Lab

3

Using Xtensa Software Tools in Xtensa Xplorer

LEARNING OBJECTIVES:

- Using the Xtensa software tools in Xtensa Xplorer
- Completing a software development flow, including compiling and linking an example program and running the application on the Instruction Set Simulator (ISS)
- Debugging the program using Xplorer and profiling the code to find hot spots
- Exploring various cache combinations

Lab Duration:
60 min

- Lab Prerequisites:

Before proceeding with this lab, we recommend viewing the Xtensa Xplorer Tutorials to become familiar with the Xtensa Xplorer application and terminology. To test your understanding of this lab, answer the questions throughout this lab. The answers are in the Answers and Solutions section.

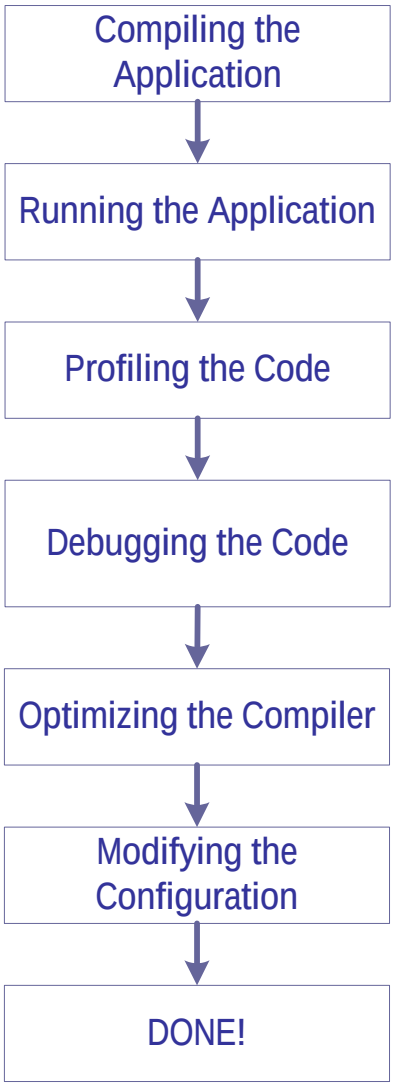
Lab Requirements:

If you are completing this lab in the Tensilica training laboratory, your PC is ready for use. Otherwise, you may need to ensure your PC is prepared with the necessary licenses. To follow the procedures in this lab, you need the following licenses:

- ISS
- XT-XCC compiler
- Xtensa Xplorer Processor Developer's Edition

Contact your instructor if you have any licensing questions.

Lab Flow Diagram



Introduction

In this lab, you will become familiar with the Xtensa software tools available in Xtensa Xplorer. This lab also introduces you to the methodology of optimizing your code using the profiling capabilities of the Xtensa software tools to locate the hot spots in your code. You will also be introduced to the debugger and learn how to use the Xtensa software tools to compare the performance of multiple configurations of the basic core or variations on the core you have designed.

The workspace used in this lab contains a configuration of the Xtensa processor, a system using this configuration, and an application called **byteswap**. We will first compile the byteswap application on the Xtensa configuration and then use a custom TIE instruction to optimize the application and improve its performance.

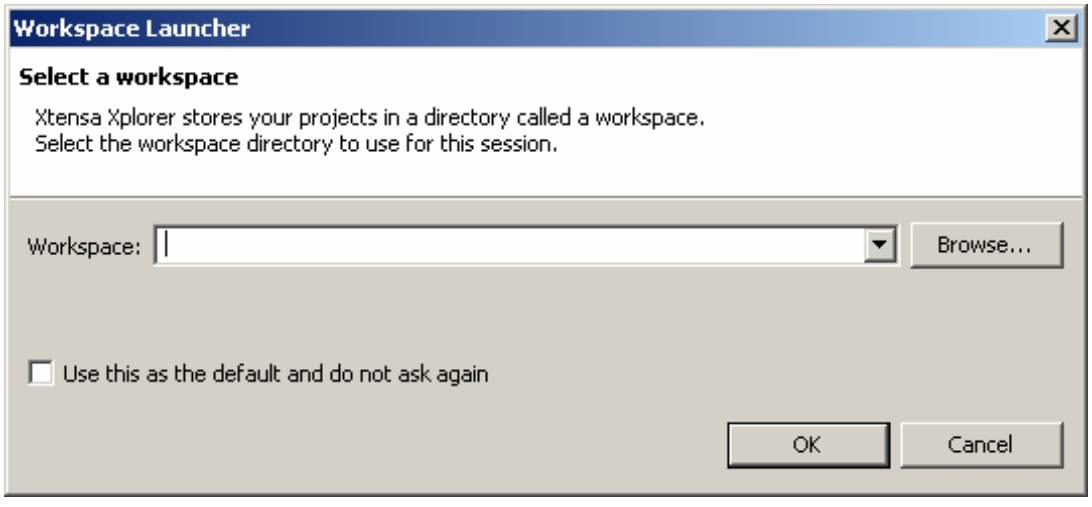
In the later labs, you will learn to extend the Xtensa processor with new custom designed instructions that are described in the Tensilica Extension Language (TIE) language. These new instructions can often dramatically increase your application performance.

Task 1. Importing the Lab Workspace

STEP 1.1: Start Xtensa Xplorer.

1. Click the shortcut to **Basic Training Xtensa LX** on the desktop.
2. Open the **Lab3** folder.

   You will see an **Xplorer** shortcut and the **BasLab3XXSWwin.xws** workspace.
3. Double-click the shortcut **Xplorer icon** to start **Xtensa Xplorer**.
4. A workspace launcher dialog box appears.



5. Browse to C:\TensilicaTraining\RA2005.2\BasicTraining\Lab3 and click OK.

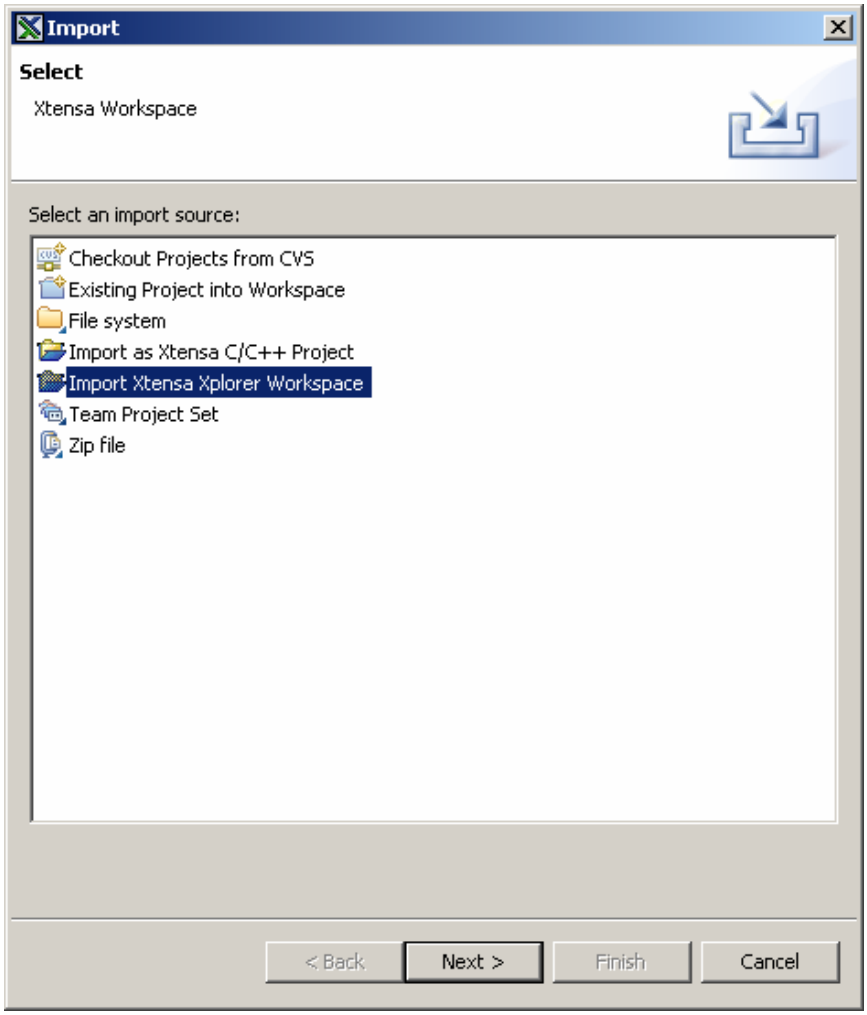
6. Click on X to remove the welcome screen.

STEP 1.2: Import the lab workspace.

