



TMS320C6000 Optimization Workshop

Lab Exercises

**OP6000 Student Notes
Revision 1.51c
March 2011**

***Technical Training
Organization (TTO)***

Notice

Creation of derivative works unless agreed to in writing by the copyright owner is forbidden. No portion of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without the prior written permission from the copyright holder.

Texas Instruments reserves the right to update this Guide to reflect the most current product information for the spectrum of users. If there are any differences between this Guide and a technical reference manual, references should always be made to the most current reference manual. Information contained in this publication is believed to be accurate and reliable. However, responsibility is assumed neither for its use nor any infringement of patents or rights of others that may result from its use. No license is granted by implication or otherwise under any patent or patent right of Texas Instruments or others.

Copyright © 1997 - 2011 by Texas Instruments Incorporated.
All rights reserved.

Training Technical Organization
Texas Instruments, Incorporated
6500 Chase Oaks Blvd, MS 8437
Plano, TX 75023
(214) 567-0973

Revision History

October 2002, Version 1.0

October 2002, Version 1.0a (Errata only)

April 2003, Version 1.1

- Lab solutions for C64x (as well as C67x)
- Labs now use CCS v2.2

January 2005, Version 1.2

- Add support for CCS 3.0
- Add tutorials for code tuning tools

August 2005, Version 1.3

- Add C64x+ CPU information

December 2007, Version 1.4

- Add support for CCS 3.3

March 2011, Version 1.51

- Partial support for CCSv4
- Updated for new devices (C674x, C66x)

Chapter 1 – Exam

It's now time to prove you've been intently listening the past hour or so. You have approximately 20 minutes to complete the exam. Afterwards, the instructor will debrief each question while your neighbor grades your exam. Have fun!

- You will have 20 minutes to complete the exam.
- Exam is open mind, open book, open eyes. Sharing answers, cheating, asking the instructor questions, anything to get the highest grade possible is completely acceptable, encouraged, and expected.
- Good luck!

Student Name: _____

1. Functional Units

- a. How many can perform an ADD? Name them. (5 pts)

- b. Which support memory loads/stores? (5 pts)

.M .S .P .D .L

2. C6000 Memories and Busing

- a. What is the advantage of L1 memory? (5 pts)

- b. What does the EDMA3 stand for ... and what does it do? (5 pts)

- c. What is an SCR master? (5 pts)

3. Conditional Code

- a. Which registers can be used as conditional registers? (5 pts)

- b. Which instructions can be conditional? (5 pts)

4. Performance

- a. What is the 'C6201 instruction cycle time? (5 pts)

- b. How can the 'C6201 execute 1600 MIPs? (10 pts)

How many MIPs can the 'C6678 execute?

- c. How many 16-bit MMACs (millions of MACs) can the 'C6201 perform? (10 pts)

How about C674x or C66x cores?

5. Coding Problems (write the code to perform the following)

- a. Move contents of A0 → A1. (5 pts)

- b. Clear register A5. (5 pts)

- c. $A2 = A0^2 + A1$ (5 pts)

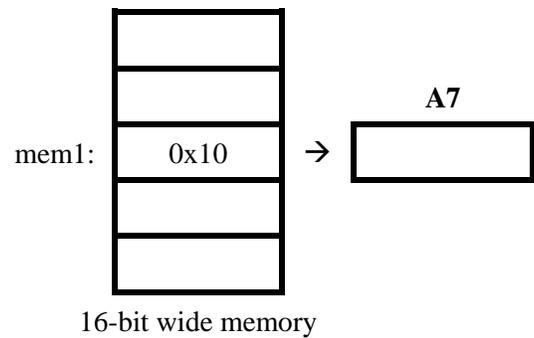
Part Number	CPU	Peak MMACS	Frequency (MHz)	Single-Cycle SRAM per core (L1)	On-Chip SRAM per core (L2)	Local SRAM (LL2)	Shared SRAM (SL2)
AVCE6467T	C64x+ ARM9	8000	1000	32KB L1P, 32KB L1D	128 KB		
DM3730-1000	C674x ARM Cortex-A8	6400	800	64 KB (ARM Cortex-A8)	?		
OMAP3530	C674x ARM Cortex-A8	4160	520	32KB L1P, 80KB L1D	?		
OMAP-L137	C674x ARM9	3000, 3648	375, 456	32KB L1P, 32KB L1D	256 KB		
OMAP-L138	C674x ARM9	3000, 3648	375, 456	32KB L1P, 32KB L1D	256 KB		
TMS320C6201-200	1 C62x	400	200	128 KB			
TMS320C6202B-300	1 C62x	600	300	384 KB			
TMS320C6203B-300	1 C62x	600	300	896 KB			
TMS320C6204-200	1 C62x	400	200	128 KB			
TMS320C6205-200	1 C62x	400	200	128 KB			
TMS320C6211B-167	1 C62x	334	167	8 Kb	64 Kb		
TMS320C6410-400	1 C64x	3200	400	32 KB	128 KB		
TMS320C6411-300	1 C64x	2400	300	32 KB	256 KB		
TMS320C6412-720	1 C64x	5760	720	32 KB	256 KB		
TMS320C6413-500	1 C64x	4000	500	32 KB	256 KB		
TMS320C6414T-1000	1 C64x	8000	1000	32 KB	1024 KB		
TMS320C6415T-1000	1 C64x	8000	1000	32 KB	1024 KB		
TMS320C6416T-1000	1 C64x	8000	1000	32 KB	1024 KB		
TMS320C6418-600	1 C64x	4800	600	32 KB	512 KB		
TMS320C6421-700	1 C64x+	5600	700	32 KB	64 KB		
TMS320C6424-700	1 C64x+	5600	700	112 KB	128 KB		
TMS320C6452-900	1 C64x+	7200	900	64 KB	1408 KB		
TMS320C6455-1200	1 C64x+	9600	1200	64 KB	2048 KB		
TMS320C6457-1200	1 C64x+	9600	1200	64 KB	2048 KB		
TMS320C6472-700	6 C64x+	33600	700	32KB L1P, 32KB L1D	4416KB (less 768K)		768 KB
TMS320C6474-1200	3 C64x+	28800	1200	32KB L1P, 32KB L1D	3072 KB		
TMS320C6670	4 C66x	153000	1000, 1200	32KB L1P, 32KB L1D		256 KB * 4	2048 KB
TMS320C6672	2 C66x	80000	1000, 1250	32KB L1P, 32KB L1D		128 KB * 4	4096 KB
TMS320C6674	4 C66x	160000	1000, 1250	32KB L1P, 32KB L1D		256 KB * 4	4096 KB
TMS320C6678	8 C66x	320000	1000, 1250	32KB L1P, 32KB L1D		512 KB * 8	4096 KB
TMS320C6701-167	1 C67x	334	167	128 KB			
TMS320C6711D-200	1 C67x	400	200	8 KB	64 KB		
TMS320C6711D-250	1 C67x	500	250	8 KB	64 KB		
TMS320C6713B-300	1 C67x	600	300	8 KB	256 KB		
TMS320C6727B-350	1 C67x+	700	350	32 KB			
TMS320C6742	1 C674x	1600	200	64 KB	64 KB		
TMS320C6743	1 C674x	1600, 3000	200, 375	64 KB	128 KB		
TMS320C6745	1 C674x	3000, 3648	375, 456	64 KB	256 KB		
TMS320C6746	1 C674x	3000, 3648	375, 456	64 KB	256 KB		
TMS320C6747	1 C674x	3000, 3648	375, 456	64 KB	256 KB		
TMS320C6748	1 C674x	3000, 3648	375, 456	64 KB	256 KB		
TMS320C6A8168	1 C674x ARM Cortex-A8	12000	1500	32KB L1P, 32KB L1D	256 KB		
TMS320DM642-720	C64x+ ARM9	5760	720	32 KB	256 KB		
TMS320DM6437-700	C64x+	5600	700	32KB L1P, 80KB L1D	128 KB		
TMS320DM6446-810	C64x+ ARM9	6480	810	32KB L1P, 80KB L1D	64 KB		
TMS320DM6467-594	C64x+ ARM9	4752	594	32KB L1P, 32KB L1D	128 KB		
TMS320DM6467-729	C64x+ ARM9	5832	729	32KB L1P, 32KB L1D	128 KB		
TMS320DM6467T-1000	C64x+ ARM9	8000	1000	32KB L1P, 32KB L1D	128 KB		
TMS320DM648-1100	C64x+ ARM9	8800	1100	32KB L1P, 32KB L1D	512 KB		
TMS320DM8168	1 C674x ARM Cortex-A8	12000	1000	32KB L1P, 32KB L1D	256 KB		
VCE6467T	C64x+ ARM9	8000	1000	32KB L1P, 32KB L1D	128 KB		

d. If $(B1 \neq 0)$ then $B2 = B5 * B6$ (10 pts)

e. Load an *unsigned* constant (19ABCh) into register A6. (5 pts)

f. Load A7 with the contents of mem1 and post increment the selected pointer. (10 pts)

*Basically, we want you to load the value 0x10 (at memory address **mem1**) into register A7.*



Chapter 1 Exam Scoring

Points Earned (Questions #1-5): _____/100

Grade (circle one):

90-100 A

80-89 B

0-79 C

(no one likes Ds or Fs!)

Page left intentionally blank.

Lab 2 – Overview

To begin Lab 2, we need to prepare our lab workstation by setting up Code Composer Studio (CCS). After this is accomplished, you can complete the main exercises and optional exercises.

Main Exercises

Lab 2 consists of four parts:

A. Prepare your lab workstation

Setup CCS preferences as needed for this workshop.

B. Build and debug C code using the *Debug* configuration

A dot-product (sum-of-products) function provides a simple, but important, algorithm to familiarize us with compiling and debugging C code within Code Composer Studio. Build options and configurations, command line interface, breakpoints, performance profiling, and using the Watch Window are some of the various features explored.

C. Use the *Release* configuration to quickly optimize C code

Once your C code has been debugged and is working, the next step is to optimize with the compiler's optimizer. Using the profiler, we'll contrast the efficiency of the compiler's optimizer.

D. Benchmark using the Profiler tool

Parts B and C used the simple clock-oriented benchmarking methodology. Part D introduces the Profiler Setup and Viewer windows. The profiling interface will be further explored as we examine the tuning tools later in the workshop.

Optional Exercises

Three optional exercises are available. The optional exercises are provided for those of you who finish the main exercises early; they extend the learning of the concepts presented in this chapter. If you do not get the chance to complete them during the assigned lab time, please feel free to work on them before or after class, or take them home (take-home solutions will be provided).

- **Lab2e – Customize Your Workspace**
- **Lab2f – Using GEL**
- **Lab2g – Graphing Data**

C64x, C64+, C671x, or C672x Exercises?

We support four processor types in these workshop lab exercises. Please see the specific callouts for each processor as you work. Overall, there are very little differences between the procedures.

Lab Exercises

- ◆ Which processor do you want to use in lab?
- ◆ We provide instructions and solutions for the C64x, C64x+, C671x and C672x.
- ◆ We try to callout the few differences in lab steps as explicitly as possible.

Lab2 – Using CCS with the Debug Configuration

Adding files to the project

You can add files to a project in one of three ways

- Select the menu **Project**→**Add files to Project...**
- Right-click the project icon in the Project Explorer window and select **Add files...**
- Drag and drop files from Windows Explorer onto the project icon

25. Using one of these methods, add the following files from `c:\op6000\labs\lab2` to your project:

67	MAIN.C C67.CDB C67CFG.CMD	64	MAIN.C C64.CDB C64CFG.CMD
-----------	---------------------------------	-----------	---------------------------------

When these files have been added, the project should look like:





Lab 2a: Prepare Lab Workstation

The computers used in TI's classrooms and dedicated workshops may be configured for one of ten different courses. The last class taught may have been DSP/BIOS, C2000 or C5000 workshop. To provide a consistent starting point for all users, we need to have you complete a few steps to reset the CCS environment to a known starting point.

Computer Login

1. **If the computer is not already logged-on, check to see if the log-on information is posted on the workstation. If not, please ask your instructor.**

CCS Setup

Code Composer Studio (CCS) can be used with various TI processors – such as the C6000 and C5000 families – and each of these has various target-boards (simulators, EVMs, DSKs, and XDS emulators). Code Composer Studio must be properly configured using the CCS_Setup application.

In this workshop, you should initially configure CCS for the processor that you selected. Use one of the following:

C64xx CPU Cycle Accurate Simulator, Little Endian

C64x+ CPU Cycle Accurate Simulator, Little Endian

C67xx CPU Cycle Accurate Simulator, Little Endian

C672x CPU Simulator, Little Endian (located in the C67xx family in Setup CCStudio)

Between you and your lab partner, pick the processor that best meets your needs. In any case, the learning objectives will be the same whichever target you choose.

2. **Start the CCS Setup utility using its desktop icon:**



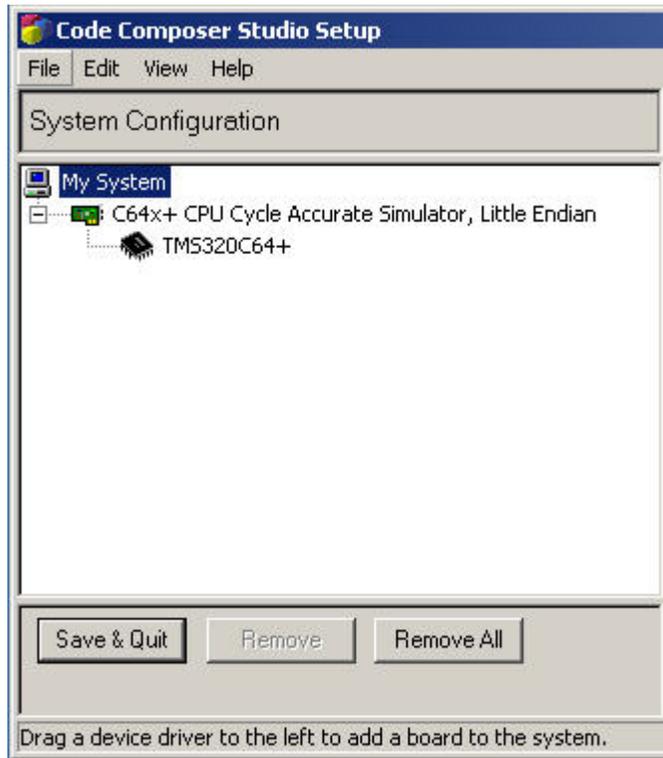
Note: Be aware there are two CCS icons, one for setup, and the other to start the CCS application. You want the **CCS Setup C6000** icon.

If you cannot find the desktop icon shortcut, try looking for it under the Windows Start menu:
Start→*Programs*→*Texas Instruments*→*Code Composer Studio 3.3*→*Setup Code Composer Studio 3.3*

If all else fails, you should be able to find the program itself at:

C:\CCStudio_v3.3\cc\bin\cc_setup.exe

3. When you open CC_Setup you should see a screen similar to this:



4. Clear the previous configuration.

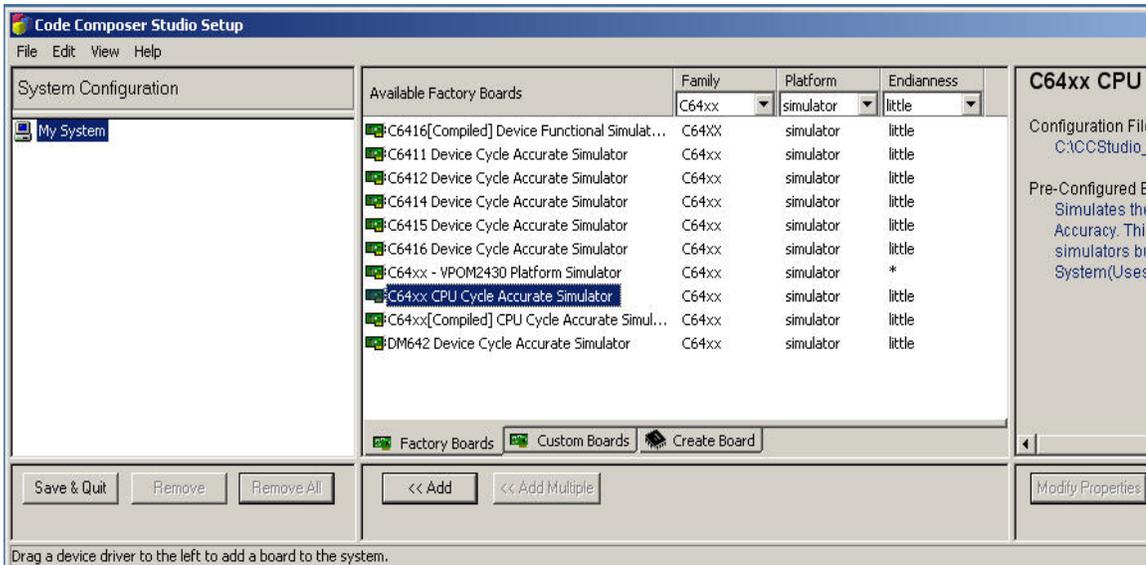
Before you select a new configuration you should delete the previous configuration if one exists. Click the *Remove All* button. CC_Setup will ask if you really want to do this, choose "Yes" to clear the configuration.



5. Select a new configuration from the list by selecting the appropriate *Family*, *Platform*, and *Endianness* filters. Select the appropriate simulator from the list and click the “<<Add” button.

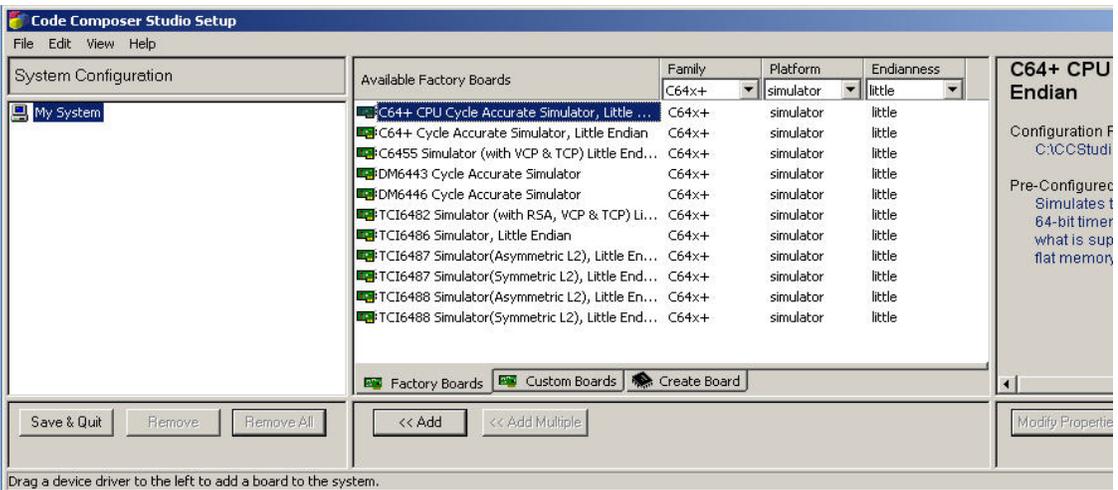
64

If you want to experiment with the C64x devices in this workshop, please choose the *C64xx CPU Cycle Accurate Simulator, Little Endian* simulator and click **Add**:



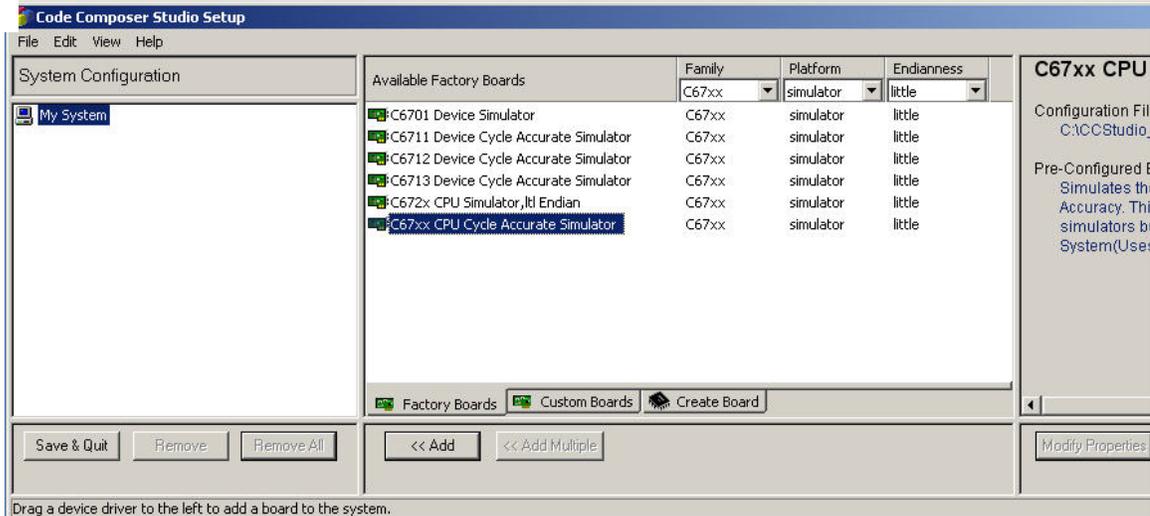
64+

If you want to experiment with the C64x devices in this workshop, please choose the *C64x+ CPU Cycle Accurate Simulator, Little Endian* simulator and click **Add**:



671x

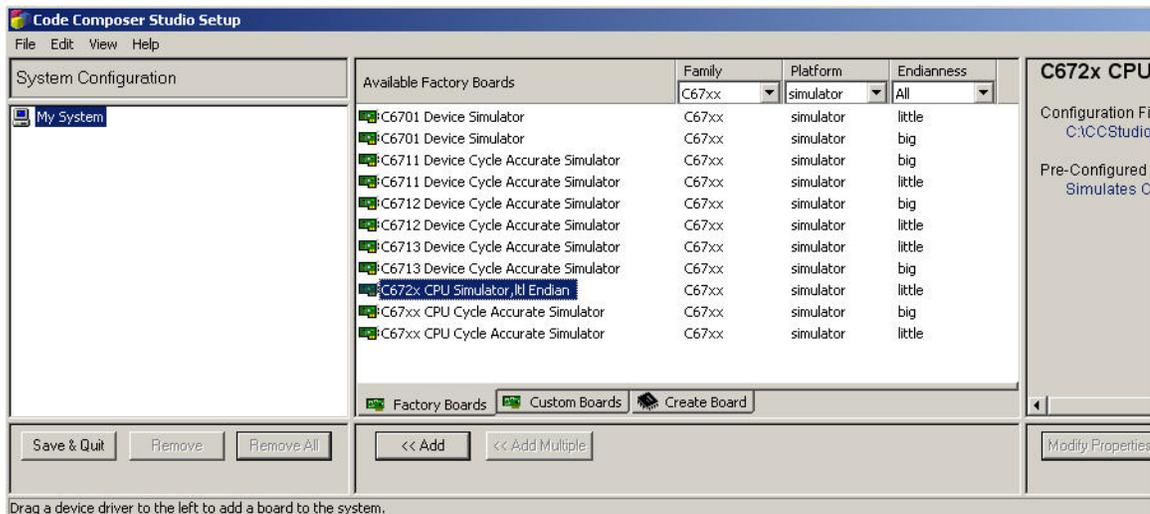
If you want to experiment with the C67x devices in this workshop, please choose the *C67xx CPU Cycle Accurate Sim, Little Endian* simulator and click **Add**:



672x

If you want to experiment with the C672x devices in this workshop, please choose the *C672x CPU Simulator, Little Endian* simulator and click **Add**.

Note: The C672x CPU Simulator is located in the **C67xx Family** in CCS Setup and not in the C672x Family.



6. Save and Quit the *System Configuration* dialog.
7. Select “Yes” to start CCS upon exiting CCS setup.

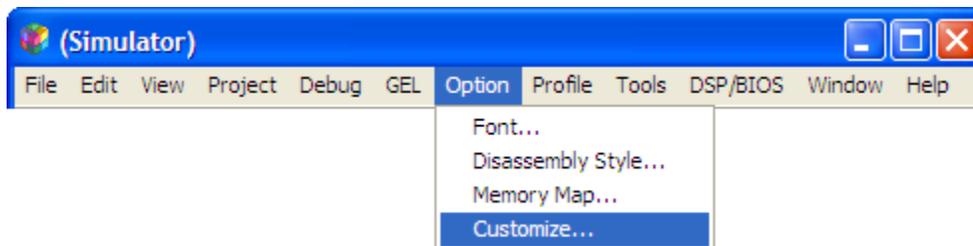


Set CCS – Customize Options

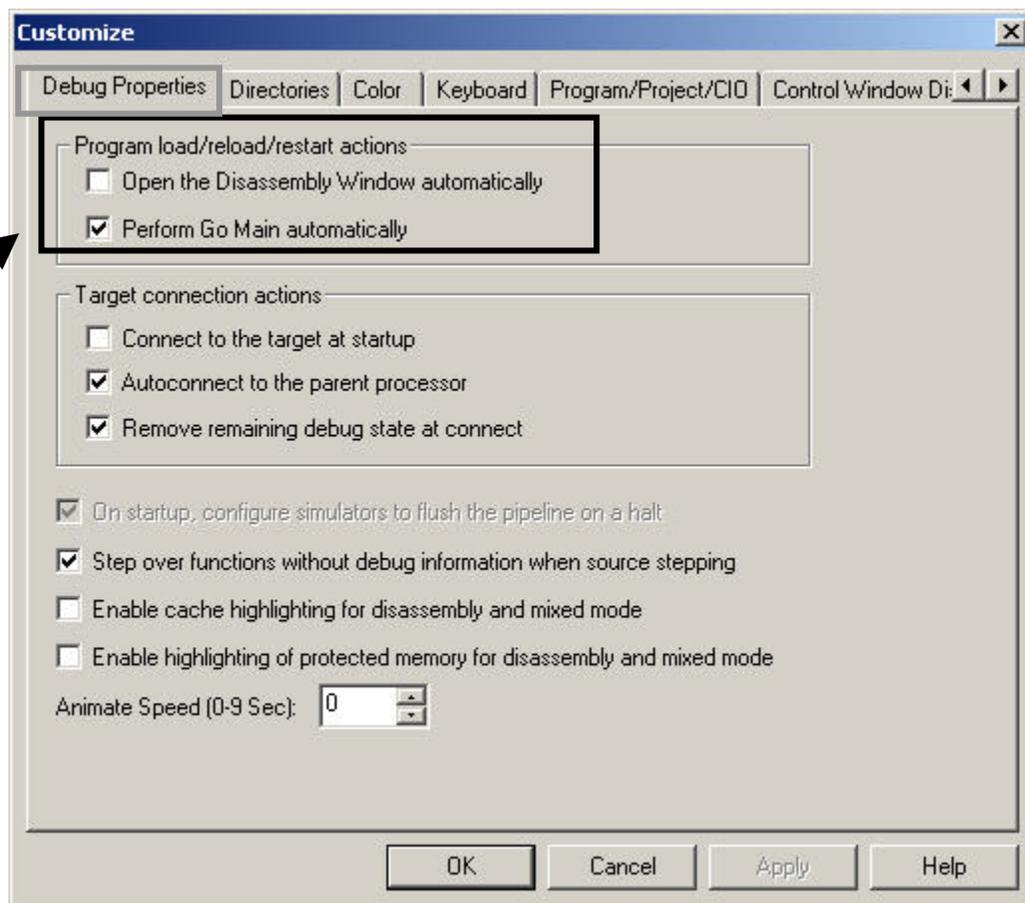
There are a few option settings that need to be verified before we begin. Otherwise, the lab procedure may be difficult to follow.

- Disable open Disassembly Window upon load
- Go to main() after build
- Program load after build
- Clear breakpoints when loading a new program
- Set CCS Titlebar information

8. Open the Options→Customize Dialog.



9. Set Debug Properties



Here are a couple options that can help make debugging easier.

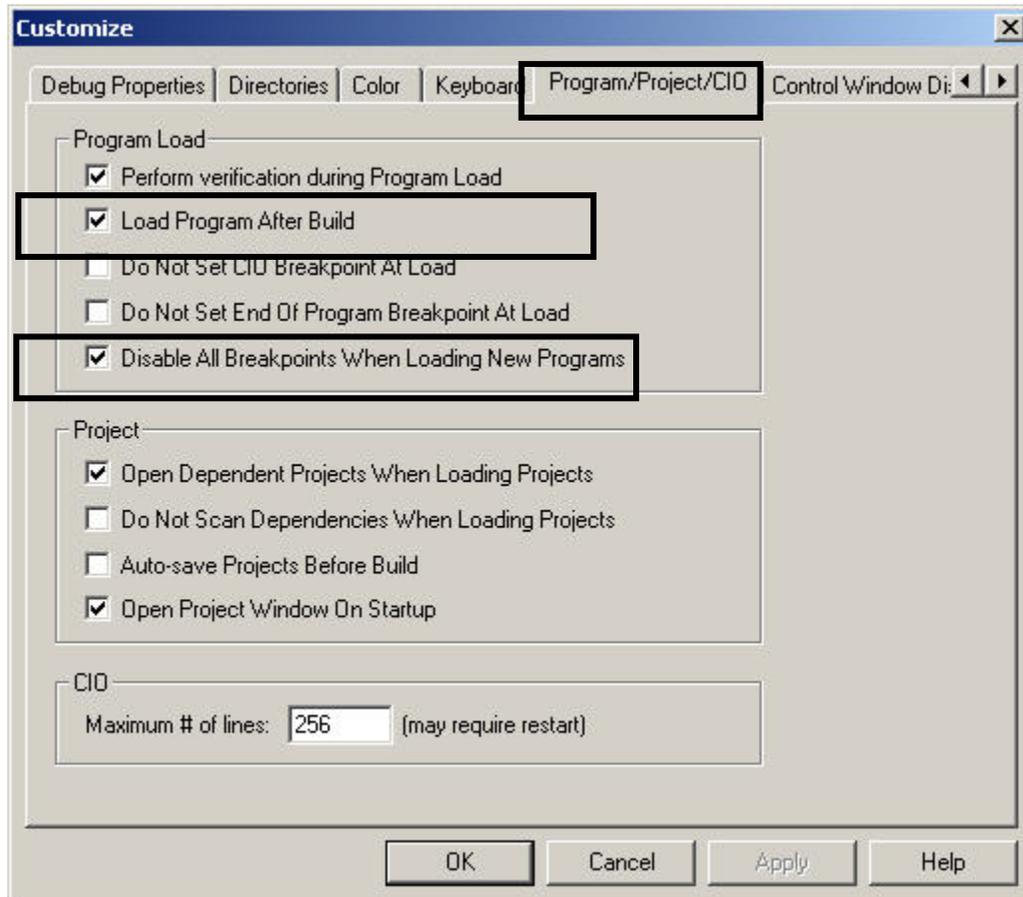
- Unless you want the *Disassembly Window* popping up every time you load a program (which annoys many folks), deselect this option.
- Many find it convenient to choose the “Perform Go Main automatically”. Whenever a program is loaded the debugger will automatically run thru the compilers initialization code to your main() function.

10. Set Program Load Options

On the “Program/Project/CIO” tab, select the two following options:

- Load Program After Build
- Clear All Breakpoints When Loading New Programs

By default, these options are not enabled, though a previous user of your computer may have already enabled them.



Conceptually, the CCS Integrated Development Environment (IDE) is made up of two parts:

- **Edit** (and Build) programs
Uses editor and code gen tools to create code.
- **Debug** (and down Load) programs
Communicates with processor/simulator to download and run code.

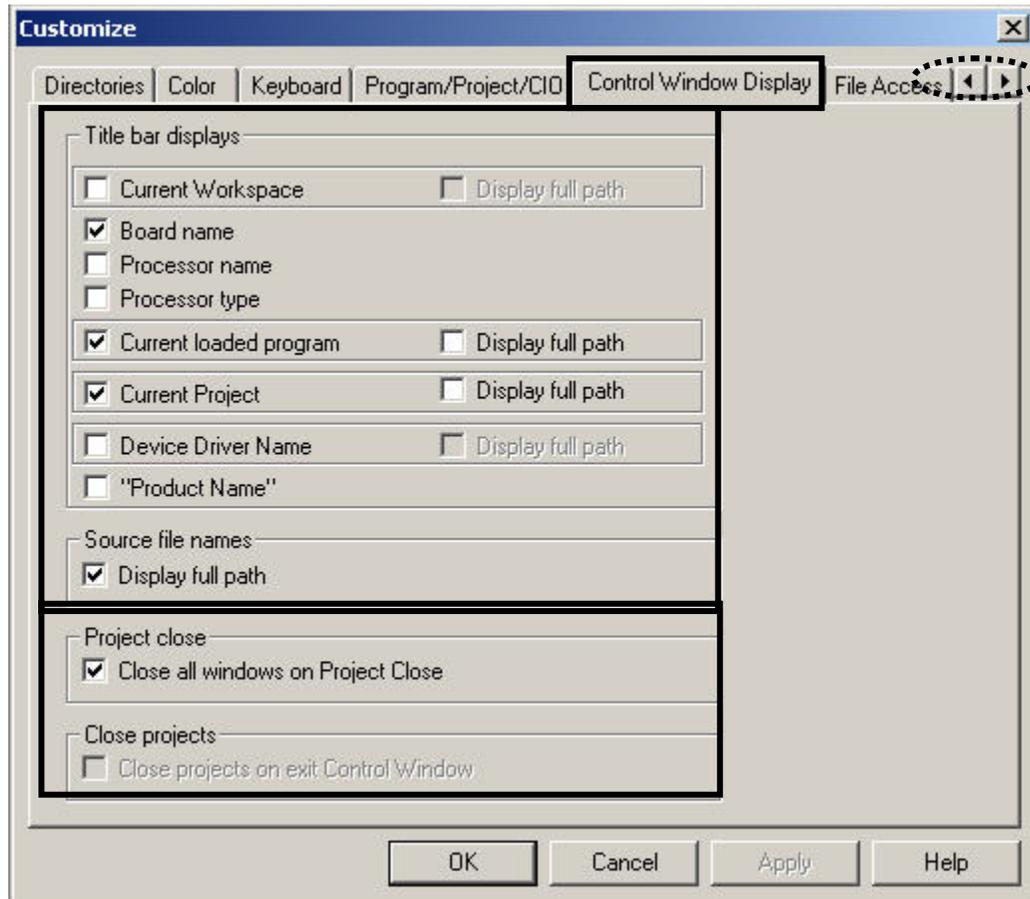
The *Load Program After Build* option automatically loads the program (.out file) created when you build a project. If you disabled this automatic feature, you would have to manually load the program via the **File→Load Program** menu.

Note: You might even think of IDE as standing for Integrated Debugger Editor, since those are the two basic modes of the tool

11. CCS Title Bar Properties

CCS allows you to choose what information you want displayed on its titlebar.

Note: To reach this tab of the “Customize” dialog box, you may have to scroll to the right using the arrows in the upper right corner of the dialog.



- We have chosen the “Board Name”, “Current Project”, and “Currently loaded program”.
 - The first item allow you to quickly confirm the chosen target (simulator, DSK, etc.).
 - The other two let us quickly determine which project is active and what program we have loaded. Notice how these correlate to the two parts of CCS: Edit and Debug.
- For our convenience we have also enabled *Display full path* and *Close all windows on Project Close*.

Now you're done with the Workstation Setup, please continue with the Lab2b exercise ...

Lab 2b: Using CCS with the Debug Configuration

The dot-product equation (sum-of-products) provides a simple, but important, function to familiarize us with compiling and debugging C code within Code Composer Studio.

Files in Project

LAB2.PJT	You create this!
LAB.TCF	You create this, too.
MAIN.C	Provided for you.

main.c – Source Code

```

/* MAIN.C */

/* Prototype */
int dotp(short *m, short *n, int count);

/* Include Files */
#include "data.h"

/* Definitions */
#define COUNT 256

/* Declarations */
short a[COUNT] = {A_ARRAY};
short x[COUNT] = {X_ARRAY};
int y = 0;

/* Code */
main()
{
    y = dotp(a, x, COUNT);
}

int dotp(short *m, short *n, int count)
{
    int i;
    int sum = 0;

    for (i=0; i < count; i++)
    {
        sum = sum + m[i] * n[i];
    }
    return(sum);
}

```

data.h

```

#define A_ARRAY 1,2,3,...,254,255,256
#define X_ARRAY 256,255,254,...,3,2,1

```

Create project file

1. Start CCS, if it's not already open.
2. Create a new project called **lab2.pjt** in the **c:\op6000\labs\lab2** folder.

Project → **New...**

You will encounter the *Project Creation* dialog. Fill in the *Project Name* and *Location* as shown below. Be careful that you specify the path correctly. A common mistake is to add one too many \lab2\ folders to the Location entry.

a) Type in lab name

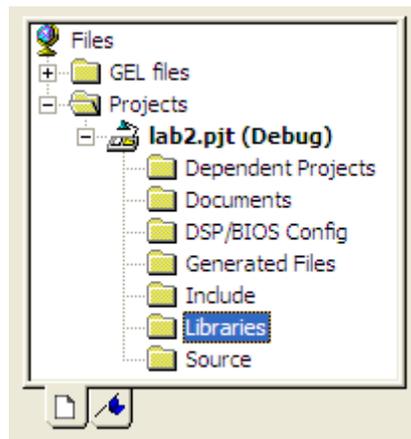
b) Click [...] and chose folder

c) Verify target processor

64, 64+

Choose TMS320C64XX if you are using the C64x or C64x+ simulator.

3. View the newly created project in CCS by examining the *Project View* window.

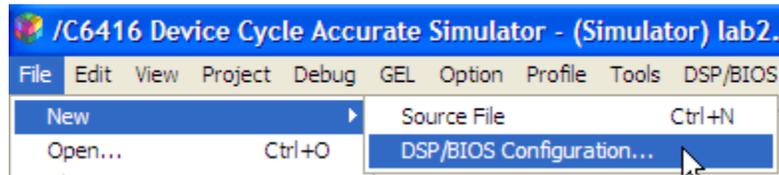


You may want to expand the **lab2.pjt** icon to observe the project contents. (Of course, it should be empty at this time.)

Create a TCF file

As was mentioned during the discussion, configuration database files (*.TCF) control a range of CCS capabilities. They are created by the *Config Tool*. In this lab, the TCF file will be used to automatically create the reset vector and perform memory management. More features of TCF files will be explored in later chapters.

4. Create a new TCF file (*DSP/BIOS Configuration...*) as shown below:



5. CCS asks you to select a template configuration file.
TI has configured a number of TCF template files to minimize your effort in customizing a configuration for your target platform. Pick one of these four TCF template files:

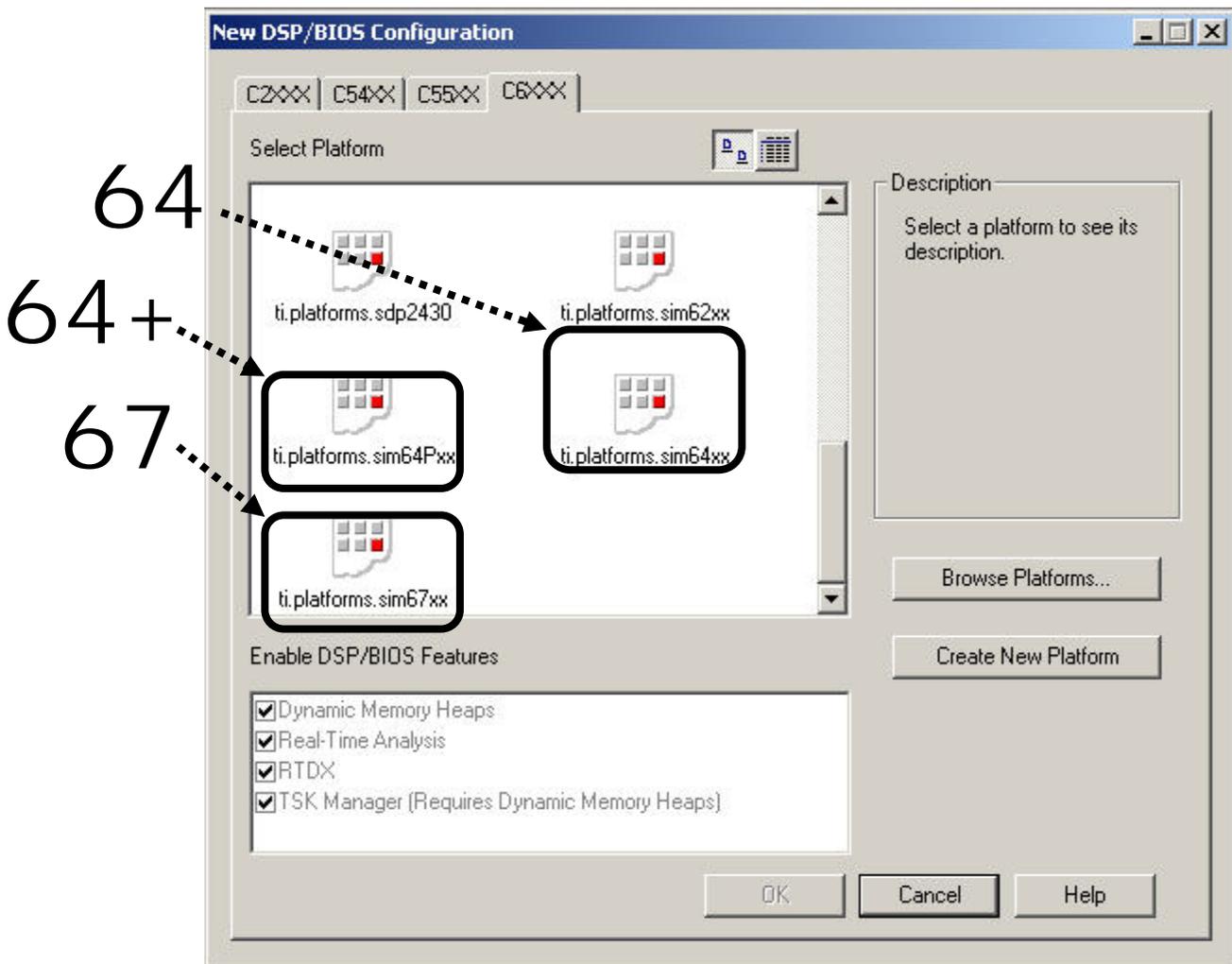
ti.platforms.sim64xx

ti.platforms.sim64Pxx

ti.platforms.sim67xx

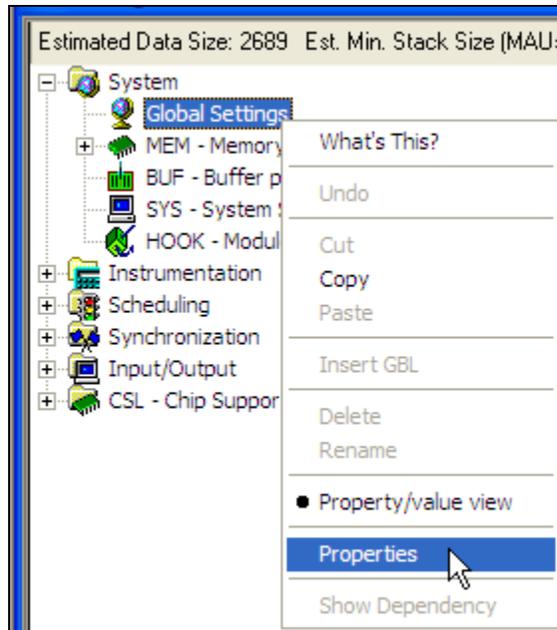
ti.platforms.padk6727

depending upon which platform you are using.



