Lab

6

Writing and Testing Your First TIE Instruction

LEARNING OBJECTIVES:

- Profiling to find hot spots
- Developing a TIE instruction based on the Fusion technique
- Benchmarking the new TIE instruction and comparing performance

Lab Duration:
45 min

Lab Prerequisites:

- How to import a workspace
- How to create, run, debug, and profile am Xtensa C/C++ project
- How to build a processor configuration and use active set

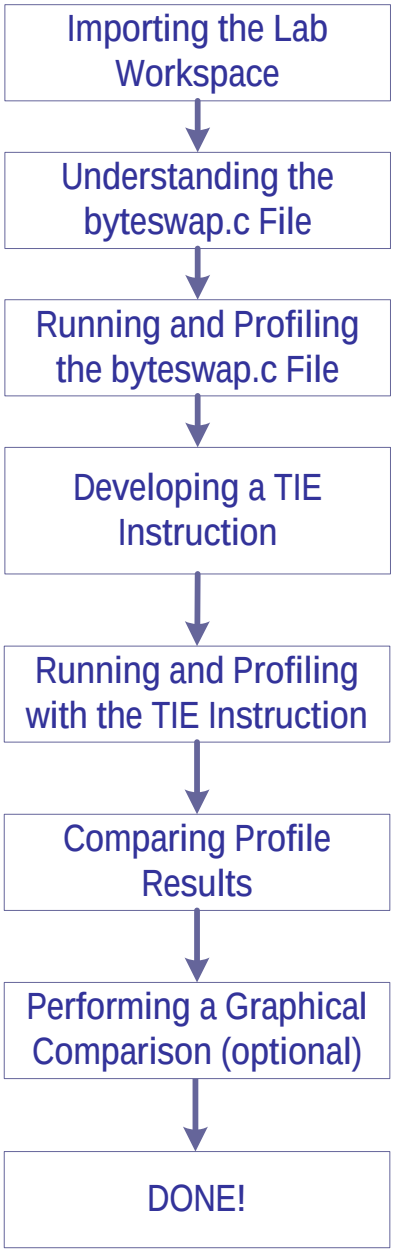
Lab Requirements:

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
- TIE 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 implement a simple **BYTESWAP** instruction in TIE. The **GOLDEN_BYTESWAP()** function is written in C and is developed to perform endian conversion. This function is often required in systems that interface peripherals of differing endian formats. The performance of the byteswap function can be a critical bottleneck in a design.

In this lab, we will optimize the system by replacing the GOLDEN_BYTESWAP() function with a single TIE instruction. When replacing a function with a TIE instruction, it is important to test the TIE instruction against a reference function. Byteswap.c provides a test bench to validate the TIE instruction. The test bench performs 10,000 endian conversions. It can be used for the GOLDEN_BYTESWAP() function and the TIE implementation. The output of the GOLDEN_BYTESWAP() function will be used to compare against the TIE instruction. This tests functional equivalency between the TIE instruction and the C-implementation.

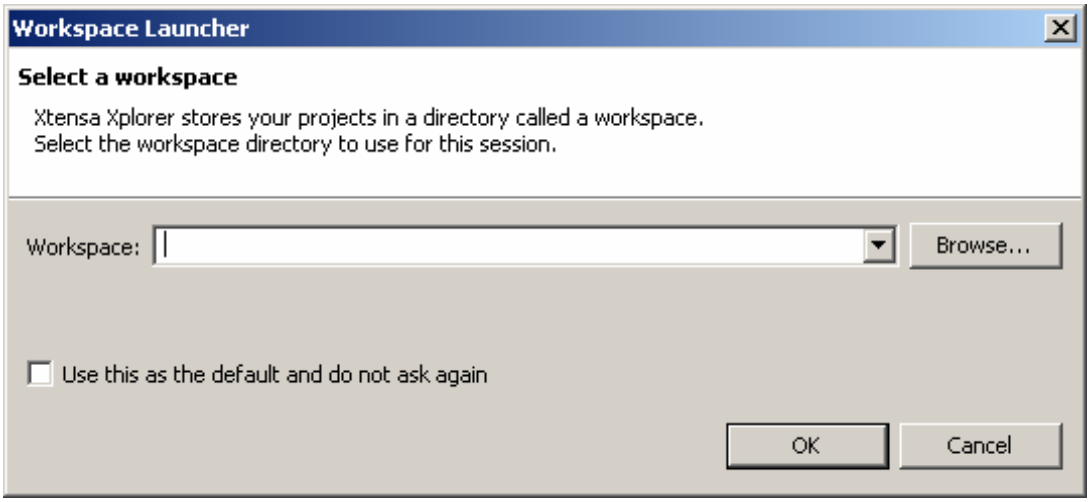
To test your understanding, answer the questions throughout this lab. You can find the answers in the Answers and Solutions section.

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 Lab6 folder.

   You will see an **Xplorer** shortcut and the **BasLab6TIEwin.xws** workspace.
3. Double-click the shortcut **Xplorer** to start **Xtensa Xplorer**.
4. A workspace launcher dialog box appears.
5. Browse to C:\TensilicaTraining\RA2005.2\BasicTraining\Lab6 and click OK.