The workspace for this lab is **BasLab3XXSWwin.xws**. This workspace contains:

- Processor configuration: XtTrain_core0
- Xtensa C/C++ Project: byteswap
- TIE file: byteswap.tie

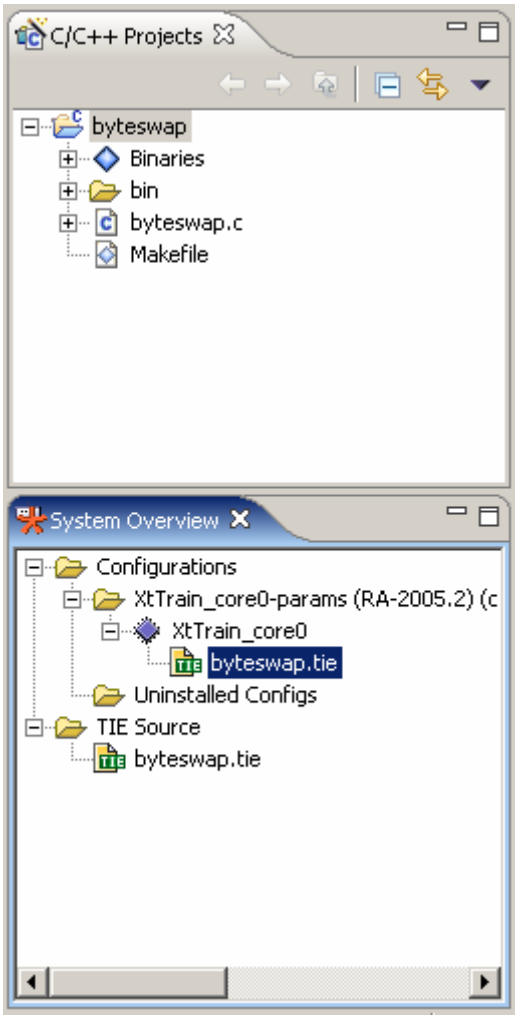
1. From the **File** menu, select **Import**.
2. Select **Import Xtensa Xplorer Workspace** and click **Next**.



3. Browse to and select the workspace for this lab -- **BasLab3XXSWwin.xws**. It is located at to C:\TensilicaTraining\RA2005.2\BasicTraining\Lab3.

4. Click **Finish**.

5. Check the processor configuration and software project against the information listed at the beginning of this task.



Task 2. Understanding the byteswap.c File

We have just imported the workspace with an Xtensa C/C++ project, **byteswap**. Now we will look at the source code and discuss what this code does.

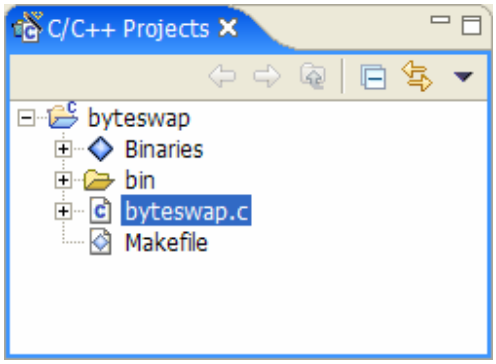
STEP 2.1: Open the C/C++ Perspective.

1. Click the **C/C++ Perspective** icon  .

   A perspective is a collection of views. The **C/C++ Perspective** contains views that are useful in developing Xtensa C/C++ applications.

STEP 2.2: Open the byteswap.c source code.

1. From the C/C++ Projects View, expand the software project, **byteswap**. Double-click the C file, **byteswap.c.**



The byteswap.c source code displays in the Xplorer Editor window, as follows:

byteswap.c

```
/*********************************************************************
  Byteswap.c
  This example compares an endian-conversion implemented in C and TIE
 *********************************************************************/

#include <stdio.h>

/* Number of Iterations to run */
#define NUM 10000
/* Random data used to test the byteswap instruction */
#define N    64
unsigned data[N] = {
    0x7edb1c67, 0x159f51b7, 0xfb17d999, 0xdeab3047, 0x580b9b31,
    0xb87db5b9, 0xbb91a3d3, 0x07e90569, 0x185f16e9, 0xd921d90f,
    0xe3f90331, 0xb277491b, 0x342b7edd, 0xda8fc287, 0x3bfd6d2b,
    0xca1b8237, 0xa0350575, 0x01096dc5, 0x9b43b3d5, 0xf74da1eb,
    0x68c16b2f, 0x61078e47, 0xf06900d9, 0x7e45f6c3, 0x2889a9a1,
    0xae37b263, 0x28033079, 0xfdeb7f9f, 0x5fbffe7b, 0xea81c641,
    0xf3a18c91, 0x0ee59eb7, 0xab0b5683, 0xf505f6e9, 0x70c9e795,
    0xc28d2c9b, 0xda8f1899, 0xf91bf539, 0xaff7178d, 0x01f9eb35,
    0xe8e750b1, 0xbd5398e3, 0x1b9fd11d, 0xccf358c5, 0xd2233add,
    0xd273e375, 0xbf33e281, 0x58ffe2e5, 0x4acd2e41, 0xa27f6353,
    0x6e17ce89, 0x10597985, 0x56e7e81d, 0x5fa9f6bb, 0xcaa9c7a3,
    0x70f581ef, 0xc0e936c7, 0xd365eebf, 0x2d3f0acf, 0xcb7f29c1,
    0x70c704af, 0x0d5b9251, 0x6b259aa9, 0xe25b19f5
};
/* global states used by the C implementation */
static unsigned GOLDEN_COUNT;
static unsigned GOLDEN_SWAP;

/* C implementation of byteswap to test against TIE implementation*/
    static unsigned
GOLDEN_BYTESWAP(unsigned s)
{
    unsigned ss = (s<<24) | ((s<<8)&0xff0000) | ((s>>8)&0xff00) |    (s>>24);
    GOLDEN_COUNT = GOLDEN_COUNT + 1;
```

STEP 2.3: Understanding the functionality of the byteswap.c program

The byteswap.c program contains a loop that invokes GOLDEN_BYTESWAP() 10,000 times and compares it to the result returned by the BYTESWAP() function. As the name suggests,

GOLDEN_BYTESWAP is the golden model or the pre-verified functional model of this function. BYTESWAP is the call to the TIE instruction that we will create later. The input to GOLDEN_BYTESWAP() comes from a table of random data. The data is used to verify the TIE instruction against the result produced by GOLDEN_BYTESWAP().

The GOLDEN_BYTESWAP function swaps the bytes in a 32-bit word in the following manner:

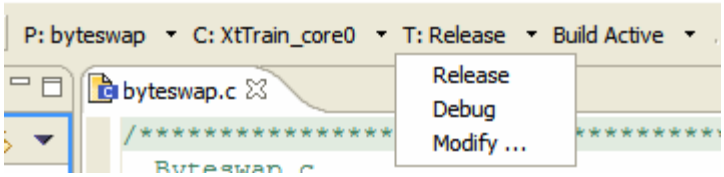
Input Word: S	d	c	b	a
Output Word: SS	a	b	c	d

Review the GOLDEN_BYTESWAP() function and you will find that it consists of a set of byte shifts and masking steps that achieve the result shown above.

Task 3. Compiling the Application

Description and Terminology: To compile an application or project in Xplorer, you have to tell Xplorer the project to compile, the processor configuration to compile the project on and the build target. A *build target* is a set of build properties (compiler, assembler, linker options). We call these the active project, active configuration, and the active target, and together we refer to them as the **Active Set**.

1. Select the active set using the drop-down menus in the toolbar on the top of Xplorer.



2. Select **byteswap** as the active project, **XtTrain_core0** as the active configuration, and **Debug** as the active target.

3. Click the **Build Active** button in the toolbar. This starts compiling the active project on the active configuration using the build properties specified in the active target.

   In the lower right corner, the Console view displays the compilation output. Select the Problems view to see if there are any errors and warnings in the build process. Double-clicking the error or warning in the Problems view opens the file with the problem in the C/C++ Editor to the corresponding line.

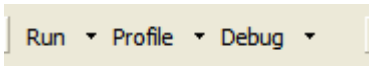
Task 4. Running the Application

Purpose: In this task, we will execute the byteswap application that we compiled in the last task. The application is executed on the Xtensa Instruction Set Simulator (ISS) and we will explore some of the profiling options available in this simulator.

Terminology and description: Xtensa Xplorer uses the concept of a "launch" as a way to remember how a binary is to be executed. It collects together all the run time settings (e.g. ISS settings, profiling and command arguments), and gives them a name so you can easily run (or debug) that launch again.

Note that a launch includes more than just the executable. The ability to specify multiple launches for a single executable is very powerful because the same executable with different datasets is a very effective way of analyzing performance of those datasets with the Benchmark perspective.

There are three buttons in the toolbar of Xplorer labeled **Run**, **Profile**, and **Debug**.

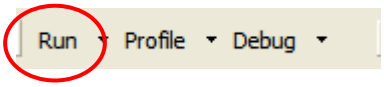


Clicking on any of these buttons launches a default run launch created by Xplorer known as the "**Auto Launch**" for that button. The auto launch associated with the *Run* button executes the binary of the application on the ISS with no profiling and no memory modeling. The auto launch associated with the *Profile* button has profiling and memory modeling enabled and the auto launch associated with the *Debug* button launches the debugger.

You can create your own run launches when you want to specify arguments to your application or change the default simulator settings that the Auto Launches use.

STEP 4.1: Executing the application using Auto Launches.

1. Click **Run** on the main menu toolbar of Xplorer.

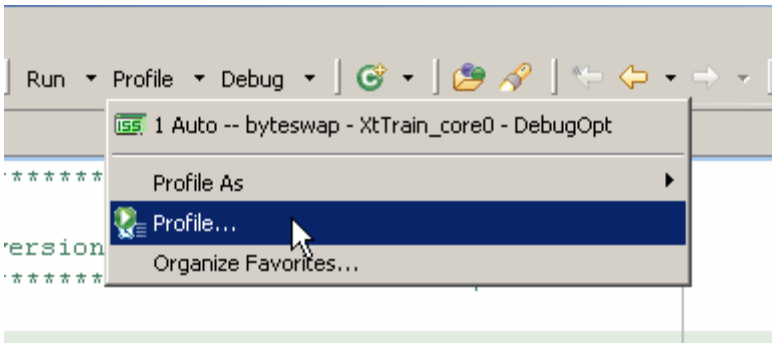


This launches the Auto Launch associated with the Run button. Xplorer executes the application and the output of the run is shown in the Console View. You should see a message that reflects that your TIE worked correctly.