6. After selecting the template file click on “OK”.
7. Since we used a generic template, let’s verify the *Global Settings* for our TCF file.

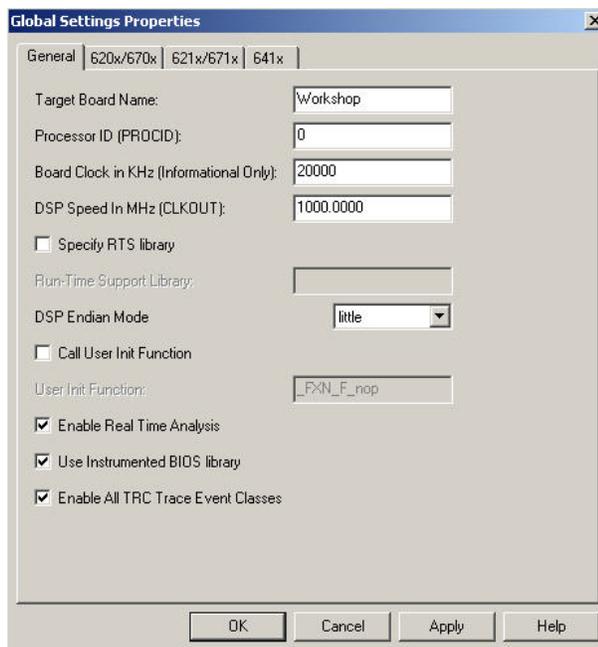
Open the *Global Settings* properties per the figure below



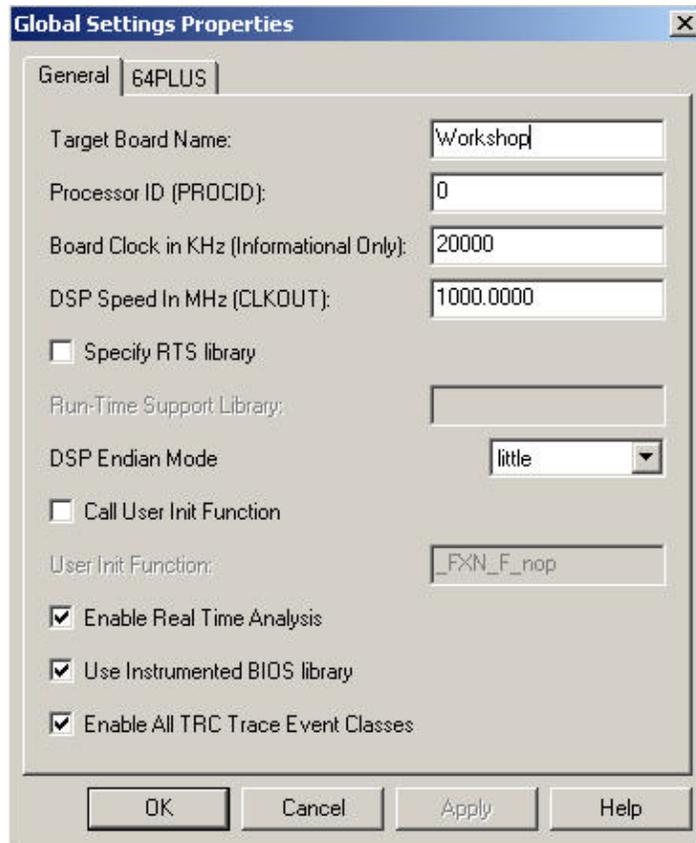
8. Edit the *Global Settings*. You'll need to change the following:

- *Target Board Name:* **Workshop** (or whatever you want to call it)
- *DSP Speed in MHz:* **300 or 1000** (67xx = 300; 64x = 1000; 64x+ = 1000)
- *DSP Endian Mode:* **little** (make sure to keep little endian)

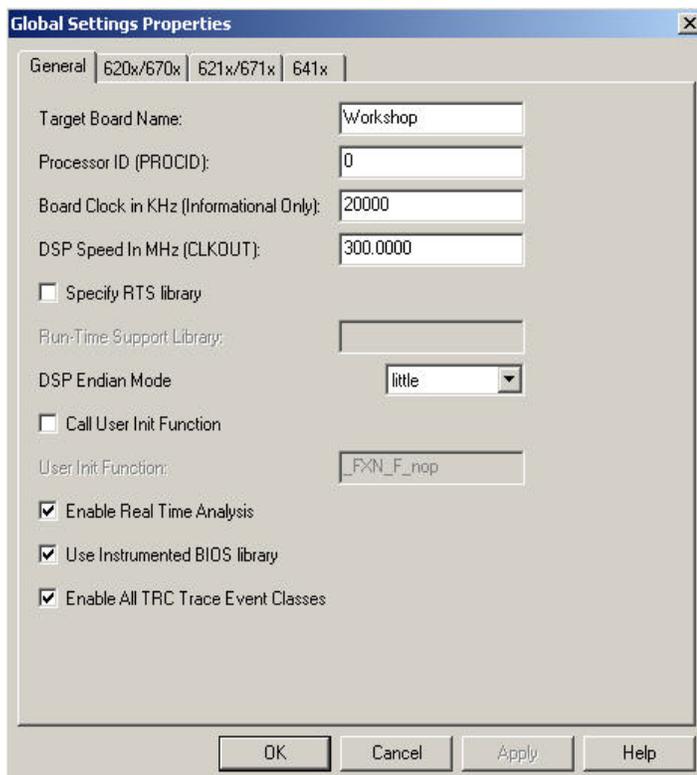
64



64+

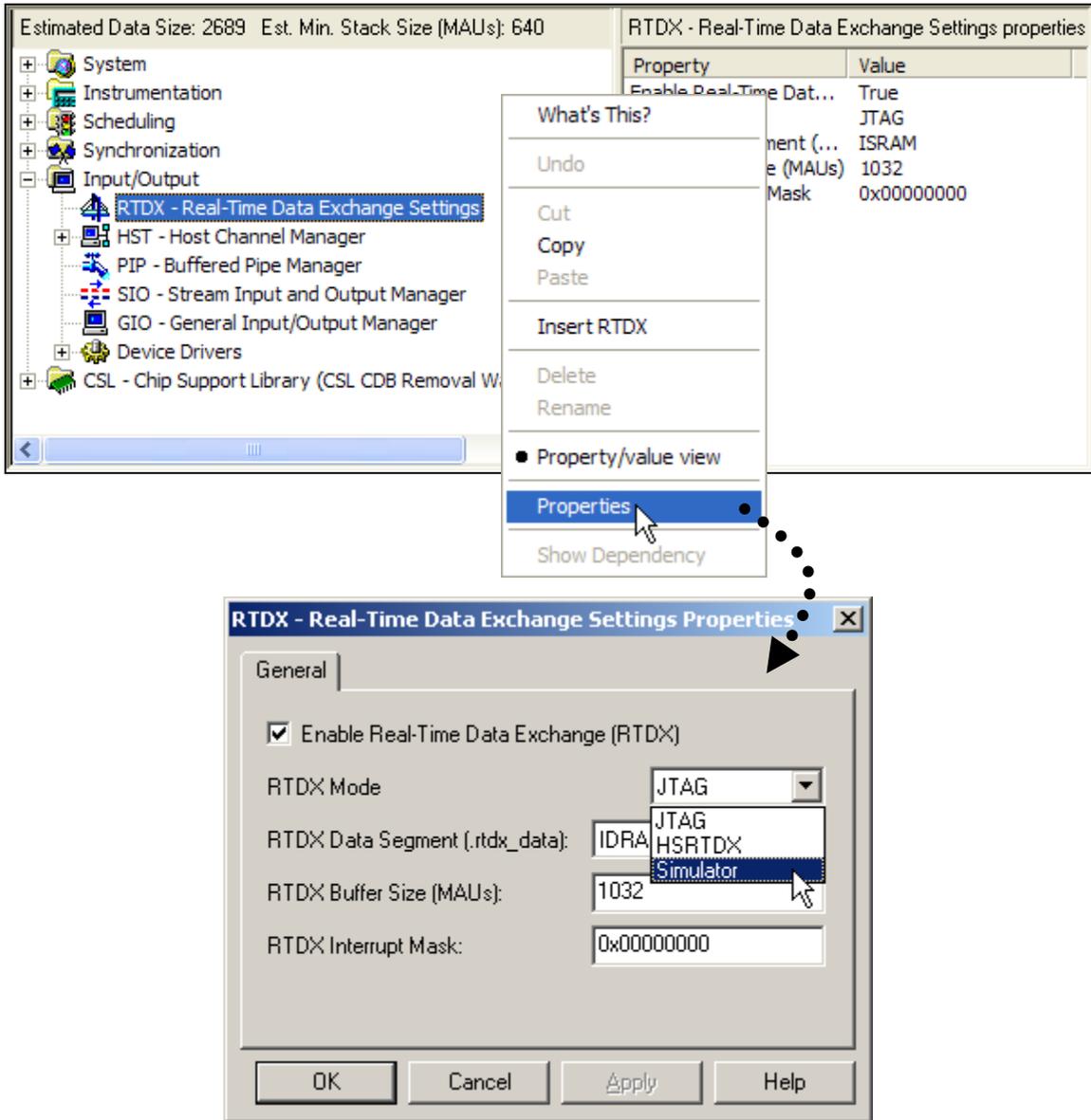


67
&
672x



- The final two TCF modifications involve changing the RTDX (real-time data transfer) mode and configuring dynamic memory. While we won't be using RTDX, this step prevents an error from appearing each time a program is loaded.

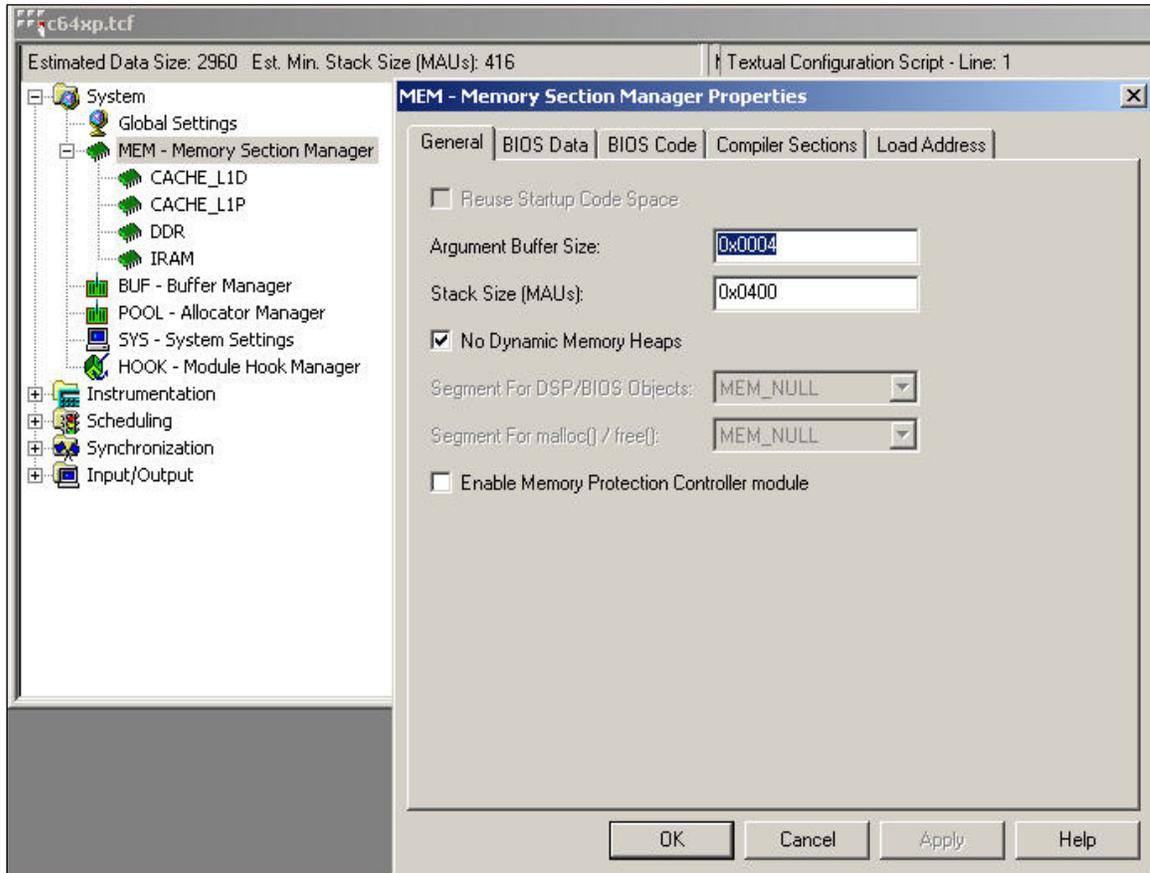
Open the RTDX properties and select "Simulator" as the RTDX Mode:



If you were to port this project to run on a DSK (or other hardware target), then you change this setting to one of the other two settings:

- JTAG is the hardware emulation protocol used in most TI compatible emulators.
- HSRTDX stands for High-Speed JTAG and is only accessible when using the XDS560 emulator.

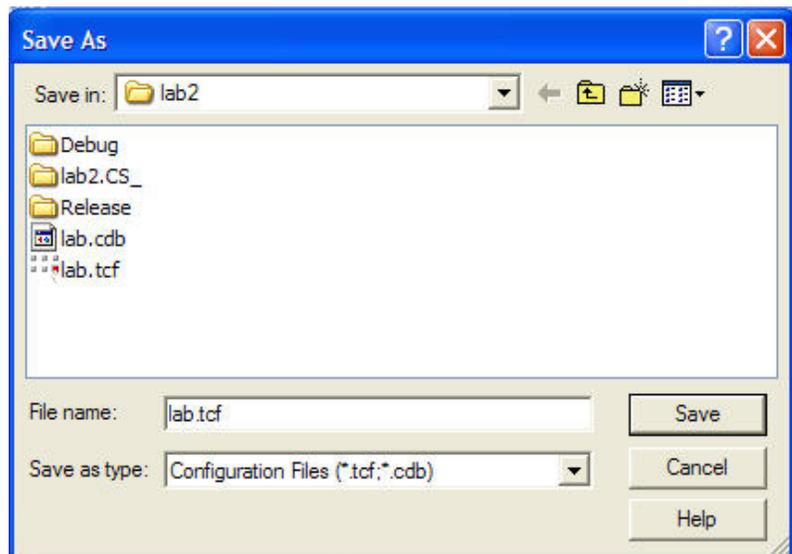
Since we will not be using dynamically allocated memory in any of our lab exercises the next step is required to reduce the memory footprint of our programs. Right click on the Memory Section Manager and select Properties. Then make certain that “No Dynamic Memory Heaps” is checked on the General tab.



10. Save your configuration file as **lab.tcf** to the **c:\op6000\labs\lab2** directory.

Note: The “Save as type:” must be **Configuration Files (*.tcf, *.cdb)**

File → Save As...



Adding files to the project

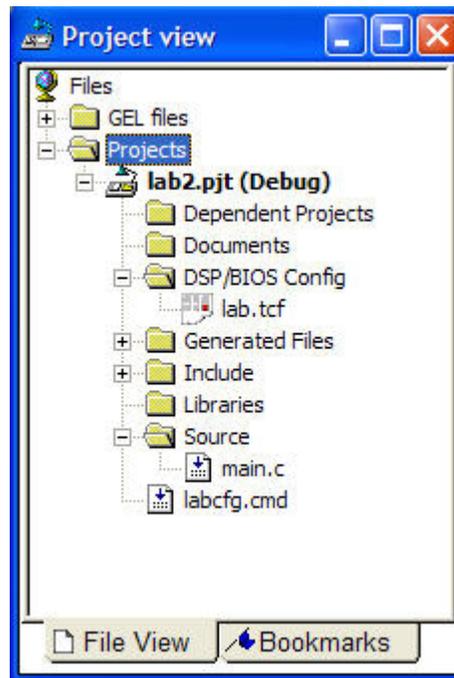
You can add files to a project in one of three ways

- Select the menu **Project**→**A**dd files to Project...
- Right-click the project icon in the Project Explorer window and select *Add files...*
- Drag and drop files from Windows Explorer onto the project icon

11. Using one of these methods, add the following files from `c:\op6000\labs\lab2` to your project:

MAIN . C
LAB . TCF
LABCFG . CMD

When these files have been added, the project should look like:



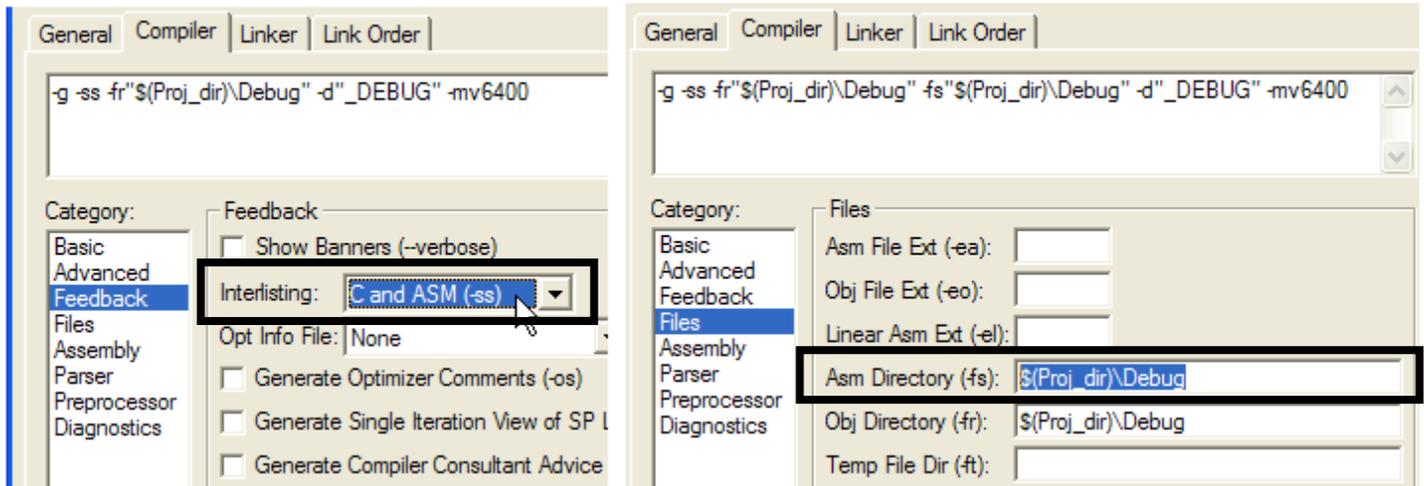
Modify Build Options

The default *Debug* build configuration added to your new project is a good starting point. We recommend you add three items to this configuration.

12. First, add the `-ss` and `-fs` compiler options.

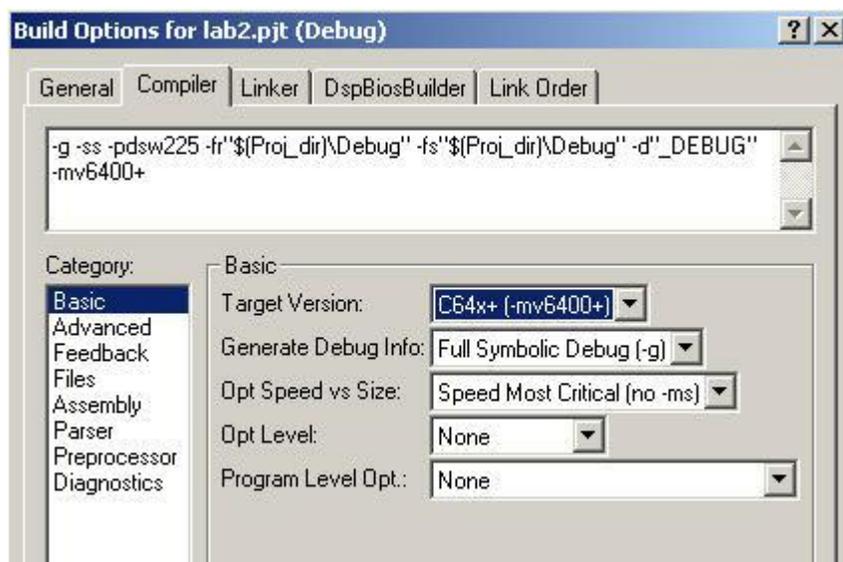
Open the project's build options: **Project**→**Build Options**

- Select the `-ss` option on the *Feedback* page
- Enter the **directory to store asm files** on the *Files* page



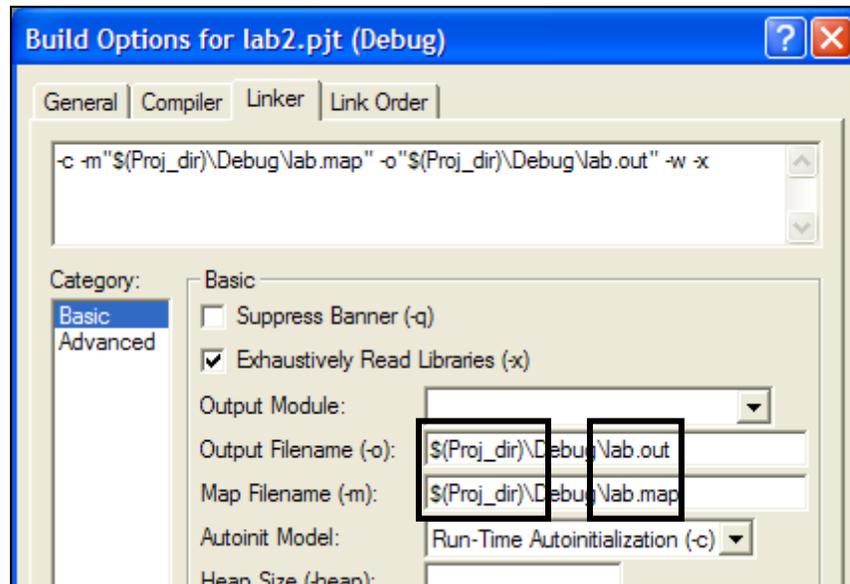
Note, the above screen captures are for a C64x system. The only thing you would see in a different system is that the `-mv` option should refer to the selected processor.

IMPORTANT- select the Basic Category on the Compiler Options tab and verify that the **Target Version** is set correctly for the processor that you chose when setting up CCS. The **Target Version** shown below is for the C64+ but yours may be different.



Modify the linker options.

13. Select the **Linker** tab of the *Build Options...* dialog box



14. Change the name of the **Output Filename (-o)** to: `$(Proj_dir)\debug\lab.out`

By default, when you create a new project, CCS names program after the project name. In most cases this works great. In our workshop, though, we wanted to use a generic output filename so that we won't have to edit the linker options in each lab.

We encourage you to use a relative path: `$(Proj_dir)\debug*.out` which tells CCS to write the output files to a folder called "debug" which is a subdirectory of the projects folder.

15. Add the name of the **Map Filename (-m)** to: `$(Proj_dir)\debug\lab.map`

- A map file is a report generated by the linker to describe its activity.
- If you do not specify a map filename, it will not be generated.
- By default, CCS does not create a map file. By specifying a map filename, CCS is notified to create the map file.

16. After the Linker options resembles the graphic above, select OK to save the values and close the dialog.

17. For good measure, save the changes you have made to the project (.pjt) file.

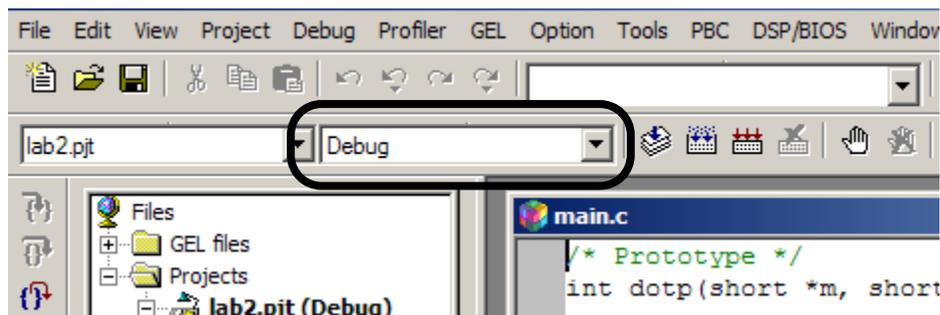
Project → Save

It's always good practice to save the project file once in a while.

Building the program (.OUT file)

Now that all the files have been added to our project, it's time to verify the build options and create the executable output program (that is, the .OUT file)

- For easy debugging, use the *Debug* configuration; this should be the default. Verify by checking that Debug is in the Project Configurations drop-down box.



- Build the program.

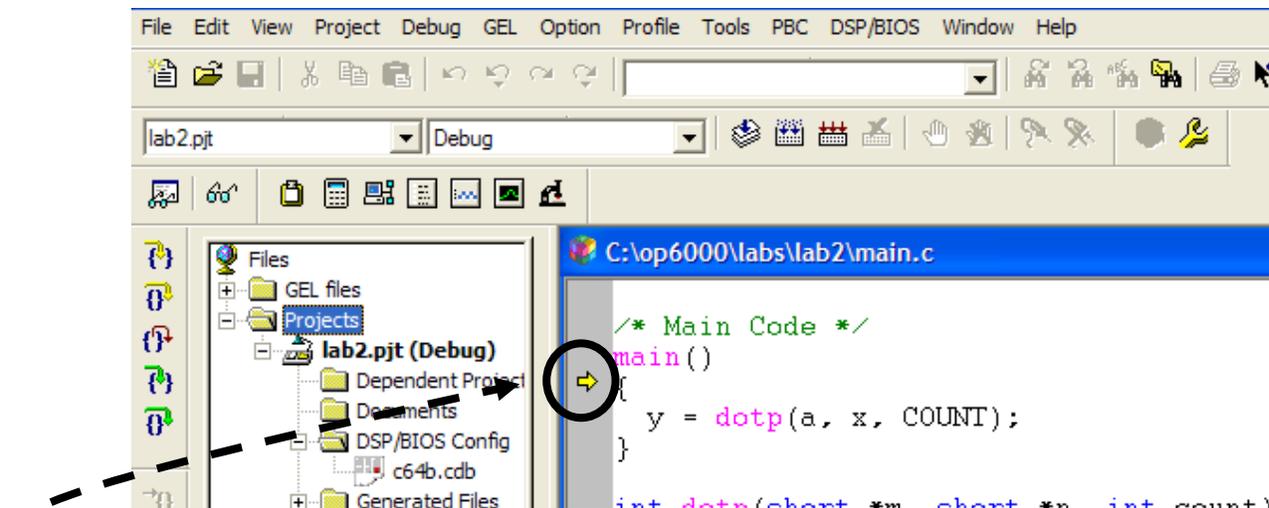
Use the **REBUILD ALL** toolbar icon:



or the **Project → Rebuild All** menu selection

Note the build progress information in the *Build Output* window that appears on the lower part of the CCS window. The build should complete with “0 Errors, 0 Warnings”.

- Once the program builds without errors the **lab.out** program should be loaded automatically. Since you enabled the *Program Load after Build* option (step 10, pg. 2-9), CCS should automatically download the program to the simulator when the build is complete.



The yellow arrow indicates the position of the program counter. Once the program is loaded, it should be pointed to the beginning of main(). Setting the *Perform Go Main Automatically* option (step 9, pg 2-8) causes CCS to run to main after being loaded. If you don't see this

arrow, then either the program wasn't loaded properly, or the *Go Main* wasn't performed. Of course, you can always run to main manually: **Debug→Go Main**

Note: Remember, the *Go Main* function tells CCS to run until it reaches the symbol `main`. If you invoke **Debug→Go Main** after the program counter has passed the `main`, the program will not be able to stop at `main`.

Note: If, for some reason, you must quit Code Composer before completing this lab you must remember to open the **lab2.pjt** project and load (**File→Load Program...**) the **lab.out** executable file.

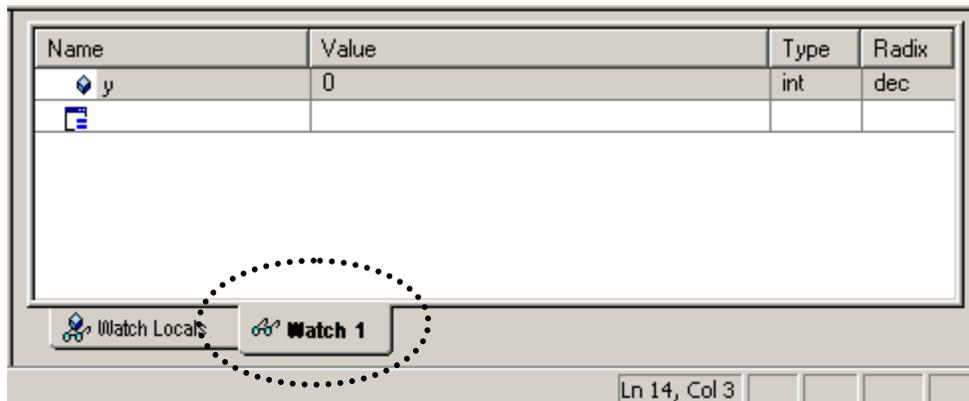
Debug Code

Now that the program has been written, compiled, and loaded, it's time to determine its logical correctness.

Watch Variables

21. Add `y` to the *Watch* window.

Select `y` in the `main.c` window
Right-click, and choose **Add To Watch Window**



After adding a variable, the *Watch* window automatically opens and `y` is added to it. Alternatively, you could have opened the watch window, selected `y` and dragged it onto the *Watch 1* window.

22. Click on the *Watch Locals* tab.

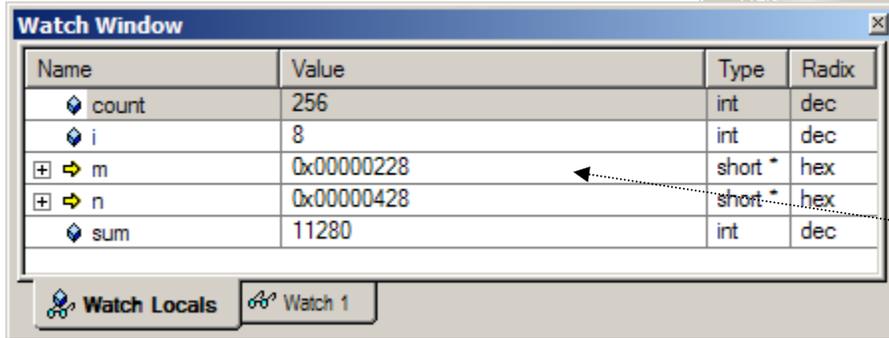
You'll notice this is empty since `main()` doesn't contain any local variables.

23. Single-step the debugger until you reach the dotp() function; it contains local variables.

Use toolbar  -or- **Debug** menu



In 2 or 3 single-steps, you'll notice that *Watch Locals* is updated:



Note

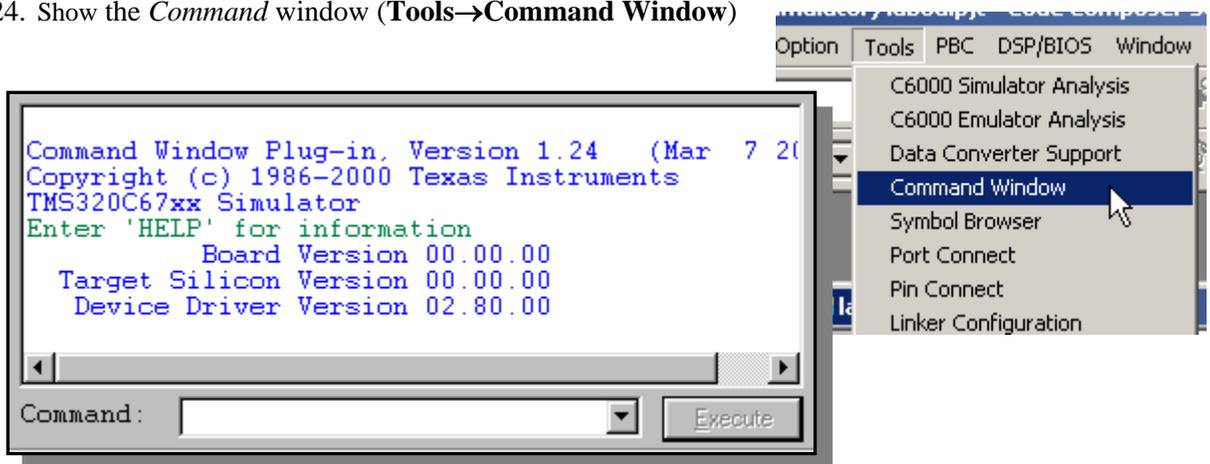
Your address values may differ from those shown.

Note: If a Watch window (other than the Watch Locals window) shows an error “unknown identifier” for a variable, don’t worry, it’s probably due to the variable’s scope. Local variables do not exist (and don’t have a value) until their function is called. Code Composer will add them to the watch window, but will indicate they aren’t valid until the appropriate function is reached.

Using the Command Window

Some of you may find it easier to use a keyboard than the mouse. To that end, CCS includes a *Command*-line window. Long time users of TI DSP tools will find these commands familiar.

24. Show the *Command* window (**Tools**→**Command Window**)



This opens up the *Command* window and docks it at the bottom of the CCS window. You may wish to resize or rearrange the windows, for your convenience. In fact, it may help to **right-click** inside the *Build* window and select **Hide** to give you more space.

25. To try out the *Command* window, let's add **a** and **x** to the *Watch* window. The *wa* (watch add) command adds a variable to the watch window. Type the following into the *Command* textbox and hit the *Execute* button (or enter key):

```
wa a
```

26. The *Command* window also represents one of the three automation methods provided within CCS. As a demonstration, we created a debugger command/log file called **watch.txt**. To invoke it, enter:

```
take watch.txt
```

You should see **a** and **x** added to the *Watch* window. Like batch files or macros, this works especially well if you want to repeat a given sequence at a later time ... or over and over.

Note: If you receive an error, "Cannot open file: "watch.txt", it's just that CCS cannot find the file. CCS looks in the path of the last file opened with **File → Open**, rather than the local path of the active project.

There are two solutions:

1. Use the full path name: **c:\op6000\labs\lab2\watch.txt**
2. Open a file from the current project's path. For example, use **File → Open** to open **main.c**. Since this file is already open, nothing really happens except making CCS use the proper default for our WATCH.TXT.

Go ahead and try **take watch.txt** again.

27. Since both **a** and **x** were in our *take* file, and we had already added **a** to the *Watch* window, it shows up twice. Let's remove one of them.

```
Select a in the Watch window and hit the delete key
```

28. Click back over to the *Watch Locals* view.
29. In the *Command* window, try the **step** command.

```
step <enter>
```

30. This should single-step the processor.

31. Try stepping multiple times.

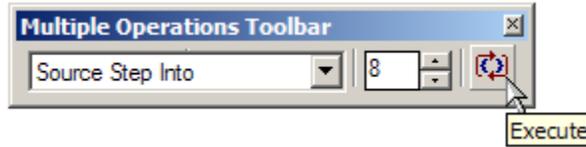
```
step 15 <enter>
```

This performs 15 single-steps. You should be able to see the *Watch Locals* updated at each step (though it goes quickly).

Multiple Operations Toolbar

CCS has a feature that provides multi-operations from a toolbar. Let's try to single-step our source code another eight times using this feature.

32. Verify the Multiple Operations toolbar is visible. It should look like:



If you cannot find it, it can be opened from the View menu:

View → Debug Toolbars → Multiple Operations

33. Set the Multiple Operations values as shown in the preceding step and execute.

```
Source Step Into
8
Execute
```

You should see it executing multiple single-steps, just as in step 31.

Run and Halt

34. Once you've had your fill of single-stepping, try running the processor:

Use toolbar  -or- **Debug:**
-or- **F5**

Step Into	F8
Step Over	F10
Step Out	Shift F7
Run	F5
Halt	Shift F5
Animate	F12

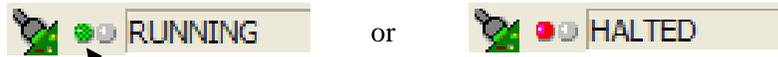
35. What happens when the processor reaches the end of the **main()** function.

One of two things can happen upon exiting main().

- CCS stops the debugger.
- The processor keeps running.

Which option occurs for you depends upon a couple factors: how you build your program, how your code was written. Most embedded systems, though, are designed to never *end*. Through the use of a *scheduler kernel* or something like a *while* loop, they loop continuously while waiting for events (i.e. interrupts) to service.

36. By checking the status bar in the lower left corner, you can determine if the processor is still running. You should see one of these messages:



Running with a *green* light means the CPU is running. **Halted** with a *red* light means it is executing.

37. If the processor is running, halt it:

Use Toolbar  -or- **Debug** → **Halt** -or- shift-F5

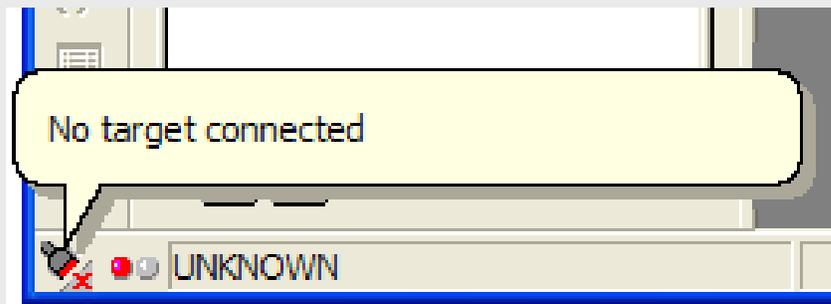
You can minimize the Disassembly Window if it pops up.

Sidebar: Connected / Disconnected from the DSP Target

You might have noticed the *plug connected* icon next to the running/halted status. This means that CCS is connected (i.e. talking to) the target.



When using the simulator, you should not have an issue with the target becoming disconnected. With a hardware target, such as a DSP Starter Kit (DSK) or your own hardware, this may not be the case. For example, what if the target was powered down while CCS was still running? Rather than *crash*, CCS signals you via the connected/disconnected icon.



Setting Software Breakpoints

You can force the program to halt execution by using a software breakpoint. Let's set one at the end of `main()` to guarantee that code execution stops.

38. Set a breakpoint at the end of `main()` to stop the processor automatically.

Click the cursor on the highlighted line below

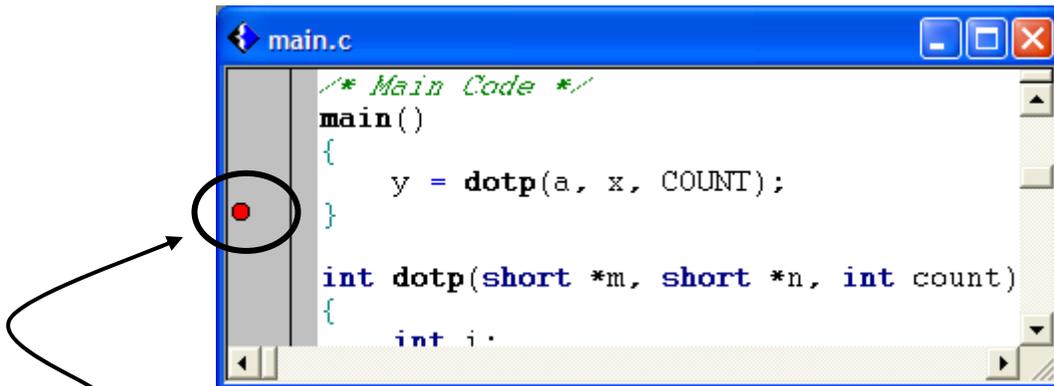
Click the **Add Breakpoint** toolbar button: 

```

/* Main Code */
main()
{
    y = dotp(a, x, 40);
→ }
    
```

Note: When selecting *Toggle Break Pt*, the break point will be added to whatever line contains the cursor. This is why we recommend clicking on the line as described above.

The code window should now appear as shown:



Alternatively, to set or remove a breakpoint, double-click on the spot where the red dot appears in the left column of the code window.

39. Before we can run our code to try out our breakpoint, we must get our Program Counter (PC) back to the address `main`. (That is, get it back to someone in code before the breakpoint.)

Debug → Restart

Note: If the breakpoint (red dot) or program-counter (yellow arrow) icons disappear after running or reset, clicking in the source window should cause them to be redrawn.

40. Run the code. It should stop at your breakpoint.

 -or- **F5** -or- Debug → Run

What's the Dot-Product Result?

41. Check the value of `y` in the watch window and write it below.

`y` = _____

The answer should be: `0x002b2b00` or `2829056` decimal

Hopefully one of these values appears in the *Watch* window.

Hint: To change the format of a *Watch* window value, click on the *Radix* cell of the entry of interest. Then select the *Property Page*. Select the type you want from the *Default Radix* drop-down style box on the *Property Page*.

Since the default for `y` is *decimal*, try *hex*.

Benchmark Code Performance

We define analyzing how long it takes code to execute code as *benchmarking* performance. In our example, we're interested in benchmarking the dot-product function. There is more than one way of benchmarking code performance within CCS, we will start with the Clock method.

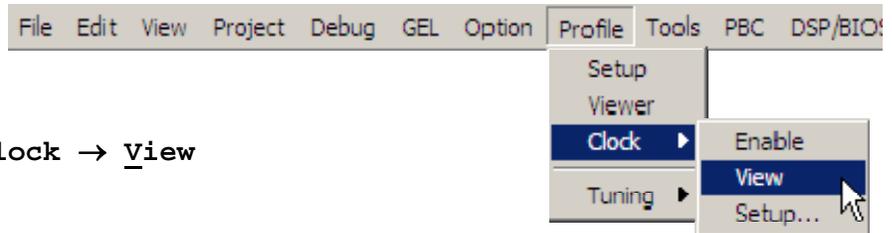
42. Before we begin, let's restart our program. Make sure the processor is halted, then:

`Debug` → `Restart`

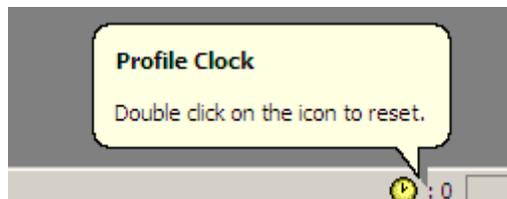
By the way, if you want to free up some extra room, you can close the *Command* window (right-click on it and select *close*.)

