change the **View update interval (msec)** to 10 and click **OK**.



5. Run the program by clicking the **Play/Resume** button (green triangle) to resume the program execution again.



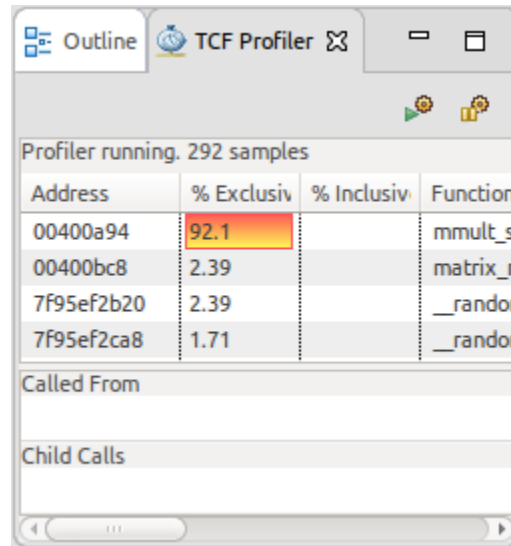
6. Navigate to where we set the breakpoint in the `matrix_multiply.c` file in the source editor and remove the breakpoint by double-clicking on line 405.



7. Run the program by clicking the **Play/Resume** button (green triangle) to resume the program execution again.





8. Examine the TCF Profiler results and notice that most of the program execution time is being spent in the **mmult_sw** function.



The screenshot shows the TCF Profiler window with the 'Profiler running. 292 samples' status. The main table displays the following data:

Address	% Exclusiv	% Inclusiv	Function
00400a94	92.1		mmult_s
00400bc8	2.39		matrix_r
7f95ef2b20	2.39		__randor
7f95ef2ca8	1.71		__randor

Below the table, there are sections for 'Called From' and 'Child Calls', both of which are currently empty.

9. The TCF Profiler will stop when the application stops (when the `led_loop_count` reaches `max_loop_count`). This takes about 45 seconds. You may stop  and disconnect  the debugger from the running system and exit the SDK.

It is clear this software application is very CPU-intensive, and we haven't even examined just how fast the software is able to complete the matrix multiply. If we had identified a multiply performance requirement, could this software have achieved it? Given that we are running Linux on the Ultra96 board that it has its own set of CPU requirements for subsystems like the X windowing manager, TCP/IP stack, USB stack, etc., running a software application that is so CPU intensive can easily cause problems if Linux is not able to keep them serviced because it is too busy doing matrix multiplies. Since we are using a Zynq UltraScale+ device with an abundance of FPGA logic resources, and FPGAs are known to be good at math functions (right?), there must be a way to offload this intensive multiply task from the CPU and accelerate it in the programmable logic. Stay tuned for Lab 2 to learn how to do just that using the Xilinx SDSoC tool.

This concludes Lab1

Appendix A: Getting Support

Avnet Support

- Technical support is offered online through the ultra96.org website support forums. Ultra96 users are encouraged to participate in the forums and offer help to others when possible.
- To access the most current collateral for the Ultra96, visit the community support page (www.ultra96.org/content/support) and click one of the icons shown below:



Support Forums



Documentation



Reference Designs
Tutorials

- Ultra96 Documentation
<http://ultra96.org/support/documentation/24166>
- Ultra96 Reference Designs
<http://ultra96.org/support/design/24166/156>

Xilinx Support

The following technical support options are available to Xilinx customers:

- Technical information is available online 24 hours a day from the [Support website](#)
- Technical Support staff are available to respond to your questions in the [Community Forums](#)
- Individual assistance from Xilinx Technical Support **may** be available through [Service Portal](#)
- Phone support is **only** available with an active open case number

Global Phone Numbers

Region	Language	Phone**	Support Hours*
North America	EN	1 800-255-7778 or +1 408-879-5199	M-F 7:00 -17:00 PST
Europe, Middle East and Africa	EN, DE, FR	00 800-5152-5152 or +353 1-461-5700	M-F 8:00 -17:00 GMT
China	CH (Mandarin), EN	+86 800 988 0218 +86 400 880 0218 (Mobile Phone)	M-F* 9:00 -18:00 CST
Taiwan	CH (Mandarin), EN	+886 2-8176-1060	M-F 9:00 -18:00 CST
Hong Kong	CH (Mandarin), EN	+852 3187-3855	M-F 9:00 -18:00 CST

* Support hours listed apply for both standard and daylight savings (summer) time. Please check the [Technical Support Holiday Calendar 2016](#) for support availability during holidays in your region.

** 00 800-5152-5152 is a international free phone (toll free) number available in the following countries: Austria, Belgium, Denmark, Finland, France, Germany, Ireland Israel, Italy, Luxembourg, Netherlands, Norway, Portugal, Spain, Sweden, Switzerland, and United Kingdom. All other countries must use +353 1-461-5700.

** For the numbers listed, '+' represents the International Direct Dialing (IDD) prefix of the country from which you are calling. Please consult your local telephone service provider for more information on specific IDD instructions.