STEP 4.2: Creating your own Run Launch.

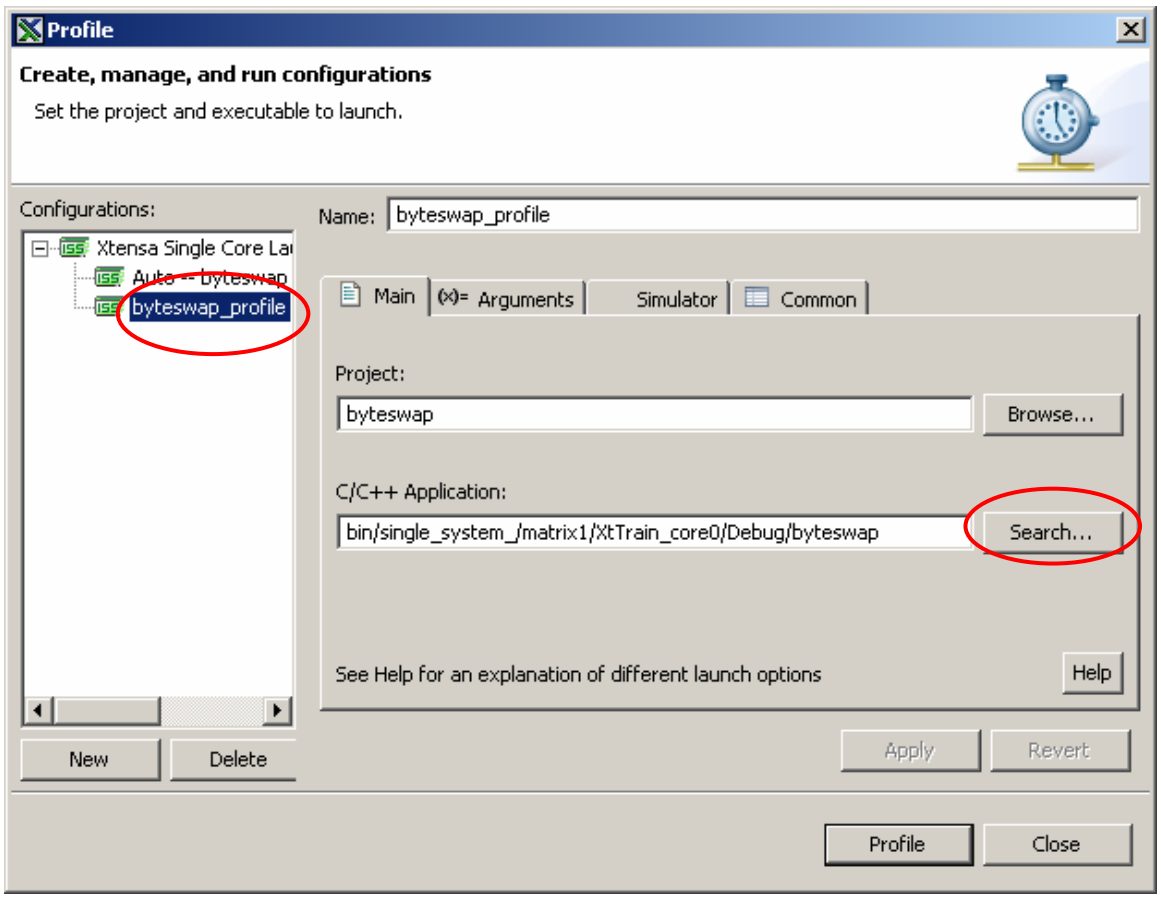
Purpose: We will show you how to create your own run launches, so that you can add your own simulator flags or arguments to the program. In this step, we will add the "pchistory" flag to simulator options.

1. Select "**Profile…**" from the Profile drop-down menu in the toolbar (or "**Run…**" from the Run drop-down menu) to open the Profile dialog box.



,

2. In the **Profile** dialog box, select **byteswap_profile** under the Xtensa Single Core Launch configuration.

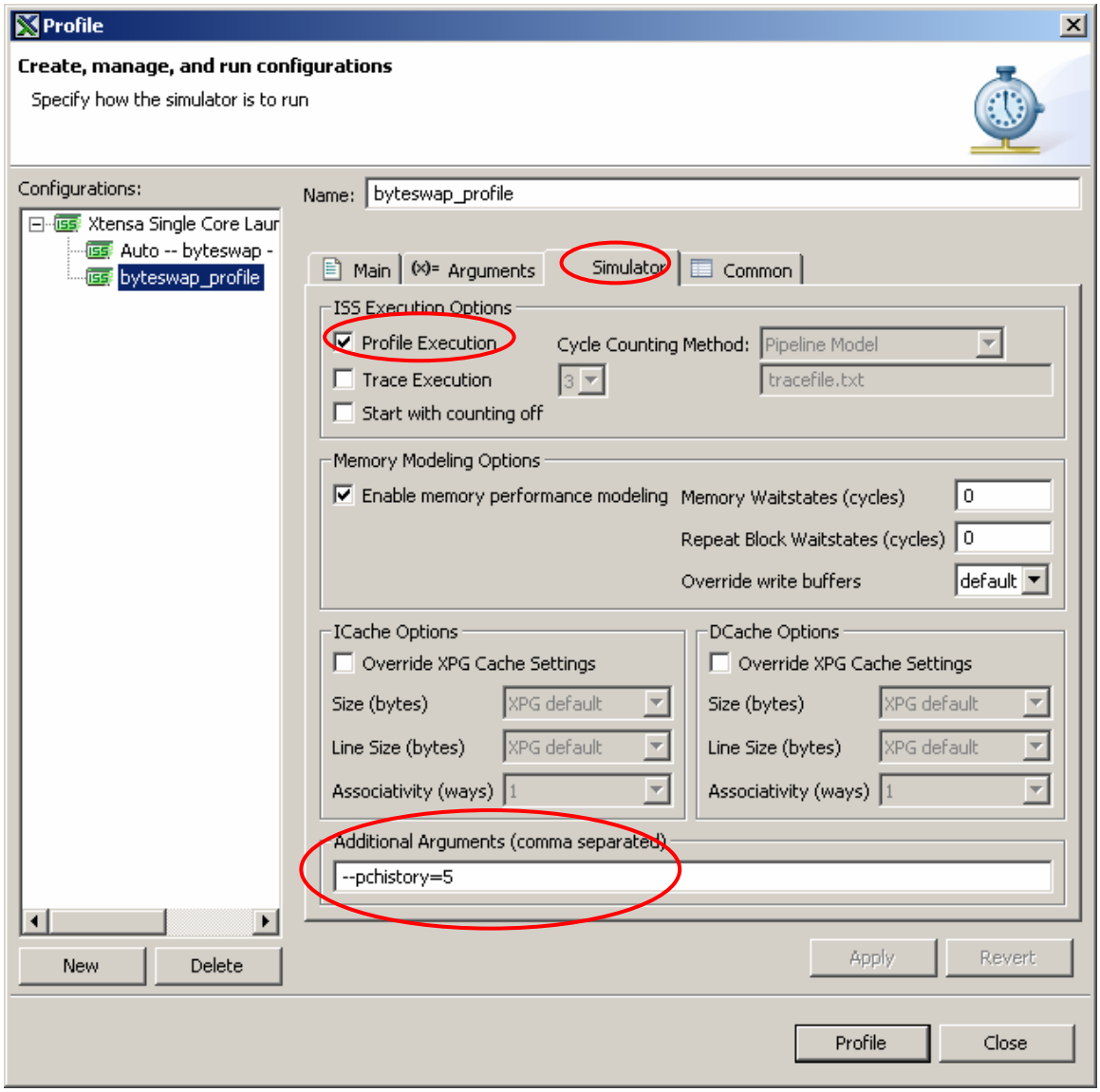


3. Select the **Main** tab, click **Search,** and then select the **byteswap-[single_system|XtTrain_core0|Debug]** program.

4. If not already enabled, select the **Simulator** tab and enable the **Profile Execution** and the **Enable memory performance modeling** options.

5. As a demonstration of how to invoke ISS Client packages through the specification of command line arguments, enter "**--pchistory=5**" in the **Additional Arguments** section. This invokes the "pchistory" client package when this run launch is invoked.

The **pchistory** client package stores the history of the last *N* program counter (PC) values. Here we specify that we want to store the last five PC values.



6. Click **Apply** and then **Profile.**

Xplorer changes to the **Benchmark Perspective** and the program executes with the output displayed in the Console view. Once the program has finished execution, the function-by-function profile of the program is displayed in the Profile view in the Benchmark perspective.

STEP 4.3: Analyze the output of the byteswap_profile launch.

1. From the Benchmark perspective, click the **Console view** tab at the bottom of the Xplorer window to review the output from the program run.

Question 1. What version of the ISS are you using and what does ISS stand for?

…….

Question 2. How long does the simulation take (in seconds)?

…….

Question 3. How many cycles does the simulation run?

…….

Question 4. What is the data cache size? What percentage of data cache reads are data cache misses?

…….