43. Open the cycle clock.

`Profiler` → `Clock` → `View`

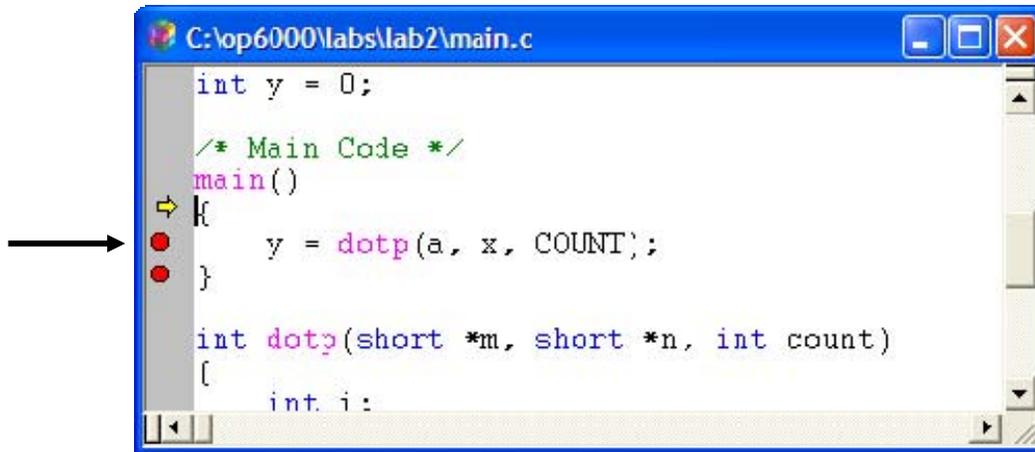


Look for the clock to appear in the lower right corner on the status bar. The balloon notice is also important, it reminds you how to zero out the Profile Clock.

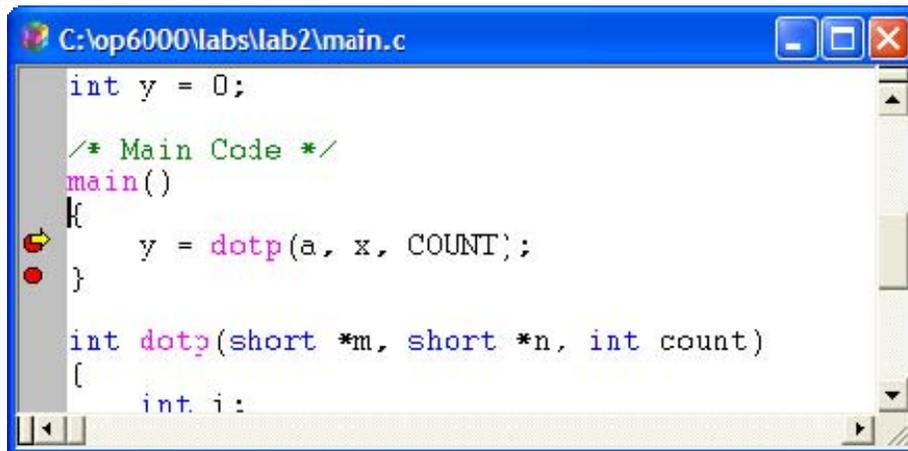


44. Before we use the clock, we need to set another breakpoint. We need two breakpoints, one before and another after the dotp() function. These will enable us to calculate the number of clock cycles it takes to execute the function.

Add the additional breakpoint indicated below.



45. First, let's run to the first breakpoint and clear the clock.



Notice the yellow arrow indicates that program execution stopped at the first breakpoint. Next, clear the profile Clock by double-clicking on it.



Once cleared, it should read zero:



46. Finally, run to the next breakpoint and record the clock value.

```

C:\op6000\labs\lab2\main.c
int y = 0;

/* Main Code */
main()
{
    y = dotp(a, x, COUNT);
}

int dotp(short *m, short *n, int count)
{
    int i:
    
```

⌚ : 8750 Ln 19, Col 1

How fast was your **dotp** function? _____ clock cycles

Was it also 8750 cycles? Since future revisions of the compiler could alter performance, or if you used different options, the results might be different.

47. Copy your result to the **Results Comparison Table** on page 2-42.

48. Halt the processor if it's still running.

Lab 2c: Benchmark the *Release* Build-Configuration

After you have verified your program works correctly, you most likely will want to improve its performance. Remember from the class discussion, the *Release* build configuration enables the optimizer.

Using the compiler's optimizer is the easiest way to speed-up your code. During the next couple days, the workshop will highlight many additional optimization techniques. In this lab, though, we use the compiler's optimizer to improve performance.

Activate and Modify the *Release* Configuration

49. Before we move on, let's be save and save our project.

Project → **Save**

50. Choose the Release build options (for optimization).

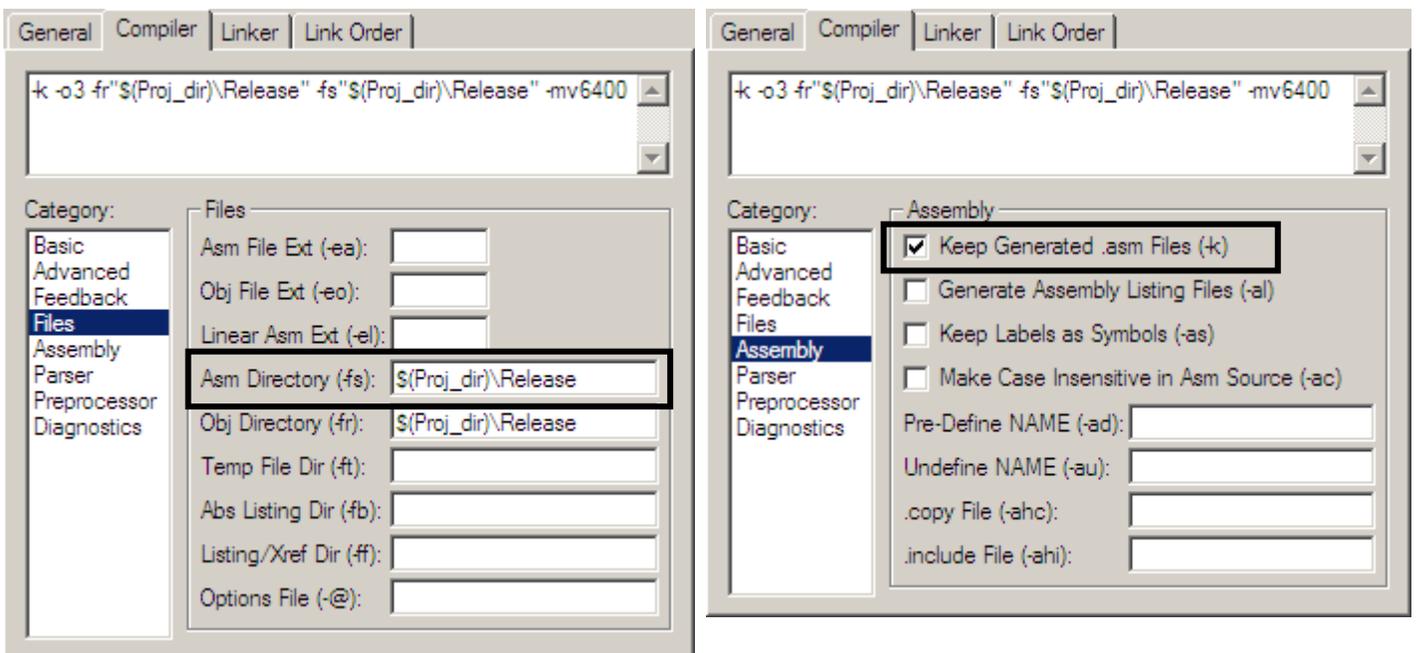
Project → **Configurations...**

Select the *Release* configuration and click the **Set Active** button, then close dialog

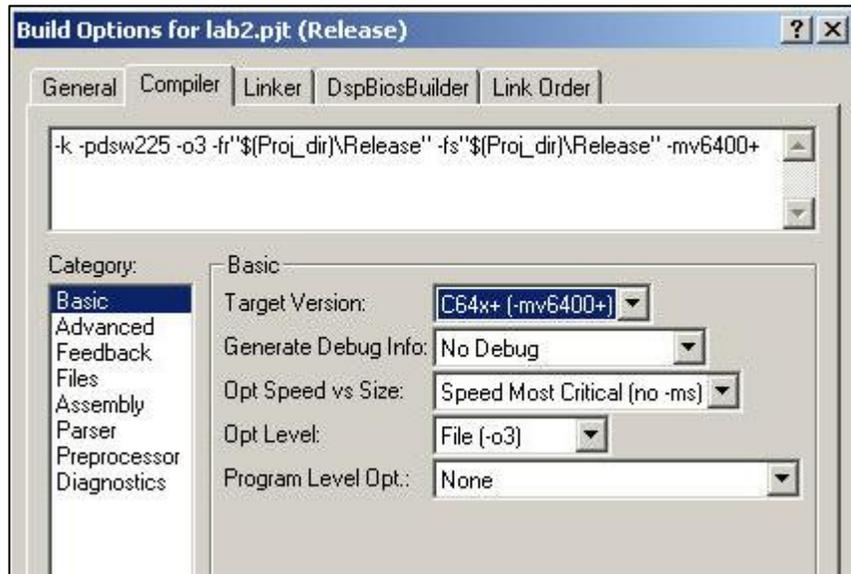
Hint: You can use the convenient build configurations drop-down box to make *Release* active. We discussed this feature in step 18 (pg 2-22).

51. Per the discussion before the lab, there are some recommended additions to the default *Release* configuration. Now that the Release build-configuration is active, you can open up the Build Configurations dialog and make the following changes.

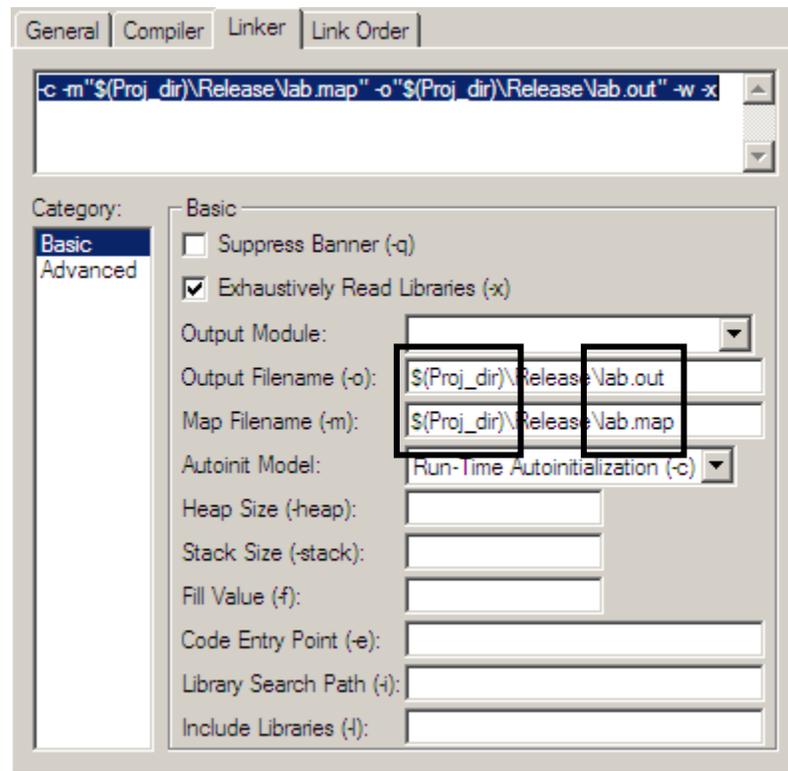
First, the Compiler:



Next, verify that the *Target Version* is set correctly for the processor that you selected:



Then the linker:

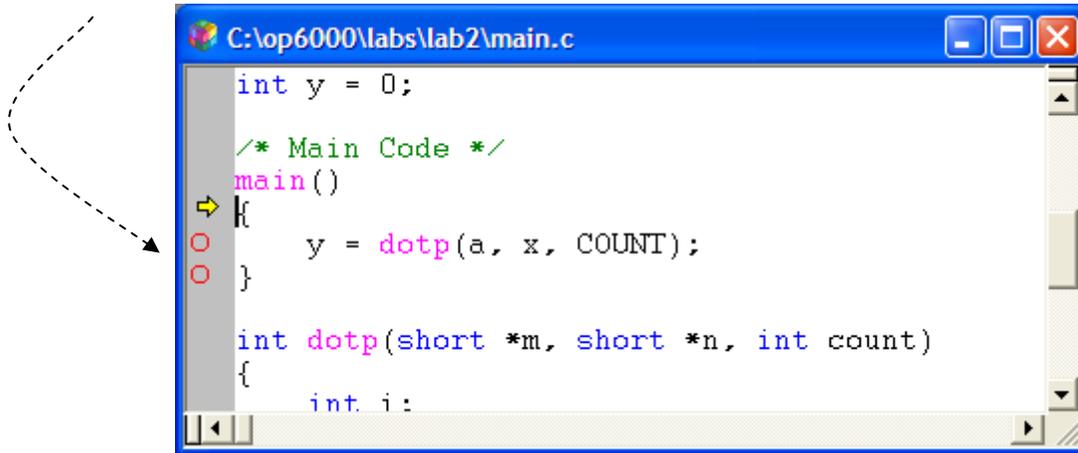


Note, the original relocatable path used in the linker works fine. For consistency, though, we chose to modify it to match the path used for the -fr and -fs options on the Compiler tab.

When finished with the changes, click OK to accept the modifications and close the dialog.

Breakpoints (again), Run and Benchmark Release

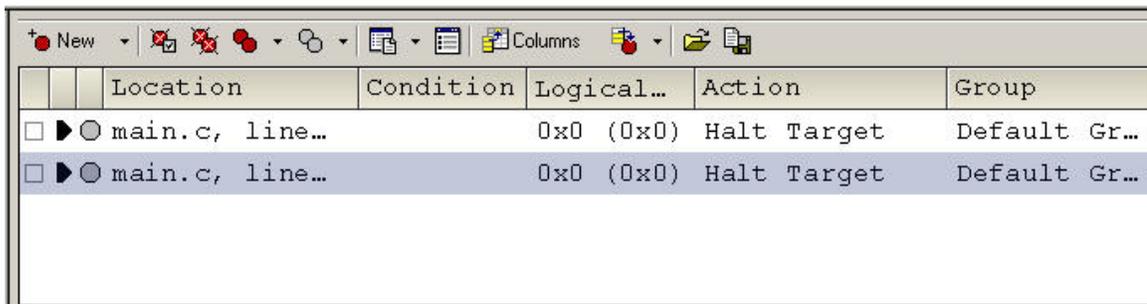
52. Rebuild and load the program (as in step 19-20).
53. Once loaded, though, note that the breakpoints you set have become disabled. You can see this from the hollow round circles where your breakpoints used to be.



Note: When loading a new program, it is desirable to disable (or remove) any breakpoints from the previous program. Rarely, do the needs of one program match another. If you prefer to disable this feature, it can be turned off via *Option*→*Customize*→*Program/Project/CIO*.

Or you can verify this from the Breakpoint dialog.

Debug → Breakpoints



If you remember from the earlier discussion, breakpoints can only be set at function-level boundaries when the optimizer is enabled – which it is in the Release build configuration.

You can go ahead and enable the second breakpoint in the dialog box, or by double-clicking on it in the source window margin.

To enable the first breakpoint, go ahead and set it as shown:

```

/* Main Code */
main()
{
    y = dotp(a, x, COUNT);
}

int dotp(short *m, short *n, int count)
{
    int i;
    int sum = 0;

    for (i=0; i < count; i++)
    {
        sum = sum + m[i] * n[i];
    }
}
    
```

The beginning and ending of main() work fine for this example. If we had other code in main, we may have needed to move our breakpoints to the beginning and ending of dotp().

54. Once the breakpoints are re-enabled, make sure the execution (yellow) arrow is at the start of main() and follow the same Clock profiling procedure used earlier (steps 42-48).

How fast was **dotp()** this time? _____ clock cycles

55. Copy this result to the **Results Comparison Table** on page 2-42.

Halt the processor if it's still running.

56. Does the code run faster using the compiler's optimizer? _____

57. Clear any breakpoints you set in the lab.

You can use **Debug → Breakpoints → Delete All** or use the toolbar button:

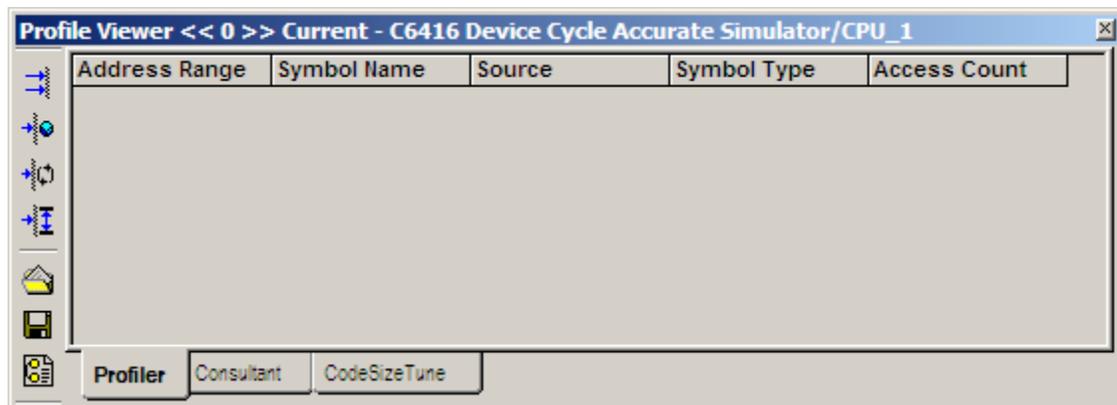
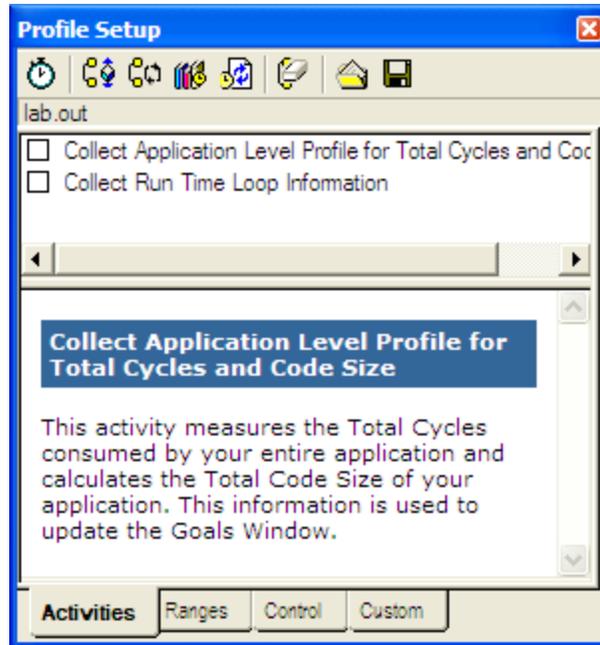


Lab 2d: Benchmarking with the Profiler

Overview

The Clock provides a simple, non-intrusive (as it sits in the status bar) method of benchmarking. When examining a large algorithm or project, though, this method is tedious. To this end, the CCS Profiler tool automates the benchmarking procedure and provides you with a table of results.

The profiler consists of two windows: *Profiler Setup* and *Profiler Viewer*.



These windows are part the entire suite of tuning tools (which include Cache-Tune and Code-Size Tune, among others). All the windows that make up the CCS interface (including all the tuning tools) are often called the *Tuning Dashboard*. In this lab, we examine only the Profiler part of the Dashboard. In future chapters, you will explore the other tuning components.

Setup the Profiler

58. **Build and load** your lab using the Debug build-configuration.

59. **Set a breakpoint** at the end of main().

60. **Open the Profile Setup window.**

Profile → Setup

61. **Enable function and loop profiling.** Set the Profile Setup window as shown.

This is a three step process:

1. Select the code we want to profile:
 - Functions
 - Loops
2. Select the items to profile on the custom tab.
3. Click "Enable Profiling" toolbar button.

62. **Open the Profile Viewer window.**

Profile → Viewer

Whether or not there is anything in this window, it will be updated when we run the program in the next step.

63. Run (F5) your program. When the program halts (on the breakpoint at the end of main), review the results in the Profile View window.

Note, you will have to scroll down to find the data we are interested in.

Address Range	Symbol Name	SLR	Symbol Type	Access Count	cycle.CPU:
0:0x38-0x3c8	c64cfg_s62-910-9	910-941:c64cfg_s	function	0	0
0:0xcd44-0xcde8	dotp	21-32:main.c	function	1	8741
0:0xcd00-0xcd44	main	16-19:main.c	function	1	8750
0:0xc8c0-0xc8dc	CACHE clean	441-484:csl_cach...loop		0	0

dotp

Notice there are two dotp symbols. Column 4 indicates one is a function, while the other is a loop inside that function. We are interested in the function.

Count

As expected, we only ran the dotp function once. Though, notice the loop ran 256 times (again, as expected).

Totals

We will examine the 8741 cycle count in the narrative below. Notice the **Inclusive/Exclusive** results. There is no difference for dotp. For main, though there is a big difference.

dotp() cycle count

So, is the 8741 cycle count what we expected? Actually, it is very close. Here are the cycle counts we got when profiling our ‘debug’ code using the Clock. Notice, there is a slight difference depending upon where we placed our breakpoints (in the calling function, or just inside the function itself).

Inner Function	Calling Function
<pre> main() { Y = dotp (); } dotp(); → { for loop(); → } </pre> <p>8741 cycles</p>	<pre> main() { → y = dotp (); → } dotp(); { for loop(); } </pre> <p>8750 cycles</p>

What do you think accounts for the difference?

What is the does *inclusive* and *exclusive* describe?

Address Range	Symbol Name	Access Count	cycle.CPU: Incl. Total	cycle.CPU: Excl. Total
0:0xfc38-0xfcc8	c64cfg.s62:910:9...	0	0	0
0:0xcd44-0xcde8	dotp	1	8741	8741
0:0xcd00-0xcd44	main	1	8750	9
0:0xc8c0-0xc8dc	CACHE clean	0	0	0

Profiler Consultant CodeSizeTune

Profiler Setup – Ranges

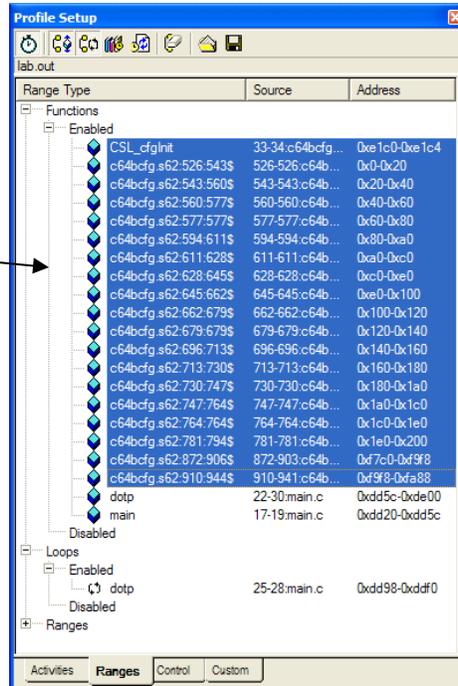
64. Reduce profiler ranges.

Since we are only looking at three of the functions/loops in the Profiler Viewer window, let's reduce the "ranges" we wish to profile. To do this:

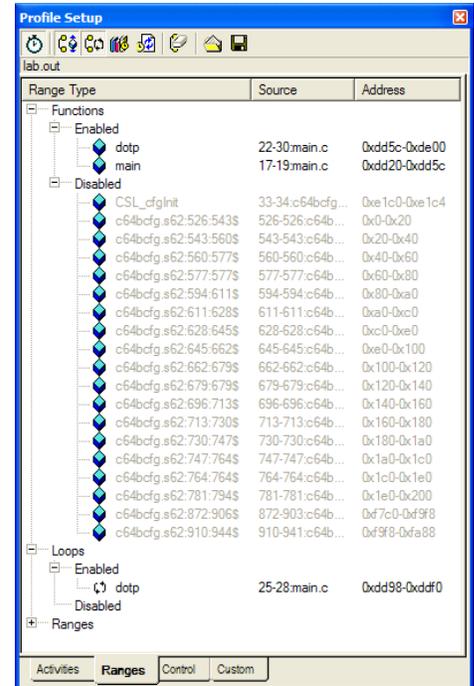
- Select the *Ranges* tab in the Profiler Setup window
- Select all the ranges except **main** and **dotp**
- Disable the selected ranges by clicking the space bar (alternatively, you disable/enable by dragging and dropping, or via a right-click menu)

Note:

The functions we are not interested in come from other code modules. In this case, they are functions created or called by the TCF file.



Profiler Setup (items selected)

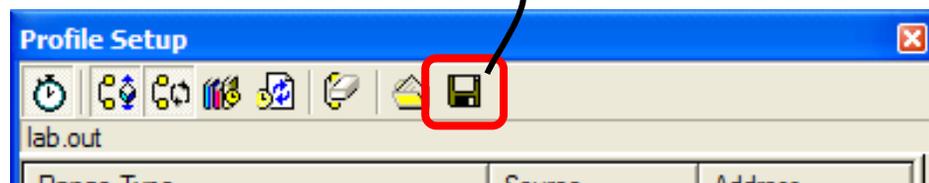


Profiler Setup (items disabled)

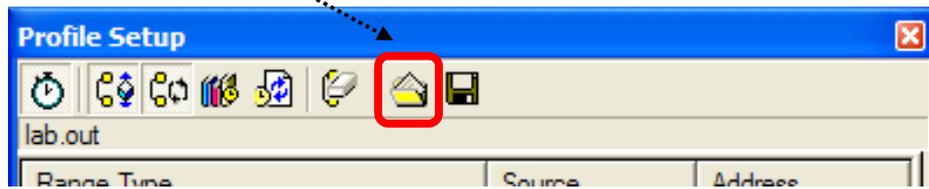
65. Save the profiler-configuration.

So that you do not have to continue disabling the profiler for the functions we are not interested in, you can save the profiler-configuration that you just created. On the Profiler Setup window toolbar, select the *Save Profiler Configuration* button (as shown below).

Save settings as: **Profiler.ini**



Notice also, the *Load Profiler Configuration* button just to the left of the save button.



Note: Now that you have this profiler configuration created, you will be able to use it for profiling code in many of the upcoming labs.

66. Profile Release build-configuration.

Let's repeat the profiler again for the Release configuration.

- Make *Release* active ←
- Build (and load) program
- Set breakpoint at the end of main()
- Run
- Compare results with previous: _____

Note

To change the build configuration to *Release*, you will have to disable the profiler. In fact, you may even have to close the Profiler Setup window.

If you have to close the Profiler Setup window, don't forget to use the *Load Profiler Configuration* upon re-opening it.

67. Here are our results:

You might notice that the dotp function is gone. Why is this? _____

Address Range	Symbol Name	Symbol Type	Access Count	cycle.CPU: Incl. Total	cycle.CPU: Excl. Total
0:0xc9c0-0xcb00	main	function	1	76	76
0:0xca20-0xca58	main	loop	30	60	60

Again, your results may differ slightly from those above. If your cycle counts are drastically different, though, please consult with your instructor.

68. Close the project and any windows associated with it.

Project → Close

Lab 2 Summary

Let's summarize our profile results.

Results Comparison Table

Lab Step	Lab2 File	Build Configuration	Cycles
Step 46 (pg 2-31)	main.c	Debug	
Step 54 (pg 2-35)	main.c	Release	

End of Lab2

We highly recommend trying the first couple optional exercises, if time is still available.



Before going on, though, please let your instructor know when you have reached this point.

Optional Exercises

Optional exercises are additional labs that you can use to extend your knowledge of the 'C6000 and CCS toolset. If you finish the main exercise early, you are welcome to begin working on these labs.

Optional 2e – Customize CCS

Add Custom Keyboard Assignment

While most CCS commands are available via hotkeys, you may find yourself wanting to modify CCS to better suit your personal needs. For example, to restart the processor, the default hotkey(s) are:

`D`ebug → Restart

CCS lets you remap many of these functions. Let's try remapping *Restart*.

1. Start CCS if it isn't already open.
2. Open the CCS customization dialog.

`O`ption → Customize...

3. Choose the *Keyboard* tab in the customize dialog box.
4. Scroll down in the *Commands* list box to find **Debug → Restart** and select it.

In CCS 3.3, you may notice a three-key shortcut has already assigned. Let's go ahead and assign another, easier to use key.

5. Click the **A**dd button.
6. When asked to, "*Press new shortcut key*", press:

`F`4

We already checked and this one isn't assigned within CCS, by default.

7. Click OK twice to close the dialog boxes.
8. From now on, to **Restart** and **Run** the CPU, all you need to do is push **F4** then **F5**.

Customize your Workspace

You may not find the default workspace for CCS as convenient as you'd like. If that's the case, you can modify as needed.

9. Close CCS if it's open, and then open CCS.

This forces CCS back to its default states (i.e. no breakpoints, profiling, etc.).

10. Move the toolbars around as you'd like them to be.

For example, you may want to close the BIOS toolbar and then move the Watch toolbar upwards so that you free up another ½ inch of screen space.

11. If you want the Project and File *open* dialogs to default to a specific path, you need to open a project or file from that path.

12. Make sure you close any project or file from step 11.

13. Save the current workspace.

File → Workspace → Save Workspace As...

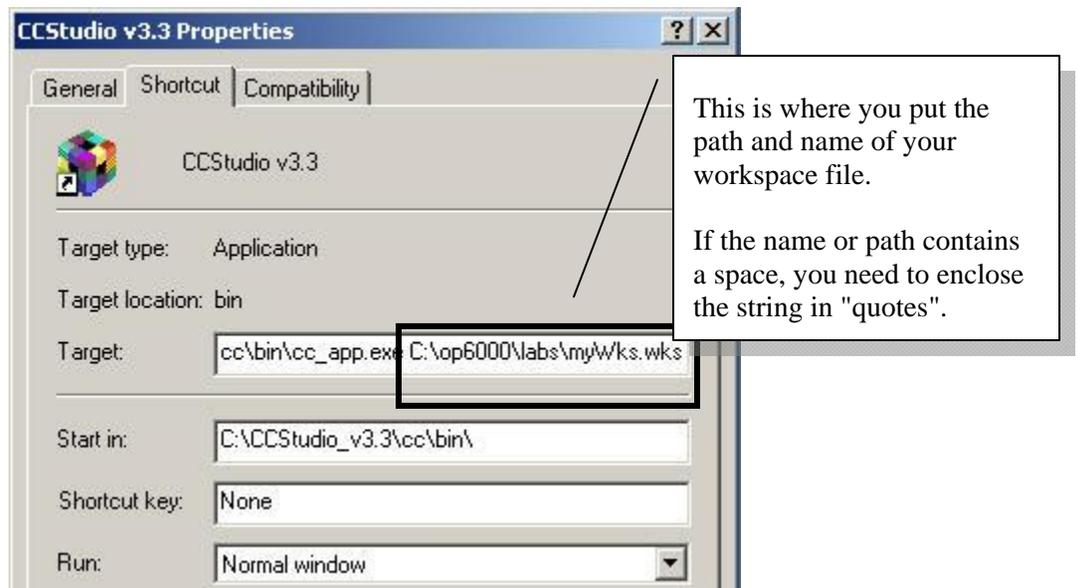
Save this file to a location you can remember. For example, you might want to save it to:

C:\op6000\labs

14. Close CCS.

15. Change the properties of the CCS desktop icon.

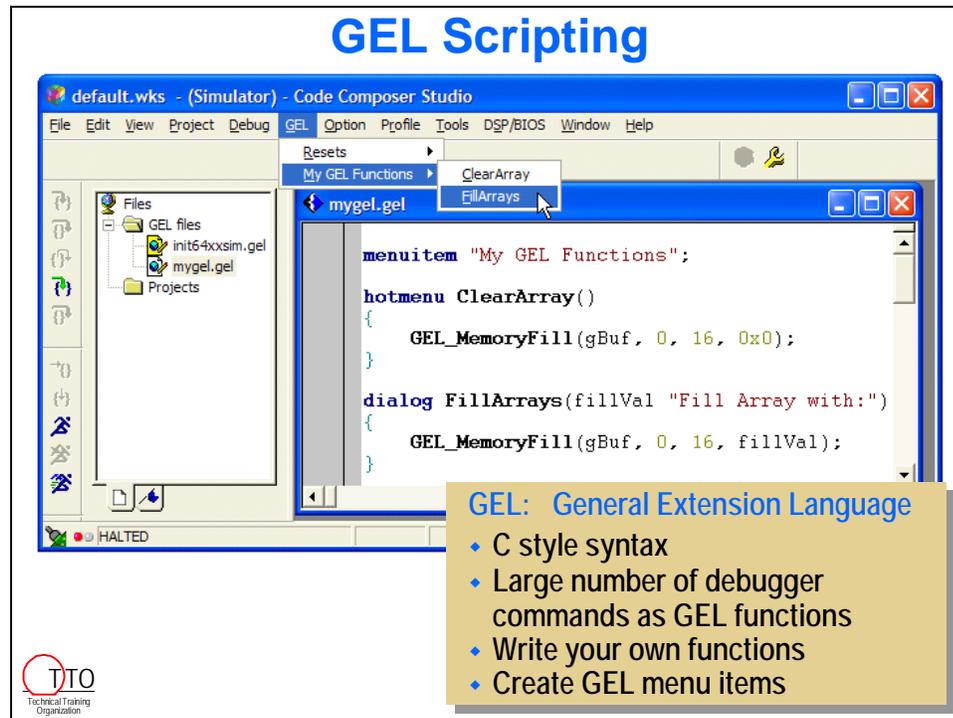
Right-click on the CCS desktop icon
Add your workspace to the *Target*, as shown below



16. Open up CCS and verify it worked.

Optional Lab 2f - Using GEL Scripts

GEL stands for General Extension Language, a fancy name for a scripting tool. You can use GEL scripts to automate processes as you see necessary. We'll be using a few of them in the lab in just a few minutes....



Using GEL Scripts

When debugging, you often need to fill memory with a known value prior to building and running some new code. Instead of constantly using the menu commands, let's create a GEL (General Extension Language) file that automates the process. GEL files can be used to execute a string of commands that the user specifies. They are quite handy.

1. **Start CCS and open project lab2.pjt, if it isn't already open.**
2. **Create a GEL file** (GEL files are just text files)

File → New → Source File

3. **Save the GEL file**

Save this file in the lab2 folder. Pick any name you want that ends in *.gel.

File → Save

We chose the name **mygel.gel**.

4. Create a new menu item

Create a new menu item (that will appear in CCS menu “GEL”) called “My GEL Functions”.

```
menuitem "My GEL Functions";
```

You can access all of the pre-defined GEL commands by accessing:

Help → Contents

Select the Index tab and type the word “GEL”.

5. Create a submenu item using hotmenu

The *menuitem* command that we used in the previous step will place the title “My GEL Functions” under the GEL menu in CCS. When you select this menu item, we want to be able to select two different operations. Submenu items are with the *hotmenu* command.

Enter the following into your GEL file: (Don’t forget the semicolon – as with C, it’s important!)

```
hotmenu HelloWorld ( )
{
    GEL_OpenWindow("Fred", 0, 5);
    GEL_TextOut("Hello World\n", "Fred", 2, 1, 1);
}
```

HelloWorld doesn’t really do anything, but it’s an example of how you can write from a GEL script to the user interface.

6. Create another submenu item to clear our arrays

Use the following commands to create a submenu item to clear the memory arrays. Again, you can look this up in the help index:

```
hotmenu ClearArrays()
{
    GEL_MemoryFill(a, 0, 128, 0x0);
    GEL_MemoryFill(x, 0, 128, 0x0);
}
```

The MemoryFill command requires the following info:

- Address
- Type of memory (data = 0)
- Length
- Memory fill pattern.

This example will fill our arrays (a, x) with zeros.

By the way, if our arrays each contain 256 elements, why did we use a fill length of 128?

7. Add one more menu item to fill the arrays

In this example, we want to ask the user to enter a value to write to each location in memory. Rather than using the *hotmenu* command, the *dialog* command allows us to query the user.

Enter the following:

```
dialog FillArrays(fillVal "Fill Arrays with:")
{
    GEL_MemoryFill(a, 0, 128, fillVal);
    GEL_MemoryFill(x, 0, 128, fillVal);
}
```

8. Save then Load your new GEL file

To use a GEL file, it must be loaded into CCS. When loaded, it shows up in the CCS *Explorer* window in the *GEL* folder.

File → Save

File → Load GEL and select your GEL file

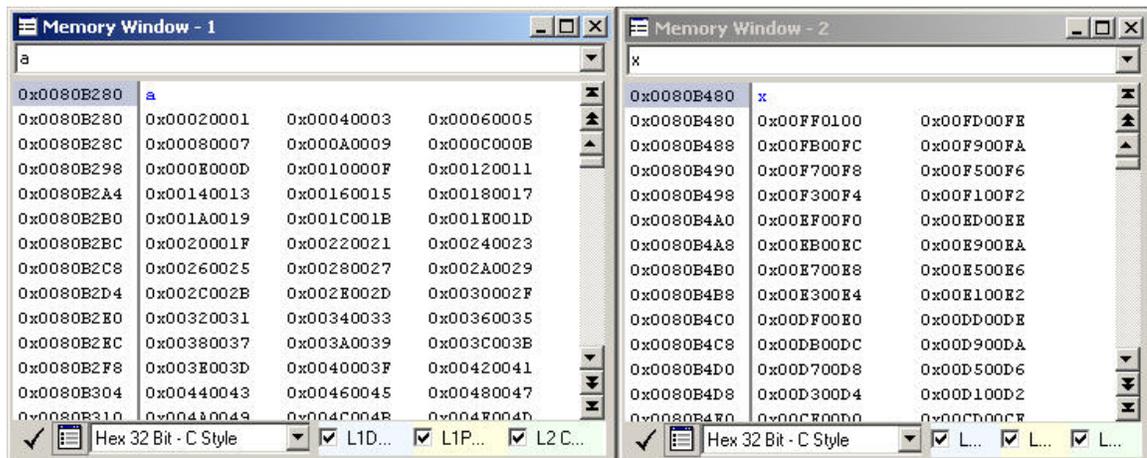
9. Let's try out your first GEL script

GEL → My GEL Functions → HelloWorld

10. Show arrays in *Memory* window

Without looking at the arrays, it will be hard to see the effect of our scripts. Let's open two *Memory* windows to view *a* and *x*.

View → Memory...



A couple notes about memory windows:

- If you enter a symbol, for example *a*, and you receive an error stating it cannot be found, make sure the program (.out file) that contains the symbol has been loaded (File-Load Program).
- You may find that array *a* begins with symbol *.bss* rather than *a*. In this case, these two symbols share the same memory address, but only one can be displayed. (The *.bss* symbol will be discussed in Chapter 12.)

- C Style adds 0x in front of the number, TI Style doesn't.

11. Now, try the other two GEL functions.

```
GEL → My GEL Functions → ClearArrays
```

```
GEL → My GEL Functions → FillArrays
```

You can actually use this GEL script throughout the rest of the workshop. It is a very handy tool. Feel free to add or delete commands from your new GEL file as you do the labs.

12. Review loaded GEL files.

Within the CCS *Explorer* window (on the left), locate and expand the GEL files folder. CCS lists all loaded GEL files here.

Hint: If you modify a loaded GEL file, before you can use the modifications you must reload it. The easiest way to reload a GEL file:

- (1) Right-click the GEL file in the CCS *Project Explorer* window
- (2) Pick *Reload* from the right-click popup menu

Optional Lab 2g – Graphing Data with CCS

The optional lab uses sine-wave and filter functions to generate an output that's easy to graph. After generating two sine waves and adding them, the filter demodulates one of the waveforms.

Create project

1. Close your Lab 2 project if it is still open.
2. Using Windows Explorer, *COPY* the following files from `c:\op6000\labs\lab2` to `c:\op6000\labs\lab2g`. You can use several methods to perform the copy. We prefer the method where you select the files that you want to copy, select Copy (Ctrl + C) to copy them, then navigate to the destination and select Paste (Ctrl + V) to paste them.



Please make sure you *COPY* the files.

LAB2.PJT
 C64.TCF, C64XP.TCF, C67.TCF or C672X.TCF
 C64CFG.CMD, C64XPCFG.CMD, C67CFG.CMD, or C672XCFG.CMD

Note: Please make sure to **copy** the above files correctly. We want to make sure that you can come back to any lab folder and find the files for that lab. If you are not careful, it is easy to move files around and rather than copying them. Make sure to **copy** the correct files from and to the correct place. The .TCF and .CMD file that you chose to copy will depend on which processor you selected when setting up CCS.

3. While still inside Windows Explorer, make sure that you are in the `c:\op6000\labs\lab2g` directory.
4. Use Windows Explorer to rename the `c:\op6000\labs\lab2g\lab2.pjt` file to `lab2g.pjt` by right clicking on it and choosing rename.
5. Now go over to Code Composer Studio (or open it up if you closed it) and open `lab2g.pjt`.

PROJECT → OPEN

Choose `c:\op6000\labs\lab2g\lab2g.pjt`

You will receive an error indicating that `main.c` cannot be found.



Remove **main.c** file from the project by using one of the two methods listed below:

- Click "Remove" in the preceding dialog
- Select the files in the Project Explorer window and press the delete key

6. Go to the **Project** menu and click on **Compile File**. This will build the linker command file.
7. Add the following files to the project:

sinefilter.C	(provided for you)
lowcoefficients.C	(provided for you)
c672xcfg.cmd	(linker command file from the previous step)

8. Build and load the project.

Setup graph windows

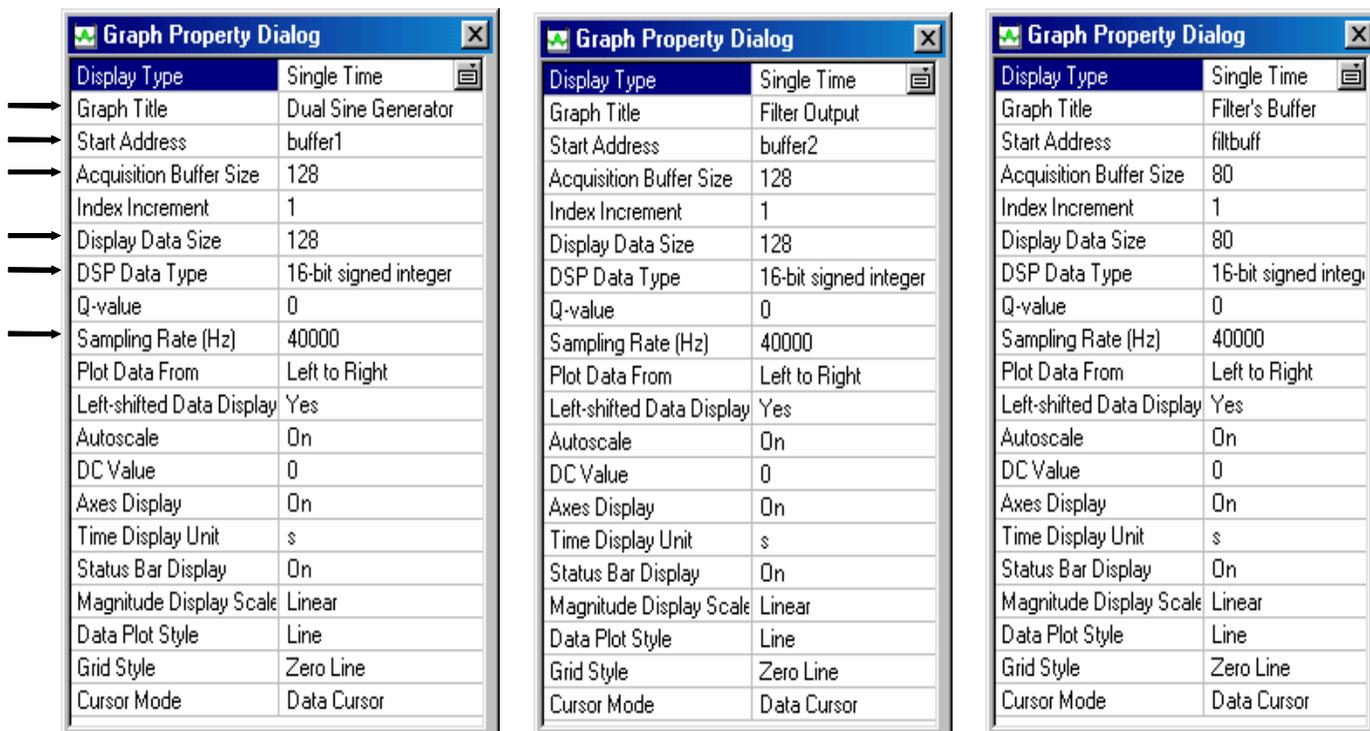
Let's setup three graph windows:

- `buffer1[]`: Sinegen-output / Filter-input
Buffer used for graphing data.
Graph should show a sinusoid with an obvious carrier.
- `buffer2[]`: Filter output
Buffer used for graphing data.
Graph should appear as a 1KHz sine-wave (using LowCoefficients.C)
- `filtbuff[]`: Buffer used by the filter.
It stores the 80 term delay line for the filter.
Watch as the data *slides* across the window.