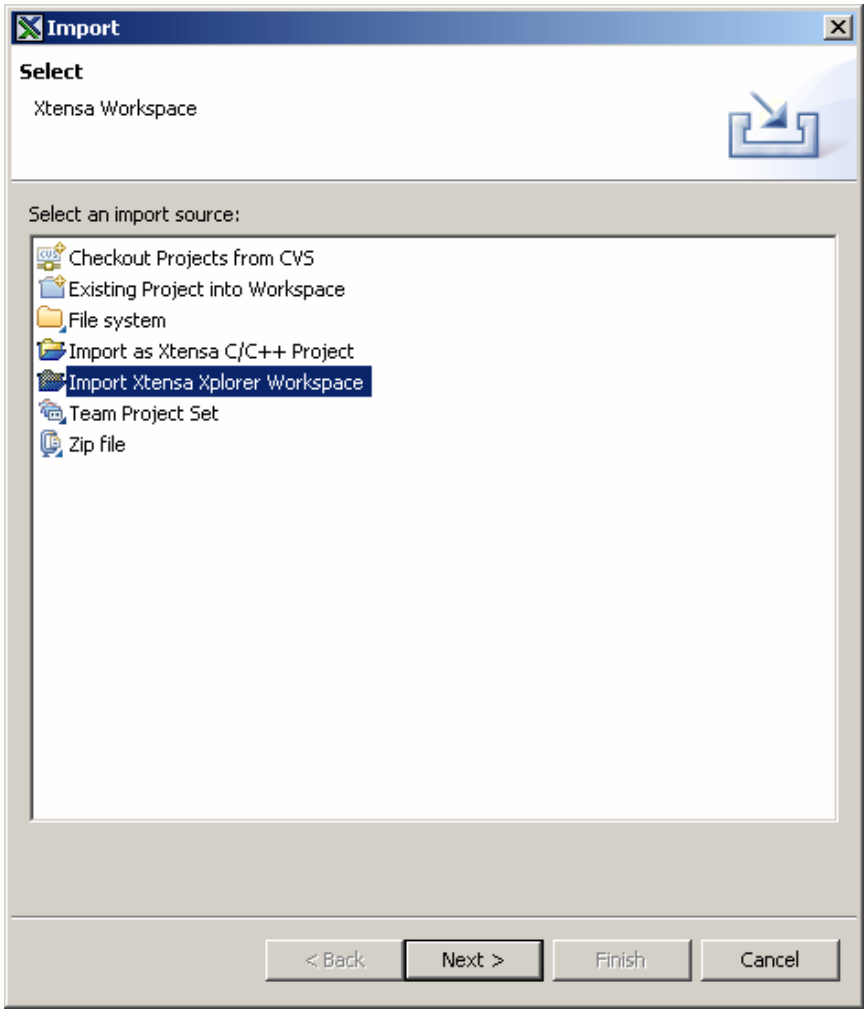
6. Click on X to remove the welcome screen.

STEP 1.2: Import the lab workspace.

The workspace for this lab is **BasLab6TIEwin.xws**. This workspace contains:

- Processor configuration: XtTrain_core0
- Xtensa C/C++ Project: byteswap

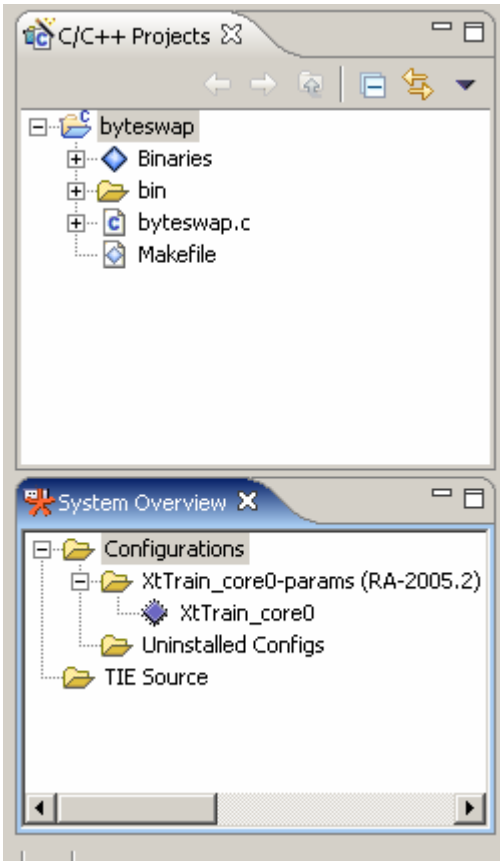
1. From the **File** menu, select **Import**.
2. Select Import Xtensa Xplorer Workspace and click **Next**.



3. Browse to and select the workspace for this lab – **BasLab6TIEwin.xws**. It is located at to C:\TensilicaTraining\RA2005.2\BasicTraining\Lab6.

4. Click **Finish**.

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


C/C++ Projects
byteswap
Binaries
bin
byteswap.c
Makefile
System Overview
Configurations
XtTrain_core0-params (RA-2005.2)
XtTrain_core0
Uninstalled Configs
TIE Source

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.

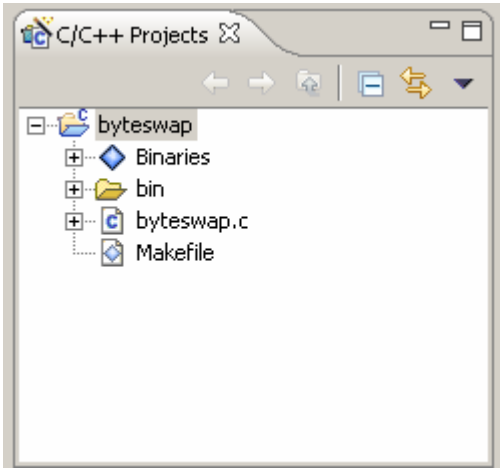
STEP 2.1: Open the C/C++ Perspective.

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**.
2. 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 byteswap.c code.

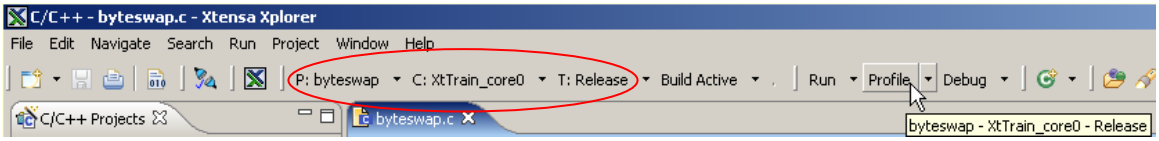
The byteswap.c program contains a loop that invokes GOLDEN_BYTESWAP() 10,000 times. The input to GOLDEN_BYTESWAP() comes from a table of random data. The data is used to verify GOLDEN_BYTESWAP() and the TIE instruction that we will create later.