Question 5. How can you increase the data cache size for your simulation? ***Hint***: *Review the Simulator section of the launch.*

…….

Question 6. What was the last value of the program counter (PC)? What about the last five PCs, and what option did we use to display that value?

…….

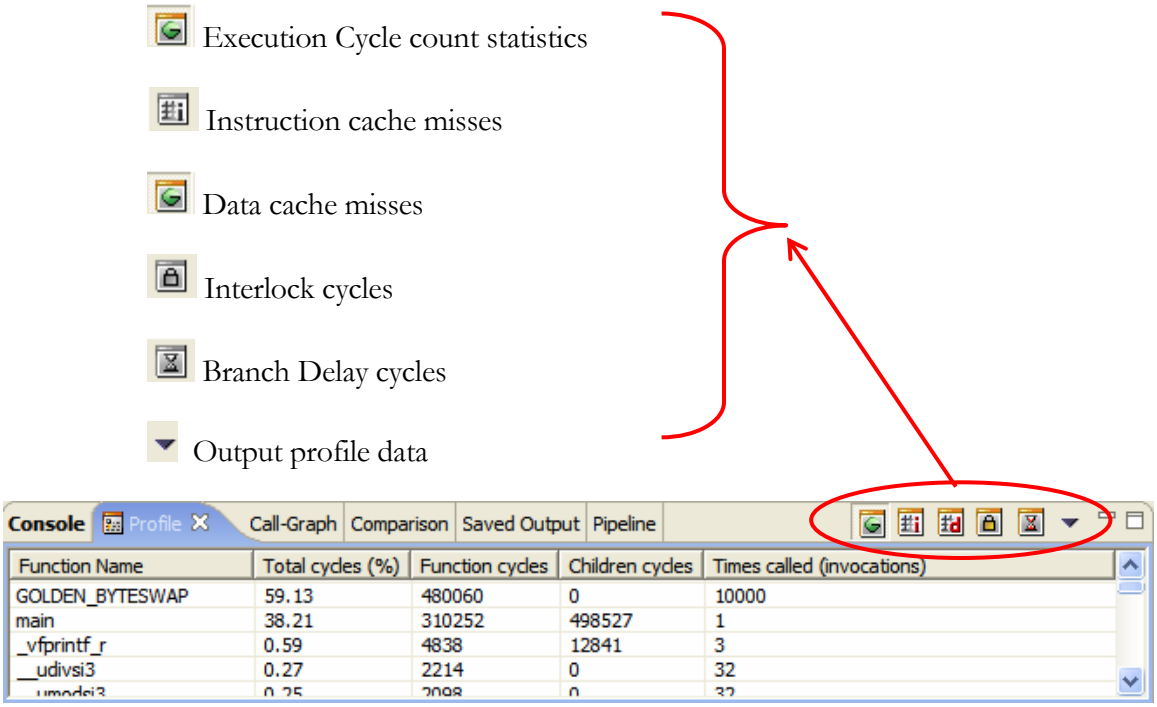
Question 7. What does memory performance modeling do, as the memory we model is zero wait states? **Hint**: *Modify the launch to remove the memory modeling. Pay attention to the cache misses.*

…….

Task 5. Viewing the Profiling Results

Purpose: We will now examine the profiling results obtained from running the application on the simulator in the last task.

1. In the Benchmark perspective, click the **Profile** view tab at the bottom of the Xplorer window to access the statistics of the byteswap_profile run launch we executed. The title bar of the Profile View contains six icons, as shown from left to right below:



Question 8. Which routine of our application takes the most cycles? How many times was it called?

…….

Question 9. How many times did main() have an instruction cache miss? **Hint**: *To sort a column, click the head of the column.*

…….

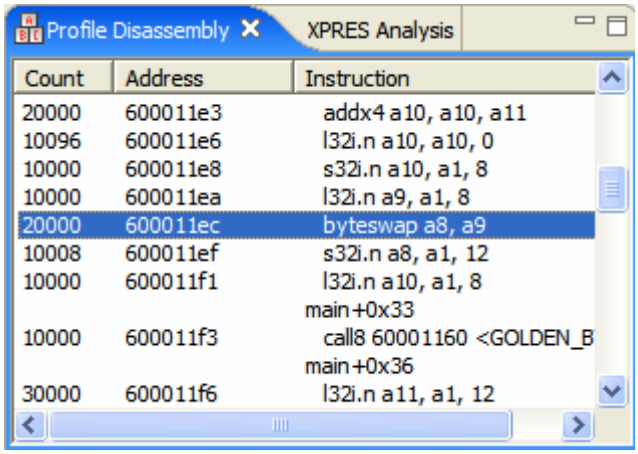
STEP 5.2: Analyze the TIE instruction byteswap.

1. Select the **Execution Cycle Count Statistics** icon (shown above).

2. Select the `main` function in the Profile view.

3. In the Profile Disassembly view (right side of the Xplorer window), scroll through the assembly code and select the TIE instruction, **byteswap**.

4. The first column in the Profile Disassembly view, Count, displays the cycle count for each instruction. Note that in the Editor view, the C Source line is within a loop of 10,000.



Question 10. How many cycles did the byteswap instruction take in the cycle count? How many bubbles (interlocks) did it produce?
Hint**: *Switch to the interlock statistics and select main() to access the Profile Disassembly listing. Take note of the address of the instruction.***

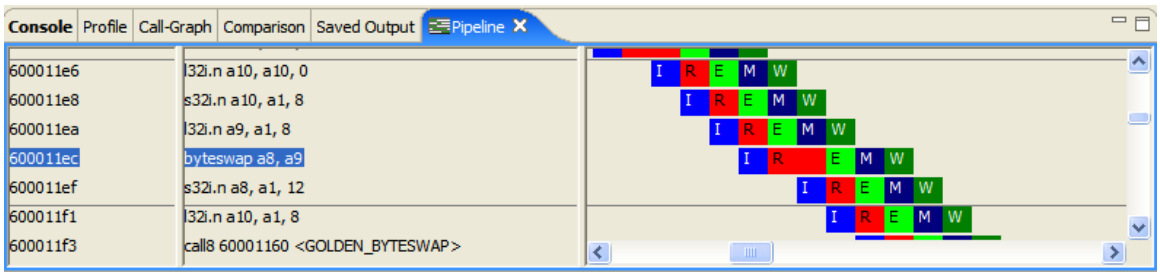
…….

STEP 5.3: Use the Pipeline view.

1. Click the **Pipeline view tab** at the bottom of the Xplorer window.

   The Pipeline View displays a statistical view of how each instruction flows through the processor pipeline.

2. The second column in the Pipeline view lists the instructions. Find the byteswap instruction in this column and double-click it. This will draw the pipeline diagram for this instruction to the beginning of the third column.



Question 11. Does this view explain why the byteswap TIE instruction accumulated twice as many cycles as the "for loop" called for? Why is there an interlock?

…….

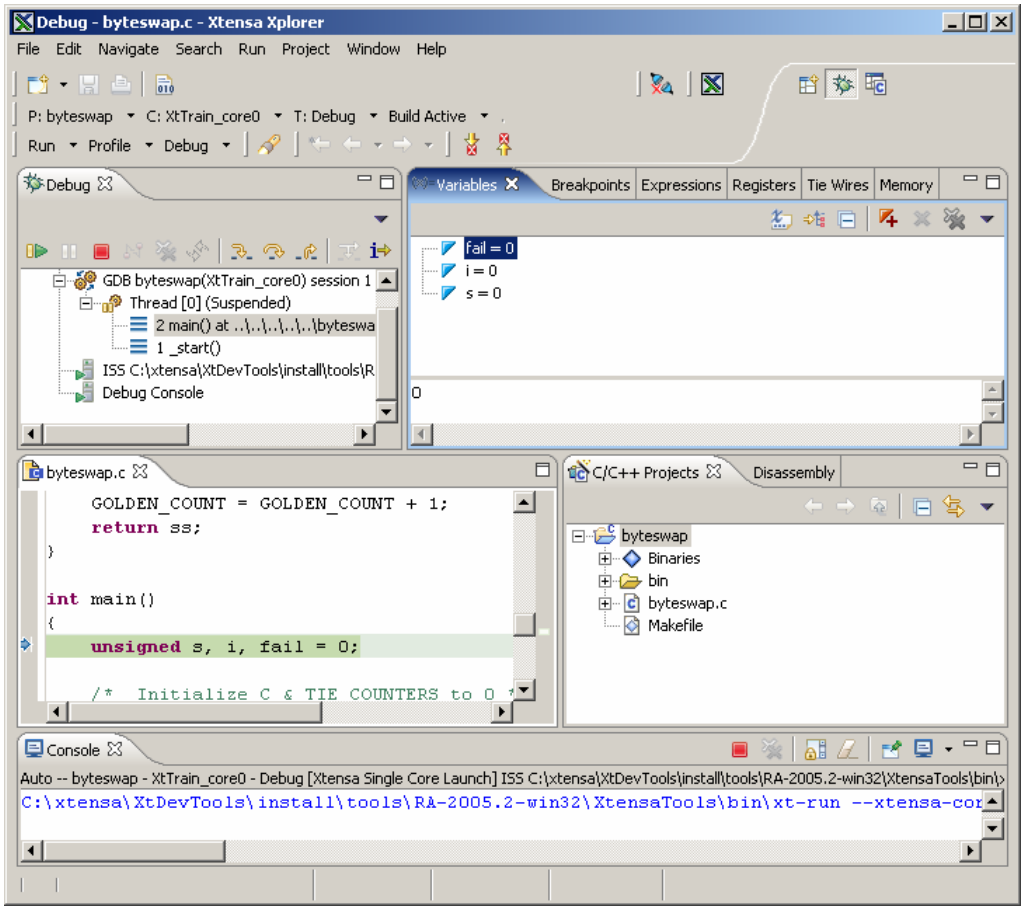
Task 6. Debugging the Code

Purpose: We will now learn how to debug our application code in Xplorer. We will single step through the instructions during the debug session and also learn how to watch the values of expressions during debugging.
Terminology and description: The Xtensa Xplorer debugger is built on top of `xt-gdb`, the command line debugger. The debugger launch definitions are the same launches as those created for execution. When using the debugger, it is better to set the compiler optimization level to zero. This is because compiler optimization strategies can create binary code that is difficult for the debugger to map back into the C source code.