9. Setup graph windows as shown:

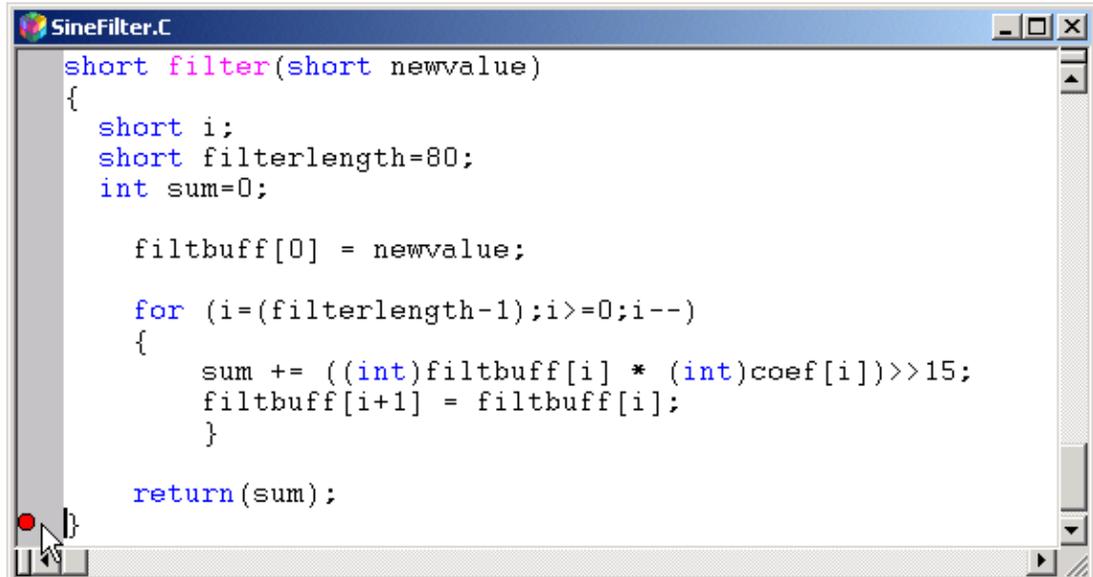
View → Graph → Time/Frequency

Setup three graphs with the following properties:



Setup breakpoint for animation

10. Setup up the following breakpoint:



```

short filter(short newvalue)
{
    short i;
    short filterlength=80;
    int sum=0;

    filtbuff[0] = newvalue;

    for (i=(filterlength-1);i>=0;i--)
    {
        sum += ((int)filtbuff[i] * (int)coef[i])>>15;
        filtbuff[i+1] = filtbuff[i];
    }

    return(sum);
}

```

11. Save the current window layout as a **Workspace**:

File → **Workspace** → **Save Workspace As:**

Give it a name such as **lab2g.wks**

12. **Restart** then **Run**.

The program should run to your breakpoint.

Look at the Data Visually

13. **Animate** your code.

Use  -or- **Debug** → **Animate** -or- 

You should see the graphs updating.

14. Now go to the Dual Sine Generator graph (*buffer1*), **RIGHT-CLICK** the right button of the mouse in this window and select **PROPERTIES**. Change the display type to

FFT Magnitude

15. Also, change the Filter Output graph (*buffer2*) to FFT Magnitude.

Notice that Dual Sine Generator graph (*buffer1*) has two spikes at 1Khz and 2Khz while Filter Buffer graph (*buffer2*) should only have a single spike at 1Khz. The filter removes one of the frequencies.

Other things to try in Lab 2g ...

- Try out the workspace saved earlier:
 - Quit Code Composer
 - Start Code Composer
 - Load LAB2g.WKSIsn't that easier than opening up all those graphs again?
- Change the filter coefficients:
 - Delete LowCoefficients.C from the project
 - Add HighCoefficients.C to the project
 - Rebuild the program and animate it again.
Now `buffer2` should only have a 2KHz spike.

Lab2h - Using CCSv4

Introduction

Our goal with Lab2h is to replicate a big part of our earlier lab exercise, but using the new CCSstudio version 4 (CCSv4).

We'll explore:

- Workspaces
- Perspectives
- Creating a CCS managed-make project
- Debug/Release compiler options
- DSP/BIOS 5.xx Configuration Files (.tcf)
- Target Configuration Files (i.e. CCS Setup)

Lab Topics

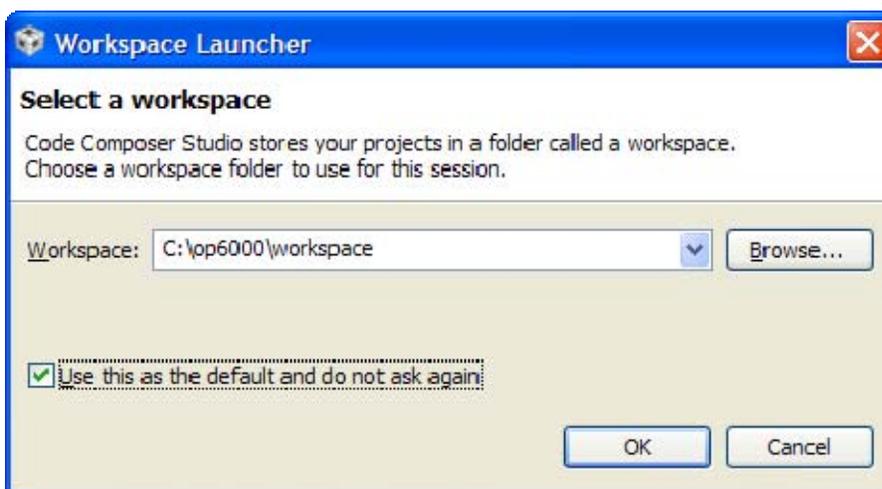
Lab2h - Using CCSv4.....	2-1
<i>Lab Topics.....</i>	2-2
<i>Open CCSv4.....</i>	2-3
<i>Creating a New CCSv4 Project</i>	2-4
New Project Wizard.....	2-4
Adding files to the Project.....	2-6
Build ... and try to Run	2-9
<i>Verify, Build ... and Run</i>	2-11
Start the Debugger.....	2-11
Breakpoint and Run.....	2-12
Clock your code.....	2-12
Profile your code	2-12
<i>Build Code for Release</i>	2-14
Fix the Release Build Configuration	2-15
Test Updated Release Build Config's Performance	2-16

Open CCSv4

1. Find the appropriate icon on the desktop to open CCSv4.



2. Select a workspace – we used (and recommend for this class) `C:\op6000\workspace`.

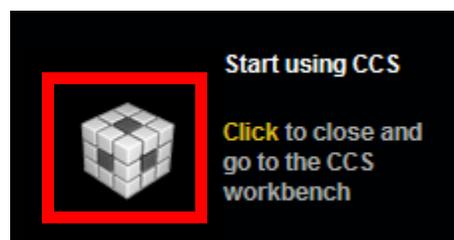


In Eclipse (i.e. CCSv4) a *workspace* is the repository for project information. The best analogy I can come up with is the iTunes database – which is the repository for all the metadata about your music and video. That is, the song files can exist *within* the iTunes database folder, or elsewhere on your drive. In any case, though, the iTunes database holds the information on what you have and where it's located.

Some users prefer to create a separate workspace for each project; others prefer to have all their projects in a single workspace - it's mostly a style preference. For this lab, we recommend following our directions ... we suspect that after you've used Eclipse/CCSv4 for a little while, you'll figure out which way you like best.

3. Click the cube ...

If your workspace is new, you will need to click the cube to start the IDE where you'll edit and debug your programs.



Creating a New CCSv4 Project

New Project Wizard

Conceptually, projects are very similar between CCSv3.3 and CCSv4. Practically, there's quite a few differences. What has changed the most is that the project settings are stored in a fashion created by the Eclipse IDE community – a set of xml files. Thankfully, TI has created a straightforward new project wizard that works quite well ... as long as you don't move thru it too fast and miss an important setting or two.

4. Create a new CCS project titled **lab2h** – locating it in the **lab2h_ccsv4** folder.

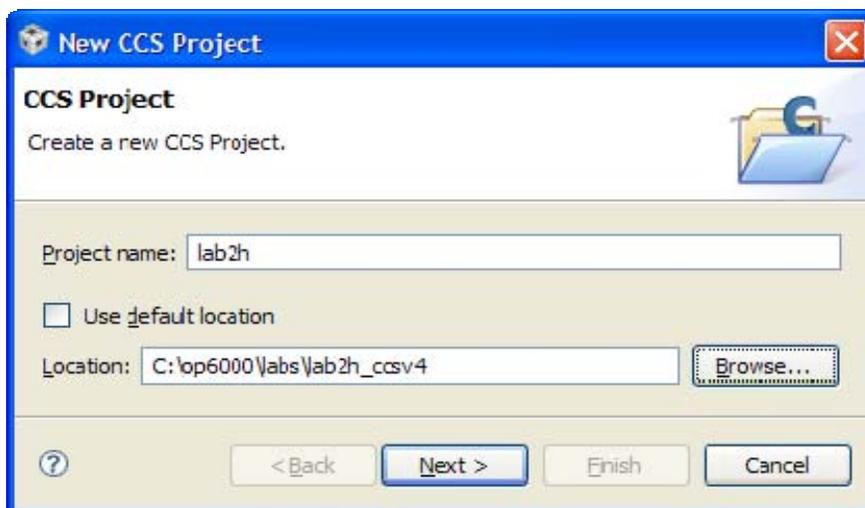
File → New → CCS Project

Fill out the resulting dialog as shown, then hit *Next >*.

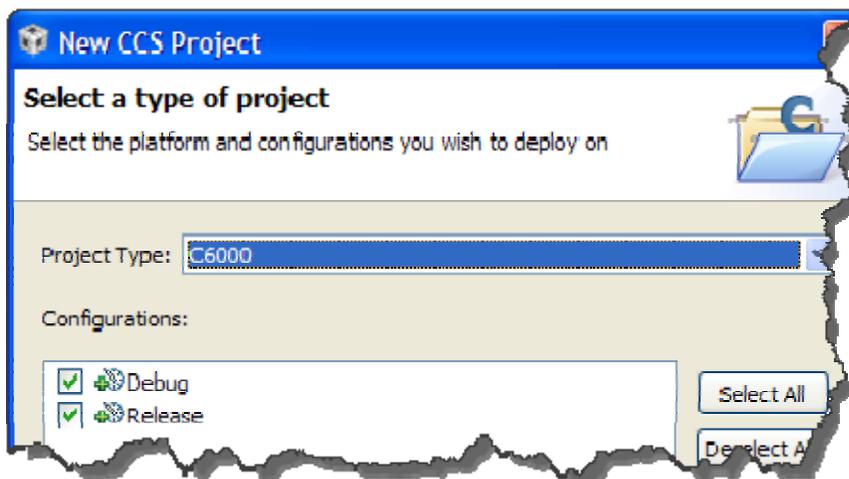
By default, CCS chooses the *workspace* to store your new project. However, we want to put our project (and it's files) into our *labs* project folder.

So, make sure you uncheck the “*Use default location*” checkbox and enter the project name “*lab2h*”. Then,

browse to the directory: `C:\op6000\labs\lab2h`

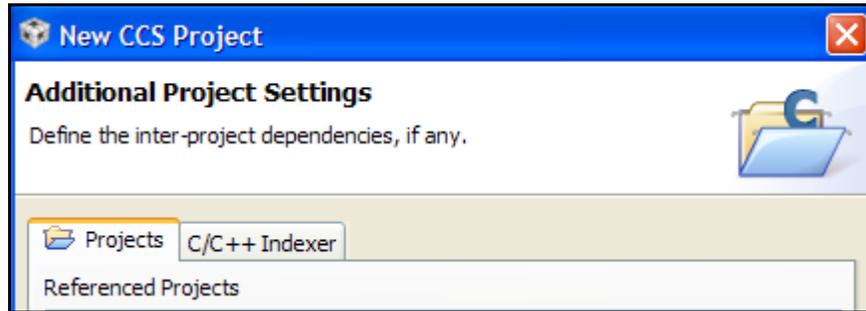


5. Choose the **Project Type** and default **Build Configurations**.



6. Select *Next* – as there are no *Add'l Project Settings*.

On this screen, just click *Next*, as there are no additional project settings.



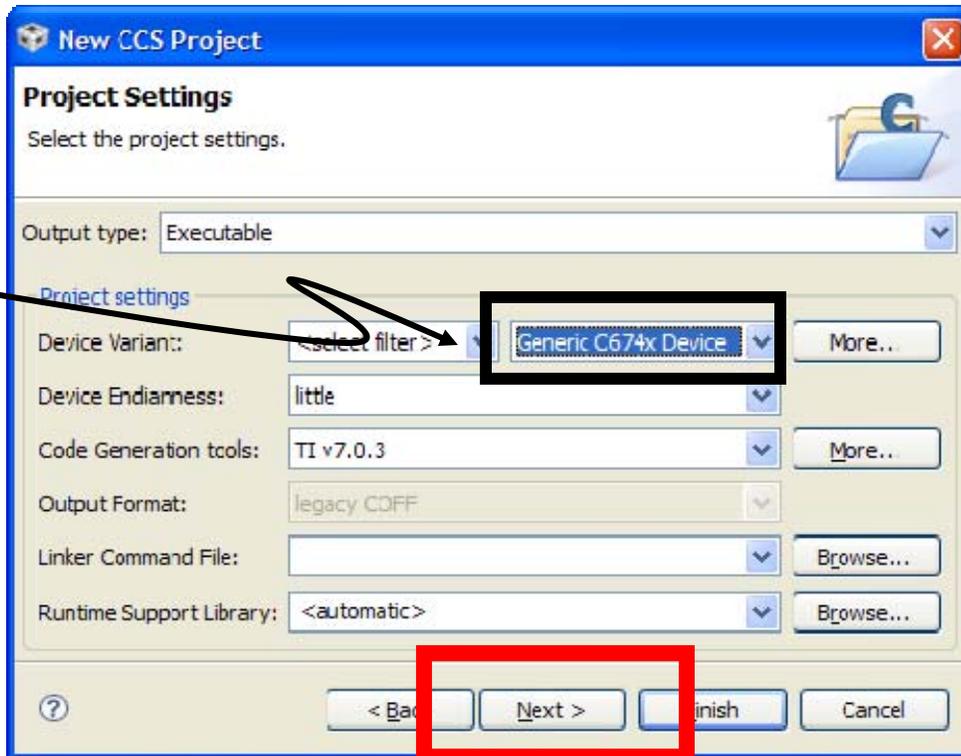
7. Select Project Settings ... then hit *Next* !!!

We are now at one of the critical and most useful capabilities of CCSv4. We can apply a number of settings on a *per-project* basis, such as: Compiler/Code-gen tools version, Endianness, target configuration (not in this dialog, but we'll show you this soon).

Older versions of CCS (e.g. CCSv3.3) required you to set these on a global basis in the CCS Configuration Manager (which we think was much less convenient).

Device Specific Step

Pick the device type based on C6000 generation you are using in this workshop

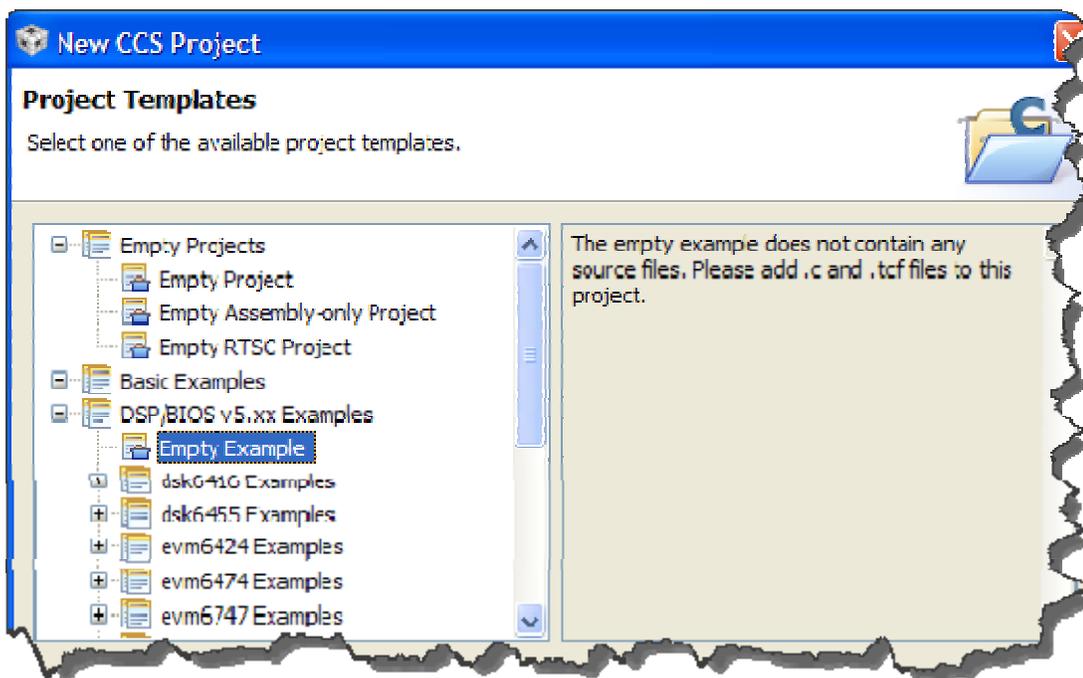


Pick the device generation appropriate for your needs ... then hit *Next*.

Note: If you aren't using BIOS, Codec Engine, XDC or any example program from TI, you could click *Finish*, but since we're using the BIOS configuration file to simplify our device configuration (i.e. reset/interrupt vectors), we want to choose *Next*.

8. Select a project template – we’ll use: DSP/BIOS v5.xx Empty Example.

This sets up the correct paths to the BIOS libraries and configuration tools.



Finally, you can click *Finish* when you’re done selecting the *Empty Example*.

Adding files to the Project

Adding Source Files

9. Add the program source files to your new project.

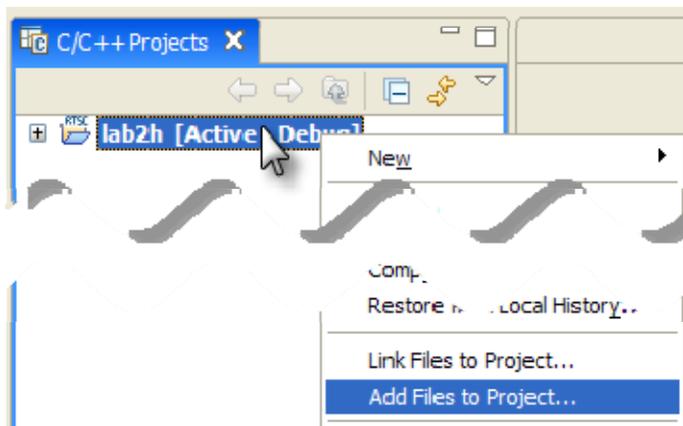
Right-click on the project and choose the *Add Files to Project...* menu command.

Add the following two files to the project from our earlier lab exercise:

```
main.c  
data.h
```

You can find these files in:

```
C:\op6000\labs\lab2
```



Note: *What’s the difference between Link and Add?*

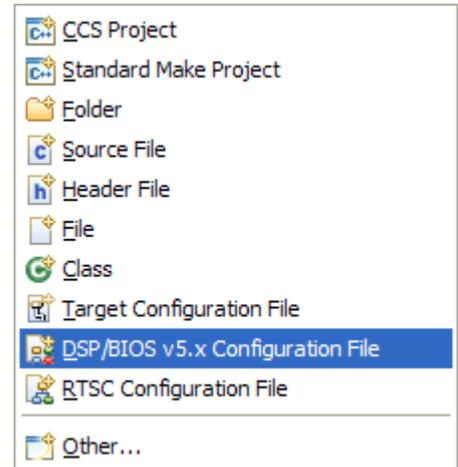
CCSv3.3 only supported *linking* – where file was only referenced from it’s original location. In CCSv4, though, *adding* a file copies it into the project’s directory (which is what we want to do for this example lab).

Adding TCF File

10. Create a new BIOS Configuration file (.tcf) and add it to the project.

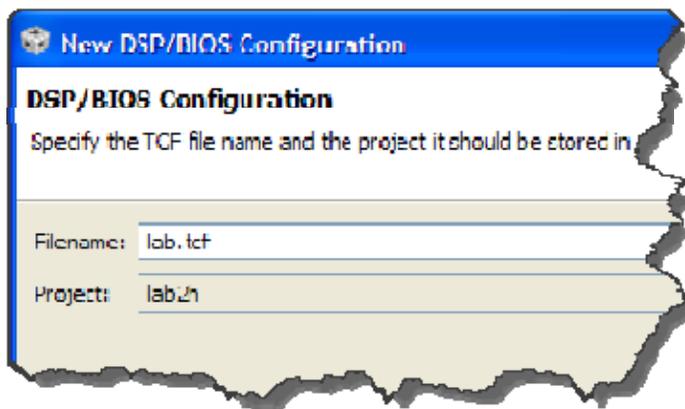
Conceptually, this is similar to the step we performed earlier in the lab exercise for CCSv3.3, though you'll notice the *new tcf file* dialog is a bit different.

File → New → DSP/BIOS v5.x Configuration File



11. Name the file lab.tcf and locate it in the project's folder.

You may need to *Browse* to select the project's path. The project path may look like lab2h or /lab2h.



Once complete, click *Next >*

12. Specify a platform to build for:

You may remember that we needed to pick the appropriate TCF seed file in our CCSv3.3. project.

Similarly, in CCSv4, we'll choose the platform we are building for, then the wizard will import that file (you can see the results if you look at the tcf file's javascript code).

You need to pick the platform appropriate for your lab selection. Four platforms are shown here, the other two you might need to use include:

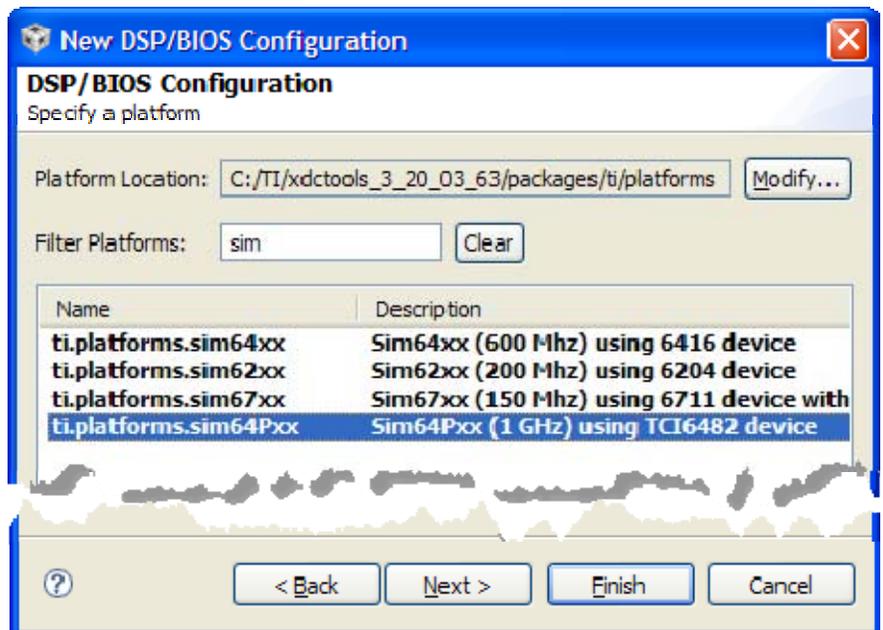
C672x use:

ti.platforms.padk6727

C674x use:

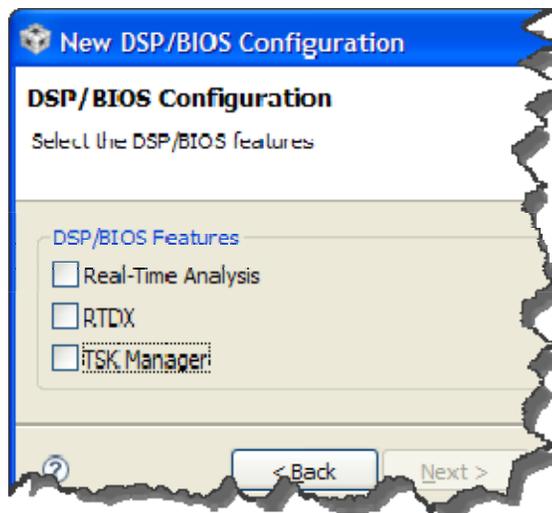
ti.platforms.evm6748

Select *Next >*



13. Deselect unnecessary BIOS components:

This last step of the .tcf wizard is not really that big of deal. It just gives you the opportunity to remove some components of DSP/BIOS if you're not planning to use them.

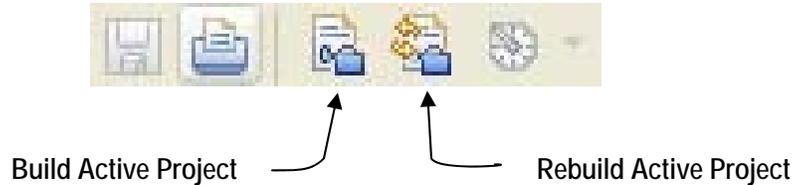


In our case, we won't be using these BIOS modules. You can easily enable (or disable) these modules later on by editing the .tcf file (either graphically or textually). In our case, this will just save us a small amount of additional code space.

Build ... and try to Run

14. Build your program.

Since we are using the same code as earlier, we shouldn't get any build errors.



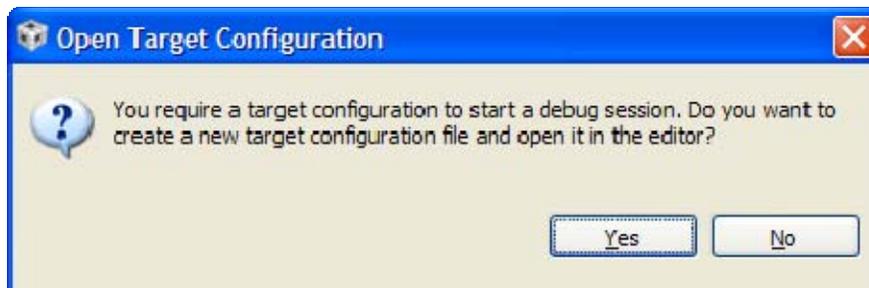
15. Let's try to debug our project. Click the *Debug* button.



This button should:

- Start the debugger
- Connect to the target platform (simulator, in our case)
- Load the program

But in this case, you should actually see it provide the following error dialog:



This error is telling us that we don't have a Target board/simulator configured for your project. Remember back to CCSv3.3 when we had to run *CCS Setup* before we could use CCS? In that case, CCS Setup set the default target board for use with all CCS projects – if you wanted to change boards, you had to re-run *CCS Setup*.

In CCSv4, the target configuration is available on a per-project basis. Even more than that, you can actually have more than one target configuration, then switch between them, as needed.

CCSv4 uses a Target Configuration File (.ccxml) to specify a target board. They can be copied from project-to-project; referenced from a common location; or, you can specify a workspace default.

In our case, we're going to create a new .ccxml file and add it to our project.

Create a Target Configuration File

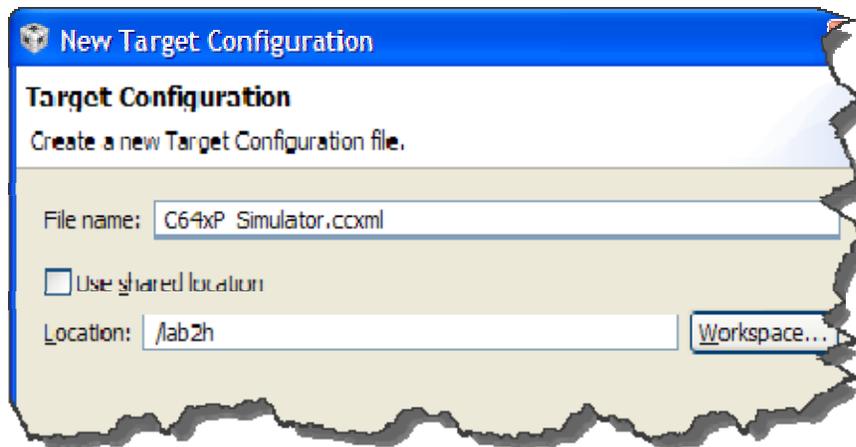
16. Create a Target Configuration File (.ccxml).

If you clicked “Yes” to the previous dialog box, you’re all ready to create a new target configuration file.

On the other hand, if you clicked “No”, then you’ll need to start the New Target Configuration File wizard: `File` → `New` → `Target Configuration File`

However you get there, you should see the following file dialog:

Device Specific Step



We recommend you choose a meaningful name. In our case, we chose to call it:

```
<CPU core name>_ Simulator.ccxml
```

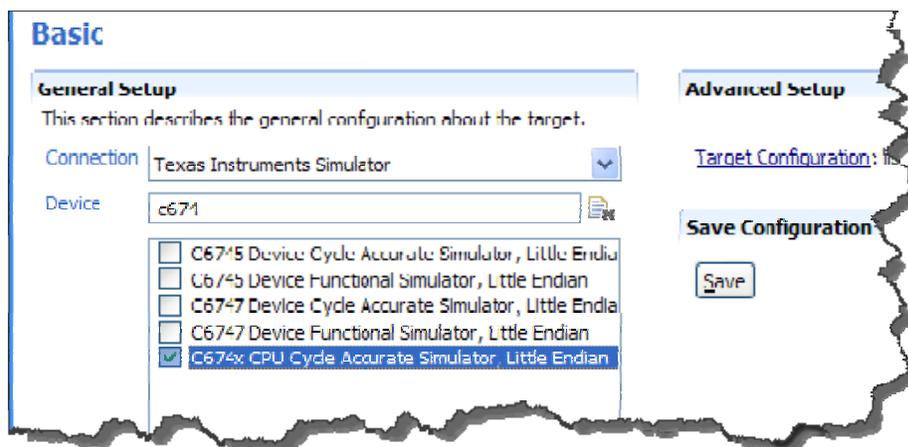
and place it into the lab folder (rather than a shared location). Again, this is one of those user style-choice decisions.

Click *Finish* when complete.

Device Specific Step

17. Edit the .ccxml file.

Once you finish the previous step, you’ll notice the .ccxml file is immediately opened for you to edit. Once again, it should look very similar to the *CCS Setup* dialog used for CCSv3.3.



Note: Please choose the same simulator you selected when configuring CCSv3.3.

Verify, Build ... and Run

18. Verify your project files.

At this point, your project should look something like this. (Though, your .ccxml file may be named differently.)



19. Build your program again

Start the Debugger

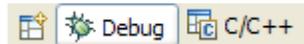
20. Click the 'bug' again to start the debugger.

This time, you shouldn't have any problems getting the debugger to run.

How do I know the debugger is running?

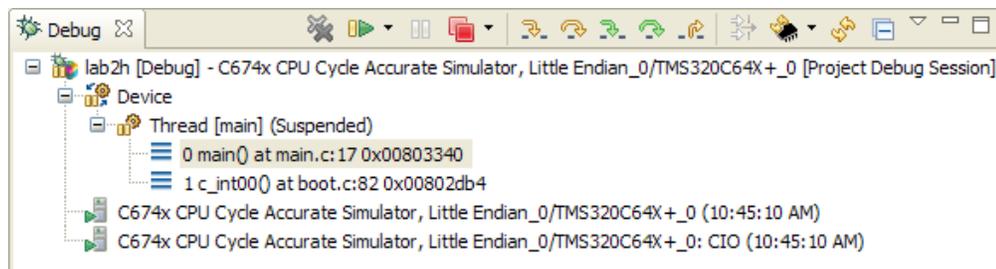
Look for two things:

1. The display should switch to the *Debug* perspective.



Eclipse *perspectives* provide a way to organize windows, toolbars, etc. Two of the default perspectives are *Debug* and *C/C++*. (The latter is convenient for editing code.) Look up perspectives in the CCS (or Eclipse) help for more info.

2. The Debug window.



The *Debug* window provides visibility to the target board; including a call stack of code running on the target.

We recommend you explore the *Debug* toolbar. Find these icons, which we'll be using:

- *Run*
- *Halt*
- *Terminate All*
- *Step Into* (and *Assembly Step Into*)
- *Restart*

Breakpoint and Run

21. Set a breakpoint at the end of main (as we did in the CCSv3.3 version of the lab).

22. Run to the breakpoint and check the result for *y* – is it correct?

You can check the result by hovering over *y*.
Alternatively, you can add it to the *Watch* window.

Hint: If you haven't used Eclipse before, you may be tempted to click the **RED** button to halt your code. Unfortunately, the RED button is for *Terminate All*. This will actually stop and close the debugger.

If, on the other hand, your goal was to just halt the code so that you could examine a variable – or something along those lines – then you should click the **Yellow Halt** button.

Clock your code

23. Restart your program (using the *Restart* toolbar command).

24. Enable the clock.

Target → Clock → Enable

You should see the clock appear in the bottom status bar, similar to the clock in CCSv3.3. While this clock has some additional capabilities, we'll use it as we did in the earlier lab to simply count clock cycles. (And remember, double-click the clock to reset it to 0.)

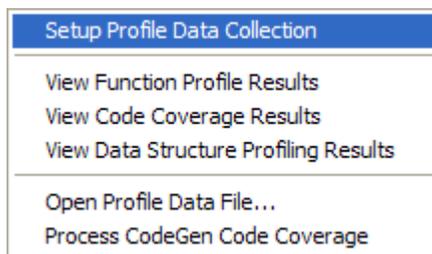
25. Run again to the breakpoint and record the number of cycles.

Debug Build Configuration: _____

Profile your code

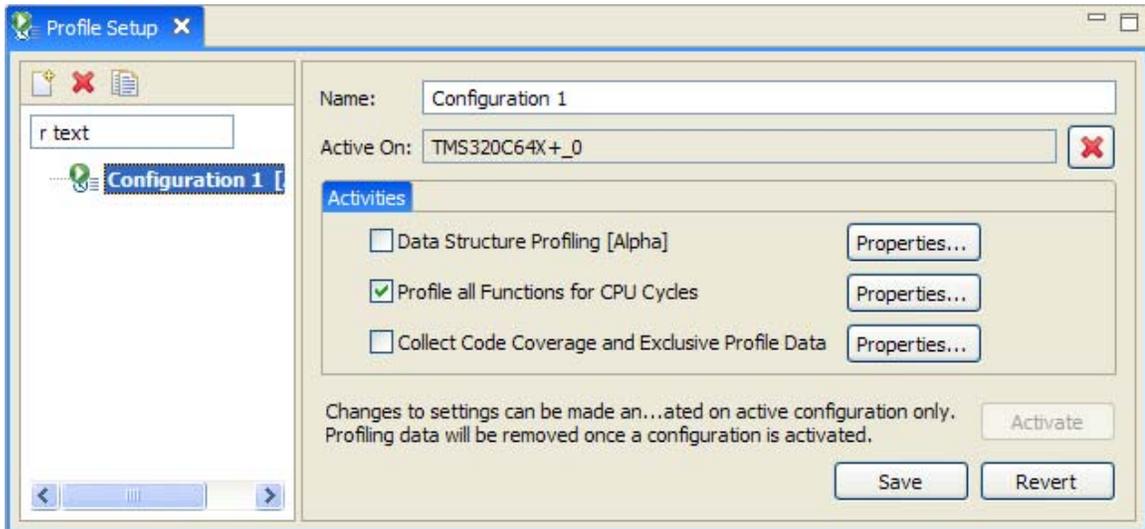
26. Open the profile settings dialog.

Tools → Profile →



27. Configure the profiler to: Profile all Functions for CPU Cycles

Just click the appropriate *checkbox* and then *Save*.

**28. Then we need to open the window to view the profiler's results:**

Tools → Profile → View Function Profile Results

29. Restart the CPU, then Run.

When the breakpoint is reached, you should see the cycles displayed in the profiler results window.

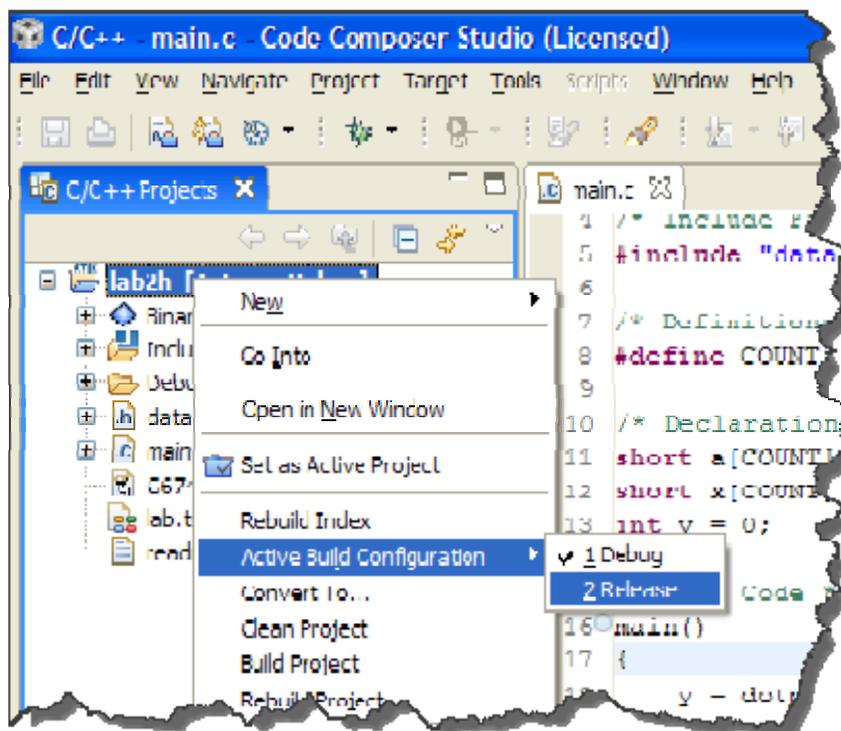
Debug (cycles): _____

Build Code for Release

30. Return to the Edit/Build mode to rebuild our code with the Release (optimized) options.

The easiest way to do this is to *Terminate* the debug session. (Red button)

31. Switch the project to the Release configuration.



32. As previously, build, run, and profile (or clock) the code.

Hint: If you click the *Debug* button, it should build a program (if necessary) and then start the debugger.

Note: If you *Terminate* a debugging session, you will need to *Re-Activate* the profiler if you want to continue using it. Just open up the Profiler Settings dialog and click the *Activate* button.

Activate

33. What performance did you get for Release?

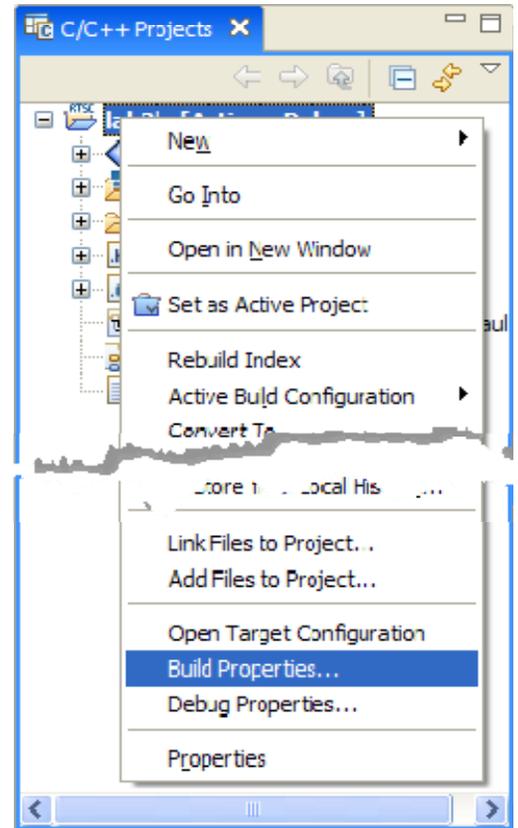
Release (cycles): _____

34. Why isn't the performance as good as we got with CCSv3.3?

Fix the Release Build Configuration

You may, or may not, have noticed that the default *Release* build configuration is not set optimally. Let's correct this and re-benchmark our code.

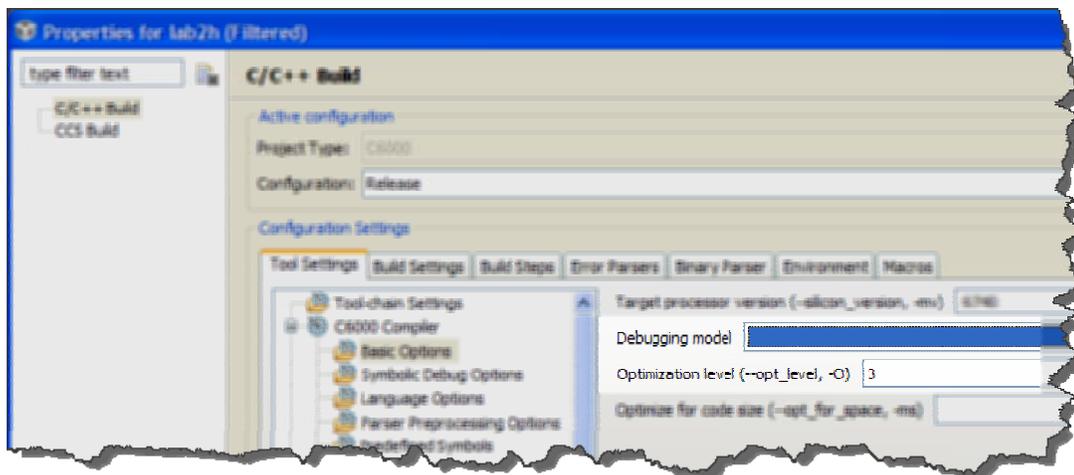
35. **Right-click on the project and select *Build Properties*.**



36. **Change the *Debugging Model* and *Optimization Level* options (as shown below).**

What they were set to before making this change? Debugging Model: _____

Optimization Level: _____



37. You may also want to keep the generated assembly code (-k option).

Hint: Look for the *Assembly Options* section under the *C6000 Compiler*.