Task 3. Running and Profiling the byteswap.c File

To find out what TIE instructions we should create, we will first run the code, profile it, and identify any hot spots. We will then choose an appropriate TIE technique to develop TIE instructions. These TIE instructions are designed to accelerate the reference code. Following are instructions to run, profile, and identify hot spots of byteswap.c.

STEP 3.1: Profiling.

1. Before we profile the code, make sure that the correct Active Set is chosen.



2. Click **Profile**. You will find that the code automatically compiles itself and, Xplorer switches to the **Benchmark Perspective** for profiling. You will find the profile information as shown below.

Console | Profile | Call-Graph | Comparison | Saved Output | Pipeline

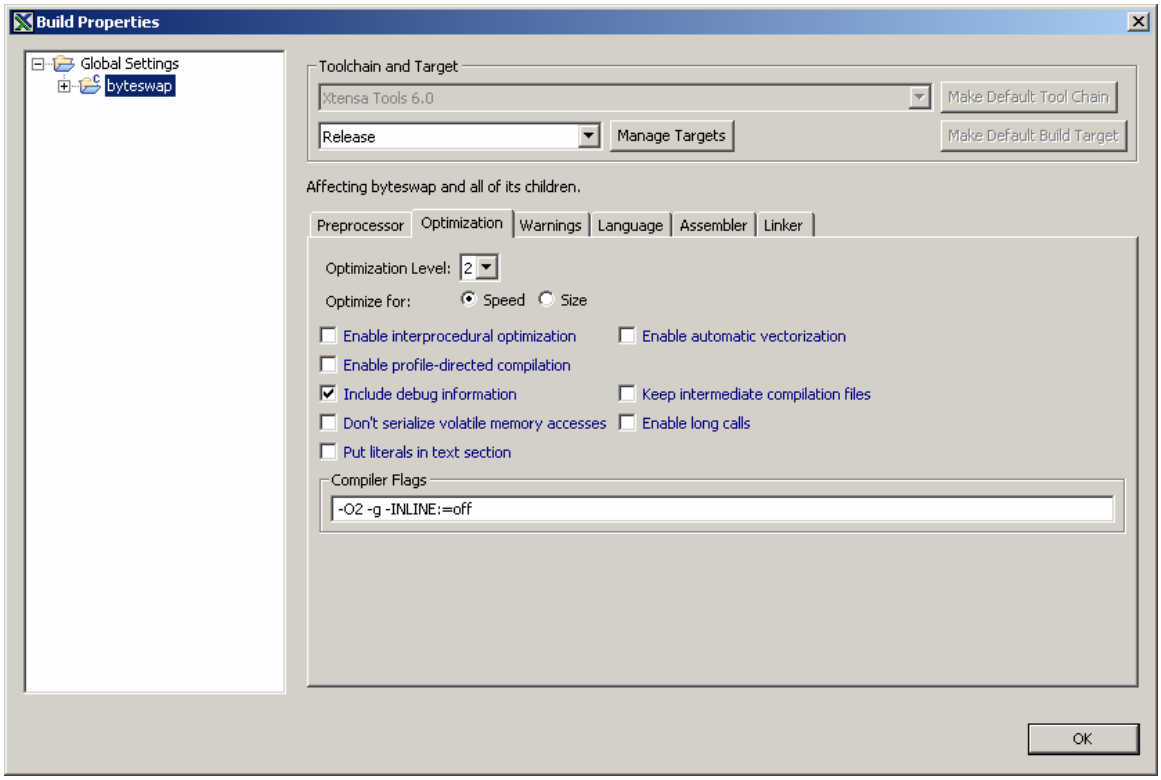
Function Name	Total cycles (%)	Function cycles	Children cycles	Times called (invocations)
_vfprintf_r	15.79	1849	6266	1
_ResetHandler	13.94	1632	0	<spontaneous>
__udivsi3	9.5	1112	0	16
__umodsi3	9.04	1058	0	16
__udivdi3	7.22	845	1085	4
__umoddi3	5.34	626	1085	4
__sfvwrite	3.99	467	608	2
_malloc_r	3.84	450	158	1
_start	3.19	374	9595	<spontaneous>
_mbtowc_r	2.37	278	0	18
_WindowUnderflow8	2.2	258	0	10
_WindowOverflow8	2.08	244	0	10
memchr	2.02	237	48	3
main	1.87	220	8569	1
memmove	1.7	200	0	3
fflush	1.7	200	189	4
__sinit	1.43	168	37	1
__smakebuf	1.11	130	764	1
exit	0.94	111	574	1
_fwalk	0.84	99	291	1
printf	0.82	96	8458	1
_sbrk_r	0.82	96	24	2
vfprintf	0.73	86	8347	1
__swsetup	0.66	78	920	1
memset	0.58	68	0	1
__do_global_dtors_aux	0.58	68	13	1
_atexit	0.53	63	0	1
__sprint	0.48	57	1127	2
_Reset_epilog	0.44	52	10	<spontaneous>
__swrite	0.42	50	88	1
__do_global_ctors_aux	0.38	45	0	1
isatty	0.38	45	43	1
_write_r	0.34	40	24	1
_fstat_r	0.3	36	0	2
localeconv	0.24	29	41	1
_exit	0.23	28	10	1
unpackdone	0.23	28	0	<spontaneous>
_WindowOverflow4	0.23	27	0	2
_cleanup_r	0.21	25	416	1
fstat	0.21	25	18	1
_localeconv_r	0.14	17	0	1
_WindowUnderflow4	0.12	15	0	1
_fini	0.11	13	81	1
_init	0.11	13	45	1
_ResetVector	0.09	11	0	<spontaneous>
_xtos_init	0.08	10	0	1
xthal_dcache_all_writeback	0.08	10	0	1
__malloc_lock	0.08	10	0	1
__malloc_unlock	0.03	4	0	1

Note that with optimization level set to 2, the xt-xcc compiler automatically inlines the GOLDEN_BYTESWAP function. That's the reason why

GOLDEN_BYTESWAP didn't show up on the profile results above. In order to view the total cycles consumed by this function, turn inline off.