STEP 6.1: Start the Debugger.

1. From Xplorer toolbar, click **Debug.**

This opens the **Debug** perspective and launches xt-gdb on the executable using a debug Auto Launch.



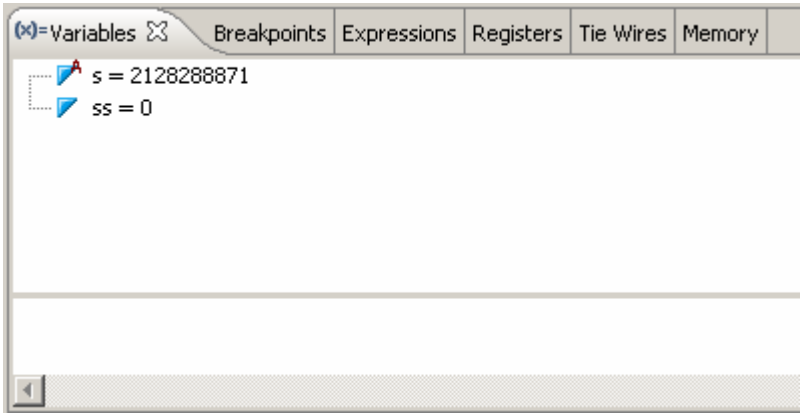
You can access the debugger control options using one of three methods:

- From the icons in this view
- From the Run menu options, or
- Using keyboard shortcuts

STEP 6.2: Use a single step to navigate the code.

1. In the title bar of the Debug view, click the **Step Into** icon  to single step through the program. Keep single stepping until you enter the GOLDEN_BYTESWAP function.

2. Select the **Variables** tab in the upper right corner view of the Debug perspective. This view displays the current values of the local variables and the arguments. In the GOLDEN_BYTESWAP function, two variables (**ss** and **s**) should be present in the **Variables** view.



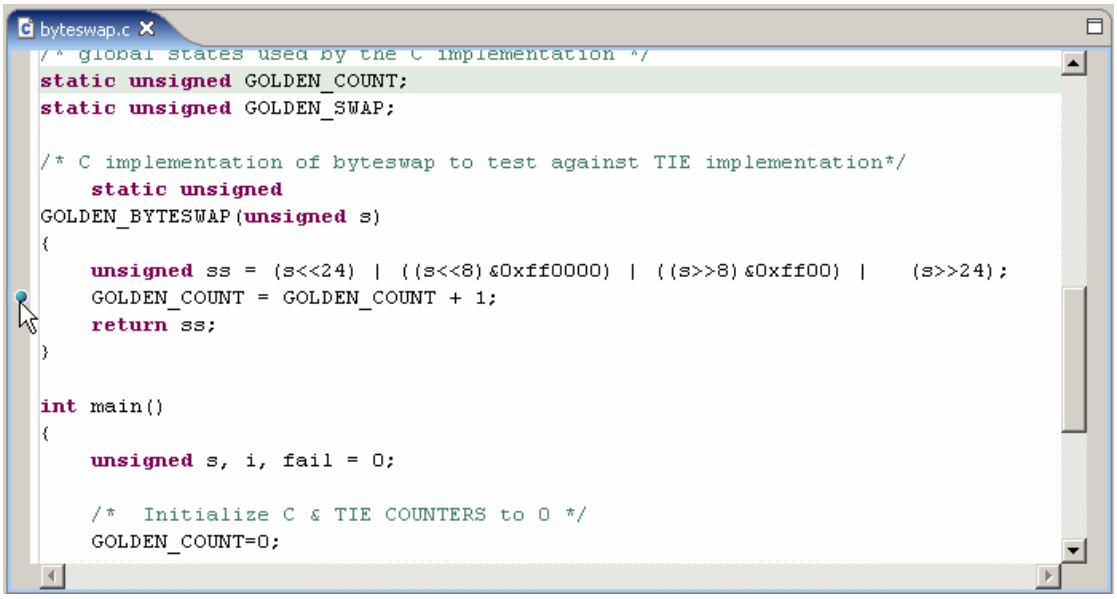
3. Continue single stepping until you reach the **return** line in the GOLDEN_BYTESWAP function.

Question 12. Looking at the values of s and ss, what function does GOLDEN_BYTESWAP perform?
Hint: *Right click each variable and change the format to* Hexadecimal.

…….

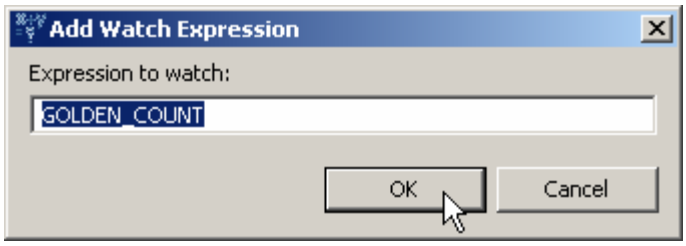
STEP 6.3: Work with expressions.

1. Set a breakpoint at the **GOLDEN_COUNT increment** line of GOLDEN_BYTESWAP function by double clicking on the column just left of the text editor window, next to this line of code. This creates a breakpoint marker at this line



2. Click the **Resume** button  or press **F8** until you reach the newly created breakpoint.

3. Double-click, then right-click the **GOLDEN_COUNT** variable and select **Add Watch Expression** to open the Add Watch Expression dialog box, and then click **OK**. This displays the value of the global variable, `GOLDEN_COUNT`, in the **Expressions** view.



Question 13. Bonus: Can you locate the register that computes the incremented GOLDEN_COUNT and then confirm its value in the Register browser?
Hint: *Click the Disassembly view to examine the assembly code.*

……

Question 14. **Bonus:** Can you locate the memory location used to hold the value of GOLDEN_COUNT and then confirm its value in the Memory browser?
Hint:: *Click the Disassembly On/Off icon and select the Registers tab and the Memory tab.*

…….

Task 7. Compiling with Optimizations

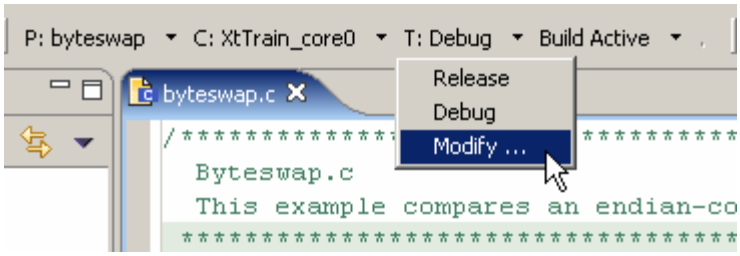
Purpose: We will now create a new compiler build target to show how the performance of the code changes when using compiler optimizations.

STEP 7.1: Create a new build target.

1. Terminate the Debug by hitting the **Terminate** button on the top of Debug Perspective.

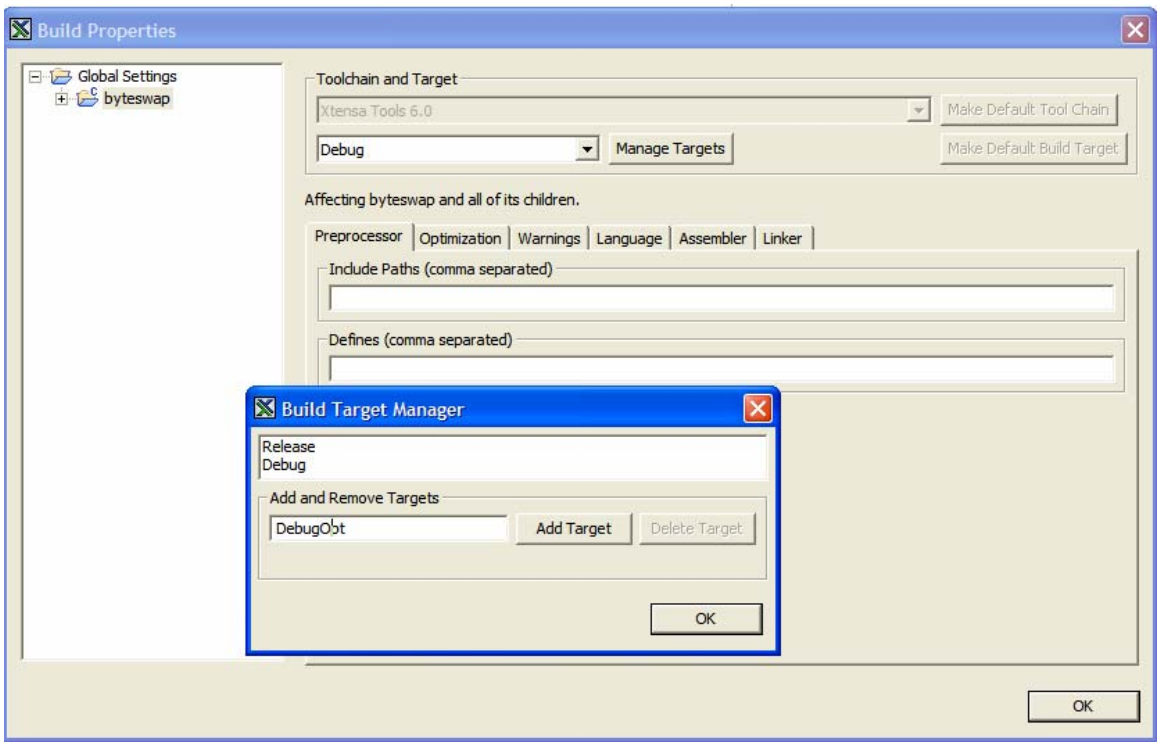
2. Open the C/C++ Development perspective.