38. Close the dialog and save the options.

Test Updated Release Build Config's Performance

39. Re-build and re-profile the code.

With the options set as we discussed in class, what is the performance ... and how does it compare to our previous (CCSv3.3) results?

. Release (cycles): _____

. How does it compare? _____

. _____

Chapter 3 – Exam

You have approximately 10 minutes to complete the exam. Afterwards, the instructor will debrief each question while your neighbor grades your exam. Have fun!

- Exam is open mind, open book, open eyes. Sharing answers, cheating, asking the instructor questions, anything to get the highest grade possible is completely acceptable, encouraged, and expected.
- Good luck.

Student Name: _____

1. Pipeline

- a. Name the three primary pipeline stages. (15 pts)

- b. Why did TI choose to make program fetches (PF pipeline stage) four cycles in length? (10 pts)

- c. Why are pipeline stages of a load (E1-E5) similar to the phases of a program fetch (PG-DP)? (10 pts)

2. Pipeline

What is a Fetch Packet (FP)? Execute Packet (EP)?

☐ (15 pts)

3. Insert the proper # NOPs in the following code:

☐ (30 pts)

```
start:      LDH   .D1   *A0, A1

            MPY   .M1   A1, A2, A3

            ADD   .L1   A1, A2, A4

            LDB   .D1   *A5, A6

            LDW   .D1   *A7, A8

            MPY   .M1   A6, A9, A10

            ADD   .L1   A9, A10, A11

            B     .S1   start
```

4. Coding

(20 pts) **Add the contents of mem1 to A5.**

The diagram shows two memory locations. On the left, a vertical stack of three yellow boxes represents memory. The middle box contains the value **10h**. Above the stack is the label **x16 mem**, and to the left is the label **mem1**. To the right of this stack is a plus sign **+**. Further right is a single yellow box representing register **A5**, which contains the value **7h**.

Grading your Exam

Scoring Chapter 3 Exam

Points Earned (Questions # 1 - 4): _____ /100

Grade (circle one):

90-100	A
80-89	B
0-79	C

(no one likes Ds or Fs!)

Page left intentionally blank

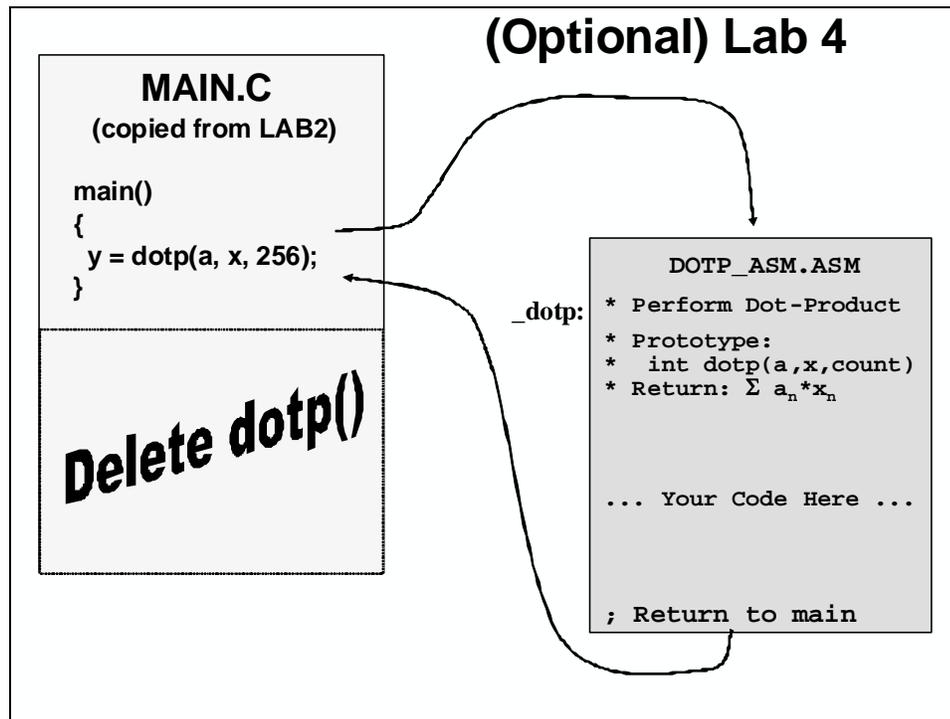
Lab 4 - Calling Assembly from C (Optional)

Lab Introduction

Lab2 used C to call and compute the dot product of two arrays. In this lab, you will use Lab2's C setup and initialization but create a separate assembly routine (called from C) that solves the dot product algorithm:

$$Y = \sum_{n=1}^{256} a(n) * x(n)$$

In Chapter's 6-8, you will use well-defined methods to optimize the dot-product routine to get to the 80 cycle loop benchmark discussed in Chapter 3. (We might not get that fast in this chapter, but we'll learn how to do it in assembly.)



This project includes the following files:

LAB4.PJT	copy from LAB2.PJT and rename
MAIN.C	copy from LAB2 folder
DOTP_ASM.ASM	dot-product routine created by you
LAB.TCF	copy from LAB2 folder
LABCFG.CMD	copy from LAB2 folder
PROFILER.INI	copy from LAB2 folder
DATA.H	provided for you

Creating the Project

1. Make sure that there are no open projects in Code Composer Studio. If there are, **close** them.
2. Using Windows Explorer, **COPY** the following files from **C:\op6000\labs\lab2** to **C:\op6000\labs\lab4**. You can use several methods to perform the copy. We prefer the method where you select the files that you want to copy, select Copy (Ctrl + C) to copy them, then navigate to the destination and select Paste (Ctrl + V) to paste them. Please make sure to pick **COPY**.

```
LAB2.PJT
MAIN.C
DATA.H
LAB.TCF
LABCFG.CMD
PROFILER.INI
```

Note: Please make sure to **COPY** the above files correctly. We want to make sure that you can come back to any lab folder and find the files for that lab. If you are not careful, it is easy to move files around and rather than copying them. Make sure to **copy** the correct files from and to the correct place.

3. While still inside Windows Explorer, make sure that you are in the **C:\op6000\labs\lab4** directory.
4. Use Windows Explorer to rename the **C:\op6000\labs\lab4\lab2.pjt** file to **lab4.pjt** by right clicking on it and choosing rename.
5. Now go over to Code Composer Studio (or open it up if you closed it) and open **lab4.pjt**.

```
PROJECT → OPEN
```

Navigate to **C:\op6000\labs\lab4** and double click on **lab4.pjt**.

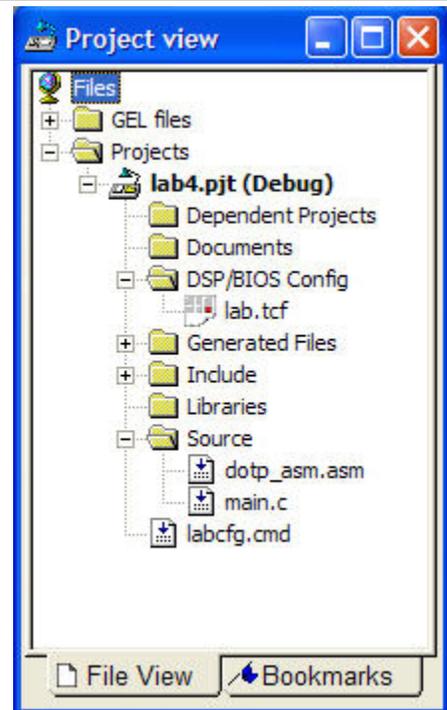
6. Now that we've converted our project from Lab 2, let's create a new ASM file for the assembly dot-product routine. Create a new source file and save it to the **Lab 4** subdirectory as **dotp_asm.asm**.

```
FILE → NEW → SOURCE FILE
FILE → SAVE AS...
```

7. Add **dotp_asm.asm** to your **lab4.pjt**:

```
PROJECT → ADD FILES TO PROJECT...
```

- The Project View in CCS should look something like this depending on your processor:



If it doesn't, double-check your work or ask the instructor for help.

Edit Source Files

- Open **main.c** and comment out (or delete) the **dotp()** function.
- We're going to need to use arguments passed to us from *main()* to solve the dot-product equation using assembly code. Fill in the blanks below with the registers that the compiler uses to pass/return arguments:
 - Use the three values “passed” by the function call: **a, x, count**

```
y = dotp(a, x, COUNT);
int dotp(short *m, short *n, int count)
```

Input Variable	C Passes Input Argument in Register
m	
n	
count	

Function Return	Register Used
Return Value	
Return Address	

Benchmark ASM Code

The Clock

One way to benchmark ASM code, is to use the Profile Clock.

16. Reset the CPU or Restart the program, then go to *main()* if you are not taken there automatically.
17. Open the Profiling Clock so that you can benchmark the function.

Profile → Clock → View

You should now see the Clock down in the lower right-hand corner of CCS:



18. Run until you reach the breakpoint at the end of *main()*.
19. How many cycles did *dotp()* require? _____ cycles
20. How does this compare to the benchmark that we got for C code in Lab 2 using the *Optimize* configuration?

The Profile View

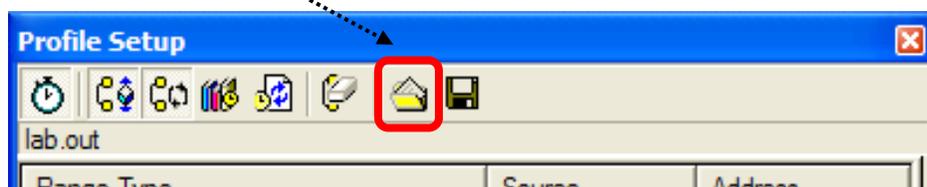
Another way to benchmark ASM code is to use the Profile View (along with Profile Setup) that we used back in Lab2.

21. Reset the CPU or Restart the program, then go to *main()* if you are not taken there automatically.
22. Open the Profile Setup window.

Profile → Setup

23. Load the profiler configuration you saved previously.

Use the *Load Profiler Configuration* button just to the left of the save button.



PROFILER.INI was the name we suggested for saving the profiler configuration.

24. Make sure that the Enable/Disable Profiling button  is turned on to enable profiling.
25. Open the Profile Viewer.

Profile → Viewer

26. Run until you reach the breakpoint at the end of *main()*.
27. Find the row for the function in *dotp_asm.asm*. You may have to scroll down to find this row.

Address Range	Symbol Name	SLR	Symbol Type	Access Count	cycle.CPU: Incl. ...	cycle.CPU: Excl. ...
0:0xe0a0-0xe0d8	dotp_asm.asm:23:...	23-38:dotp_asm.a...	function	1	4355	4355
0:0xe0e0-0xe114	main	17-19:main.c	function	1	4373	0

Note: your numbers may vary slightly from those shown above.

There should only be one entry for *dotp_asm.asm:##:##\$*. The first number is the line number for the start of the range and the second number is for the end of the range that was profiled for the ASM function.

How do these numbers compare with those that you obtained in step 19 using the Profile Clock?

28. Would it be nice to see a symbol associated with the assembly function so that it looks more like a C function in the profiler? This can be done by adding two new directives: `.asmfunc` and `.endasmfunc` to your ASM file like this:

```

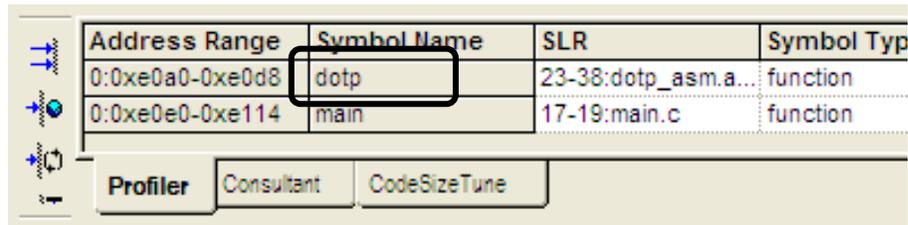
dotp_asm.asm
*   A6 = original counter value
*   B0 = working Y value
*   B4 = address of x
*****
.sect   ".text"           ; What section is this put
                        ;   so the linker can find
dotp   .asmfunc
_dotp:
...
    zero B0               ; initilize y=0
    mv   A6, A2           ; set up counter
loop  ldh  *A4++, A0      ; put the value of a into
    ldh  *B4++, A1      ; put the value of x into
    nop  4
    mpy  A0, A1, A3      ; a * x --> product
    nop  ; put counter into A2
    add  A3, B0, B0      ; product + previous Y = Y
    sub  A2, 1, A2
    [A2] b   loop        ; repeat loop counter times
    nop  5
    mv   B0, A4          ; store return value
    b    B3              ; return to main
    nop  5
    .endasmfunc
    
```

29. Using the *Debug* build configuration, rebuild all files by clicking on .

30. Reset the CPU or Restart the program, then go to `main()` if you are not taken there automatically.

31. Run until you reach the breakpoint at the end of `main()`.

32. Take a look at the Profile Viewer. Notice that it lists the function with the dotp symbol instead of using the cryptic file reference.



Address Range	Symbol Name	SLR	Symbol Type
0:0xe0a0-0xe0d8	dotp	23-38:dotp_asm.a...	function
0:0xe0e0-0xe114	main	17-19:main.c	function

Profiler Consultant CodeSizeTune

The `.asmfunc` and `.endasmfunc` make the function “visible” to the profiler and allow you to assign a more meaningful name to an ASM function. If you’d like more information on these directives, please see the online help.

Command Window

Now that we've written and debugged some assembly code, let's explore how to use one of CCS's other features, the Command Window.

33. Open the *Command Window*.

Tools → **Command Window**

and execute the following commands by typing them into the Command Window.

```
restart           Restarts the processor
go dotp          Runs to the dotp label
```

34. Single-step into your routine. Using the *CPU Registers* window, observe that your “count” register is (or gets) set to 0x00000100 (256 decimal).

35. Set a breakpoint within your ASM dotp subroutine. Do this by double-clicking on the gray left-hand column next to a line in the subroutine. Alternatively, you can use the *Command* window. To demonstrate this, here's a clip of our solution's assembly code.

```
_dotp:
        zero B0           ; initialize y=0
        mv A6, A2         ; set up counter in cond. reg

loop    ldh      *A4++, A0 ; put the value of a into A0
        ldh      *B4++, A1 ; put the value of x into A1
```

It's easy to add a breakpoint using any label (with the **ba** command). In our case, the label “loop” is part of the loop (it's the top of the loop).

```
ba loop
```

36. Run until the counter is equal to five. In the *Command* window, type the following and hit the Execute button.

```
run A2 > 5 ↵
```

Note: If you didn't use **A2** as your **count** register, replace A2 in the above command with the register you chose.

The *run* command is conditional. This *run* example says, “run while A2 > 5”. When running, every time the debugger hits a breakpoint it evaluates the expression. If still true, it automatically begins running again.

37. **Save** all of your work and **close** any open projects.

End of Lab 4 Exercise

Exercise Solutions

Mixing C and Assembly

Why Bother with Assembly?

- Total flexibility
- Understand how compiler works
- Understand how processor works

Calling Assembly from C

```
//Parent.C
int child(int, int);
int x = 7, y, w = 3;

void main (void)
{
    y = child(x, 5);
}
```

- ◆ Arguments
- ◆ Return/Result

```
//Child.C
int child(int a, int b)
{
    return(a + b);
}
```

```
;Child.ASM

.global _child
_child:
    add    a4,b4,a4
    b     b3
    nop   5
; end of subroutine
```

Accessing Global Variables from ASM

```
//Parent.C
int child2(int, int);
int x = 7, y, w = 3;

void main (void)
{
    y = child2(x, 5);
}
```

```
;Child2.ASM

.global _child2
.global _w

_child2:
    mvkl  _w , A1
    mvkh  _w , A1
    ldw   *A1, A0
```

- ◆ Use underscore when accessing C variables/labels
- ◆ Declare global labels
- ◆ Advantages of declaring variables in C?
 - Declaring in C is easier
 - Compiler does variable init (int w = 3)

Lab 5a – Writing Linear Assembly

Lab 5a introduces us to the easier method of programming with assembly mnemonics, Linear Assembly. This is a rather simple lab, but it allows us to quickly practice using the assembler directives required by the assembly optimizer.

Optional: Lab 5b gives you a chance to try calling one linear assembly routine from another. If you have time, try this optional exercise.

Overview

Similar to Chapter 4, our goal is to write the dot-product function. This time you don't need to worry about delay-slots (hence, NOPs), functional units, register names, etc.

Essentially, you will use the same project file as in Chapter 4. Just replace the assembly dot-product with a linear-assembly dot-product.

<code>LAB5a.PJT</code>	<code>copy from LAB4.PJT</code>
<code>DOTP.SA</code>	<code>you create this</code>
<code>MAIN.C</code>	<code>copy from LAB4 folder</code>
<code>LAB.TCF</code>	<code>copy from LAB4 folder</code>
<code>LABCFG.CMD</code>	<code>copy from LAB4 folder</code>
<code>PROFILER.INI</code>	<code>copy from LAB4 folder</code>
<code>DATA.H</code>	<code>provided for you</code>

Create Project

Note: If you did not complete the Optional Lab 4 exercise you should copy the above files from `C:\op6000\solutions for (your selected processor)\lab4` to `C:\op6000\labs\lab5a`.

1. Using Windows Explorer, *COPY* the following files from `C:\op6000\labs\lab4` to `C:\op6000\labs\lab5a`. Please make sure to *COPY* these files.

```
LAB4.PJT
MAIN.C
LAB.TCF
LABCFG.CMD
PROFILER.INI
```

Note: Please make sure to **copy** the above files correctly. We want to make sure that you can come back to any lab folder and find the files for that lab. If you are not careful, it is easy to move files around instead of copying them. Please make sure to **copy** the correct files from and to the correct place.

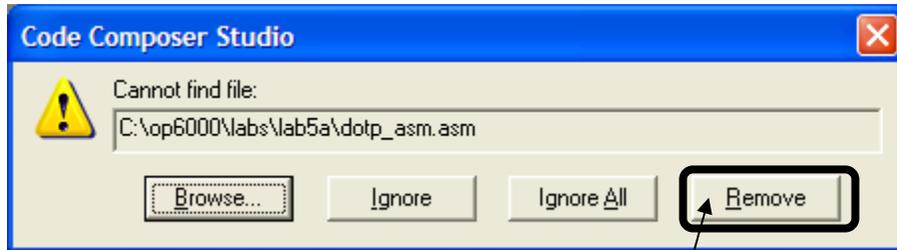
2. While still inside Windows Explorer, make sure that you are in the `C:\op6000\labs\lab5a` directory.
3. Rename the `C:\op6000\labs\lab5a\lab4.pjt` file to `lab5a.pjt`.

- Now go over to Code Composer Studio (or open it up if you closed it) and open **lab5a.pjt**.

PROJECT → OPEN

Navigate to **C:\op6000\labs\lab5a** and double click on **lab5a.pjt**.

- We didn't copy the `dotp_asm.asm` file from Lab 4, but it is still listed in the project. Therefore, CCS cannot find the file in the project directory. You should receive an error that looks something like this:



Remove the file from the project by clicking on **Remove**.

- Create a new source file and save it to the **Lab5a** subdirectory as **dotp.sa**.

FILE → NEW → SOURCE FILE
FILE → SAVE AS...

- Add **dotp.sa** to your **lab5a.pjt**:

PROJECT → ADD FILES TO PROJECT...

Edit Source Files

- Make sure the `dotp()` function is still commented out or deleted from the **main.c** source file.
- Add the following code to **dotp.sa**. (You can either type in the following code or cut-and-paste it from the supplied "**dotp template.sa**" file.)

```
        .global  _dotp

_dotp:
    zero    y
loop:
    ldh     *ap++, a
    ldh     *xp++, x
    mpy     a, x, prod
    add     prod, y, y
    sub     cnt, 1, cnt
[cnt] b    loop
```

Notice, this code is similar to the assembly code used in Chapter 4.

10. Well, you've created most of the linear assembly file. Did you already notice something is missing? Please add the required assembler directives. (Turn the page if you need a hint.)

You should also add these assembler directives:

```
.cproc  <arg>, <arg>, <arg>
.reg    <arg>, <arg>, <arg>, <arg>
.return <arg>
.endproc
```

Build, Run, and Profile (Debug)

11. Build the project using the **Debug** options.

12. Verify that the code works.

Run to the end of *main()* and check your answer. What's the value of *y*?

y = _____

13. Reset or Restart the program, then go to *main()*.
14. Use one of the Profiling Methods, either the Profile Clock or the Profile Setup/Viewer, to benchmark the Linear Assembly *dotp* function.
15. Run until you reach the end of *main()*.
16. How many cycles did *dotp()* require? _____ cycles

Using Optimization

Once you've verified that the code works, go back and reconfigure the options to enable optimization.

17. Rebuild your code with the *Release* configuration that we modified back in Lab 2.
18. Profile the *Release* configuration.

Dotp cycle count: _____

y = _____

19. How does this compare with your previous results?

Dot-product Version	Cycles
Lab2 (Debug options)	
Lab2 (Optimize options)	
Lab4 (standard assembly)	
Lab5a (step 16: Debug options)	
Lab5a (step 18: Release options)	

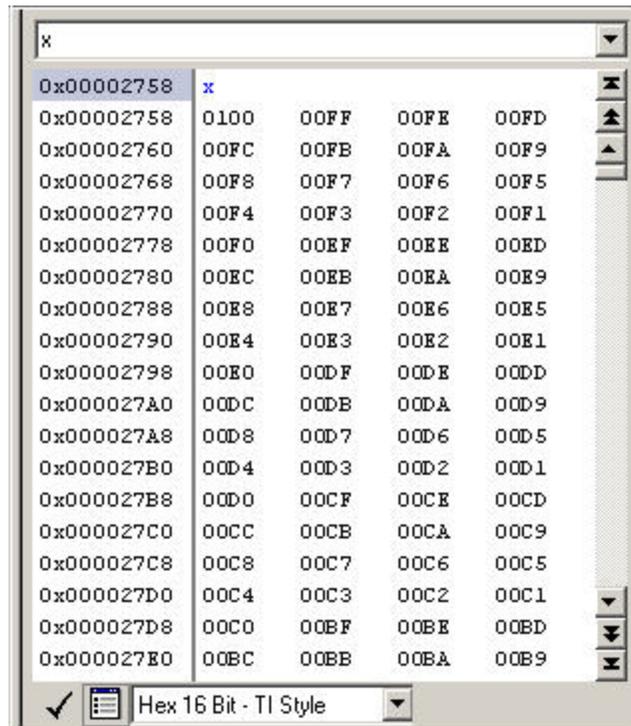
Did the compiler beat either standard or linear assembly code?
 (We'll find out why in Chapter 7.)

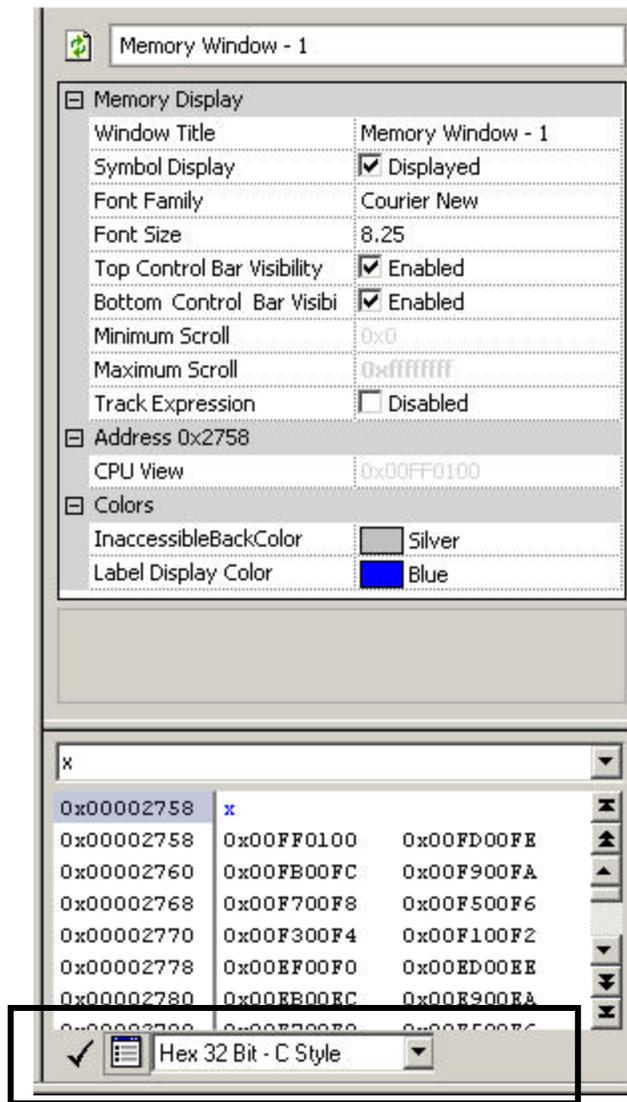
Viewing Memory

There are many methods of looking at data in memory using CCS. Earlier we used the *Watch* and *Core CPU Registers* windows to view results. You can also open a *Memory* window to examine your systems memory.

20. Open a memory windows to view array `x[]`.

View → Memory...





The *Open Property Window* icon  is used to display the properties for the memory window.

- Set the address to a. Notice, that CCS also displays the hex address.
- Track Expression, when not disabled, tells CCS that you want to continue to watch “a”, even if the symbol’s address changes on subsequent builds. Note: Normally do not check “disabled”.
- Select the 16-bit format using the drop down box, since our data is 16 bits wide.

21. Change Memory window's format to *16-Bit Hex – C Style*

Use the drop-down selection list

22. What is the difference between the *TI* and *C* styles?

Many users prefer the *TI* style since it takes up less space.

23. In the *Memory* window, what is the first value of **x**: 0x_____

x[0] should equal 0x0100 (or 256 decimal).

24. Now, change the *Memory* window of **x** to display:

8-bit Hex – *TI* Style

25. What are the first two values for **x**? 0x_____ 0x_____

26. If the 16 bit value is 0x0100, why does the first byte contain 0x00 and the second byte contain 0x10?

27. Go ahead and try a couple other formats, just to see how they appear:

32-bit Hex – *TI* Style

16-bit Unsigned Int

By changing the format, are we in any way changing the actual values in memory? Why?

When would you choose to use one format over the another? _____

28. **Close** the project.



Before going on, let your instructor know when you have reached this point.

Optional: Lab 5b - Calls in Linear Assembly

The goal is replace the `main()` C function with one written in Linear Assembly.

Files in Lab

<code>LAB5b.PJT</code>	copy from <code>LAB5a.PJT</code>
<code>GLOBALS.C</code>	create from <code>MAIN.C</code>
<code>MAIN5b.SA</code>	create from scratch
<code>DOTP.SA</code>	copy from Lab 5a folder
<code>LAB.TCF</code>	copy from Lab 5a folder
<code>LABCFG.CMD</code>	copy from Lab 5a folder
<code>PROFILER.INI</code>	copy from Lab 5a folder

Create Project

1. *COPY* the following files from `C:\op6000\labs\lab5a` to `C:\op6000\labs\lab5b`. Please make sure to *COPY* these files.

```
LAB5a.PJT
MAIN.C
DOTP.SA
LAB.TCF
LABCFG.CMD
PROFILER.INI
```

2. While still inside Windows Explorer, make sure that you are in the `C:\op6000\labs\lab5b` directory.
3. Rename the `C:\op6000\labs\lab5b\lab5a.pjt` file to `lab5b.pjt`.
4. In CCS, open `lab5b.pjt`.
5. Open `main.c` and save it as `globals.c`.

FILE → SAVE AS...

We'll edit this file in the upcoming steps.

6. Remove `main.c` from the project.
7. Create a new source file and save it to the `Lab5b` subdirectory as `main5b.sa`.
8. Add `globals.c` and `main5b.sa` to your `lab5b.pjt`.

Edit Source Files

9. Modify `globals.c` by commenting out (or deleting) the function `main()`.
Don't delete the global variable definitions, though.
10. What does our new linear assembly main function need to do?

Hint: Did you mention passing arguments in your answer above? If not, don't forget that this is an important step for calling `dotp()`. You might want to refer back to page 5-11 in your Student Notes to see how to do this.

11. Go ahead and type-in your new `main()` function into `main5b.sa`.

Build and Run

12. Build the project for `Debug` -- without optimization.

Hint: If you have trouble building your program with the linear assembly version of `main()`, here's a troubleshooting tip. The most common error is trying to pass C variables from the `main5b.sa` routine. For example you cannot use the following:

```
.call result = _dotp(a, x, count)
```

where `a`, `x`, and `count` are C variables. Remember that this is assembly code (albeit, easy assembly code). The `.call` statement requires *registers* as arguments (e.g. registers declared with `.reg`).

13. Reset and go to `main()`. Do you reach it? _____

14. Now, single-step from `main()`. Does it reach your dot-product routine? _____

15. **Save** all of your work and **close** any open projects.

Congratulations! You have called one linear assembly routine from another.

Lab 7

Using Linear Assembly, optimize your dot-product algorithm using WORD-WIDE optimization; that is, replace LDH with LDW and use both MPY and MPYH.

Lab 7

Lab 7a: (OPTIONAL) Optimize your dot-product routine using LDW in place of LDH (using Linear Assembly)

LDH
LDH
MPY
ADD

x40
⋮

→

LDW
LDW
MPY
MPYH
ADD
ADD

x20
⋮

7b. Use LDDW's and the DOTP2 instruction

7c. (OPTIONAL) Try writing the LDW version of the dot-product using Standard Assembly (rather than Linear Asm)

LAB7a.PJT DOTP.SA MAIN.C LAB.TCF LABCFG.CMD PROFILER.INI DATA.H	copy from LAB5a.PJT and rename copy from LAB5a folder copy from LAB5a folder copy from LAB5a folder copy from LAB5a folder copy from LAB5a folder provided for you
--	---

Lab 7a (If using the C64x (or above) we suggest skipping to Lab 7b)

- Using Windows Explorer, **COPY** the following files from C:\op6000\labs\lab5a to C:\op6000\labs\lab7a. Please make sure to **COPY** these files.

```
LAB5a.PJT
MAIN.C
DOTP.SA
LAB.TCF
LABCFG.CMD
PROFILER.INI
```

Note: Please make sure to **copy** the above files correctly.

- While still inside Windows Explorer, make sure that you are in the C:\op6000\labs\lab7a directory, then ...

3. Rename the **C:\op6000\labs\lab7\lab5a.pjt** file to **lab7a.pjt**.
4. Now go over to Code Composer Studio (or open it up if you closed it) and open **lab7a.pjt**.
5. Open **DOTP.SA** and modify it to implement the LDW optimization; i.e. load two shorts (as words) and process them both per loop.
6. Rebuild (using *Debug* options) your project and verify proper operation.
7. Profile this new routine: _____ cycles using *Debug*
8. Rebuild, Run, and Profile your code with the *Release* configuration.
How fast is your LDW based routine with optimization turned on? _____ cycles
How does this compare with DOTP.SA from Lab 5a? How about Lab 4? _____

9. What is the size of your DOTP.SA routine in Lab 7a? _____
10. **Close** the project when completed.

If you still have time left, please move on to Lab 7b.

Lab 7b

Once you've got the LDW version of the dot-product running, try implementing the algorithm on the 'C64x or C64x+ using the DOTP2 instruction. Like Lab 5a, implement this lab in linear assembly.

Change to C64x Simulator

67

1. Since this is a 'C64x based lab, we need to reconfigure the CCS setup. Launch CCS Setup:

67+

File → Launch Setup

2. When setup opens, the "Import Configuration" window appears. Follow the procedure found at the start of Lab 2. These are the basic steps:
 - Clear the previous system configuration (button on right).
 - Scroll through the list to find *C64xx CPU Cycle Accurate Sim, Ltl Endian*. Select this item and click **Import**
 - Quit and save CCS Setup. When asked, allow CCS Setup to restart CCS for you.

Create Project

3. Using Windows Explorer, COPY the following files from **C:\op6000\labs\lab7a** to **C:\op6000\labs\lab7b**. Please make sure to **COPY** these files.

LAB7a.PJT
 MAIN.C
 DOTP.SA
 PROFILER.INI
 LAB.TCF

If you were doing 67xx labs don't copy this file, we'll create it.

LABCFG.CMD

If you were doing 67xx labs don't copy this file, we'll create it.

4. Rename the LAB7a.PJT to LAB7b.PJT. Save it in the C:\op6000\labs\lab7b folder. You may need to remove the floating-point TCF and CMD files from the project when you first start CCS.

67

5. Create LAB.TCF file based on the ti.platforms.sim64xx.tci template.

67+

FILE → NEW → DSP/BIOS Configuration...

6. Choose the ti.platforms.sim64xx.tci template file.
7. Modify the RTDX Mode and Dynamic Memory Heaps (as you did for the C67.tcf file in Lab 2 – Step 9).
8. Save the new .tcf file as lab.tcf to the C:\op6000\labs\lab7b folder.
9. Open up the project build options and change the target processor by using the -mv6400 switch.
10. Select LAB.TCF in the Project Explorer window. Click on: **Project → Compile File** to create the linker command file.

11. If they are not already included, add the following files to your LAB7b.PJT project:
LAB.TCF, LABCFG.CMD

12. Replace the MPY/MPYH instructions in DOTP.SA with DOTP2 instructions.

_____ How many results are you calculating per loop when using DOTP2?

13. Build using *Debug* and validate your answer.

14. Use the *Release* configuration to rebuild and profile with optimization.

15. Rebuild with full optimization and benchmark your new routine: _____ cycles.

How does this compare with DOTP.SA from Lab 7a? _____

16. How does it compare with earlier results?

17. Are these the results that you expected using DOTP2? Can you think of any other way to improve this benchmark? If so, give it a try.

18. **Save** all of your work and **close** any open projects.

67
67+

19. When finished with Lab 7b, change the CCS setup back to the original floating-point simulator that you were using. Make sure you reset the CCS Environment Options as explained in Lab 2 so your program will automatically build and load.

Optional - Lab 7c

Try implementing the most optimized dotp procedure that you can in standard assembly (.ASM) rather than Linear Assembly (.SA). Remember, linear assembly uses special directives and labels that are not available to standard assembly. In standard assembly you must use registers, define memory setups, and include nops where necessary.

Procedure

1. Copy the following files from **C:\op6000\labs\lab7a** to **C:\op6000\labs\lab7c**. Please make sure to **COPY** these files.

```
LAB7a.PJT
MAIN.C
LAB.TCF
LABCFG.CMD
PROFILER.INI
```

2. Create the file **DOTP_ASM.ASM**.
3. Open **LAB7C.PJT** and remove any files that CCS complains about.
4. Add the **DOTP_ASM.ASM** file to your **LAB7c.PJT** project.
5. Edit **DOTP_ASM.ASM** to implement the optimizations that you feel will give you the most optimized code.
6. Rebuild using **Debug** and validate your answer.
7. Benchmark your routine and fill in the following table.

Dot-product Version	Cycles
Lab 4 (standard assembly, LDH)	
Lab 7a (linear assembly, using LDW)	
Lab 7b (linear assembly, using DOTP2)	
Lab 7c (standard assembly, LDW)	

8. Which method seems to give you the best results? Why?

9. **Save** all of your work and **close** any open projects.

Page left intentionally blank.

Lab 9

There are three different groups of instructions for Lab 9a and Lab 9b depending on your choice of processor. You will find directions for navigating through these groups of instructions on the following page.

Lab 9a investigates the effects of *Alias Disambiguation*, *Balancing Resources* on the A and B sides of the processor, and Packed Data Optimization.

Lab 9b looks at the benefit of performing *Program Level Optimization*.

Lab 9c provides an opportunity to use the Compiler Consultant. This lab is based on a tutorial included in the Help section of Code Composer Studio. The tutorial requires using the C64x processor so it will be necessary to switch to it for this lab.

C64x

Lab9a: Go to page 9 - 4

Lab9b: Go to page 9 - 15

Lab9c: Go to page 9 - 61

C64xp

Lab9a: Go to page 9 - 18

Lab9b: Go to page 9 - 29

Lab9c: Go to page 9 - 61

C67

Lab9a: Go to page 9 - 32

Lab9b: Go to page 9 - 43

Lab9c: Go to page 9 - 61

C672x

Lab9a: Go to page 9 - 46

Lab9b: Go to page 9 - 57

Lab9c: Go to page 9 - 61

Lab 9a for 'C64x

We're going to explore using other pragmas and tools that we discussed in this chapter to maximize the efficiency of some more C code.

Save a Copy of Lab9a

Before we get started, we want to save a copy of the files that we will be working with. We want you to do this, so that you know that we aren't trying to do anything "tricky".

1. Use Windows Explorer to copy *C:\op6000\labs\lab9a* folder to *C:\op6000\labs\lab9b*.

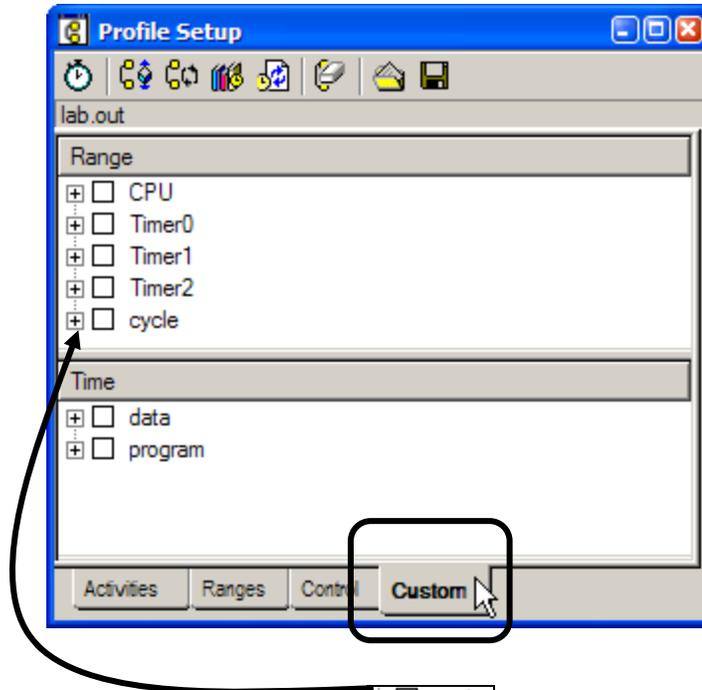
Load Project and Set Compiler Options

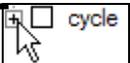
2. If Code Composer Studio is not already running, start it.
3. Open the *lab9a64* project found in folder *C:\op6000\labs\lab9a\lab9a64.pjt*.
4. Expand this project and make sure it contains the following files:
 - *wvs.c*
 - *main.c*
 - *lnk.cmd*
 - *C:\CCStudio\c6000\cgtools\lib\rts6400.lib*
5. Make sure the Build Configuration is set to *Debug*.
6. Build, load, & run your program to main (go main if not taken there automatically).

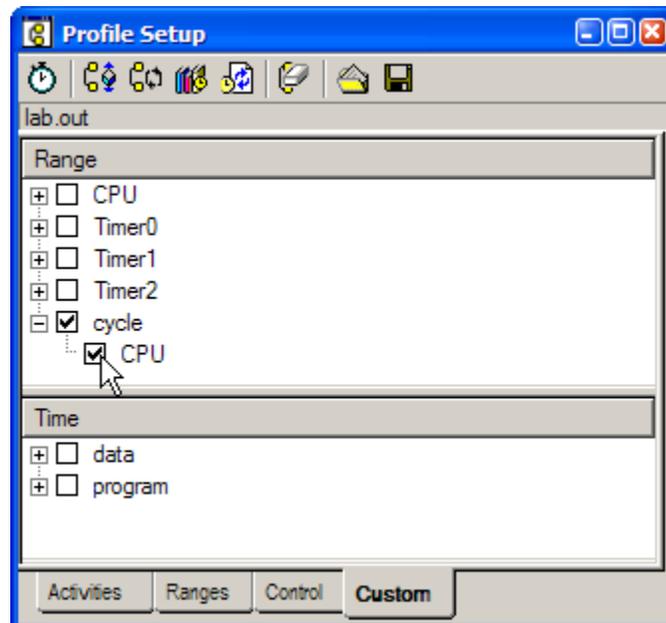
Set up Profiler

7. Open the Profile Setup window and click on the *Custom* tab at the bottom of the window.

The Custom tab allows you to configure exactly what simulation events are captured by the profiler.



8. Click on the plus sign next to the cycle entry, .
9. Click the box next to the CPU entry under cycle.



The cycle:CPU event counts the number of cycles executed by the CPU while ignoring memory and cache effects.

- Click on the Enable/Disable All Functions icon  in the Profile Setup window to **enable** the profiling of all functions.

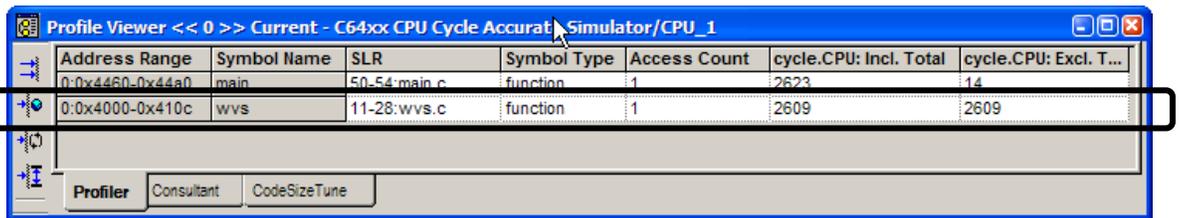
- Enable profiling by clicking on the  icon.

- Open the Profile Viewer.

Profile → Viewer

- Run the code.

- Record the Incl Total (Cycle Count) for the `wvs` function for Lab9a in the table on p. 9-17 under “No Optimization”.