STEP 3.2: Identify the hot spots.

For the purpose of this lab, turn inlining off by adding a compiler flag –INLINE:=off. You should have learned how to set the compiler flag in previously lab. If necessary, please refer to the previous lab for instructions.



1. After turning inlining off, profile the code again. This time, after switching to the Benchmark perspective, you may be asked whether you want to replace the old profile results.

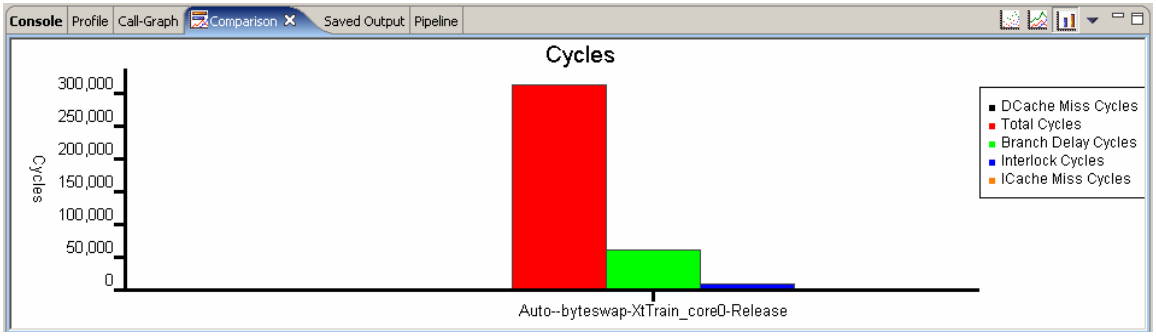
2. Click **Yes** to Continue. The profile information now contains the GOLDEN_BYTESWAP function.

Console | Profile | Call-Graph | Comparison | Saved Output | Pipeline

Function Name	Total cycles (%)	Function cycles	Children cycles	Times called (invocations)
GOLDEN_BYTESWAP	64.18	200025	0	10000
main	32.13	100149	208589	1
_vfprintf_r	0.59	1863	6243	1
_ResetHandler	0.52	1632	0	<spontaneous>
__udivsi3	0.35	1110	0	16
__umodsi3	0.33	1058	0	16
__udivdi3	0.27	845	1084	4
__umoddi3	0.2	644	1084	4
_malloc_r	0.14	446	148	1
__sfvwrite	0.14	445	615	2
_start	0.12	374	309535	<spontaneous>
memchr	0.08	254	47	3
_mbtowc_r	0.08	266	0	18
_WindowUnderflow8	0.07	243	0	10
_WindowOverflow8	0.07	239	0	10
memmove	0.06	200	0	3

This view contains several pieces of useful profiling information.

3. Click the **Comparison** tab to chart the benchmark data.



The chart shows the data cache miss cycles, total cycles taken by the code, branch delay cycles, interlock cycles and instruction cache miss cycles.

Question 1. Which function takes the most number of cycles?

……

Question 2. How many cycles does that function take?

……

Task 4. Developing a TIE Instruction

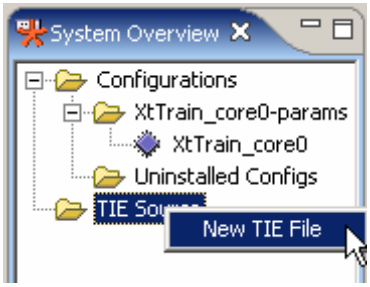
There are several mask and shift instructions in the **GOLDEN_BYTESWAP()** function**.** In addition, there is an add operation to increment the GOLDEN_COUNT value. This value shows how many times this operation is executed. We will use the fusion technique to combine these operations.

When you think in terms of hardware, the mask and shift operations are "re-wiring" certain bits from the input to the output. In this task, we will develop a TIE instruction that implements this "re-wiring". This TIE instruction will also fuse the "rewiring" operation with the add operation. The add operation increments GOLDEN_COUNT.

STEP 4.1: Create a TIE file.

1. Open the C/C++ Perspective.

2. From the System Overview View, right-click **TIE Source** and select **New TIE File** to open the Create TIE File dialog box.