3. To open the Build Properties dialog box, click the Active Target drop-down menu, and click **Modify…**



4. In the Build Properties dialog box, expand **Global Settings** on the left side and select the **byteswap** project.

5. Click **Manage Targets**, and then type **DebugOpt** in the Add and Remove Targets text box of the Build Target Manager dialog box.

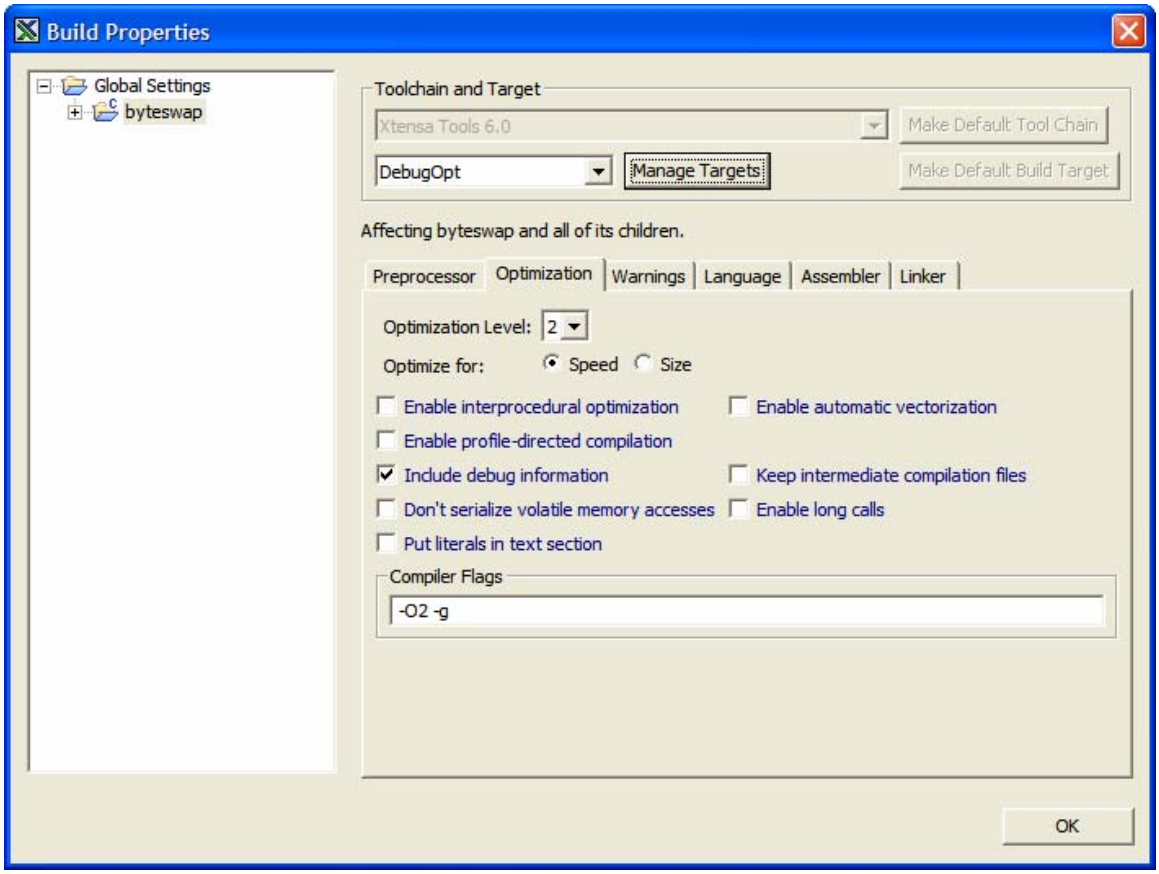


TIP: Do not use punctuation in this box. Xplorer enables the Add Target button only when there is a valid name.

6. Click **Add Target** to define the **DebugOpt** target options for the byteswap SW project and then click **OK**. The newly created **DebugOpt** target should automatically be selected in the Build Properties dialog box.

STEP 7.2: Select the Build Target options.

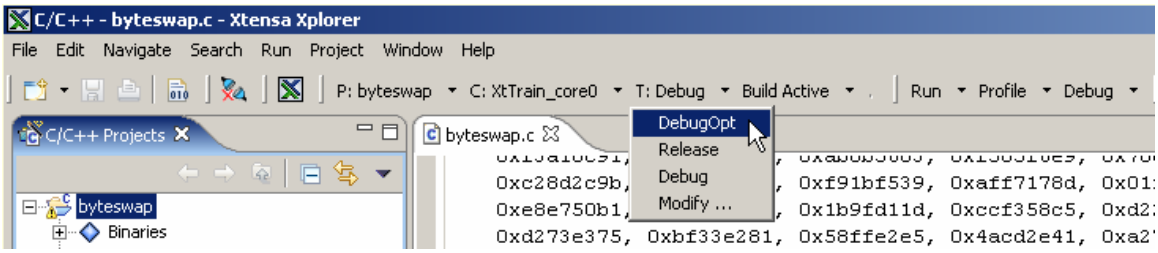
1. Select the **Optimization** tab in the Build Properties dialog box and select the "**Include debug information**" option – this selects the "-g" compiler flag.



2. Click **OK** to accept the new settings.

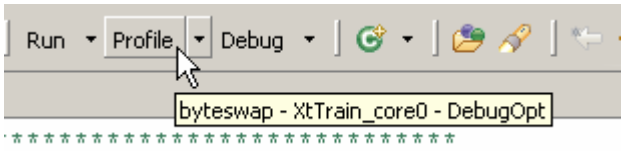
STEP 7.3: Profiling with the new Build Target.

1. Select **DebugOpt**. as the active target as shown below.



2. Click the **Build Active** button to compile the project for the new build target.

3. Click the **Profile** button.



   Xplorer changes to the Benchmark Perspective and executes the binary for the active set.

4. In the **Profile view** at the bottom of the Xplorer window in the Benchmark perspective, click **main** to open the Profile Disassembly for the `main()` function in the top right corner of the Xplorer window.

Question 15. What is the cycle count for the byteswap TIE instruction now? How is this different from the previous cycle count?

…….

Question 16. What happened to the call to GOLDEN_BYTESWAP?

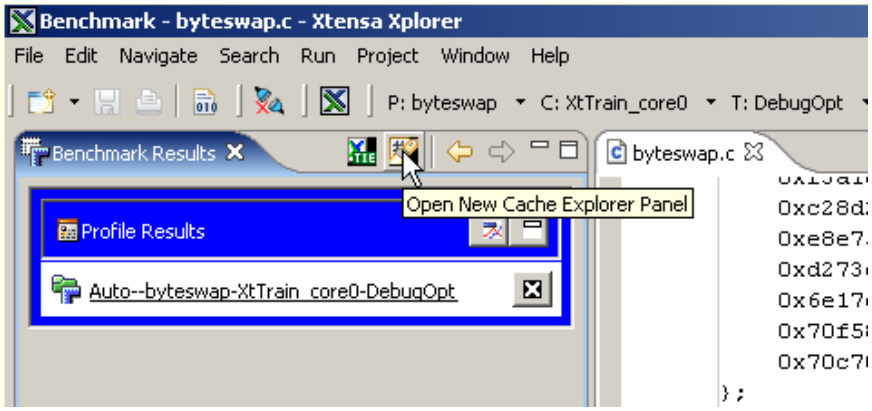
…….