Revision History

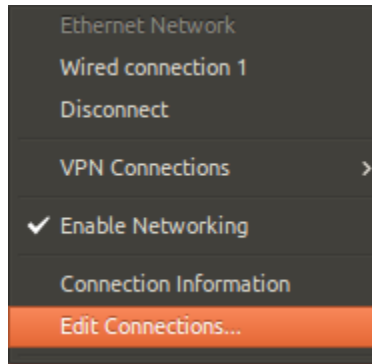
Date	Version	Revision
15 Sep 18	00	Preliminary release
17 Sep 18	01	KK review notes

Appendix

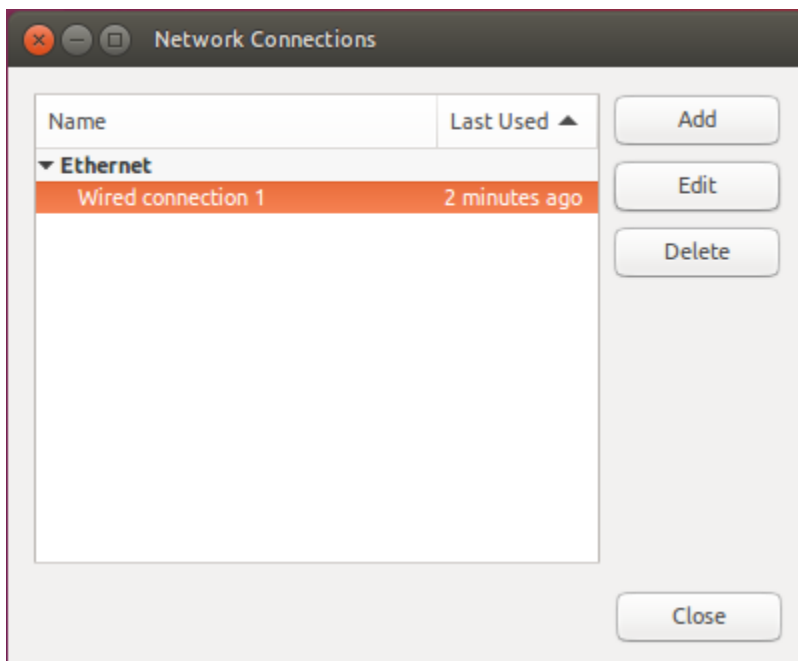
Instructions to replace instructions above, depending on how the Xilinx laptop for XDF is setup.

12. Set the IP address of the Ubuntu laptop to 192.168.11.12. The sudo password is **<tbd>**.

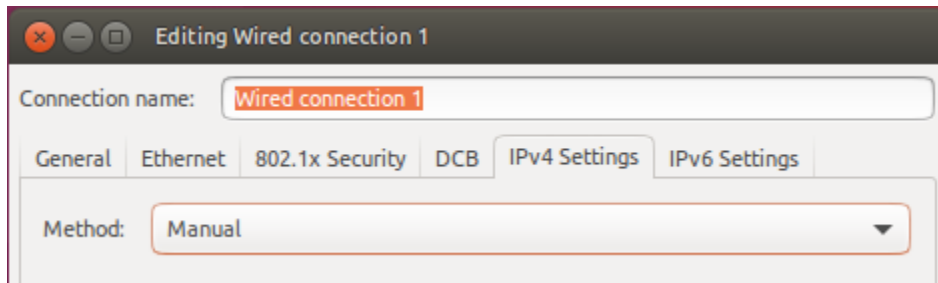
- a. Right-click to the  icon on the top Ubuntu toolbar and select **Edit Connections...**



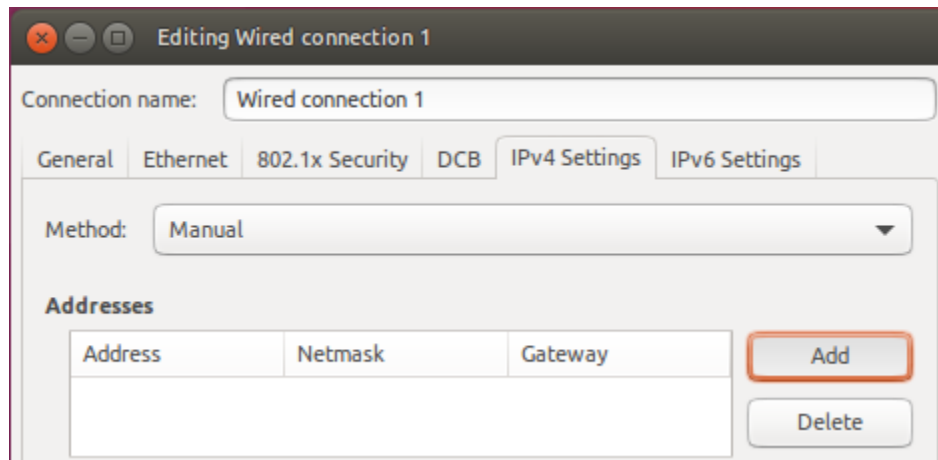
- b. Select **Wired Connection 1** and click on **Edit**.



- c. Click on the **IPv4 Settings** tab and change the **Method** to **Manual**.



- d. Click on the Add button to set a new IP address.



- e. Click in the space under Address to set the new IP address as follows. Click **Save** to continue.
- Address: 192.168.11.12
 - Netmask: 255.255.255.0

Editing Wired connection 1

Connection name: Wired connection 1

General Ethernet 802.1x Security DCB IPv4 Settings IPv6 Settings

Method: Manual

Addresses

Address	Netmask	Gateway
192.168.11.12	255.255.255.0	

Add Delete

DNS servers:

Search domains:

DHCP client ID:

☐ Require IPv4 addressing for this connection to complete

Routes...

Cancel Save

- f. X
- g. X
- h. x