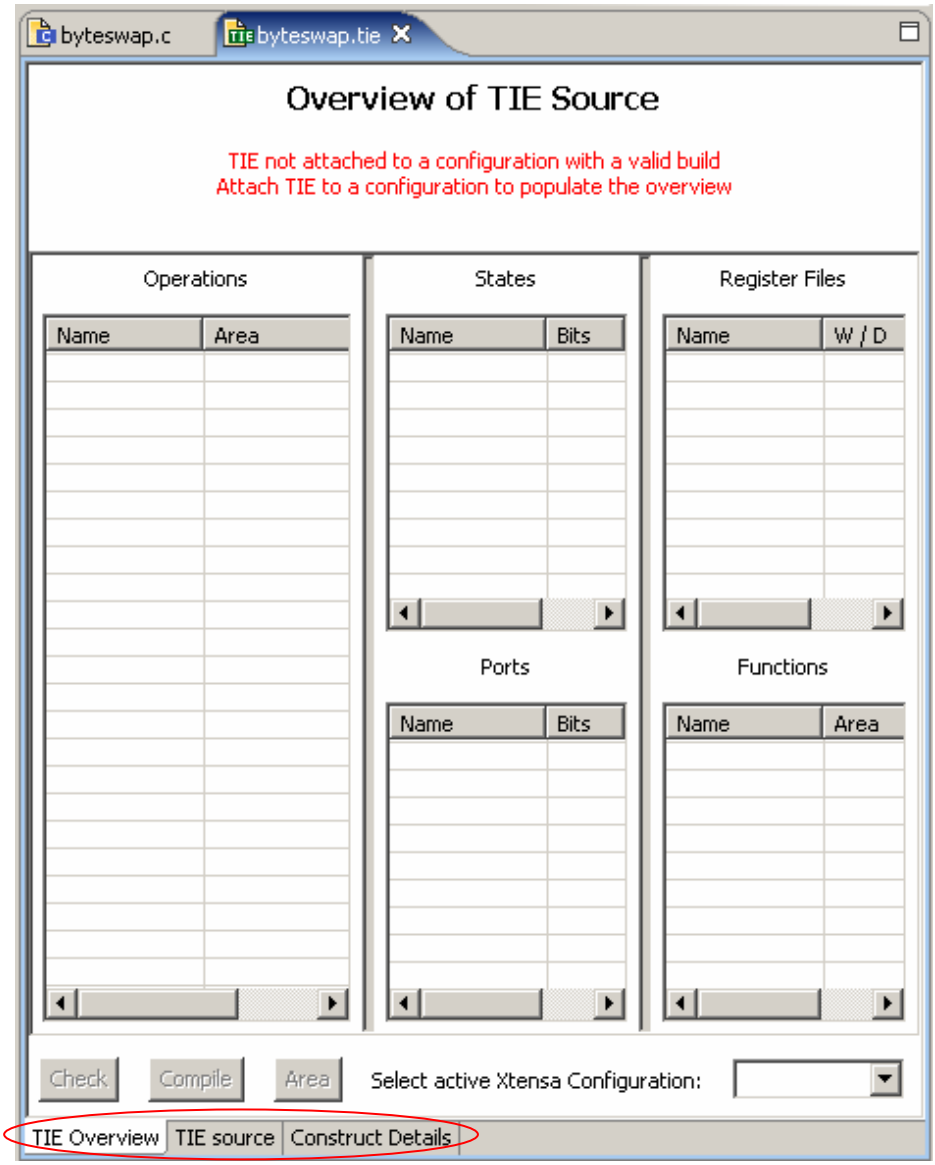


3. Type the TIE file name, **byteswap.tie**, and click **Next**.

4. From the Select a Configuration dialog box, select **<none>** and click **Finish**.

At this point of the lab, we will not attach the TIE file to the XtTrain_core0 processor configuration. Instead, we will make a clone of the **XtTrain_core0** processor configuration and then attach this TIE file to the clone. This way we keep the original processor configuration for comparison.

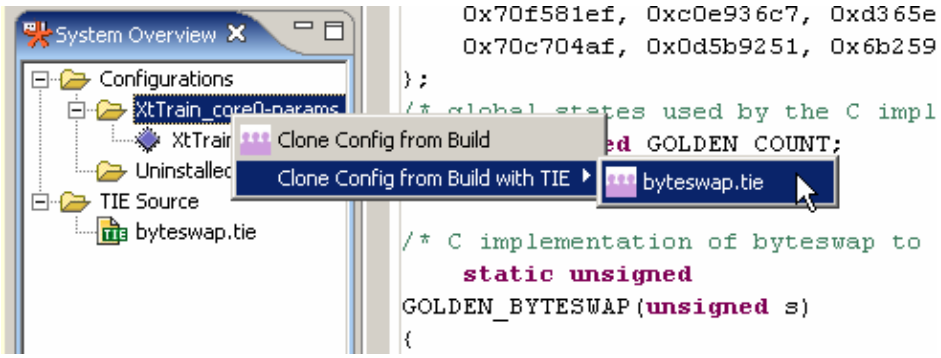
You will see **byteswap.tie** in the Xplorer's Editor View. This view contains the following tabs at the bottom of the window: **TIE Overview, TIE source**, and **Construct Details**. Later we will click the **TIE source tab** to develop the TIE instruction.



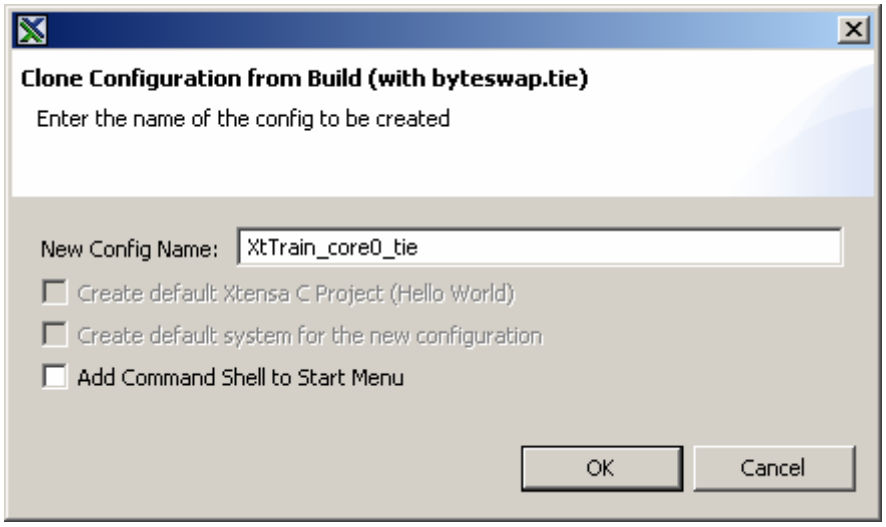
STEP 4.2: Clone a processor configuration and attach the TIE file.

We are cloning a processor configuration to be able to keep the original configuration for development purposes.

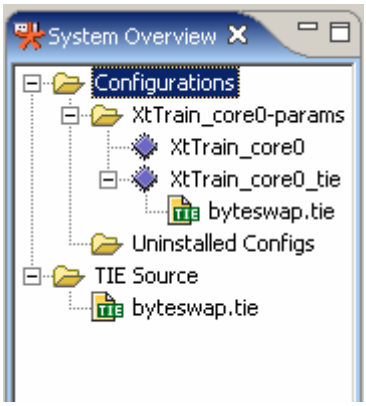
1. From the System Overview View, expand **Configurations** and right-click **XtTrain_core0-params**, select **Clone Config with TIE**, and then select **byteswap.tie**.