Task 8. Modifying Core Configurations

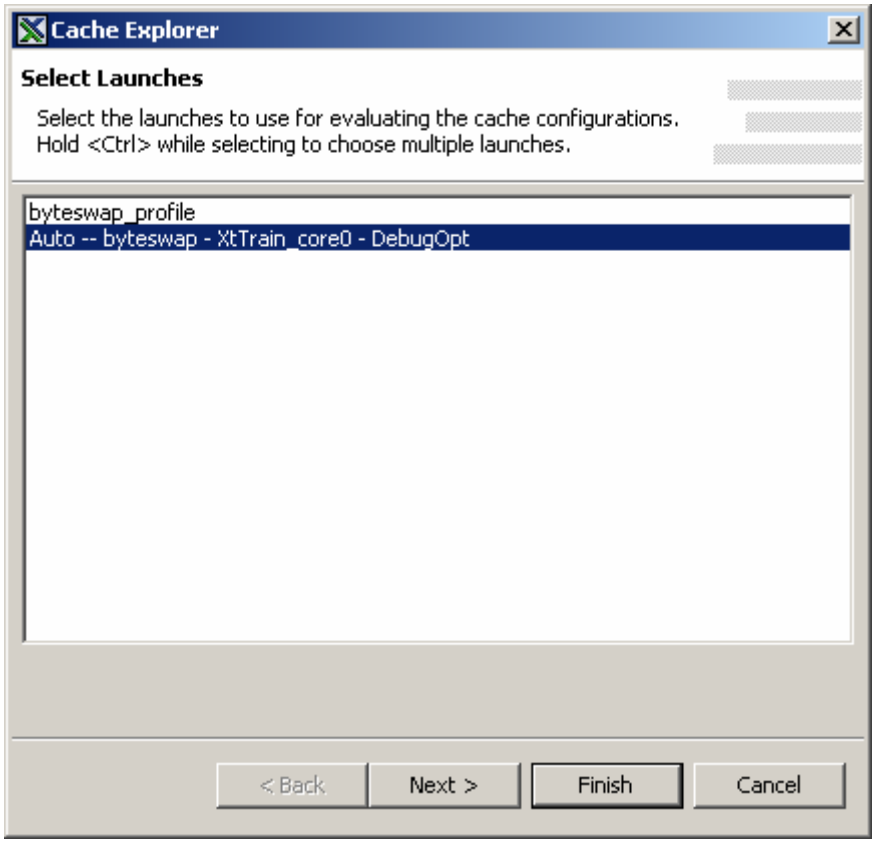
Purpose: We will now learn how to modify the configuration of the Xtensa processor core and examine the impact of these modifications on the performance results for our application code. We will also learn how to view the comparisons between different configurations.
Terminology and description: Because our example is not data intensive, you may wonder how small we can make the cache and what are the effects of this change. The ISS permits us to change these configuration parameters for simulation purposes. In addition, Xplorer includes a valuable tool that enables visualization of comparisons between multiple processor configurations.

STEP 8.1: Run Cache Explorer .

1. In the Benchmark perspective, click the **Open New Cache Explorer Panel** button.



2. To select a launch to be evaluated for different cache configurations, select **Auto—byteswap-XtTrain_core0-DebugOpt** and click **Next.**



For a more realistic analysis, we will add the memory and repeat block wait states as 5 in both cases:

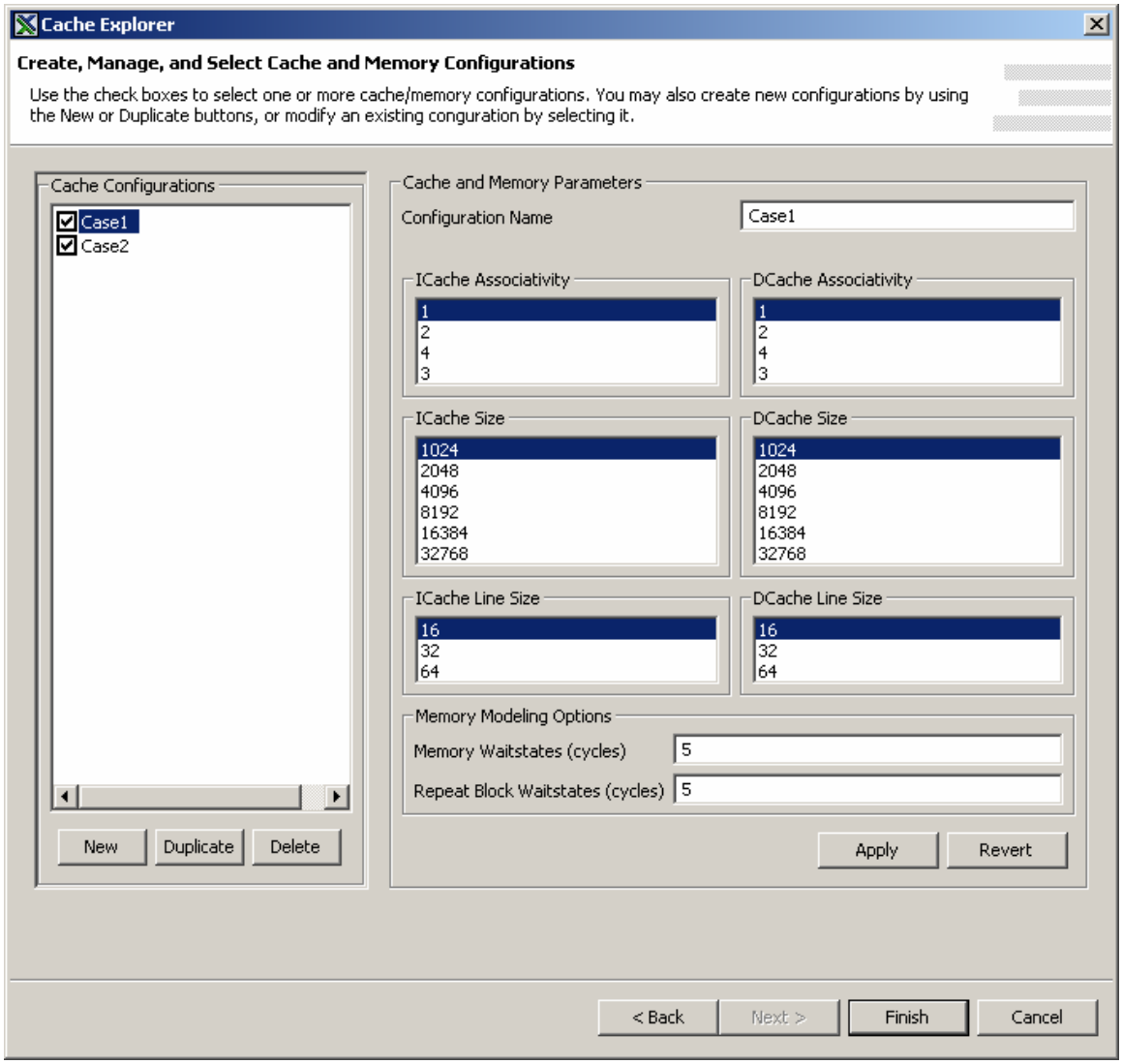
- **Case1: Icache(Direct, 1K 16 bytes/line), Dcache(Direct, 1K 16 bytes/line)**
- **Case2: Icache(Direct, 4K 16 bytes/line), Dcache(Direct, 4K 16 bytes/line)**

STEP 8.2: Create Case 1.

1. Type **Case1** in the **Configuration Name** field.
2. Select the cache and memory parameters listed above and then click **Apply**.

STEP 8.3: Create Case 2.

1. Click **New** and type **Case2** as the configuration name.
2. Enter the cache and memory parameters as listed previously and then click **Apply**.
3. Select the checkboxes for **Case1** and **Case2** in the Cache Configurations section, and then click **Finish**.

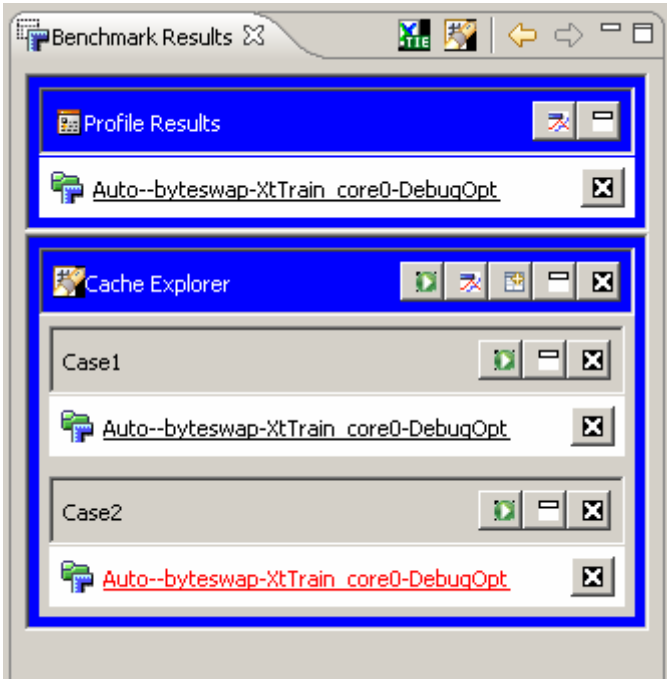


Xplorer executes the program with the defined memory and cache parameters. After a few seconds both Case1 and Case2 runs become available. The Benchmark Projects view will contain an entry for Cache Explorer with Case1 and Case2.

Note: The selection order is important.

STEP 8.4: Create a comparison graph.

1. Click the **Compare all Cache Configurations** button in the Cache Explorer Panel to compare benchmark results for configuring cache differently.



2. The comparison chart showing both cases appears in the Comparison view.

3. Double-click the title bar to maximize the Comparison view. Notice the row of icons on the title bar.

4. Use the drop-down menu on the right side of the graph to display the Cache Misses and Application Size comparison chart.

Question 17. What information does each of the graphs display? **(Hint: Check the Cycles, Cache Misses and Application Size)**

…….

Question 18. How much performance degradation occurred with Case 1? How about the cache misses?

…….

Answers / Solutions

Question 1. What version of the ISS are you using and what does ISS stand for?