Address Range	Symbol Name	SLR	Symbol Type	Access Count	cycle.CPU: Incl. Total	cycle.CPU: Excl. T...
0:0x4460-0x44a0	main	50-54:main.c	function	1	2623	14
0:0x4000-0x410c	wvs	11-28:wvs.c	function	1	2609	2609

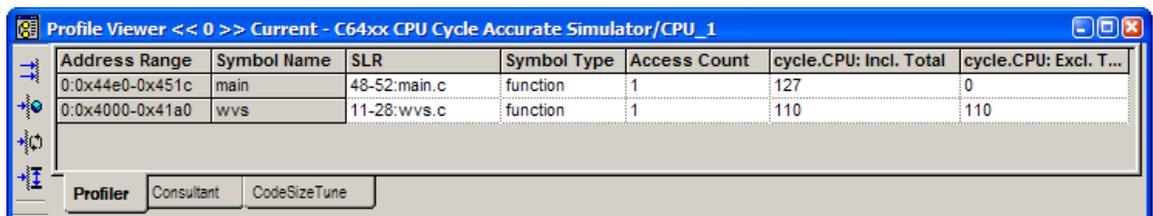
Optimizing the Code

- Open the `wvs.c` file. This C code is a vector summation of two weighted vectors. We will be using this code throughout this lab.

- Change the Build Configuration to *Release*.

- Rebuild the code.

- Run the code. Record the number of cycles from the Profile Viewer in the table on page 9-17.



Address Range	Symbol Name	SLR	Symbol Type	Access Count	cycle.CPU: Incl. Total	cycle.CPU: Excl. T...
0:0x44e0-0x451c	main	48-52:main.c	function	1	127	0
0:0x4000-0x41a0	wvs	11-28:wvs.c	function	1	110	110

19. Open the *wvs.asm* file in the *C:\op6000\labs\lab9a\Release_64* directory.

This file is available because of the `-k` compiler option we set.

The *.asm* file contains software pipeline information. Scroll down through this file and find the Software Pipeline Information Comments. Do you see “ Searching for a software pipeline schedule at ... `ii = 11`”?

`ii=11` stands for iteration interval equals 11. This implies that each iteration of the loop takes eleven cycles, or that there are eleven cycles per loop iteration.

Record the number of "Cycles per Iteration" for *Optimize* in the table on page 9-17.

With 8 resources available every cycle on such a small loop, we would expect this loop to do better than this.

Q. Why did the loop start searching for a software pipeline at `ii=11` (for an 11 cycle loop)?

A. Because 3 pointers are used in this function (`xptr`, `yptr`, & `zptr`). If all the pointers point to a different location in memory, there is no dependency. However since all three pointers are passed into `lab9a`, there is no way for the compiler to be sure they don't alias, or point to the same location. This is a memory alias disambiguation problem. The compiler must be conservative to guarantee correct execution. Unfortunately, the requirement for the compiler to be conservative can have dire effects on the performance of your code.

We know from looking at the calling function in `main`, that in fact, these pointers all point to separate arrays in memory. However, from the compiler's local view of `lab9a`, this information is not available.

Q. How can you pass more information to the compiler to improve its performance?

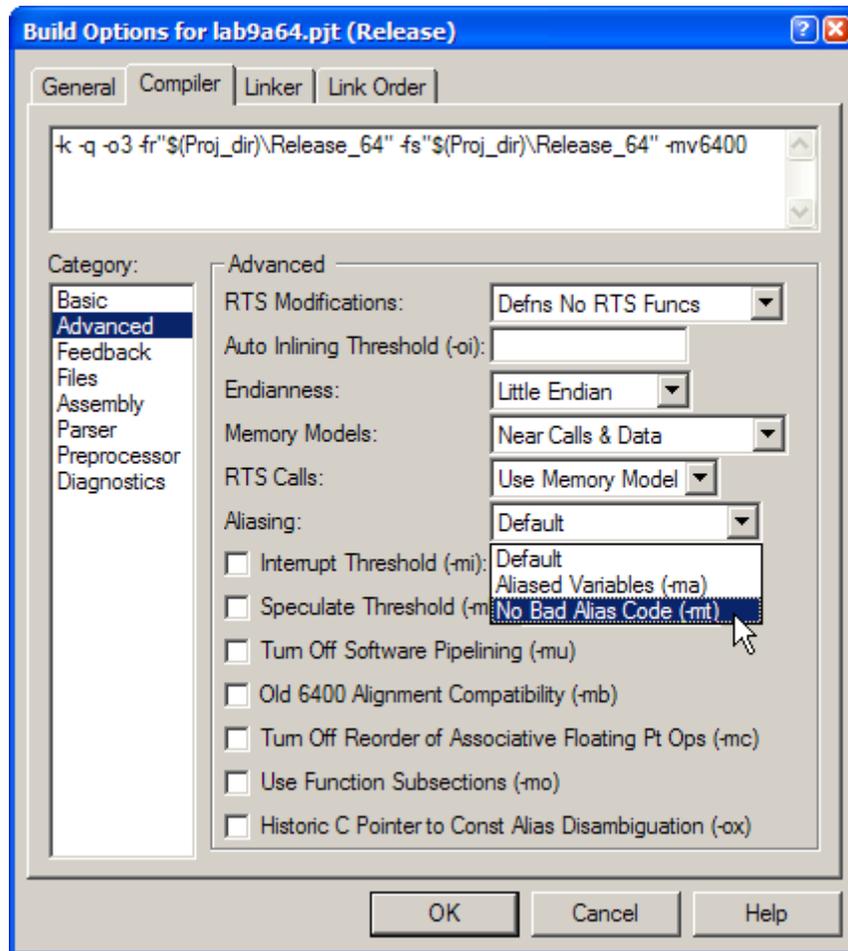
Solving Alias Disambiguation Problems

A special option in the compiler, `-mt`, tells the compiler to ignore alias disambiguation problems like the one described at the beginning of this lab. Try using this option to rebuild the original lab and look at the results.

20. Turn on the `-mt` compiler option.

Project → *Build Options* → *Compiler* tab → *Advanced* Category.

Pull down *Aliasing* menu and select *No Bad Alias Code (-mt)* and then press OK.



21. Rebuild, Reload, & Go Main.

22. Hide the Build window.

23. Run the code.

24. Record the Incl Total (Cycle Count) of `wvs()` in the table on p. 9-17 under “`-mt` options”.

25. Record the number of "Cycles per Iteration" for *Release* in the table on page 9-17 under "-mt options".

Open the *wvs.asm* file. Under the Global File Parameters, the following line implies that `-mt` was used:

```
;* Memory Aliases : Presume not aliases (optimistic)
```

The compiler now knows that there aren't supposed to be any aliases in this project. What if some functions had aliases and others didn't? We need something that is a little more specific.

Adding a Restrict Qualifier

26. Open the Build Options for the *Release* configuration and remove the `-mt` option.
27. Now open the *wvs.c* file.

Because we know that `xptr` & `yptr` are actually separate arrays in memory from `w_sum`, we can declare that nothing else points to these objects. To do this, we must add a `restrict` type qualifier.

Modify the function definition to be as follows:

```
void wvs(short * restrict xptr, short * restrict yptr, short *zptr, short *w_sum, int N)
```

Save and close *wvs.c*.

28. We must also modify *main.c* to reflect this function definition. Open *main.c* and change the external statement to read:

```
extern void wvs(short * restrict, short * restrict, short *, short *, int);
```

Save and close *main.c*.

29. Rebuild, Reload, & Go Main.
30. Run the code.
31. Record the Incl Total (Cycle Count) of lab9a in the table on p. 9-17 under "Restrict Qualifier".
32. Open *wvs.asm* and scroll down to the Software Pipeline Information.

Q. How many cycles is the Software Pipelined Loop now? _____

Now the loop carried dependency bound is zero. By simply passing more information to the compiler, we allowed it to improve an 11-cycle loop to a 2-cycle loop.

Record the number of cycles per iteration in the table on page 9-17.

33. Scroll down to the Software Pipeline Information until you see

```
;* Loop Carried Dependency Bound (^) : 0
:* ii = 2 Schedule found with 6 iterations in parallel
```

This indicates that a 2-cycle loop was found, which is similar to using the `-mt` compiler option. However, the `restrict` keyword is function specific and gives the user more control.

Q. How can we optimize our code more?

Balance the Resources

34. Let's analyze the feedback to determine what improvements could be made. Open `wvs.asm`.

Scroll down to the Software Pipeline Information. The first iteration interval (ii) attempted was two cycles because the Partitioned Resource Bound is two. We can see the reason for this if we look below at the `.D` unit and the `.T` address paths. This loop requires two loads (from `xptr` and `yptr`) and one store (to `w_sum`) for each iteration of the loop.

Each memory access requires a `.D` unit for address calculation, and a `.T` address path to send the address out to memory. Because the C6000 has two `.D` units and two `.T` address paths available on any given cycle (A side and B side), the compiler must partition at least two of the operations on one side (the A side). That means that these operations are the bottleneck in resources (highlighted with an `*`) and are the limiting factor in the Partitioned Resource Bound. The feedback in `wvs.asm` shows that there is an imbalance in resources between A and B side due, in this case, to an odd number of operations being mapped to two sides of the machine.

Q. Is it possible to improve the balance of resources?

A. One way to balance an odd number of operations is to unroll the loop. Now, instead of three memory accesses, you will have six, which is an even number. You can only do this if you know that the loop counter is a multiple of two; otherwise, you will incorrectly execute too few or too many iterations. In main, `LOOPCOUNT` is defined to be 40, which is a multiple of two, so you are able to unroll the loop.

Q. Why did the compiler not unroll the loop?

A. In the limited scope of `wvs.c`, the loop counter is passed as a parameter to the function. Therefore, it might be any value from this limited view of the function. To improve this scope you must pass more information to the compiler. One way of doing this is with a `MUST_ITERATE` pragma.

35. Open *wvs.c* and insert a `MUST_ITERATE` pragma above the for loop that says the loop always iterates in multiples of 4s but always iterates at least 40 times. It should look like this:

```
#pragma MUST_ITERATE ( 40, , 4 );
```

Unrolling a loop can incur some minor overhead in loop setup. The compiler does not unroll loops with small loop counts because unrolling may not reduce the overall cycle count. If the compiler does not know what the minimum value of the loop counter is, it will not automatically unroll the loop. Again, this is information the compiler needs but does not have in the local scope of *wvs.c*. You know that `LOOPCOUNT` is set to 40, so you can tell the compiler that `N` is greater than some minimum value. The `MUST_ITERATE` pragma passes these two valuable pieces of information.

36. Save & close *wvs.c*. Close any open files. Leave the Profile Viewer open.

37. Rebuild, Reload, & Go Main.

38. Run the code.

39. Record the Incl Total (Cycle Count) of *wvs()* in the table on p. 9-17 under “`MUST_ITERATE`”.

Remember, no code was altered in this lab. Only additional information was passed via the `MUST_ITERATE` pragma. We simply guarantee the compiler that the trip count (in this case the trip count is `N`) is a multiple of four and that the trip count is greater than or equal to 40. The first argument for `MUST_ITERATE` is the minimum number of times the loop will iterate. The second argument is the maximum number of times the loop will iterate. The trip count must be evenly divisible by the third argument, the factor.

Note: Choose a trip count large enough to tell the compiler that it is more efficient to unroll the loop. Always specify the largest minimum trip count that is safe.

Open the *wvs.asm* file. Notice the following things in the Software Pipeline information feedback:

Loop Unroll Multiple:4x

This loop has been unrolled by a factor of four.

ii=5 Schedule found with 3 iterations in parallel

You can tell by looking at the .D units and .T address paths that this 5-cycle loop comes after the loop has been unrolled because the resources show a total of six memory accesses. Therefore, our new effective loop iteration interval is 5/4 or 1.25 cycles. The "3 iterations in parallel" means that the loop is working on 3 different results at the same time. This doesn't have any bearing on our performance.

Minimum Safe Trip Count: 2 (after unrolling)

This is because we specify the count of the original loop to be greater than or equal to forty and a multiple of four and after unrolling, the loop count has to be even.

Therefore, by passing information without modifying the loop code, compiler performance improves from a 11-cycle loop to 2 cycles and now to 1.25 cycles.

Record the number of cycles per iteration in the table on page 9-17.

Q. Is this the lower limit?

Packed Data Optimization Increases Memory Bandwidth

The last optimization produced a 5-cycle loop that performed 4 iterations of the original *vector sum of two weighted vectors*. This means that each iteration of our loop now performs eight memory accesses, eight multiplies, four adds, four shift operations, a decrement for the loop counter, and a branch. You can see this in the feedback of *wvs.c*.

40. Open *wvs.asm*.

The six memory accesses appear as .D and .T units. The four multiplies appear as .M units. The two shifts and the branch show up as .s units. The decrement and the two adds appear as .LSD units.

By analyzing this part of the feedback, we can see that resources are most limited by the memory accesses; hence, the reason for an asterisk highlighting the .D units and .T address paths.

Q. Does this mean that we cannot make the loop operate any faster?

A. Further insight into the 'C6000 architecture is necessary here.

The C62x fixed-point device loads and/or stores 32 bits every cycle. In addition, the C67x floating-point and the C64x fixed-point device loads two 64-bit values each cycle. In our example, we load four 16-bit values and store two 16-bit values every five cycles. This means we only use 16 bits of memory access every cycle. Because this is a resource bottleneck in our loop, increasing the memory access bandwidth further improves the performance of our loop.

In the unrolled loop generated from the `MUST_ITERATE` pragma, we load four consecutive 16-bit elements with `LDNDWs` from both the `xptr` and `yptr` array.

Q. Why not use an aligned `LDDW` to load one 64-bit element, with the resulting register pair load containing the first element in one-half of the lower 32-bit register and the second element in the other half and the third and fourth in the other register?

A. This is called Packed Data optimization. Four 16-bit loads are effectively performed by one single 64-bit load instruction.

Q. Why didn't the compiler do this automatically in the `MUST_ITERATE` code?

A. Again, the answer lies in the amount of information the compiler has access to from the local scope of the program.

In order to perform a `LDDW` (64-bit load) on the C64x cores, the address must be aligned to a double word-address; otherwise, incorrect data would be loaded. An address is double word-aligned if the lower three bits of the address are zero. Unfortunately, in our example, the `wvs()` function does not have knowledge as to the `xptr` and `yptr` address values. Therefore, the compiler is forced to be conservative and assume that these pointers might not be aligned. Once again, we can pass more information to the compiler, this time via the `_nassert` statement.

41. Open *wvs.c*. Add the following piece of code on the first line:

```
#define DWORD_ALIGNED(x) (_nassert(((int) (x) & 0x7) == 0))
```

Also add `DWORD_ALIGNED` definitions for `xptr` & `yptr` after the short `w1,w2;` statement.

```
DWORD_ALIGNED(xptr);  
DWORD_ALIGNED(yptr);
```

By asserting that `xptr` and `yptr` addresses “anded” with `0x7` are equal to zero, the compiler knows that they are double word aligned. This means the compiler can perform LDDW and packed data optimization on these memory accesses.

Hint: If you need would like a refresher on how `DATA_ALIGN` and `_nassert()` are working together to produce optimal code, refer back to the material on page 9-19 of the Student Notes.

42. Save and close *wvs.c*.
43. Rebuild, Reload, & Go Main.
44. Run the code.
45. Record the Incl Total (Cycle Count) of *wvs()* in the table on p. 9-17 under “`_nassert (double word-aligned)`”.
46. Open *wvs.asm*.
Success! The compiler has fully optimized this loop. You can now achieve 4 iterations of the loop every three cycles for **.75** cycle per iteration throughout.
The `.D` and `.T` resources now show 3 (two LDDWs and four STHs for four iterations of the loop).
47. Close *lab9a64.pjt* and any files associated with it.

Move on to Lab9b for the ‘C64x

Lab9b for 'C64x

Program Level Optimization

Now we have learned how to pass information to the compiler. This increased the amount of information visible to the compiler from the local scope of each function.

Q. Is this necessary in all cases?

A. No, not in all cases. First, if this information already resides locally inside the function, the compiler has visibility here and `restrict` and `MUST_ITERATE` statements are not usually necessary. For example, if `xptr` and `yptr` are declared as local arrays, the compiler does not assume a dependency with `w_sum`. If the loop count is defined in the function or if the loop is simply described to be from one to forty, the `MUST_ITERATE` pragma is not necessary.

Second, even if this type of information is not declared locally, the compiler can still have access to it in an automated way by giving it a program level view.

The C6000 compiler provides two valuable switches, which enable program level optimization: **-pm** and **-op2**. When these two options are used together, the compiler can automatically extract all of the information we passed in Lab9a.

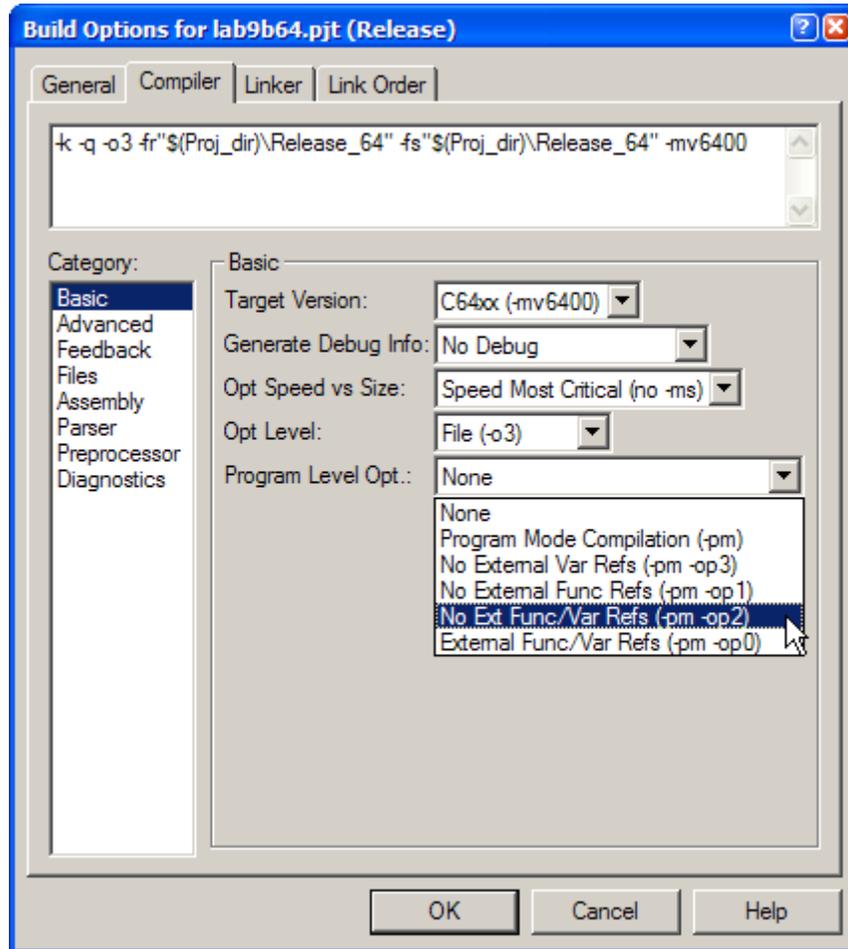
1. Make sure that all files from lab9a are closed.
2. We are now going to use the copy of the Lab 9a files that you made at the beginning of that lab. Navigate to the `C:\op6000\labs\lab9b` folder.
3. Rename the `C:\op6000\labs\lab9b\lab9a64.pjt` to `C:\op6000\labs\lab9b\lab9b64.pjt`.
4. Open the `lab9b64.pjt` file.
5. Change to the *Release* configuration.

6. To tell the compiler to use program level optimization (`-pm` and `-op2`), enter the following

Project → Build Options → Compiler tab → Basic Category

Open *Program Level Optimization* drop-down menu.

Select **No External Func/Var Refs**. Press OK



This adds `-pm` and `-op2` to the command line.

7. Rebuild, Reload, & Go Main.
 8. If the Profile Viewer closed with *lab9a64.pjt*, reopen it.

Profile → Viewer

9. Run the code.
 10. Record the Incl Total (Cycle Count) of `wvs()` in the table on page 9-17 under “Program Level Opt (`-pm` & `-op2`)”.

11. Now try to open *wvs.asm* to get the Cycles per Iteration. Can you find the file where it should be? _____

Note: Remember that `-pm` combines all of your files into one file before compiling. Therefore there is not a *wvs.asm* file.

12. Open *main.asm* in the `C:\op6000\labs\lab9b\Release_64` directory and find the *wvs()* function. Use the Software Pipelined Information embedded in the file to record the "Cycles per Iteration" under Program Level Opt.
13. Compare the improvements in the following table:

Optimization Comparison Table for the 'C64x

Optimization Method	Incl Tot/Cycle Count	Cycles per Iteration
No Optimization		N/A
Release		
-mt option		
Restrict Qualifier		
MUST_ITERATE		
_nassert (double word-aligned)		
Program Level Opt (-pm & -op2)		

14. **Save** your work and **close** the Lab 9b project.

End of Lab9b for 'C64x, move on to Lab9c on page 9 - 61

Lab 9a for 'C64x+

We're going to explore using other pragmas and tools that we discussed in this chapter to maximize the efficiency of some more C code.

Save a Copy of Lab9a

Before we get started, we want to save a copy of the files that we will be working with. We want you to do this, so that you know that we aren't trying to do anything "tricky".

1. Use Windows Explorer to copy *C:\op6000\labs\lab9a* folder to *C:\op6000\labs\lab9b*.

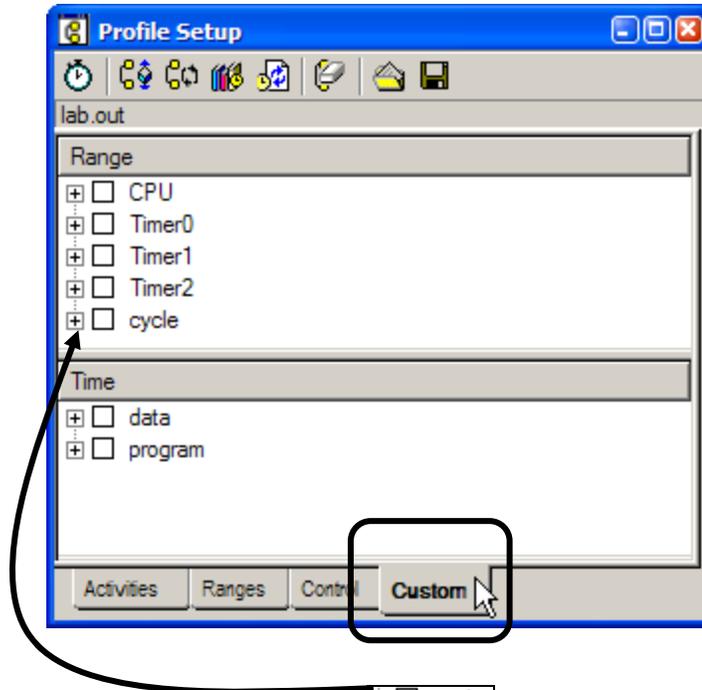
Load Project and Set Compiler Options

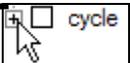
2. If Code Composer Studio is not already running, start it.
3. Open the *lab9a64p* project found in folder *C:\op6000\labs\lab9a\lab9a64p.pjt*.
4. Expand this project and make sure it contains the following files:
 - wvs.c
 - main.c
 - lnk.cmd
 - C:\CCStudio\c6000\cgtools\lib\rts64plus.lib
5. Make sure the Build Configuration is set to *Debug*.
6. Build, load, & run your program to main (go main if not taken there automatically).

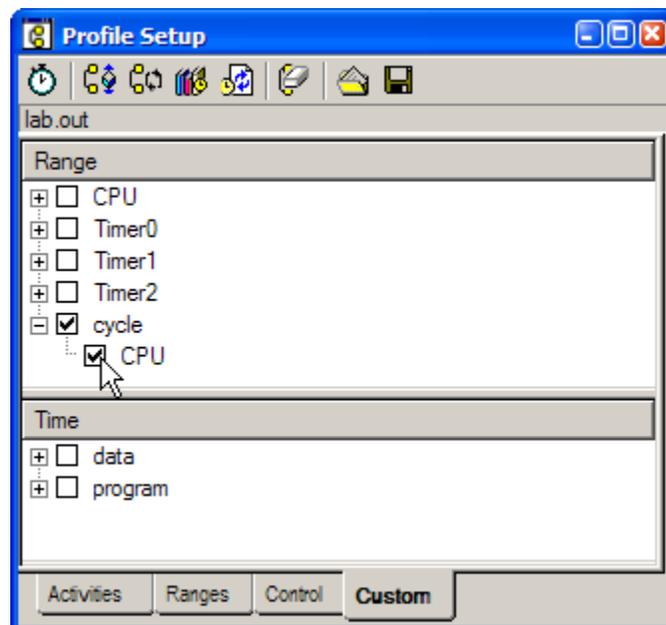
Set up Profiler

7. Open the Profile Setup window and click on the *Custom* tab at the bottom of the window.

The Custom tab allows you to configure exactly what simulation events are captured by the profiler.



8. Click on the plus sign next to the cycle entry, .
9. Click the box next to the CPU entry under cycle.



The cycle:CPU event counts the number of cycles executed by the CPU while ignoring memory and cache effects.

- Click on the Enable/Disable All Functions icon  in the Profile Setup window to **enable** the profiling of all functions.

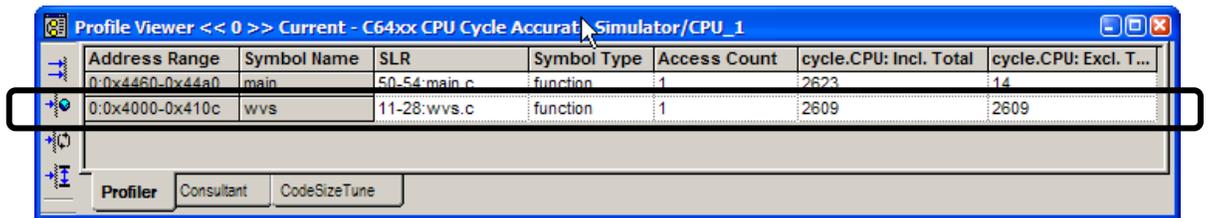
- Enable profiling by clicking on the  icon.

- Open the Profile Viewer.

Profile → Viewer

- Run the code.

- Record the Incl Total (Cycle Count) for the `wvs` function for Lab9a in the table on p. 9-31 under “No Optimization”.



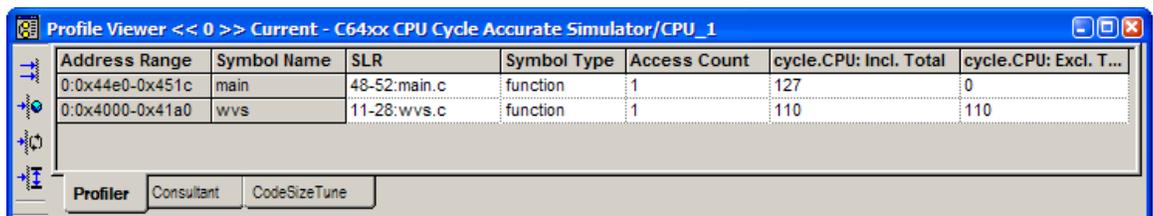
Optimizing the Code

- Open the `wvs.c` file. This C code is a vector summation of two weighted vectors. We will be using this code throughout this lab.

- Change the Build Configuration to *Release*.

- Rebuild the code.

- Run the code. Record the number of cycles from the Profile Viewer in the table on page 9-31.



19. Open the *wvs.asm* file in the *C:\op6000\labs\lab9a\Release_64p* directory.

Note

The version of the compiler we're using actually performs better than discussed here. So, you may see an "ii=2".

This file is available because of the `-k` compiler option we set.

The *.asm* file contains software pipeline information. Scroll down through this file and find the Software Pipeline Information Comments. Do you see “ Searching for a software pipeline schedule at ... ii = 11”?

ii=11 stands for iteration interval equals 11. This implies that each iteration of the loop takes eleven cycles, or that there are eleven cycles per loop iteration.

Record the number of "Cycles per Iteration" for *Optimize* in the table on page 9-31.

With 8 resources available every cycle on such a small loop, we would expect this loop to do better than this.

Q. Why did the loop start searching for a software pipeline at ii=11 (for an 11 cycle loop)?

A. Because 3 pointers are used in this function (*xptr*, *yptr*, & *zptr*). If all the pointers point to a different location in memory, there is no dependency. However since all three pointers are passed into *lab9a*, there is no way for the compiler to be sure they don't alias, or point to the same location. This is a memory alias disambiguation problem. The compiler must be conservative to guarantee correct execution. Unfortunately, the requirement for the compiler to be conservative can have dire effects on the performance of your code.

We know from looking at the calling function in *main*, that in fact, these pointers all point to separate arrays in memory. However, from the compiler's local view of *lab9a*, this information is not available.

Q. How can you pass more information to the compiler to improve its performance?

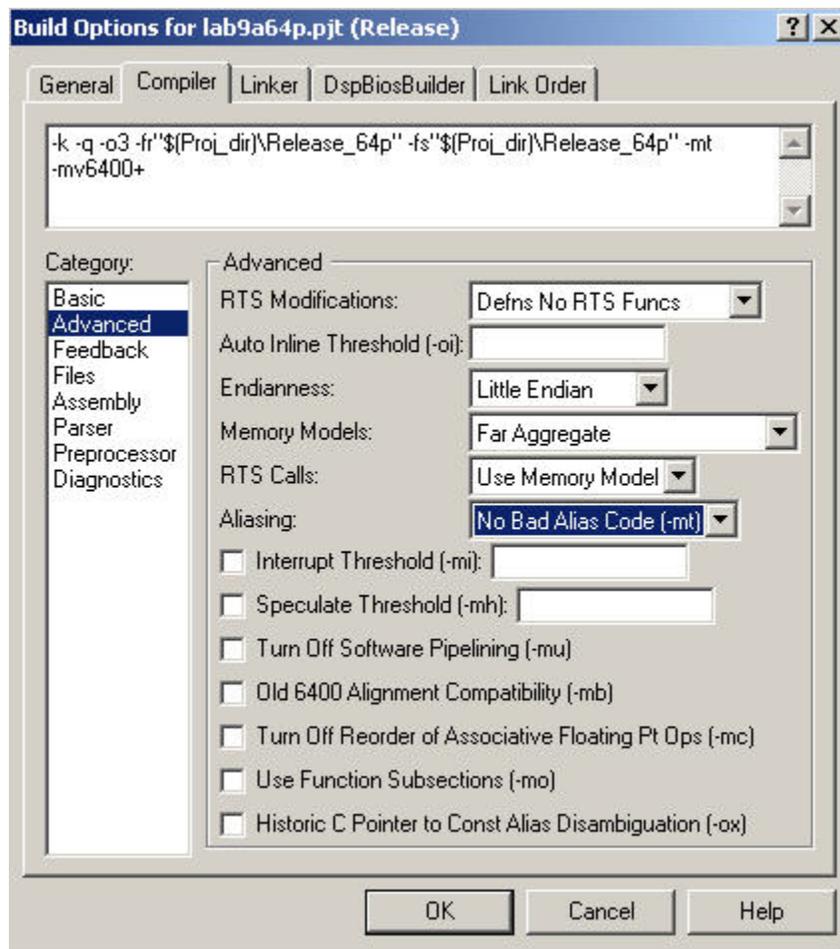
Solving Alias Disambiguation Problems

A special option in the compiler, `-mt`, tells the compiler to ignore alias disambiguation problems like the one described at the beginning of this lab. Try using this option to rebuild the original lab and look at the results.

20. Turn on the `-mt` compiler option.

Project → *Build Options* → *Compiler* tab → *Advanced* Category.

Pull down *Aliasing* menu and select *No Bad Alias Code (-mt)* and then press OK.



21. Rebuild, Reload, & Go Main.

22. Hide the Build window.

23. Run the code.

24. Record the Incl Total (Cycle Count) of `wvs()` in the table on p. 9-31 under “`-mt` options”.

25. Record the number of "Cycles per Iteration" for *Release* in the table on page 9-31 under "-mt options".

Open the *wvs.asm* file. Under the Global File Parameters, the following line implies that `-mt` was used:

```
;* Memory Aliases : Presume not aliases (optimistic)
```

The compiler now knows that there aren't supposed to be any aliases in this project. What if some functions had aliases and others didn't? We need something that is a little more specific.

Adding a Restrict Qualifier

26. Open the Build Options for the *Release* configuration and remove the `-mt` option.
27. Now open the *wvs.c* file.

Because we know that `xptr` & `yptr` are actually separate arrays in memory from `w_sum`, we can declare that nothing else points to these objects. To do this, we must add a `restrict` type qualifier.

Modify the function definition to be as follows:

```
void wvs(short * restrict xptr, short * restrict yptr, short *zptr, short *w_sum, int N)
```

Save and close *wvs.c*.

28. We must also modify *main.c* to reflect this function definition. Open *main.c* and change the external statement to read:

```
extern void wvs(short * restrict, short * restrict, short *, short *, int);
```

Save and close *main.c*.

29. Rebuild, Reload, & Go Main.
30. Run the code.
31. Record the Incl Total (Cycle Count) of lab9a in the table on p. 9-31 under "Restrict Qualifier".
32. Open *wvs.asm* and scroll down to the Software Pipeline Information.

Q. How many cycles is the Software Pipelined Loop now? _____

Now the loop carried dependency bound is zero. By simply passing more information to the compiler, we allowed it to improve an 11-cycle loop to a 2-cycle loop.

Record the number of cycles per iteration in the table on page 9-31.

33. Scroll down to the Software Pipeline Information until you see

```
;* Loop Carried Dependency Bound (^) : 0
:* ii = 2 Schedule found with 6 iterations in parallel
```

This indicates that a 2-cycle loop was found, which is similar to using the `-mt` compiler option. However, the `restrict` keyword is function specific and gives the user more control.

Q. How can we optimize our code more?

Balance the Resources

34. Let's analyze the feedback to determine what improvements could be made. Open `wvs.asm`.

Scroll down to the Software Pipeline Information. The first iteration interval (ii) attempted was two cycles because the Partitioned Resource Bound is two. We can see the reason for this if we look below at the `.D` unit and the `.T` address paths. This loop requires two loads (from `xptr` and `yptr`) and one store (to `w_sum`) for each iteration of the loop.

Each memory access requires a `.D` unit for address calculation, and a `.T` address path to send the address out to memory. Because the C6000 has two `.D` units and two `.T` address paths available on any given cycle (A side and B side), the compiler must partition at least two of the operations on one side (the A side). That means that these operations are the bottleneck in resources (highlighted with an `*`) and are the limiting factor in the Partitioned Resource Bound. The feedback in `wvs.asm` shows that there is an imbalance in resources between A and B side due, in this case, to an odd number of operations being mapped to two sides of the machine.

Q. Is it possible to improve the balance of resources?

A. One way to balance an odd number of operations is to unroll the loop. Now, instead of three memory accesses, you will have six, which is an even number. You can only do this if you know that the loop counter is a multiple of two; otherwise, you will incorrectly execute too few or too many iterations. In main, `LOOPCOUNT` is defined to be 40, which is a multiple of two, so you are able to unroll the loop.

Q. Why did the compiler not unroll the loop?

A. In the limited scope of `wvs.c`, the loop counter is passed as a parameter to the function. Therefore, it might be any value from this limited view of the function. To improve this scope you must pass more information to the compiler. One way of doing this is with a `MUST_ITERATE` pragma.

35. Open *wvs.c* and insert a `MUST_ITERATE` pragma above the for loop that says the loop always iterates in multiples of 4s but always iterates at least 40 times. It should look like this:

```
#pragma MUST_ITERATE ( 40, , 4 );
```

Unrolling a loop can incur some minor overhead in loop setup. The compiler does not unroll loops with small loop counts because unrolling may not reduce the overall cycle count. If the compiler does not know what the minimum value of the loop counter is, it will not automatically unroll the loop. Again, this is information the compiler needs but does not have in the local scope of *wvs.c*. You know that `LOOPCOUNT` is set to 40, so you can tell the compiler that `N` is greater than some minimum value. The `MUST_ITERATE` pragma passes these two valuable pieces of information.

36. Save & close *wvs.c*. Close any open files. Leave the Profile Viewer open.

37. Rebuild, Reload, & Go Main.

38. Run the code.

39. Record the Incl Total (Cycle Count) of *wvs()* in the table on p. 9-31 under “`MUST_ITERATE`”.

Remember, no code was altered in this lab. Only additional information was passed via the `MUST_ITERATE` pragma. We simply guarantee the compiler that the trip count (in this case the trip count is `N`) is a multiple of four and that the trip count is greater than or equal to 40. The first argument for `MUST_ITERATE` is the minimum number of times the loop will iterate. The second argument is the maximum number of times the loop will iterate. The trip count must be evenly divisible by the third argument, the factor.

Note: Choose a trip count large enough to tell the compiler that it is more efficient to unroll the loop. Always specify the largest minimum trip count that is safe.

Open the *wvs.asm* file. Notice the following things in the Software Pipeline information feedback:

Loop Unroll Multiple:4x

This loop has been unrolled by a factor of four.

ii=5 Schedule found with 3 iterations in parallel

You can tell by looking at the .D units and .T address paths that this 5-cycle loop comes after the loop has been unrolled because the resources show a total of six memory accesses. Therefore, our new effective loop iteration interval is 5/4 or 1.25 cycles. The "3 iterations in parallel" means that the loop is working on 3 different results at the same time. This doesn't have any bearing on our performance.

Minimum Safe Trip Count: 2 (after unrolling)

This is because we specify the count of the original loop to be greater than or equal to forty and a multiple of four and after unrolling, the loop count has to be even.

Therefore, by passing information without modifying the loop code, compiler performance improves from a 11-cycle loop to 2 cycles and now to 1.25 cycles.

Record the number of cycles per iteration in the table on page 9-31.

Q. Is this the lower limit?

Packed Data Optimization Increases Memory Bandwidth

The last optimization produced a 5-cycle loop that performed 4 iterations of the original *vector sum of two weighted vectors*. This means that each iteration of our loop now performs eight memory accesses, eight multiplies, four adds, four shift operations, a decrement for the loop counter, and a branch. You can see this in the feedback of *wvs.c*.

40. Open *wvs.asm*.

The six memory accesses appear as .D and .T units. The four multiplies appear as .M units. The two shifts and the branch show up as .s units. The decrement and the two adds appear as .LSD units.

By analyzing this part of the feedback, we can see that resources are most limited by the memory accesses; hence, the reason for an asterisk highlighting the .D units and .T address paths.

Q. Does this mean that we cannot make the loop operate any faster?

A. Further insight into the 'C6000 architecture is necessary here.

The C62x fixed-point device loads and/or stores 32 bits every cycle. In addition, the C67x floating-point and the C64x fixed-point device loads two 64-bit values each cycle. In our example, we load four 16-bit values and store two 16-bit values every five cycles. This means we only use 16 bits of memory access every cycle. Because this is a resource bottleneck in our loop, increasing the memory access bandwidth further improves the performance of our loop.

In the unrolled loop generated from the `MUST_ITERATE` pragma, we load four consecutive 16-bit elements with `LDNDWs` from both the `xptr` and `yptr` array.

Q. Why not use an aligned `LDDW` to load one 64-bit element, with the resulting register pair load containing the first element in one-half of the lower 32-bit register and the second element in the other half and the third and fourth in the other register?

A. This is called Packed Data optimization. Four 16-bit loads are effectively performed by one single 64-bit load instruction.

Q. Why didn't the compiler do this automatically in the `MUST_ITERATE` code?

A. Again, the answer lies in the amount of information the compiler has access to from the local scope of the program.

In order to perform a `LDDW` (64-bit load) on the C64x cores, the address must be aligned to a double word-address; otherwise, incorrect data would be loaded. An address is double word-aligned if the lower three bits of the address are zero. Unfortunately, in our example, the `wvs()` function does not have knowledge as to the `xptr` and `yptr` address values. Therefore, the compiler is forced to be conservative and assume that these pointers might not be aligned. Once again, we can pass more information to the compiler, this time via the `_nassert` statement.

41. Open *wvs.c*. Add the following piece of code on the first line:

```
#define DWORD_ALIGNED(x) (_nassert(((int) (x) & 0x7) == 0))
```

Also add `DWORD_ALIGNED` definitions for `xptr` & `yptr` after the short `w1,w2;` statement.

```
DWORD_ALIGNED(xptr);  
DWORD_ALIGNED(yptr);
```

By asserting that `xptr` and `yptr` addresses “anded” with `0x7` are equal to zero, the compiler knows that they are double word aligned. This means the compiler can perform LDDW and packed data optimization on these memory accesses.

Hint: If you need would like a refresher on how `DATA_ALIGN` and `_nassert()` are working together to produce optimal code, refer back to the material on page 9-19 in the Student Notes.

42. Save and close *wvs.c*.
43. Rebuild, Reload, & Go Main.
44. Run the code.
45. Record the Incl Total (Cycle Count) of *wvs()* in the table on p. 9-31 under “_nassert (double word-aligned)”.
46. Open *wvs.asm*.
- Success! The compiler has fully optimized this loop. You can now achieve 4 iterations of the loop every three cycles for **.75** cycle per iteration throughout.
- The .D and .T resources now show 3 (two LDDWs and four STHs for four iterations of the loop).
47. Close *lab9a64p.pjt* and any files associated with it.

Move on to Lab9b for the ‘C64xp

Lab9b for 'C64xp

Program Level Optimization

Now we have learned how to pass information to the compiler. This increased the amount of information visible to the compiler from the local scope of each function.

Q. Is this necessary in all cases?

A. No, not in all cases. First, if this information already resides locally inside the function, the compiler has visibility here and `restrict` and `MUST_ITERATE` statements are not usually necessary. For example, if `xptr` and `yptr` are declared as local arrays, the compiler does not assume a dependency with `w_sum`. If the loop count is defined in the function or if the loop is simply described to be from one to forty, the `MUST_ITERATE` pragma is not necessary.

Second, even if this type of information is not declared locally, the compiler can still have access to it in an automated way by giving it a program level view.

The C6000 compiler provides two valuable switches, which enable program level optimization: **-pm** and **-op2**. When these two options are used together, the compiler can automatically extract all of the information we passed in Lab9a.