2. For the **New Config Name**, enter **XtTrain_core0_tie**, then click **OK** to close the dialog box.



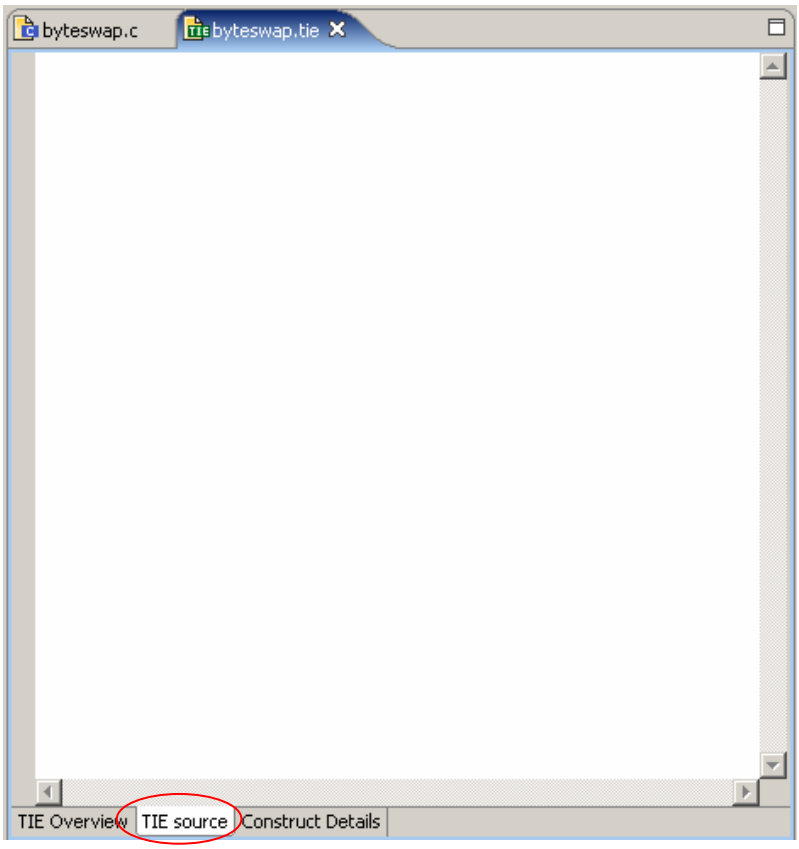
The name of the original processor configuration is **XtTrain_core0**. The clone processor configuration name is **XtTrain_core0_tie**. These two processor configurations are shown under **XtTrain_core0-params** in the System Overview View's Configurations folder.



STEP 4.3: Develop the TIE instruction.

1. Expand **TIE Source** in the System Overview View, and double-click **byeswap.tie** to open the file in Xplorer's Editor View.

2. Click the **TIE source** tab.



This is where you will develop your TIE instruction.

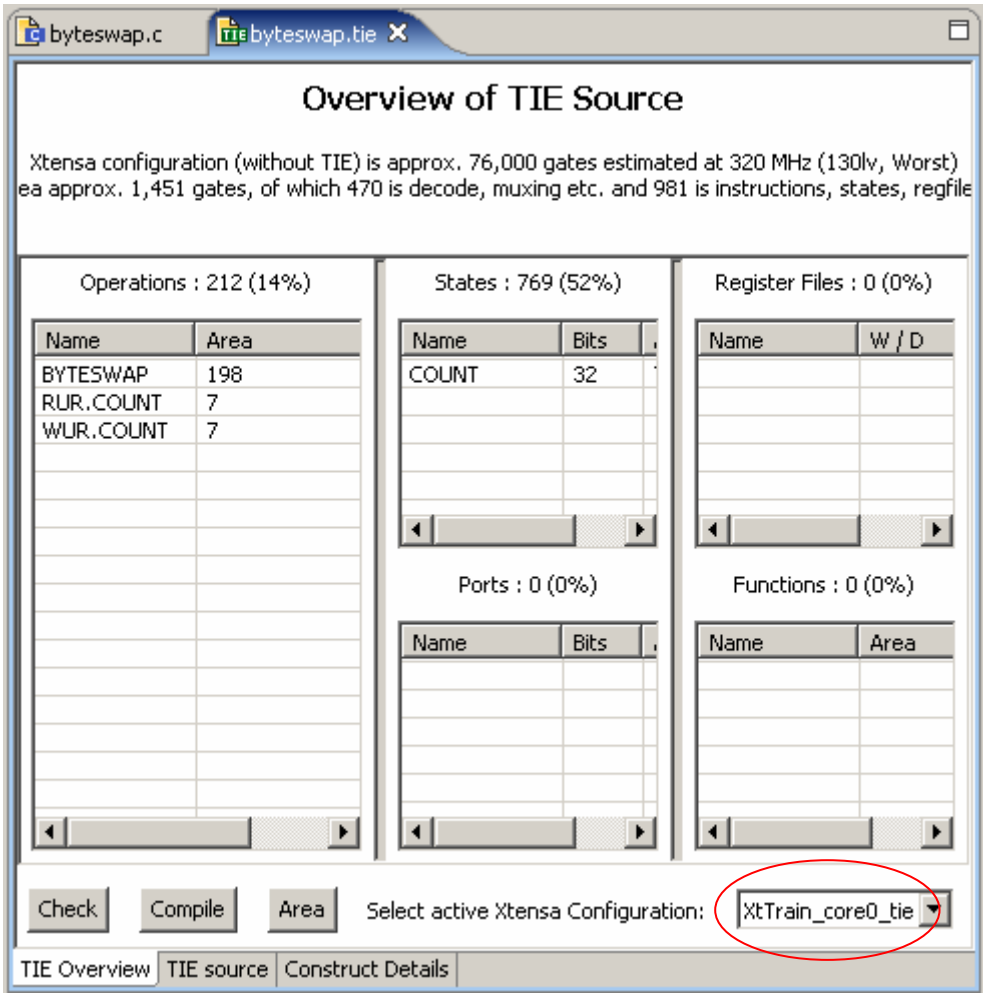
Following are some hints to help you develop your TIE instruction (see the Original C Source Code on page 76 of this document):

- It should be a single "operation" statement
- The input to the operation should come from an AR register
- The output of the operation should go to an AR register
- Use "assign" and bit selection to perform the "rewiring"
- Create a 32-bit storage for COUNT that is used to store how many times this TIE instruction has been used
- Add into the operation the incrementing of COUNT

3. Save the file.

STEP 4.4: Compile the created TIE.