Answer: ISS version 6.0.2
ISS stands for Instruction Set Simulator

Question 2. How long does the simulation take?

Answers vary for this question. This is the amount of wall clock time your computer needs to complete the simulation.

Question 3. How many cycles does the simulation run?

Answer: **811,821 cycles**. The answer can vary. Check the console output.

Question 4. What is the data cache size? What percentage of data cache reads are data cache misses?

Answer: Data Cache: 1,024 bytes (1KB), direct mapped, 16-byte line size

0.05% of data cache reads are data cache misses.

Question 5. How can you increase the data cache size for your simulation?

Answer: Although you need to create a new processor configuration on the Xtensa Processor Generator to change the cache size, for simulation purposes you can change the simulator's cache size to gain insight into what is the optimal cache size for your application.

The Simulator tab of the Run Launch Dialog contains these switches. For data and instruction cache, we can select the size, number of ways, and the size of the line to change the simulation cache parameters.

Question 6. What was the last value of the program counter (PC)? What about the last five PCs, and what option did we use to display that value?

Answer: From the Console output, we can see that the last PC value is part of the

normal summary.

```
current pc = 0x60005d9d
```

The last five PC values come from the –pchistory=5 simulator option. The last five PC values are:

```
Backtrace of the last 5 PCs for core example_core1:
    0x6000c6a7
    0x60005d91
    0x60005d94
    0x60005d97
    0x60005d9a
The above result can vary. Check the console output.
```

Question 7. **What does memory performance modeling do, since the memory we model is zero wait stats?**

Answer: **Memory performance modeling enables memory subsystem modeling, including simulation of caches and internal RAM/ROM behavior (not only the external memory). Therefore, cache miss penalties are taken into account in the simulation.**

Question 8. **What routine of our application takes the most cycles? How many times was it called?**

Answer: **From the Profile display on the cycle count statistics view, we can see that the function with the largest cycle count is GOLDEN_BYTESWAP with 480060 cycles and 59.13% of the accumulated cycles. This function was called 10,000 times.**

In the Call-Graph view, we can identify that `main` is the only function that calls GOLDEN_BYTESWAP.

Question 9. **How many times did `main()` have an instruction cache miss?**

Answer: **From the Profile display on the I-cache miss count statistics view, we can see that the main function had six cache misses in itself. In addition, the functions it called (children) had 867 I-cache misses. To locate the statistics for `main()` easily, click the** Function Name **header to sort the statistics by function name.**

Question 10. How many cycles did this instruction take in the cycle count? How many bubbles (interlocks) did it produce?

Answer: The TIE instruction byteswap took 20,000 cycles total. It produced 10,000 interlock cycles.

Question 11. Does this explain why the byteswap TIE instruction accumulated twice as many cycle as the for loop called `for`? Why is there an interlock?

Answer: The Pipeline Preview shows that there is a resource interlock in the byteswap instruction. The register it uses is loaded in the previous instruction. Therefore, a bubble needs to be inserted to wait for the register to be loaded.

Question 12. Looking at the values of s and ss, what is the function performed by GOLDEN_BYTESWAP?

Answer: The function swaps the bytes within a 32-bit word, so byte 0 swaps with byte 3 and byte 1 swaps with byte 2. So in the second data sample, s=0x159f51b7 and the output is ss=0xb7519f15.

Question 13. Bonus. Locate the register that is used to compute the incremented GOLDEN_COUNT and then confirm its value in the Register browser.

Answer: In the Disassembly view, a few lines above the retw.n instruction locate the addi.n a4, a4, 1 instruction. The register A4 is used to compute GOLDEN_COUNT. In the Register tab of the top right corner view, open the AR register (current window), where a4 should be equal to 1.

Question 14. Bonus. Locate the memory location used to hold the value of GOLDEN_COUNT and then confirm its value in the Memory browser.

Answer: In the Disassembly view, a few lines above the retw.n instruction locate the s32i.n a4, a5, 0 instruction right below the addi.n a4, a4, 1 instruction. The register A5 is used to point to the memory location holding GOLDEN_COUNT. In the Register tab of the top right corner view, open the AR register (current window), where A5 should be equal to 0x6000d278. In the Memory view, select a Memory1 tab and give the address 0x6000d278. Confirm that the value is 01.

Question 15. What is now the cycle count for the byteswap TIE instruction? How is this different from the previous cycle count?

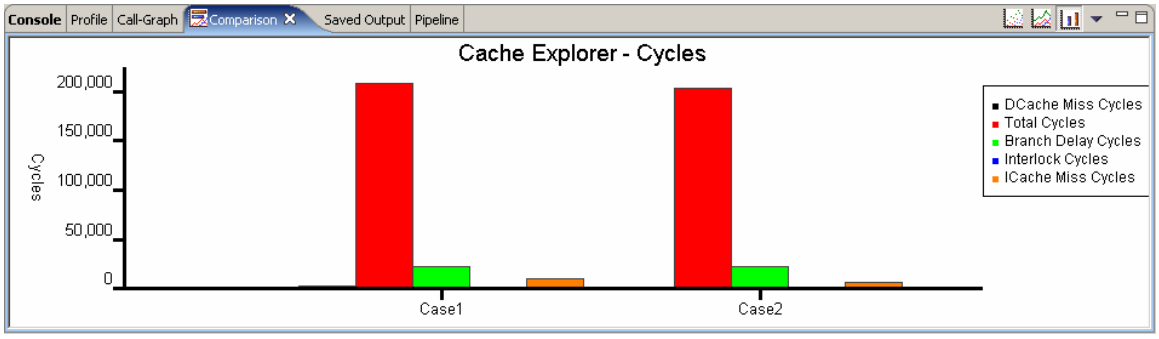
Answer: The byteswap instruction now takes only 10,000 cycles. It no longer has a bubble because the input data is not loaded in the immediately preceding instruction.

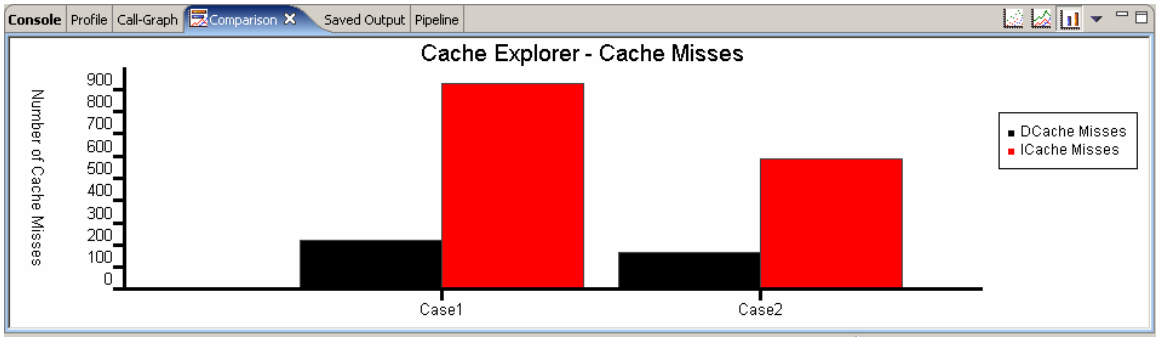
Question 16. What happened to the call to GOLDEN_BYTESWAP?

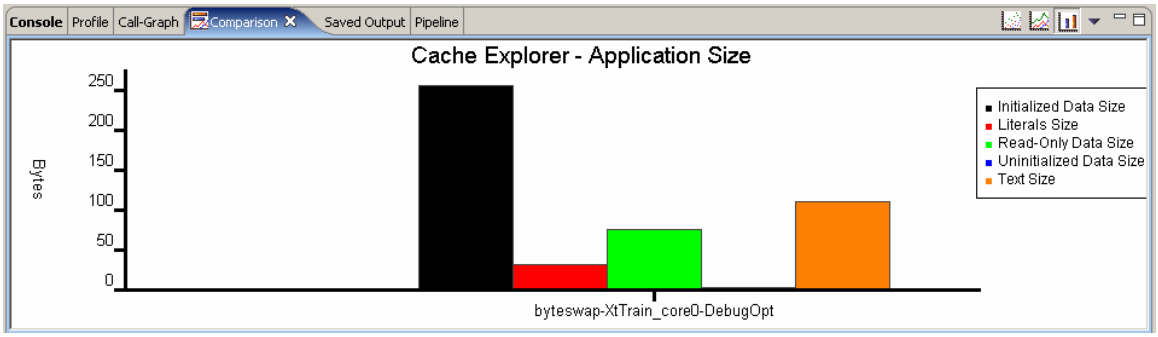
GOLDEN_BYTESWAP was optimized and now is inline. Notice the code right before the byteswap TIE instruction. That code is the GOLDEN_BYTESWAP function.

Question 17. What information does each of the graphs display? (Hint: Check the Cycles, Cache Misses and Application Size)

Answer: The cycles graph shows all the statistics (cycle count, Icache misses in cycles, Dcache misses in cycles, the branch delays in cycles, and the interlocked cycles) for both Case1 and Case2. The Cache Miss graph shows the Cache misses per occurrence, both Icache and Dcache. The Application Size graph shows initialized data size, literals size, read-only data size, uninitialized data size (bss) and text size.







Question 18. **How much performance degradation do we get with Case1? How about the cache misses?**

Case 1 is slightly slower, but either cache configuration will take about 200,000 cycles to execute the code. The cache misses for both Icache and Dcache are about 50% higher in Case1 than in Case2.