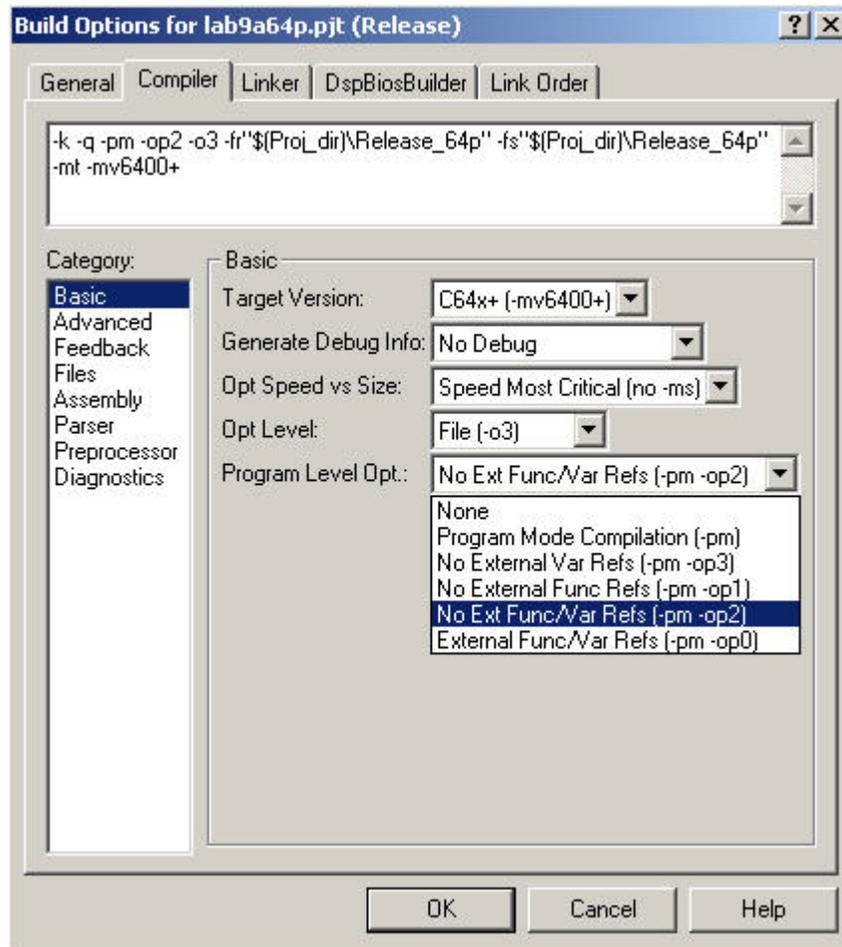
1. Make sure that all files from lab9a are closed.
2. We are now going to use the copy of the Lab 9a files that you made at the beginning of that lab. Navigate to the `C:\op6000\labs\lab9b` folder.
3. Rename the `C:\op6000\labs\lab9b\lab9a64p.pjt` to `C:\op6000\labs\lab9b\lab9b64p.pjt`.
4. Open the `lab9b64p.pjt` file.
5. Change to the *Release* configuration.

6. To tell the compiler to use program level optimization (`-pm` and `-op2`), enter the following

Project → Build Options → Compiler tab → Basic Category

Open *Program Level Optimization* drop-down menu.

Select **No External Func/Var Refs**. Press OK



This adds `-pm` and `-op2` to the command line.

7. Rebuild, Reload, & Go Main.
 8. If the Profile Viewer closed with *lab9a64p.pjt*, reopen it.

Profile → Viewer

9. Run the code.
 10. Record the Incl Total (Cycle Count) of `wvs()` in the table on page 9-31 under “Program Level Opt (`-pm` & `-op2`)”.

11. Now try to open *wvs.asm* to get the Cycles per Iteration. Can you find the file where it should be? _____

Note: Remember that `-pm` combines all of your files into one file before compiling. Therefore there is not a *wvs.asm* file.

12. Open *main.asm* in the `C:\op6000\labs\lab9b\Release_64p` directory and find the *wvs()* function. Use the Software Pipelined Information embedded in the file to record the "Cycles per Iteration" under Program Level Opt.
13. Compare the improvements in the following table:

Optimization Comparison Table for the 'C64xp

Optimization Method	Incl Tot/Cycle Count	Cycles per Iteration
No Optimization		N/A
Release		
-mt option		
Restrict Qualifier		
MUST_ITERATE		
_nassert (double word-aligned)		
Program Level Opt (-pm & -op2)		

14. **Save** your work and **close** the Lab 9b project.

End of Lab9b for 'C64x, move on to Lab9c on page 9 - 61

Lab 9a for 'C67x

We're going to explore using other pragmas and tools that we discussed in this Chapter to maximize the efficiency of some more C code.

Save a Copy of Lab9a

Before we get started, we want to save a copy of the files that we will be working with. We want you to do this, so that you know that we aren't trying to do anything "tricky".

1. Use Windows Explorer to copy *C:\op6000\labs\lab9a* folder to *C:\op6000\labs\lab9b*.

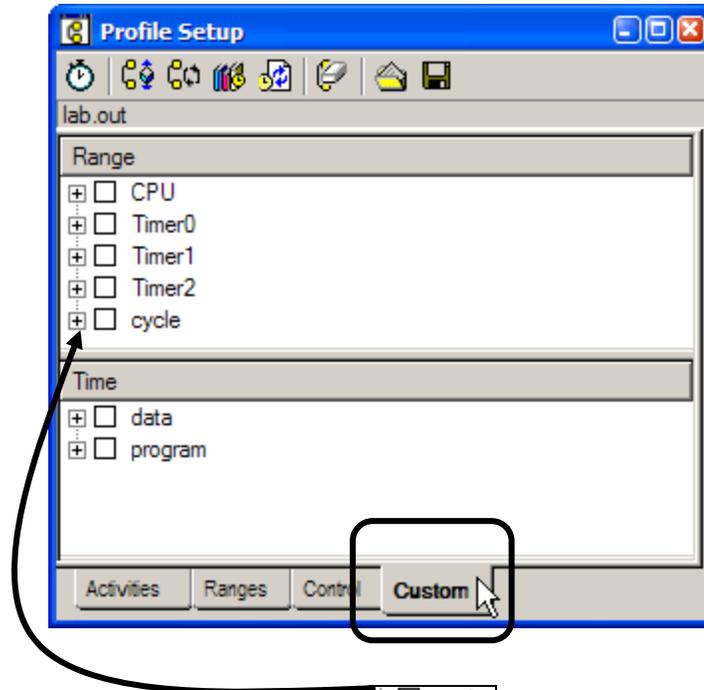
Load Project and Set Compiler Options

2. Open the *lab9a67* project found in folder *C:\op6000\labs\lab9a\lab9a67.pjt*.
3. Expand this project and make sure it contains the following files:
 - *wvs.c*
 - *main.c*
 - *lnk.cmd*
 - *C:\CCStudio\c6000\cgtools\lib\rts6700.lib*
4. Make sure the Build Configuration is set to *Debug*.
5. Build, load, & run your program to main (go main if not taken there automatically).

Set up Profiler

6. Open the Profile Setup window and click on the *Custom* tab at the bottom of the window.

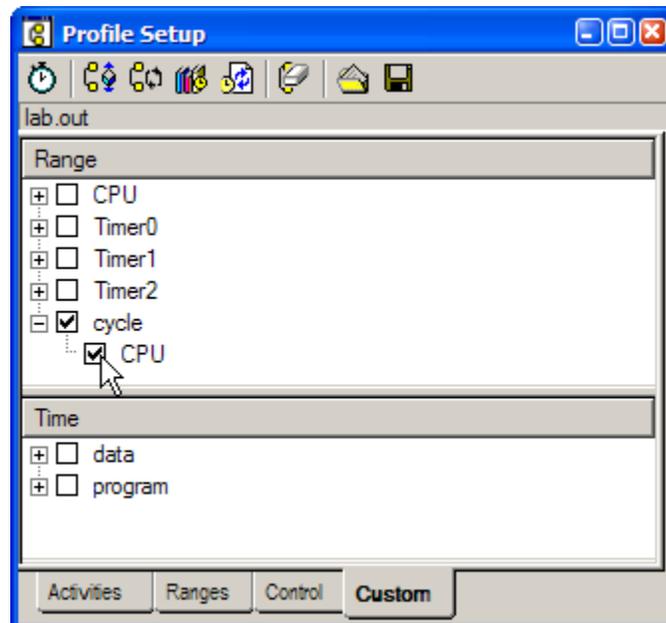
The Custom tab allows you to configure exactly what simulation events are captured by the profiler.



7. Click on the plus sign next to the cycle entry,



8. Click the box next to the CPU entry under cycle.



The cycle:CPU event counts the number of cycles executed by the CPU while ignoring memory and cache effects.

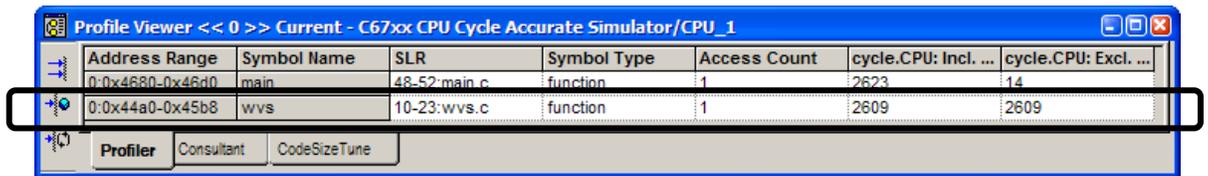
- Click on the Enable/Disable All Functions icon  in the Profile Setup window to **enable** the profiling of all functions.

- Enable profiling by clicking on the  icon.

- Open the Profile Viewer.

Profile → Viewer

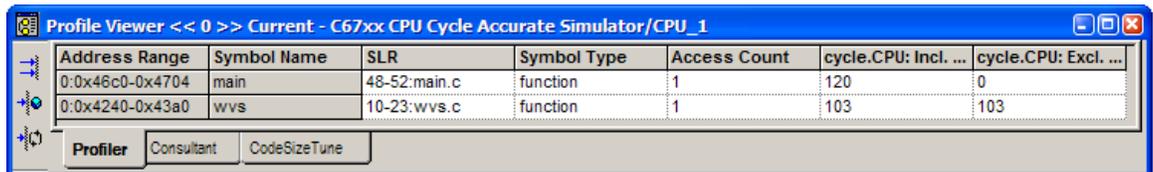
- Run the code.
- Record the Incl Total (Cycle Count) for the `wvs` function for Lab9a in the table on p. 9-45 under “No Optimization”.



Address Range	Symbol Name	SLR	Symbol Type	Access Count	cycle.CPU: Incl. ...	cycle.CPU: Excl. ...
0:0x4680-0x46d0	main	48-52:main.c	function	1	2623	14
0:0x44a0-0x45b8	wvs	10-23:wvs.c	function	1	2609	2609

Optimizing the Code

- Open the `wvs.c` file. This C code is a vector summation of two weighted vectors. We will be using this code throughout this lab.
- Change the Build Configuration to *Release*.
- Rebuild the code.
- Run the code. Record the number of cycles from the Profile Viewer in the table on page 9-45.



Address Range	Symbol Name	SLR	Symbol Type	Access Count	cycle.CPU: Incl. ...	cycle.CPU: Excl. ...
0:0x46c0-0x4704	main	48-52:main.c	function	1	120	0
0:0x4240-0x43a0	wvs	10-23:wvs.c	function	1	103	103

18. Open the *wvs.asm* file in the *C:\op6000\labs\lab9a\Release_67* directory.

This file is available because of the `-k` compiler option we set.

The *.asm* file contains software pipeline information. Scroll down through this file and find the Software Pipeline Information Comments. Do you see “ Searching for a software pipeline schedule at ... `ii = 10`”?

`ii=10` stands for iteration interval equals 10. This implies that each iteration of the loop takes ten cycles, or that there are ten cycles per loop iteration.

Record the number of "Cycles per Iteration" for *Release* in the table on page 9-45.

With 8 resources available every cycle on such a small loop, we would expect this loop to do better than this.

Q. Why did the loop start searching for a software pipeline at `ii=10` (for an 10 cycle loop)?

A. Because 3 pointers are used in this function (`xptr`, `yptr`, & `zptr`). If all the pointers point to a different location in memory, there is no dependency. However since all three pointers are passed into `lab9a`, there is no way for the compiler to be sure they don't alias, or point to the same location. This is a memory alias disambiguation problem. The compiler must be conservative to guarantee correct execution. Unfortunately, the requirement for the compiler to be conservative can have dire effects on the performance of your code.

We know from looking at the calling function in `main`, that in fact, these pointers all point to separate arrays in memory. However, from the compiler's local view of `lab9a`, this information is not available.

Q. How can you pass more information to the compiler to improve its performance?

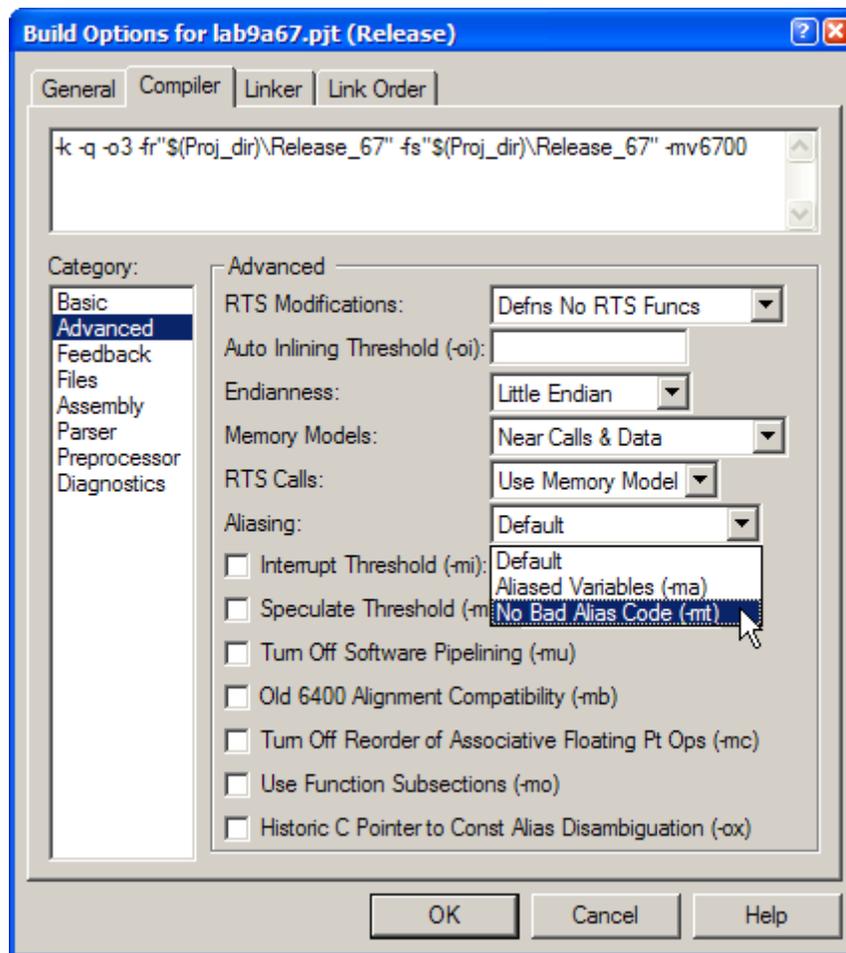
Solving Alias Disambiguation Problems

A special option in the compiler, `-mt`, tells the compiler to ignore alias disambiguation problems like the one described at the beginning of this lab. Try using this option to rebuild the original lab and look at the results.

19. Turn on the `-mt` compiler option.

Project → *Build Options* → *Compiler* tab → *Advanced* Category.

Pull down *Aliasing* menu and select *No Bad Alias Code (-mt)* and then press OK.



20. Rebuild, Reload, & Go Main.

21. Hide the Build window.

22. Run the code.

23. Record the Incl Total (Cycle Count) of `wvs()` in the table on p. 9-45 under “`-mt` options”.

24. Record the number of "Cycles per Iteration" for *Release* in the table on page 9-45 under "-mt options".

Open the *wvs.asm* file. Under the Global File Parameters, the following line implies that `-mt` was used:

```
;* Memory Aliases : Presume not aliases (optimistic)
```

The compiler now knows that there aren't supposed to be any aliases in this project. What if some functions had aliases and others didn't? We need something that is a little more specific.

Adding a Restrict Qualifier

25. Open the Build Options for the *Release* configuration and remove the `-mt` option.
26. Now open the *wvs.c* file.

Because we know that `xptr` & `yptr` are actually separate arrays in memory from `w_sum`, we can declare that nothing else points to these objects. To do this, we must add a restrict type qualifier.

Modify the function definition to be as follows:

```
void wvs(short * restrict xptr, short * restrict yptr, short *zptr, short *w_sum, int N)
```

Save and close *wvs.c*.

27. We must also modify *main.c* to reflect this function definition. Open *main.c* and change the external statement to read:

```
extern void wvs(short * restrict, short * restrict, short *, short *, int);
```

Save and close *main.c*.

28. Rebuild, Reload, & Go Main.
29. Run the code.
30. Record the Incl Total (Cycle Count) of lab9a in the table on p. 9-45 under "Restrict Qualifier".
31. Open *wvs.asm* and scroll down to the Software Pipeline Information.

Q. How many cycles is the Software Pipelined Loop now? _____

Now the loop carried dependency bound is zero. By simply passing more information to the compiler, we allowed it to improve an 10-cycle loop to a 2-cycle loop.

Record the number of cycles per iteration in the table on page 9-45.

32. Scroll down to the Software Pipeline Information until you see

```
;* Loop Carried Dependency Bound (^) : 0
:* ii = 2 Schedule found with 5 iterations in parallel
```

This indicates that a 2-cycle loop was found, which is similar to using the `-mt` compiler option. However, the `restrict` keyword is function specific and gives the user more control.

Q. How can we optimize our code more?

Balance the Resources

33. Let's analyze the feedback to determine what improvements could be made. Open `wvs.asm`.

Scroll down to the Software Pipeline Information. The first iteration interval (ii) attempted was two cycles because the Partitioned Resource Bound is two. We can see the reason for this if we look below at the `.D` unit and the `.T` address paths. This loop requires two loads (from `xptr` and `yptr`) and one store (to `w_sum`) for each iteration of the loop.

Each memory access requires a `.D` unit for address calculation, and a `.T` address path to send the address out to memory. Because the C6000 has two `.D` units and two `.T` address paths available on any given cycle (A side and B side), the compiler must partition at least two of the operations on one side (the A side). That means that these operations are the bottleneck in resources (highlighted with an `*`) and are the limiting factor in the Partitioned Resource Bound. The feedback in `wvs.asm` shows that there is an imbalance in resources between A and B side due, in this case, to an odd number of operations being mapped to two sides of the machine.

Q. Is it possible to improve the balance of resources?

A. One way to balance an odd number of operations is to unroll the loop. Now, instead of three memory accesses, you will have six, which is an even number. You can only do this if you know that the loop counter is a multiple of two; otherwise, you will incorrectly execute too few or too many iterations. In main, `LOOPCOUNT` is defined to be 40, which is a multiple of two, so you are able to unroll the loop.

Q. Why did the compiler not unroll the loop?

A. In the limited scope of `wvs.c`, the loop counter is passed as a parameter to the function. Therefore, it might be any value from this limited view of the function. To improve this scope you must pass more information to the compiler. One way of doing this is with a `MUST_ITERATE` pragma.

34. Open `wvs.c` and insert a `MUST_ITERATE` pragma above the for loop that says the loop always iterates in multiples of 4s but always iterates at least 40 times. It should look like this:

```
#pragma MUST_ITERATE ( 40, , 4 );
```

Unrolling a loop can incur some minor overhead in loop setup. The compiler does not unroll loops with small loop counts because unrolling may not reduce the overall cycle count. If the compiler does not know what the minimum value of the loop counter is, it will not automatically unroll the loop. Again, this is information the compiler needs but does not have in the local scope of `wvs.c`. You know that `LOOPCOUNT` is set to 40, so you can tell the compiler that `N` is greater than some minimum value. The `MUST_ITERATE` pragma passes these two valuable pieces of information.

35. Save & close `wvs.c`. Close any open files. Leave the Profile Viewer open.

36. Rebuild, Reload, & Go Main.

37. Run the code.

38. Record the Incl Total (Cycle Count) of `wvs()` in the table on p. 9-45 under “`MUST_ITERATE`”.

Remember, no code was altered in this lab. Only additional information was passed via the `MUST_ITERATE` pragma. We simply guarantee the compiler that the trip count (in this case the trip count is `N`) is a multiple of four and that the trip count is greater than or equal to 40. The first argument for `MUST_ITERATE` is the minimum number of times the loop will iterate. The second argument is the maximum number of times the loop will iterate. The trip count must be evenly divisible by the third argument, the factor.

Note: Choose a trip count large enough to tell the compiler that it is more efficient to unroll the loop. Always specify the largest minimum trip count that is safe.

Open the *wvs.asm* file. Notice the following things in the Software Pipeline information feedback:

Loop Unroll Multiple:2x

This loop has been unrolled by a factor of two.

ii=3 Schedule found with 5 iterations in parallel

You can tell by looking at the .D units and .T address paths that this 3-cycle loop comes after the loop has been unrolled because the resources show a total of six memory accesses. Therefore, our new effective loop iteration interval is $3/2$ or 1.5 cycles. The "5 iterations in parallel" means that the loop is working on 5 different results at the same time. This doesn't have any bearing on our performance.

Minimum Safe Trip Count: 4 (after unrolling)

This is because we specify the count of the original loop to be greater than or equal to forty and a multiple of four and after unrolling, the loop count has to be even.

Therefore, by passing information without modifying the loop code, compiler performance improves from a 10-cycle loop to 2 cycles and now to 1.5 cycles.

Record the number of cycles per iteration in the table on page 9-45.

Q. Is this the lower limit?

Packed Data Optimization Increases Memory Bandwidth

The last optimization produced a 3-cycle loop that performed 2 iterations of the original *vector sum of two weighted vectors*. This means that each iteration of our loop now performs six memory accesses, four multiplies, two adds, two shift operations, a decrement for the loop counter, and a branch. You can see this in the feedback of *wvs.c*.

39. Open *wvs.asm*.

The six memory accesses appear as .D and .T units. The four multiplies appear as .M units. The two shifts and the branch show up as .s units. The decrement and the two adds appear as .LSD units.

By analyzing this part of the feedback, we can see that resources are most limited by the memory accesses; hence, the reason for an asterisk highlighting the .D units and .T address paths.

Q. Does this mean that we cannot make the loop operate any faster?

A. Further insight into the 'C6000 architecture is necessary here.

The C62x fixed-point device loads and/or stores 32 bits every cycle. In addition, the C67x floating-point and the C64x fixed-point device loads two 64-bit values each cycle. In our example, we load four 16-bit values and store two 16-bit values every five cycles. This means we only use 16 bits of memory access every cycle. Because this is a resource bottleneck in our loop, increasing the memory access bandwidth further improves the performance of our loop.

In the unrolled loop generated from the `MUST_ITERATE` pragma, we load two consecutive 16-bit elements with LDHs from both the `xptr` and `yptr` array.

Q. Why not use an aligned LDDW to load one 64-bit element, with the resulting register pair load containing the first element in one-half of the lower 32-bit register and the second element in the other half and the third and fourth in the other register?

A. This is called Packed Data optimization. Four 16-bit loads are effectively performed by one single 64-bit load instruction.

Q. Why didn't the compiler do this automatically in the `MUST_ITERATE` code?

A. Again, the answer lies in the amount of information the compiler has access to from the local scope of the program.

In order to perform a LDDW (64-bit load) on the C67x cores, the address must be aligned to a double word-address; otherwise, incorrect data would be loaded. An address is double word-aligned if the lower three bits of the address are zero. Unfortunately, in our example, the `wvs()` function does not have knowledge as to the `xptr` and `yptr` address values. Therefore, the compiler is forced to be conservative and assume that these pointers might not be aligned. Once again, we can pass more information to the compiler, this time via the `_nassert` statement.

40. Open *wvs.c*. Add the following piece of code on the first line:

```
#define DWORD_ALIGNED(x) (_nassert(((int) (x) & 0x7) == 0))
```

Also add `DWORD_ALIGNED` definitions for `xptr` & `yptr` after the short `w1,w2;` statement.

```
DWORD_ALIGNED(xptr);  
DWORD_ALIGNED(yptr);
```

By asserting that `xptr` and `yptr` addresses “anded” with `0x7` are equal to zero, the compiler knows that they are double word aligned. This means the compiler can perform LDDW and packed data optimization on these memory accesses.

Hint: If you need would like a refresher on how `DATA_ALIGN` and `_nassert()` are working together to produce optimal code, refer back to the material on page 9-19 in the Student Notes.

41. Save and close *wvs.c*.
42. Rebuild, Reload, & Go Main.
43. Run the code.
44. Record the Incl Total (Cycle Count) of *wvs()* in the table on p. 9-45 under “_nassert (double word-aligned)”.
45. Open *wvs.asm*.
- Success! The compiler has fully optimized this loop. You can now achieve 4 iterations of the loop every four cycles for **1** cycle per iteration throughout.
- The `.D` and `.T` resources now show 3 (two LDDWs and four STHs for four iterations of the loop).
46. Close *lab9a67.pjt* and any files associated with it.

Move on to Lab9b for the ‘C67x

Lab9b for 'C67x

Program Level Optimization

Now we have learned how to pass information to the compiler. This increased the amount of information visible to the compiler from the local scope of each function.

Q. Is this necessary in all cases?

A. No, not in all cases. First, if this information already resides locally inside the function, the compiler has visibility here and `restrict` and `MUST_ITERATE` statements are not usually necessary. For example, if `xptr` and `yptr` are declared as local arrays, the compiler does not assume a dependency with `w_sum`. If the loop count is defined in the function or if the loop is simply described to be from one to forty, the `MUST_ITERATE` pragma is not necessary.

Second, even if this type of information is not declared locally, the compiler can still have access to it in an automated way by giving it a program level view.

The C6000 compiler provides two valuable switches, which enable program level optimization: **-pm** and **-op2**. When these two options are used together, the compiler can automatically extract all of the information we passed in Lab9a.

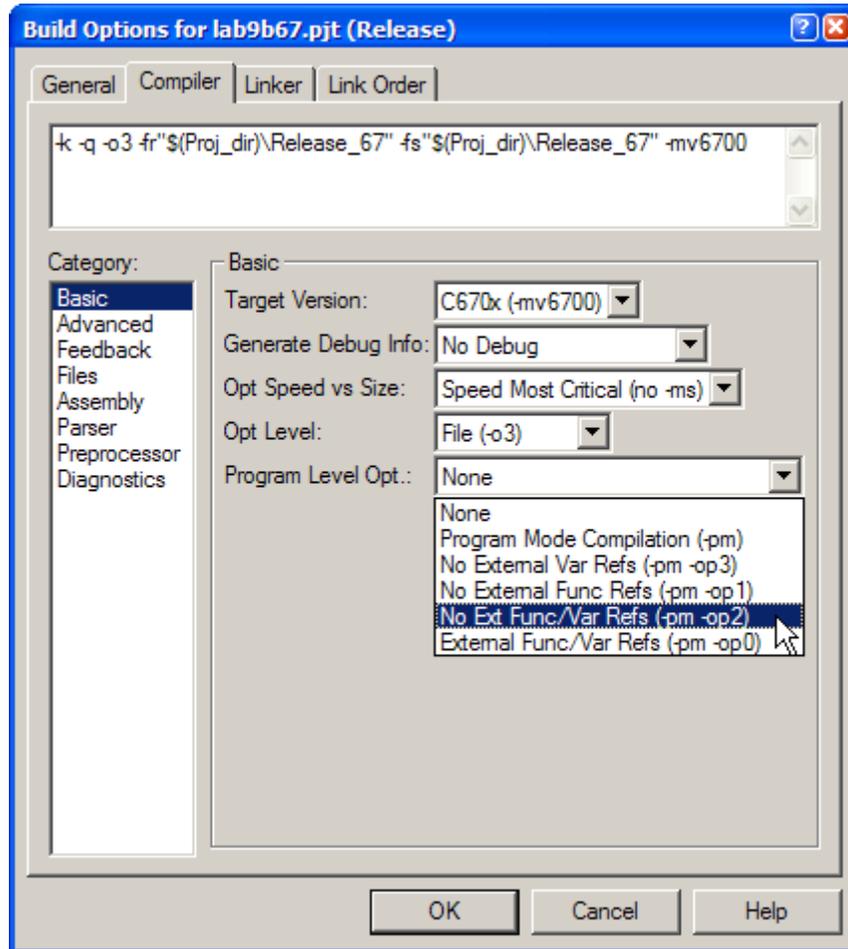
1. Make sure that all files from lab9a are closed.
2. We are now going to use the copy of the Lab 9a files that you made at the beginning of that lab. Navigate to the `C:\op6000\labs\lab9b` folder.
3. Rename the `C:\op6000\labs\lab9b\lab9a67.pjt` to `C:\op6000\labs\lab9b\lab9b67.pjt`.
4. Open the `lab9b67.pjt` file.
5. Change to the *Release* configuration.

6. To tell the compiler to use program level optimization (`-pm` and `-op2`), enter the following

Project → Build Options → Compiler tab → Basic Category

Open *Program Level Optimization* drop-down menu.

Select **No External Func/Var Refs**. Press OK



This adds `-pm` and `-op2` to the command line.

7. Rebuild, Reload, & Go Main.
 8. If the Profile Viewer closed with *lab9a67.pjt*, reopen it.

Profile → Viewer

9. Run the code.
 10. Record the Incl Total (Cycle Count) of `wvs()` in the table on page 9-45 under “Program Level Opt (`-pm` & `-op2`)”.

11. Now try to open *wvs.asm* to get the Cycles per Iteration. Can you find the file where it should be? _____

Note: Remember that `-pm` combines all of your files into one file before compiling. Therefore there is not a *wvs.asm* file.

12. Open *main.asm* in the `C:\op6000\labs\lab9b\Release_67` directory and find the *wvs()* function. Use the Software Pipelined Information embedded in the file to record the "Cycles per Iteration" under Program Level Opt.
13. Compare the improvements in the following table:

Optimization Comparison Table for 'C67x

Optimization Method	Incl Tot/Cycle Count	Cycles per Iteration
No Optimization		N/A
Release		
-mt option		
Restrict Qualifier		
MUST_ITERATE		
_nassert (double word-aligned)		
Program Level Opt (-pm & -op2)		

14. **Save** your work and **close** the Lab9b project.

15. End of Lab9b for 'C67x, move on to Lab9c on page 9 - 61

Lab 9a for 'C672x

We're going to explore using other pragmas and tools that we discussed in this Chapter to maximize the efficiency of some more C code.

Save a Copy of Lab9a

Before we get started, we want to save a copy of the files that we will be working with. We want you to do this, so that you know that we aren't trying to do anything "tricky".

47. Use Windows Explorer to copy *C:\op6000\labs\lab9a* folder to *C:\op6000\labs\lab9b*.

Load Project and Set Compiler Options

48. Open the *lab9a67p* project found in folder *C:\op6000\labs\lab9a\lab9a67p.pjt*.

49. Expand this project and make sure it contains the following files:

- *wvs.c*
- *main.c*
- *lnk.cmd*

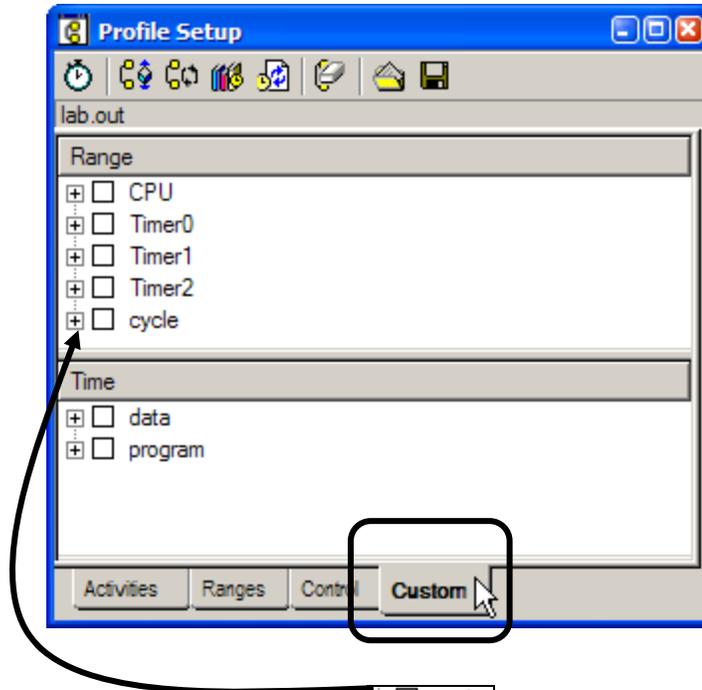
50. Make sure the Build Configuration is set to *Debug*.

51. Build, load, & run your program to main (go main if not taken there automatically).

Set up Profiler

52. Open the Profile Setup window and click on the *Custom* tab at the bottom of the window.

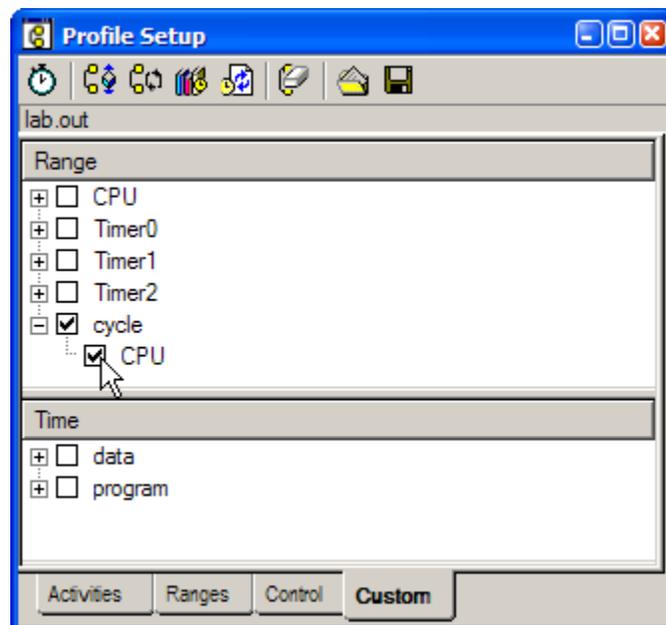
The Custom tab allows you to configure exactly what simulation events are captured by the profiler.



53. Click on the plus sign next to the cycle entry,



54. Click the box next to the CPU entry under cycle.



The cycle:CPU event counts the number of cycles executed by the CPU while ignoring memory and cache effects.

55. Click on the Enable/Disable All Functions icon  in the Profile Setup window to **enable** the profiling of all functions.

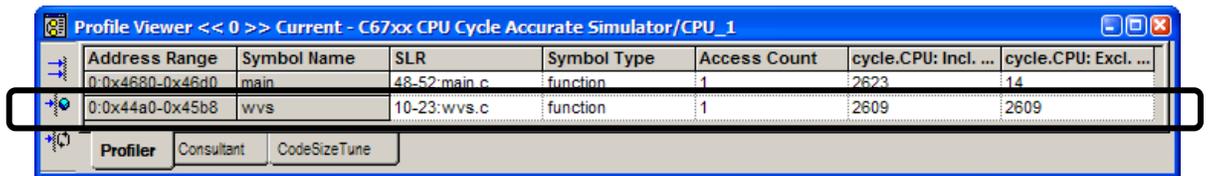
56. Enable profiling by clicking on the  icon.

57. Open the Profile Viewer.

Profile → Viewer

58. Run the code.

59. Record the Incl Total (Cycle Count) for the `wvs` function for Lab9a in the table on p. 9-45 under “No Optimization”.



Address Range	Symbol Name	SLR	Symbol Type	Access Count	cycle.CPU: Incl. ...	cycle.CPU: Excl. ...
0:0x4680-0x46d0	main	48-52:main.c	function	1	2623	14
0:0x44a0-0x45b8	wvs	10-23:wvs.c	function	1	2609	2609

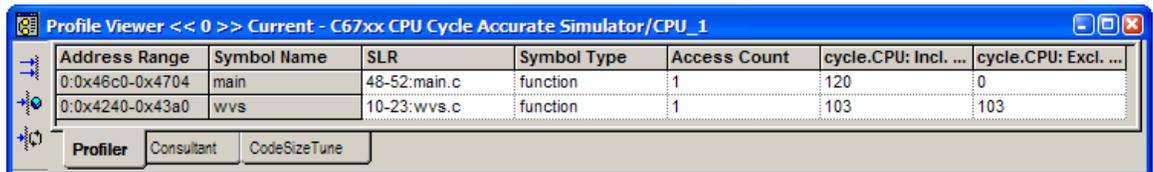
Optimizing the Code

60. Open the `wvs.c` file. This C code is a vector summation of two weighted vectors. We will be using this code throughout this lab.

61. Change the Build Configuration to *Release*.

62. Rebuild the code.

63. Run the code. Record the number of cycles from the Profile Viewer in the table on page 9-45.



Address Range	Symbol Name	SLR	Symbol Type	Access Count	cycle.CPU: Incl. ...	cycle.CPU: Excl. ...
0:0x46c0-0x4704	main	48-52:main.c	function	1	120	0
0:0x4240-0x43a0	wvs	10-23:wvs.c	function	1	103	103

64. Open the *wvs.asm* file in the *C:\op6000\labs\lab9a\Release_67p* directory.

This file is available because of the `-k` compiler option we set.

The *.asm* file contains software pipeline information. Scroll down through this file and find the Software Pipeline Information Comments. Do you see “ Searching for a software pipeline schedule at ... `ii = 10`”?

`ii=10` stands for iteration interval equals 10. This implies that each iteration of the loop takes ten cycles, or that there are ten cycles per loop iteration.

Record the number of "Cycles per Iteration" for *Release* in the table on page 9-45.

With 8 resources available every cycle on such a small loop, we would expect this loop to do better than this.

Q. Why did the loop start searching for a software pipeline at `ii=10` (for an 10 cycle loop)?

A. Because 3 pointers are used in this function (`xptr`, `yptr`, & `zptr`). If all the pointers point to a different location in memory, there is no dependency. However since all three pointers are passed into `lab9a`, there is no way for the compiler to be sure they don't alias, or point to the same location. This is a memory alias disambiguation problem. The compiler must be conservative to guarantee correct execution. Unfortunately, the requirement for the compiler to be conservative can have dire effects on the performance of your code.

We know from looking at the calling function in `main`, that in fact, these pointers all point to separate arrays in memory. However, from the compiler's local view of `lab9a`, this information is not available.

Q. How can you pass more information to the compiler to improve its performance?

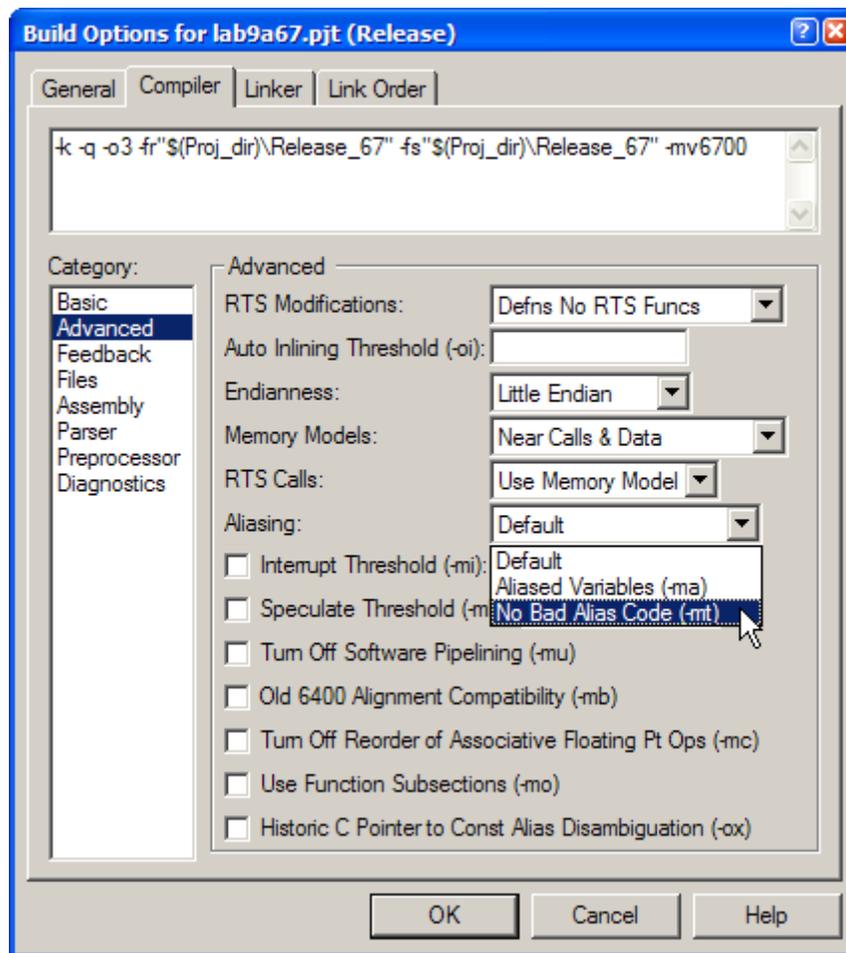
Solving Alias Disambiguation Problems

A special option in the compiler, `-mt`, tells the compiler to ignore alias disambiguation problems like the one described at the beginning of this lab. Try using this option to rebuild the original lab and look at the results.

65. Turn on the `-mt` compiler option.

Project → *Build Options* → *Compiler* tab → *Advanced* Category.

Pull down *Aliasing* menu and select *No Bad Alias Code (-mt)* and then press OK.



66. Rebuild, Reload, & Go Main.

67. Hide the Build window.

68. Run the code.

69. Record the Incl Total (Cycle Count) of `wvs()` in the table on p. 9-45 under “`-mt` options”.

70. Record the number of "Cycles per Iteration" for *Release* in the table on page 9-45 under "-mt options".

Open the *wvs.asm* file. Under the Global File Parameters, the following line implies that `-mt` was used:

```
;* Memory Aliases : Presume not aliases (optimistic)
```

The compiler now knows that there aren't supposed to be any aliases in this project. What if some functions had aliases and others didn't? We need something that is a little more specific.

Adding a Restrict Qualifier

71. Open the Build Options for the *Release* configuration and remove the `-mt` option.
72. Now open the *wvs.c* file.

Because we know that `xptr` & `yptr` are actually separate arrays in memory from `w_sum`, we can declare that nothing else points to these objects. To do this, we must add a restrict type qualifier.

Modify the function definition to be as follows:

```
void wvs(short * restrict xptr, short * restrict yptr, short *zptr, short *w_sum, int N)
```

Save and close *wvs.c*.

73. We must also modify *main.c* to reflect this function definition. Open *main.c* and change the external statement to read:

```
extern void wvs(short * restrict, short * restrict, short *, short *, int);
```

Save and close *main.c*.

74. Rebuild, Reload, & Go Main.
75. Run the code.
76. Record the Incl Total (Cycle Count) of lab9a in the table on p. 9-45 under "Restrict Qualifier".
77. Open *wvs.asm* and scroll down to the Software Pipeline Information.

Q. How many cycles is the Software Pipelined Loop now? _____

Now the loop carried dependency bound is zero. By simply passing more information to the compiler, we allowed it to improve an 10-cycle loop to a 2-cycle loop.

Record the number of cycles per iteration in the table on page 9-45.

78. Scroll down to the Software Pipeline Information until you see

```
;* Loop Carried Dependency Bound (^) : 0
:* ii = 2 Schedule found with 5 iterations in parallel
```

This indicates that a 2-cycle loop was found, which is similar to using the `-mt` compiler option. However, the `restrict` keyword is function specific and gives the user more control.

Q. How can we optimize our code more?

Balance the Resources

79. Let's analyze the feedback to determine what improvements could be made. Open `wvs.asm`.

Scroll down to the Software Pipeline Information. The first iteration interval (ii) attempted was two cycles because the Partitioned Resource Bound is two. We can see the reason for this if we look below at the `.D` unit and the `.T` address paths. This loop requires two loads (from `xptr` and `yptr`) and one store (to `w_sum`) for each iteration of the loop.