1. Click the **TIE Overview** tab in the Editor View.



2. If not already selected, select **XtTrain_core0** as the active Xtensa Configuration option. This ensures that the TIE Compiler (tc) will compile your TIE for the correct configuration.

3. Click **Check** to check the TIE. Correct any errors in your TIE.

4. Click **Compile** to compile the TIE for the processor configuration. Correct any errors in your TIE. Check your TIE against the example in the back of the document.

   Compiling a TIE file creates the necessary updates for the software tools.

Task 5. Running and Profiling with the New TIE Instruction

To use the new TIE instruction, we need to make changes to the C-code. These changes facilitate profiling and benchmark comparison later in the lab. We will also create a new launch for this new binary.

STEP 5.1: Modify byteswap.c to use the TIE instruction.

When compiling a software project in Xtensa Xplorer, a compiler flag referring to the specific processor configuration is automatically included. This compiler flag can be very useful. The compiler flag uses the form of CONFIG_<*processor configuration name*>. In this case, the compiler flag is **CONFIG_XtTrain_core0_tie**.

First, we need to instruct the C compiler that we are using the new TIE instruction. To do so, we include the TIE header file.

1. Add these lines to the beginning of **byteswap.c**

```
#ifdef CONFIG_XtTrain_core0_tie
#include <xtensa/tie/byteswap.h>
#endif
```

2. Then we initialize the storage element, **COUNT**, in our TIE instruction by using the C-intrinsics, **WUR_<*user-defined state*>** and **RUR_<*user-defined state*>**. These C-intrinsics are automatically created when a **state** has the **add_read_write** modifier.

   Add these lines at the correct location in **byteswap.c**

```
#ifdef CONFIG_XtTrain_core0_tie
WUR_COUNT(0);
#endif
```

3. Now, instead of replacing GOLDEN_BYTESWAP() with the TIE instruction BYTESWAP, we will verify the TIE instruction BYTESWAP against GOLDEN_BYTESWAP(). We will do this by adding a comparison check inside the loop to compare the result of the TIE instruction, BYTESWAP**,** against GOLDEN_BTESWAP().

   Replace the following code

```
GOLDEN_BYTESWAP(s);
```

   with

```
#ifdef CONFIG_XtTrain_core0_tie
   if (GOLDEN_BYTESWAP(s) != BYTESWAP(s)) fail++;
#else
   GOLDEN_BYTESWAP(s);
#endif
```

4. Replace the following code

```
printf("Swapped %d words \n", i);
```

with

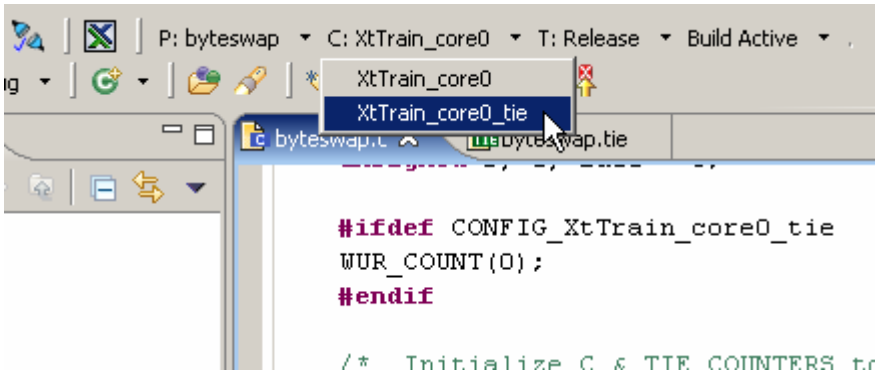
```
#ifdef CONFIG_XtTrain_core0_tie
   printf("State COUNT=%d\n", RUR_COUNT());
   printf("Swapped %d words \n", i);
   printf("%s\n", fail ? "Mismatch detected" : "Your TIE works!");
#else
   printf("Swapped %d words \n", i);
#endif
```

5. Save the file.

STEP 5.2: Build the modified byteswap.c file in the new system.

First, we need to set up the build for the new system.

1. Change the active configuration from XtTrain_core0 to XtTrain_core0_tie.



We need to tell Xplorer that we want to build the software project **byteswap** for the core **XtTrain_core0_tie** in the new system.

2. Click **Build Active.**

Xplorer builds the project. Correct any build errors at this time.

3. Click **Profile** to view the Benchmark results.

Task 6. Comparing Profile Results

Now we have the profiling information for both processor configurations. We can compare the performance gain from the TIE instruction.

STEP 6.1: Find out how many cycles the TIE instruction takes.

In the Profile view:

1. Click the **Function cycles** column in the Profile view to sort data on function cycles.

   Can you find the number of cycles spent on the TIE instruction? Do you know the reason?

   The TIE instruction is a single cycle instruction. BYTESWAP() is a C macro that is translated into an assembly instruction, BYTESWAP, during compilation. It is not a real function.

   To see how many cycles are spent on the TIE instruction, BYTESWAP, use the **Profile Disassembly** view.

2. Click on the function **main** in the **Profile** view**,** and the **Profile Disassembly** view opens on the right side of the Xplorer window. If it does not open, from the **Window** menu, select **Show View**, then **Profile Disassembly**.

The **Profile Disassembly** view displays the assembly instructions that correspond to the function that you have selected in the **Profile** view. You are now seeing the assembly instructions generated for the function **main().**

Look for the TIE instruction **BYTESWAP**.

Question 3. How many cycles are spent in the TIE instruction BYTESWAP through the entire application?

……

Question 4. What is the performance gain by using TIE instruction?

……

You will see that even though this single cycle BYTESWAP is executed 10,000 times, the number of cycles associated to the execution of BYTESWAP is more than 10,000 cycles. There is pipeline interlock penalty associated to the execution of BYTESWAP. Could you identify it? One way to try to remove the interlock penalty is by using a higher degree of optimization. You could optionally try to raise the optimization level to 3 and profile this code again.

Task 7. Performing a Graphical Comparison (optional)

The profiling information contains information such as cache misses and pipeline interlocking. We will modify the code slightly so that we can graphically compare the TIE byteswap with the original C implementation of byteswap.

STEP 7.1: Make a change in the software project.

1. Replace this portion of the code

```
#ifdef CONFIG_XtTrain_core0_tie
   if (GOLDEN_BYTESWAP(s) != result) fail++;
#else
```

with

```
#ifdef CONFIG_XtTrain_core0_tie
  // if (GOLDEN_BYTESWAP(s) != result) fail++;
   BYTESWAP(s);
#else
```

2. Save the file (using **CTRL-S).**

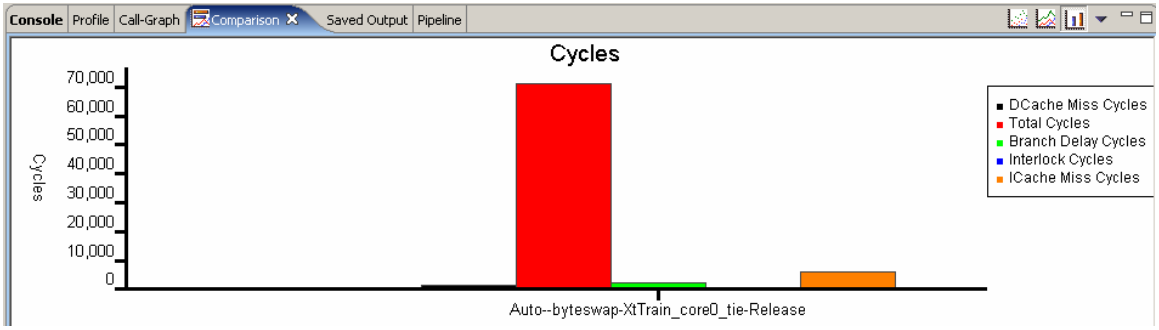
STEP 7.2: Recompile the software project.

1. Make sure you are building the correct Active Set.

Click **Profile** to begin profiling the binary.

STEP 7.3: View profiling information graphically.

1. Click the **Comparison** tab at the bottom.



2. Compare the chart results with the one for the original code shown earlier.

Conclusion

In this lab, we have created our first TIE instruction to do a byteswap. As we have seen, using typical processor operations to implement a byteswap function requires around 20 cycles. TIE is used here to accelerate byteswap. The result is a single cycle operation to compute the byteswap. This TIE instruction achieves 20 times speed up.

We have also employed a methodology to identify a hot spot in an algorithm for acceleration. We benchmarked the original C code, found the hot spot to be the byteswap function, and then developed TIE to accelerate it. This methodology helps us focus on the part of the program that could benefit the most from TIE development.

Original C Source Code

```
/*****************************************************************
  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;
   return ss;
}

int main()
{
   unsigned s, i, fail = 0;
```

```
   /*  Initialize C & TIE COUNTERS to 0 */
   GOLDEN_COUNT=0;

   for (i = 0; i < NUM; i++) {
      s = data[i % N];

      GOLDEN_BYTESWAP(s);


   }

   printf("Swapped %d words \n",GOLDEN_COUNT);
}
```

TIE Source Code

```
// declare state SWAP and COUNT
state COUNT 32 add_read_write

operation BYTESWAP {out AR outR, in AR inpR}{inout COUNT}
{
  assign outR = {inpR[7:0],inpR[15:8],inpR[23:16],inpR[31:24]};
  assign COUNT = COUNT + 1;
}
```

Modified C Source Code

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

#include <stdio.h>
#ifdef CONFIG_XtTrain_core0_tie
#include <xtensa/tie/byteswap.h>
#endif

/* 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;

/* 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;
   return ss;
}

int main()
{
   unsigned s, i, fail = 0;
```

```
   /*  Initialize C & TIE COUNTERS to 0 */
   GOLDEN_COUNT=0;
#ifdef CONFIG_XtTrain_core0_tie
    WUR_COUNT(0);
#endif

   for (i = 0; i < NUM; i++) {
      s = data[i % N];
#ifdef CONFIG_XtTrain_core0_tie
   if (GOLDEN_BYTESWAP(s) != BYTESWAP(s)) fail++;
#else
      GOLDEN_BYTESWAP(s);
#endif

   }
#ifdef CONFIG_XtTrain_core0_tie
   printf("State COUNT=%d\n",RUR_COUNT());
   printf("%s\n", fail ? "Mismatch detected" : "Your TIE works!");
#else
    printf("Swapped %d words \n", i);
#endif


}
```

Answers / Solutions]

Question 1. Which function takes the most number of cycles?

Answer: GOLDEN_BYTESWAP()

Question 2. How many cycles does that function take?

Answer: 200,025 cycles

Question 3. How many cycles are spent in the TIE instruction BYTESWAP through the entire application?

Answer: 20,000 cycles due to interlock penalty (10,000 cycles if using –O3)

Question 4. What is the performance gain by using TIE instructions?

Answer: 10 times speed up; 20 times if using –O3