Each memory access requires a `.D` unit for address calculation, and a `.T` address path to send the address out to memory. Because the C6000 has two `.D` units and two `.T` address paths available on any given cycle (A side and B side), the compiler must partition at least two of the operations on one side (the A side). That means that these operations are the bottleneck in resources (highlighted with an `*`) and are the limiting factor in the Partitioned Resource Bound. The feedback in `wvs.asm` shows that there is an imbalance in resources between A and B side due, in this case, to an odd number of operations being mapped to two sides of the machine.

Q. Is it possible to improve the balance of resources?

A. One way to balance an odd number of operations is to unroll the loop. Now, instead of three memory accesses, you will have six, which is an even number. You can only do this if you know that the loop counter is a multiple of two; otherwise, you will incorrectly execute too few or too many iterations. In main, `LOOPCOUNT` is defined to be 40, which is a multiple of two, so you are able to unroll the loop.

Q. Why did the compiler not unroll the loop?

A. In the limited scope of `wvs.c`, the loop counter is passed as a parameter to the function. Therefore, it might be any value from this limited view of the function. To improve this scope you must pass more information to the compiler. One way of doing this is with a `MUST_ITERATE` pragma.

80. Open `wvs.c` and insert a `MUST_ITERATE` pragma above the for loop that says the loop always iterates in multiples of 4s but always iterates at least 40 times. It should look like this:

```
#pragma MUST_ITERATE ( 40, , 4 );
```

Unrolling a loop can incur some minor overhead in loop setup. The compiler does not unroll loops with small loop counts because unrolling may not reduce the overall cycle count. If the compiler does not know what the minimum value of the loop counter is, it will not automatically unroll the loop. Again, this is information the compiler needs but does not have in the local scope of `wvs.c`. You know that `LOOPCOUNT` is set to 40, so you can tell the compiler that `N` is greater than some minimum value. The `MUST_ITERATE` pragma passes these two valuable pieces of information.

81. Save & close `wvs.c`. Close any open files. Leave the Profile Viewer open.
82. Rebuild, Reload, & Go Main.
83. Run the code.
84. Record the Incl Total (Cycle Count) of `wvs()` in the table on p. 9-45 under “`MUST_ITERATE`”.

Remember, no code was altered in this lab. Only additional information was passed via the `MUST_ITERATE` pragma. We simply guarantee the compiler that the trip count (in this case the trip count is `N`) is a multiple of four and that the trip count is greater than or equal to 40. The first argument for `MUST_ITERATE` is the minimum number of times the loop will iterate. The second argument is the maximum number of times the loop will iterate. The trip count must be evenly divisible by the third argument, the factor.

Note: Choose a trip count large enough to tell the compiler that it is more efficient to unroll the loop. Always specify the largest minimum trip count that is safe.

Open the *wvs.asm* file. Notice the following things in the Software Pipeline information feedback:

Loop Unroll Multiple:2x

This loop has been unrolled by a factor of two.

ii=3 Schedule found with 5 iterations in parallel

You can tell by looking at the .D units and .T address paths that this 3-cycle loop comes after the loop has been unrolled because the resources show a total of six memory accesses. Therefore, our new effective loop iteration interval is $3/2$ or 1.5 cycles. The "5 iterations in parallel" means that the loop is working on 5 different results at the same time. This doesn't have any bearing on our performance.

Minimum Safe Trip Count: 4 (after unrolling)

This is because we specify the count of the original loop to be greater than or equal to forty and a multiple of four and after unrolling, the loop count has to be even.

Therefore, by passing information without modifying the loop code, compiler performance improves from a 10-cycle loop to 2 cycles and now to 1.5 cycles.

Record the number of cycles per iteration in the table on page 9-45.

Q. Is this the lower limit?

Packed Data Optimization Increases Memory Bandwidth

The last optimization produced a 3-cycle loop that performed 2 iterations of the original *vector sum of two weighted vectors*. This means that each iteration of our loop now performs six memory accesses, four multiplies, two adds, two shift operations, a decrement for the loop counter, and a branch. You can see this in the feedback of *wvs.c*.

85. Open *wvs.asm*.

The six memory accesses appear as .D and .T units. The four multiplies appear as .M units. The two shifts and the branch show up as .s units. The decrement and the two adds appear as .LSD units.

By analyzing this part of the feedback, we can see that resources are most limited by the memory accesses; hence, the reason for an asterisk highlighting the .D units and .T address paths.

Q. Does this mean that we cannot make the loop operate any faster?

A. Further insight into the 'C6000 architecture is necessary here.

The C62x fixed-point device loads and/or stores 32 bits every cycle. In addition, the C67x floating-point and the C64x fixed-point device loads two 64-bit values each cycle. In our example, we load four 16-bit values and store two 16-bit values every five cycles. This means we only use 16 bits of memory access every cycle. Because this is a resource bottleneck in our loop, increasing the memory access bandwidth further improves the performance of our loop.

In the unrolled loop generated from the `MUST_ITERATE` pragma, we load two consecutive 16-bit elements with LDHs from both the `xptr` and `yptr` array.

Q. Why not use an aligned LDDW to load one 64-bit element, with the resulting register pair load containing the first element in one-half of the lower 32-bit register and the second element in the other half and the third and fourth in the other register?

A. This is called Packed Data optimization. Four 16-bit loads are effectively performed by one single 64-bit load instruction.

Q. Why didn't the compiler do this automatically in the `MUST_ITERATE` code?

A. Again, the answer lies in the amount of information the compiler has access to from the local scope of the program.

In order to perform a LDDW (64-bit load) on the C67x cores, the address must be aligned to a double word-address; otherwise, incorrect data would be loaded. An address is double word-aligned if the lower three bits of the address are zero. Unfortunately, in our example, the `wvs()` function does not have knowledge as to the `xptr` and `yptr` address values. Therefore, the compiler is forced to be conservative and assume that these pointers might not be aligned. Once again, we can pass more information to the compiler, this time via the `_nassert` statement.

86. Open *wvs.c*. Add the following piece of code on the first line:

```
#define DWORD_ALIGNED(x) (_nassert(((int) (x) & 0x7) == 0))
```

Also add `DWORD_ALIGNED` definitions for `xptr` & `yptr` after the short `w1,w2;` statement.

```
DWORD_ALIGNED(xptr);  
DWORD_ALIGNED(yptr);
```

By asserting that `xptr` and `yptr` addresses “anded” with `0x7` are equal to zero, the compiler knows that they are double word aligned. This means the compiler can perform LDDW and packed data optimization on these memory accesses.

Hint: If you need would like a refresher on how `DATA_ALIGN` and `_nassert()` are working together to produce optimal code, refer back to the material on page 9-19 in the Student Notes.

87. Save and close *wvs.c*.

88. Rebuild, Reload, & Go Main.

89. Run the code.

90. Record the Incl Total (Cycle Count) of *wvs()* in the table on p. 9-45 under “_nassert (double word-aligned)”.

91. Open *wvs.asm*.

Success! The compiler has fully optimized this loop. You can now achieve 4 iterations of the loop every four cycles for **1** cycle per iteration throughout.

The `.D` and `.T` resources now show 3 (two LDDWs and four STHs for four iterations of the loop).

92. Close *lab9a67.pjt* and any files associated with it.

Move on to Lab9b for the ‘C672x

Lab9b for 'C672x

Program Level Optimization

Now we have learned how to pass information to the compiler. This increased the amount of information visible to the compiler from the local scope of each function.

Q. Is this necessary in all cases?

A. No, not in all cases. First, if this information already resides locally inside the function, the compiler has visibility here and `restrict` and `MUST_ITERATE` statements are not usually necessary. For example, if `xptr` and `yptr` are declared as local arrays, the compiler does not assume a dependency with `w_sum`. If the loop count is defined in the function or if the loop is simply described to be from one to forty, the `MUST_ITERATE` pragma is not necessary.

Second, even if this type of information is not declared locally, the compiler can still have access to it in an automated way by giving it a program level view.

The C6000 compiler provides two valuable switches, which enable program level optimization: **-pm** and **-op2**. When these two options are used together, the compiler can automatically extract all of the information we passed in Lab9a.

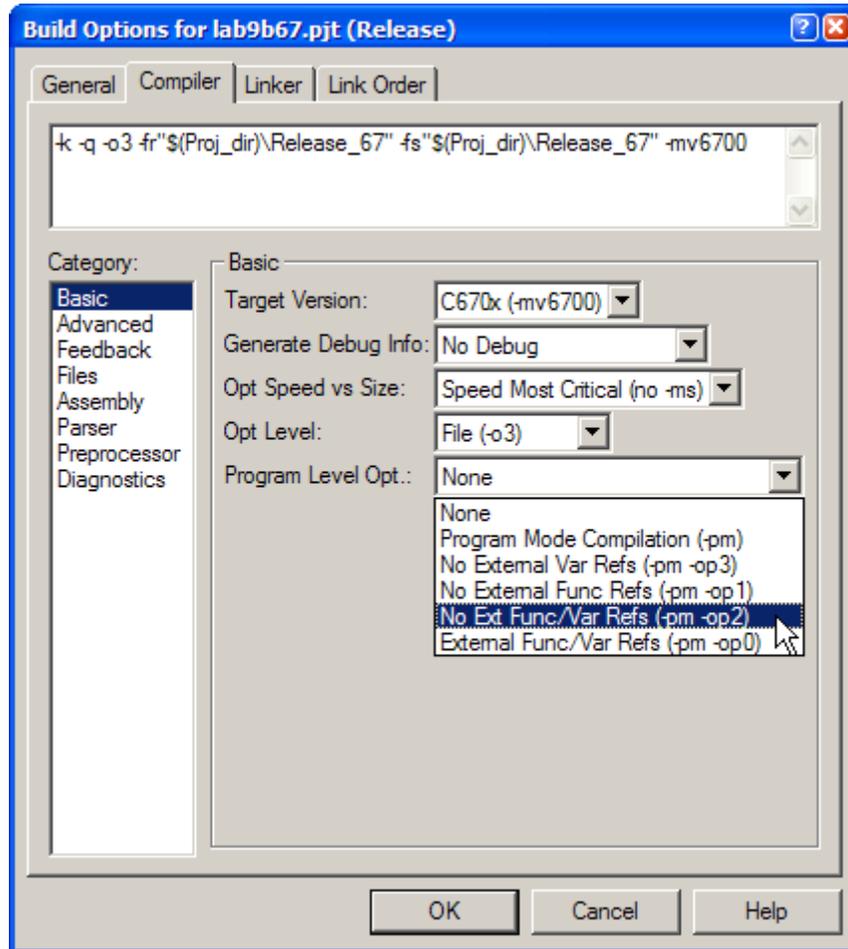
16. Make sure that all files from lab9a are closed.
17. We are now going to use the copy of the Lab 9a files that you made at the beginning of that lab. Navigate to the `C:\op6000\labs\lab9b` folder.
18. Rename the `C:\op6000\labs\lab9b\lab9a67p.pjt` to `C:\op6000\labs\lab9b\lab9b67p.pjt`.
19. Open the `lab9b67p.pjt` file.
20. Change to the *Release* configuration.

21. To tell the compiler to use program level optimization (`-pm` and `-op2`), enter the following

Project → Build Options → Compiler tab → Basic Category

Open *Program Level Optimization* drop-down menu.

Select **No External Func/Var Refs**. Press OK



This adds `-pm` and `-op2` to the command line.

22. Rebuild, Reload, & Go Main.

23. If the Profile Viewer closed with *lab9a67.pjt*, reopen it.

Profile → Viewer

24. Run the code.

25. Record the Incl Total (Cycle Count) of `wvs()` in the table on page 9-45 under “Program Level Opt (`-pm` & `-op2`)”.

26. Now try to open *wvs.asm* to get the Cycles per Iteration. Can you find the file where it should be? _____

Note: Remember that `-pm` combines all of your files into one file before compiling. Therefore there is not a *wvs.asm* file.

27. Open *main.asm* in the `C:\op6000\labs\lab9b\Release_67` directory and find the *wvs()* function. Use the Software Pipelined Information embedded in the file to record the "Cycles per Iteration" under Program Level Opt.
28. Compare the improvements in the following table:

Optimization Comparison Table for 'C67x

Optimization Method	Incl Tot/Cycle Count	Cycles per Iteration
No Optimization		N/A
Release		
-mt option		
Restrict Qualifier		
MUST_ITERATE		
_nassert (double word-aligned)		
Program Level Opt (-pm & -op2)		

29. **Save** your work and **close** the Lab9b project.

End of Lab9b for 'C67x, move on to Lab9c on page 9 - 61

Lab9c - Compiler Consultant Introduction

In this lab, you will analyze a simple loop for optimization. The Compiler Consultant Tool analyzes your application and makes recommendations for changes to optimize the performance. The tool displays two types of information for the user: Compile Time Loop Information and Run Time Loop Information. Compile Time Loop Information is created by the compiler. Each time you compile or build your code, Consultant will analyze the code and create suggestions for different optimization techniques to improve code efficiency. These suggestions include compiler optimization switches and pragmas to insert into your code that give the compiler more application information. You then have the option of implementing the advice and building the project again. The Run Time Loop Information is data gathered by profiling your application. This run time loop information helps prioritize the advice.

Learning Objectives:

- Use the Compiler Consultant Tool to create optimization advice on a sample project loop
- Introduce the different kinds of advice used to optimize the project, including Options, Alias, Alignment and Trip Count Advice
- Implement the advice, and observe the effects

Application Objective:

This lab uses the consultant project to display different features of the Compiler Consultant Tool. You will analyze the example loop using the Compiler Consultant tool throughout the optimization process. After each optimization, new advice will appear based on the changes made. Each type of advice has specific steps associated with it. You will implement these steps and view the results. Optimization will continue until the Consultant Compiler Tool has determined that the loop has been fully optimized.

Getting Started with Consultant

In this lesson, you will analyze a project called consultant.pjt. You will use the Profile Viewer to analyze the project, and the Compiler Consultant Tool will present advice on how to optimize the loop. This lab is currently based on the 'C64x simulator.

67 Change to 'C64x Simulator

1. Since this is a 'C64x based lab, we need to reconfigure the CCS setup. Launch CCS Setup:

672x **File → Launch Setup**

64xp

2. When setup opens, the "Import Configuration" window appears. Follow the procedure found at the start of Lab 4. These are the basic steps:
 - Clear the previous system configuration (button on right).
 - Scroll through the list to find *C64xx CPU Cycle Accurate Sim, Ltl Endian*. Select this item and click **Import**
 - Quit and save CCS Setup. When asked, allow CCS Setup to restart CCS for you.

3. If Code Composer Studio is not running, start it.

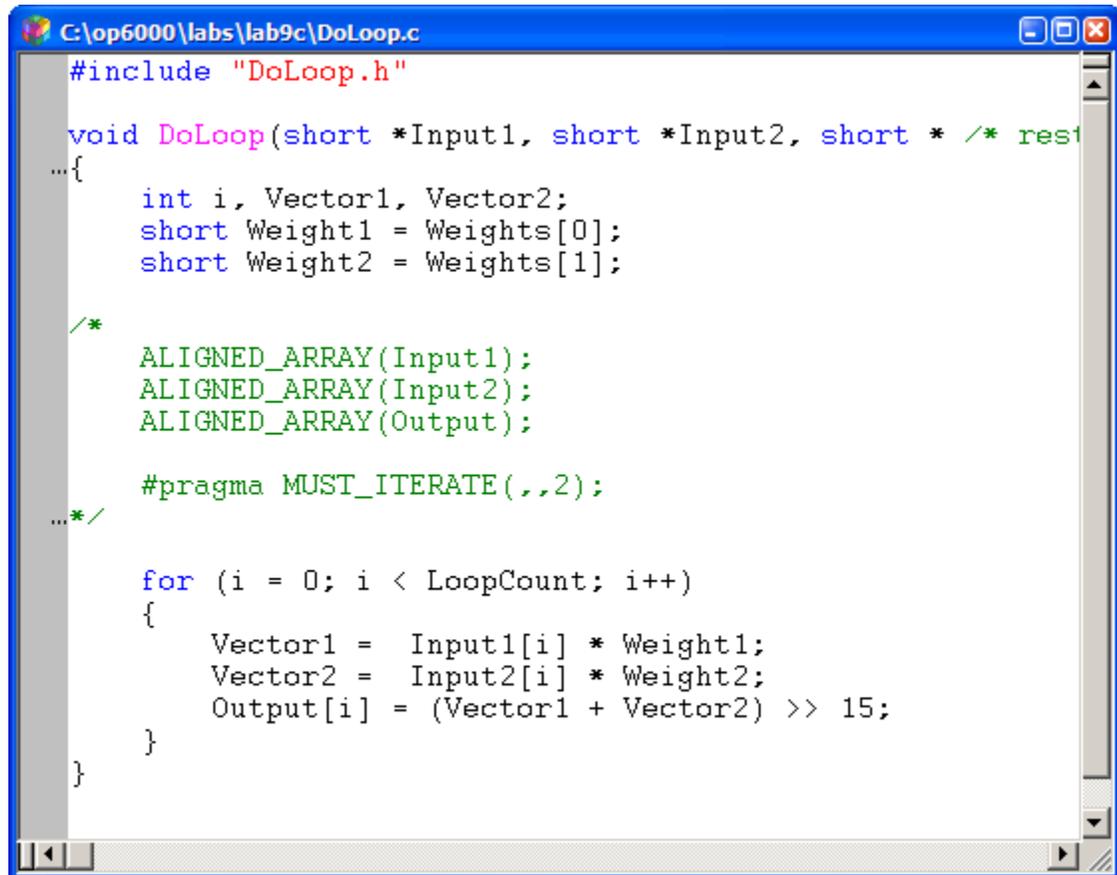
4. Disable the *Load Program after Build* and *Perform Go Main Automatically* options.

Option → Customize; Debug Properties Tab; **uncheck** Perform Go Main Automatically

Option → Customize; Program Load Options Tab; **uncheck** Load Program after Build

5. **Open consultant64.pjt** from C:\op6000\labs\lab9c.

6. Open DoLoop.c for editing by double-clicking on it in the project view.



```
C:\op6000\labs\lab9c\DoLoop.c
#include "DoLoop.h"

void DoLoop(short *Input1, short *Input2, short * /* rest
...{
    int i, Vector1, Vector2;
    short Weight1 = Weights[0];
    short Weight2 = Weights[1];

    /*
    ALIGNED_ARRAY(Input1);
    ALIGNED_ARRAY(Input2);
    ALIGNED_ARRAY(Output);

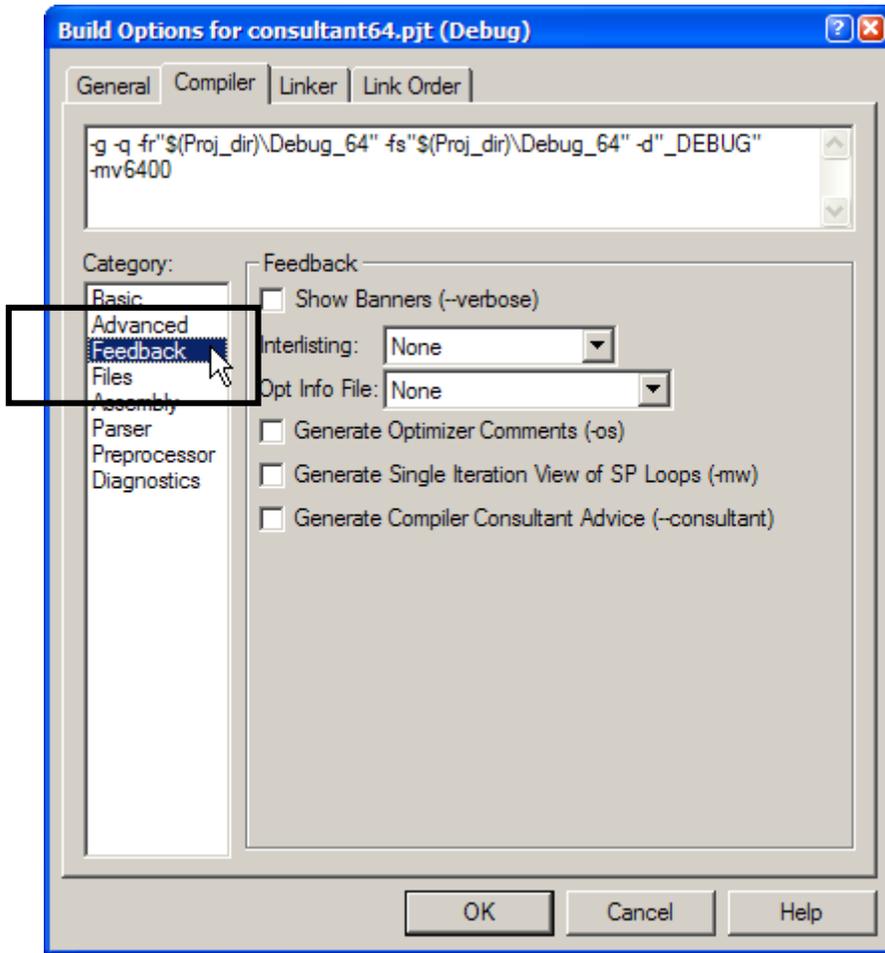
    #pragma MUST_ITERATE(,,2);
    ...*/

    for (i = 0; i < LoopCount; i++)
    {
        Vector1 = Input1[i] * Weight1;
        Vector2 = Input2[i] * Weight2;
        Output[i] = (Vector1 + Vector2) >> 15;
    }
}
```

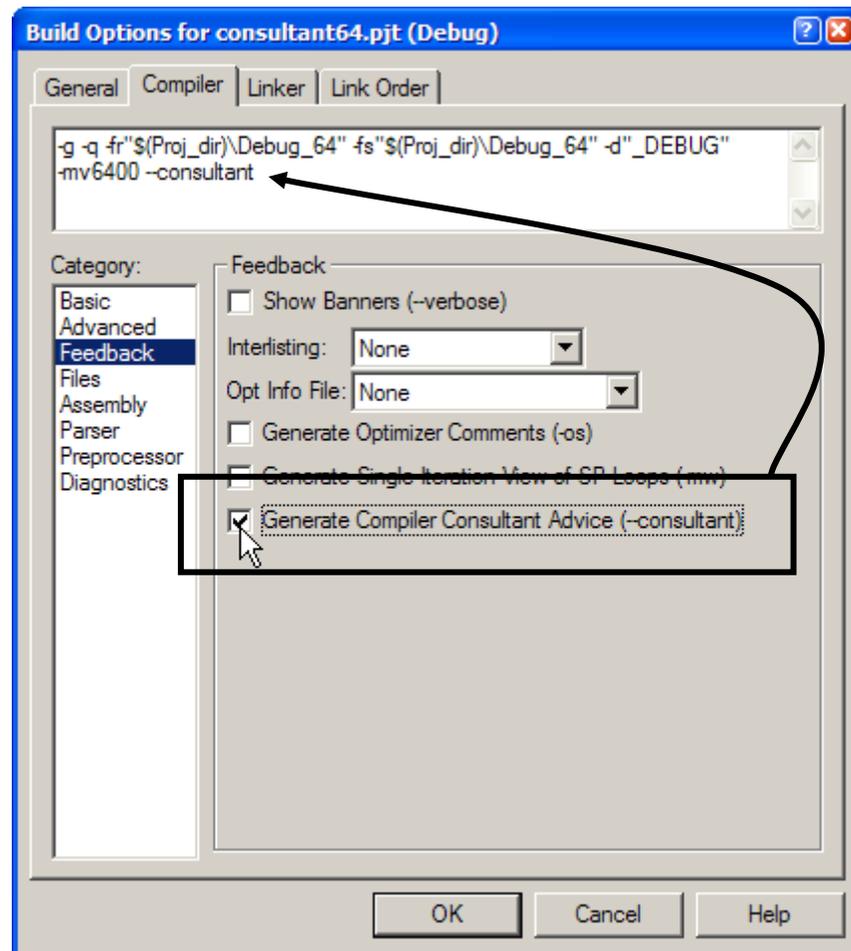
DoLoop() is called directly from main() one time. It contains only one loop. This loop is the focus of the tutorial.

7. Under the **Project** menu, choose **Build Options...**
8. In the **Build Options** dialog, click the **Compiler** tab.

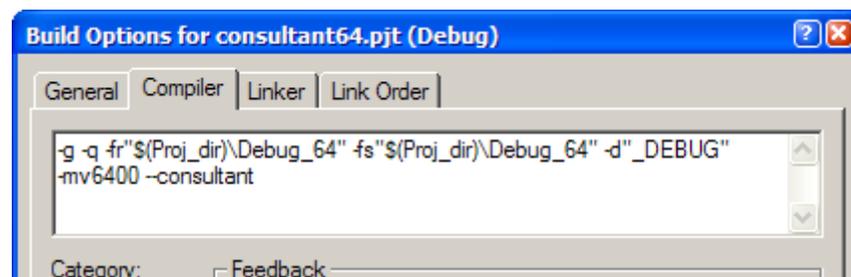
9. Click on the Feedback item in the Category list.



10. Click on the checkbox **Generate Compiler Consultant Advice (--consultant)**.



11. Your build options should resemble the following, although the path may change depending on your target:

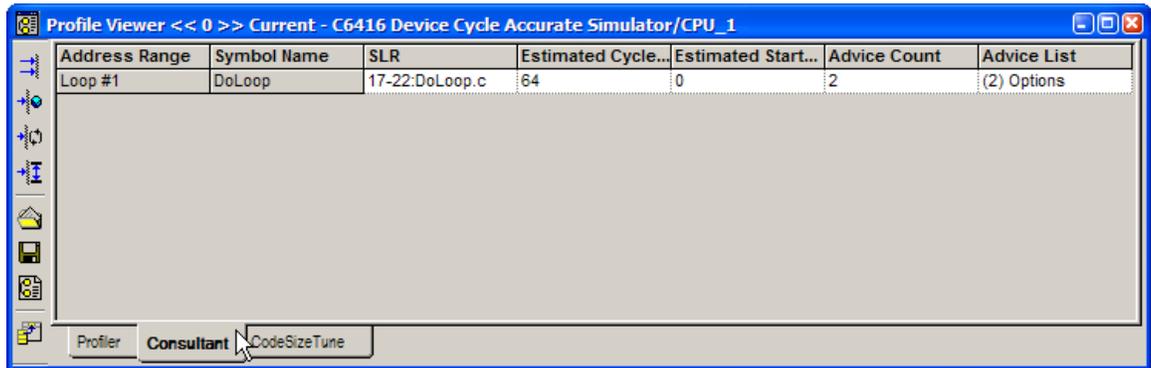


12. Click **OK** to close the **Build Options** dialog.

13. From the **Project** menu, choose **Rebuild All** or click on the **Rebuild All button** .

14. From the **Profile** menu, choose **Viewer**.

15. Once the **Profile Viewer** window appears, click on the **Consultant** tab. The one line in the Profile Viewer represents the single loop in the Consultant tutorial.

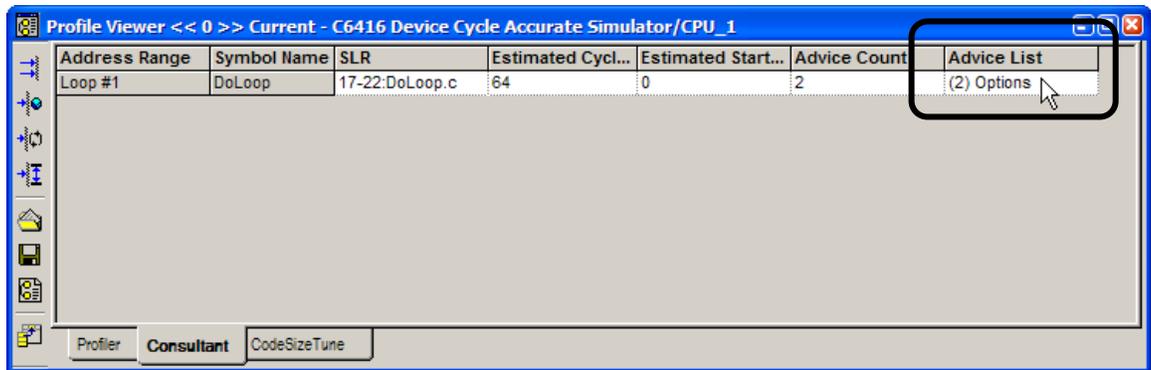


16. Notice that the Estimated Cycles Per Iteration for the loop is currently 64 cycles, and the Estimated Start Cycles is zero. Also note that the loop has two pieces of Options advice, as indicated by the Advice Count column in the Profile Viewer. For information on the columns in Profile Viewer, consult the online help file.

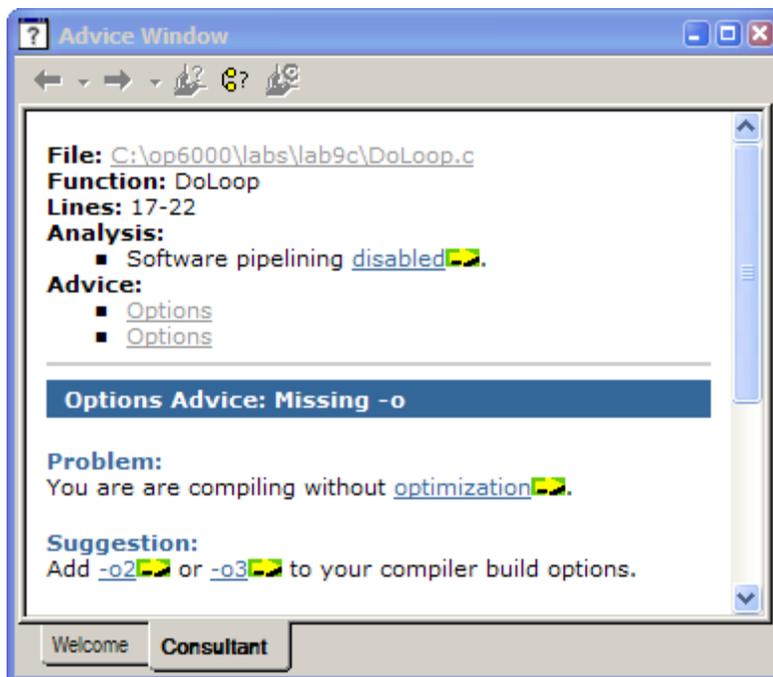
Options Advice Implementation

In this part of the lab, you will analyze the options advice given by the Consultant Tool, and implement it for the consultant.pjt project.

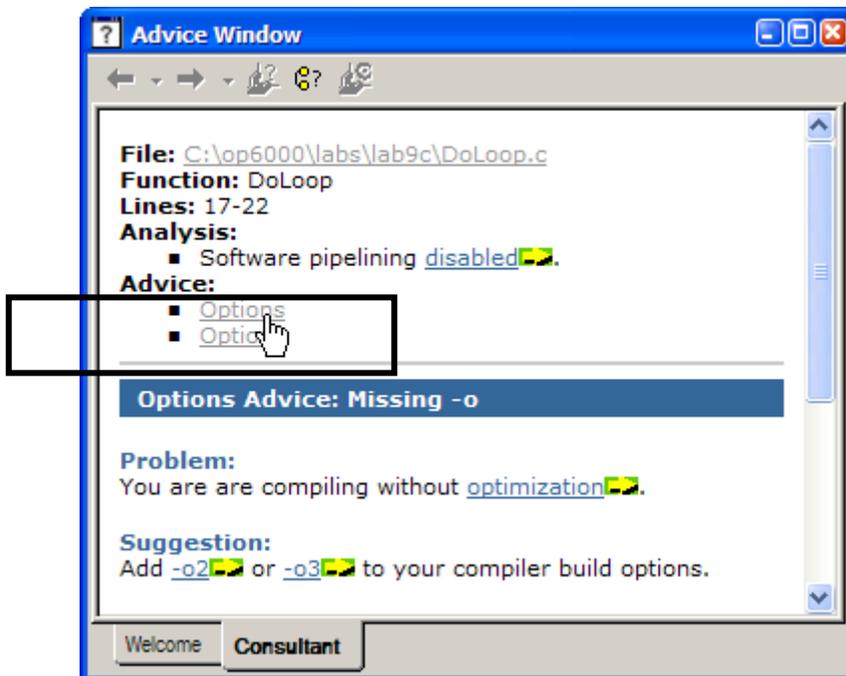
17. Double click on the **Advice List** cell for the DoLoop row.



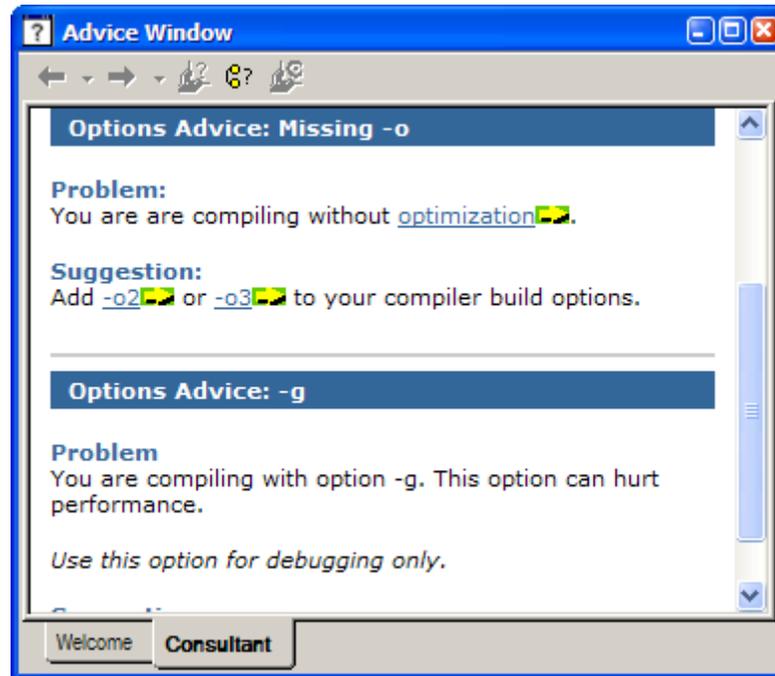
- The Advice Window appears, and the **Consultant** tab displays advice for the DoLoop function. The **Analysis** of the DoLoop functions indicates that the software pipelining optimizations in the compiler are currently disabled. Also note the two pieces of Options advice.



- Click on the link to the first piece of Options advice.



20. Read the advice offered in the suggested solution:



Note: Terminology in advice is often linked to the online help, as indicated by the online help icon: 

The suggested solution indicates you should turn on the `-o2` or `-o3` compiler option. Before doing so, let's take a quick look at the second piece of advice.

21. Scroll back to the top of the Advice window with the scroll bar and click on the second piece of advice.

The second piece of Options advice states that we are compiling with the `-g` debug option turned on. As the advice notes, compiling with `-g` hurts performance. The advice suggests turning off the `-g` option.

22. Under the **Project** menu, choose **Build Options...**

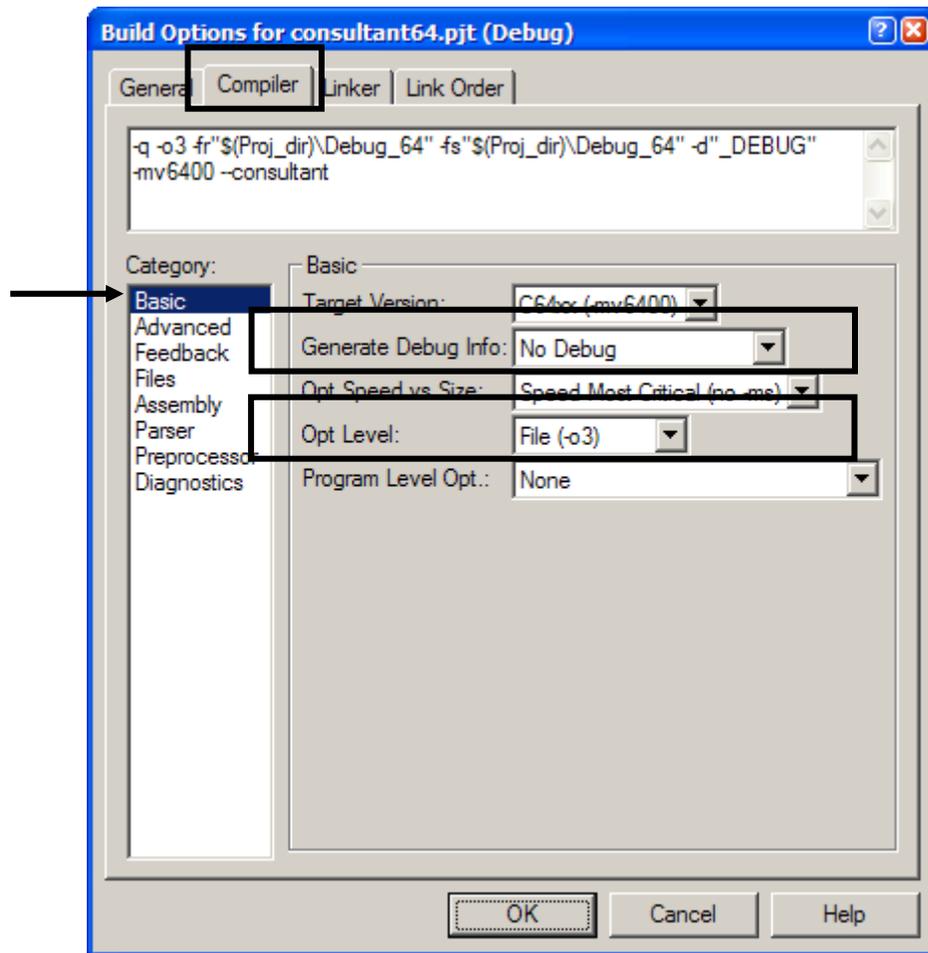
23. In the **Build Options** dialog, click the **Compiler** tab.

24. Click on the **Basic** item in the **Category** list.

25. From the **Generate Debug Info** drop down list, choose **No Debug**.

26. From the **Opt Level** drop down list, choose **File (-o3)**.

27. Your build options should be similar to:



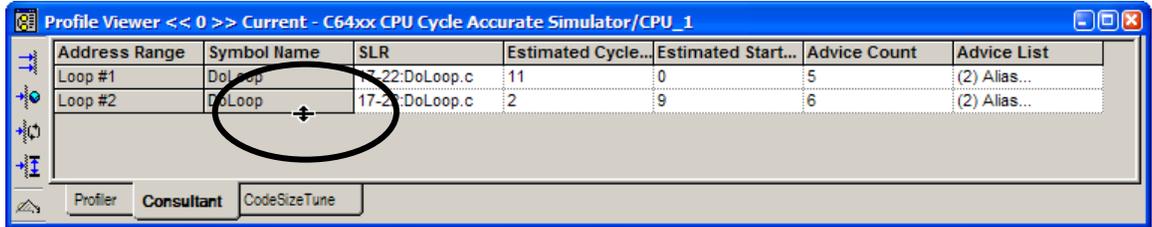
Note: We could have changed these options by switching over to the *Release* configuration, but then we would have had to add the `--consultant` option to *Release* to continue to get feedback from the compiler consultant.

28. Click **OK** to close the **Build Options** dialog.

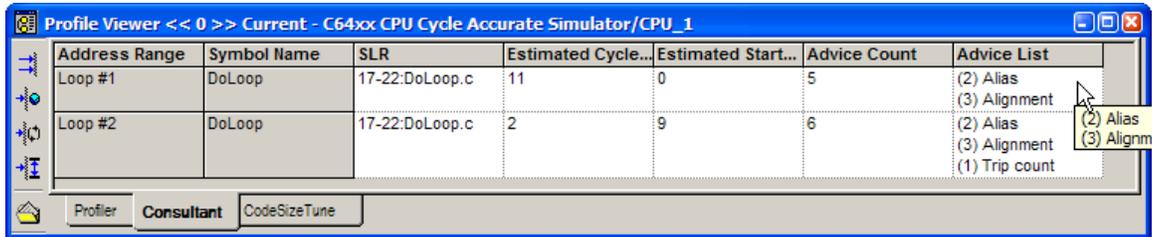
29. From the **Project** menu, choose **Build** or click on the **Incremental Build icon** .

30. After the build, the advice in the Advice Window is no longer valid, so it clears the advice and informs the user that double clicking on the Advice Count column will refresh the advice.

31. In the Profile Viewer, click on the Consultant tab if it is not active. The default row height in the profile viewer only allows you to see one advice type in the Advice List column. To increase row height, place the mouse pointer on the solid horizontal line at the bottom of the top row in the Address Range column. Double-click on that line, and the row height will automatically resize to fit all the information.



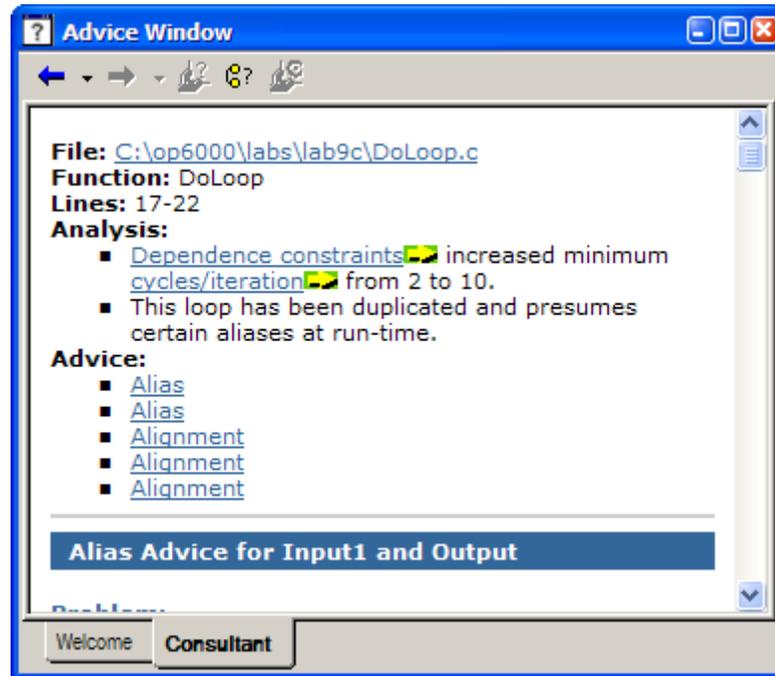
32. Double-click on the Advice Count cell for the Loop with an Estimated Cycles Per Iteration of 11 to update the Advice Window for this loop. You can also increase the row heights for the two loops in the Profile Viewer again to see all the pieces of advice.



There are now five pieces of advice for the main loop in the profile viewer, two Alias and three Alignment. Before looking at the specifics on these advice topics, the next lesson will cover loop analysis.

Loop Analysis

Look at the **Consultant** tab of the Advice Window.



The **Analysis** for this loop indicates it has been duplicated. Loop duplication can occur when the compiler is unable to determine if one or more pointers are pointing at the same memory address as some other variable. Such pointers are called *aliases*. One version of the loop presumes the presence of aliases, the other version of the loop does not. The compiler generates code that checks, at runtime, whether certain aliases are present. Then, based on that check, the compiler executes the appropriate version of the loop. The copy of the loop presently displayed in the Advice Window executes when the runtime check for aliases indicates that aliases may be present.

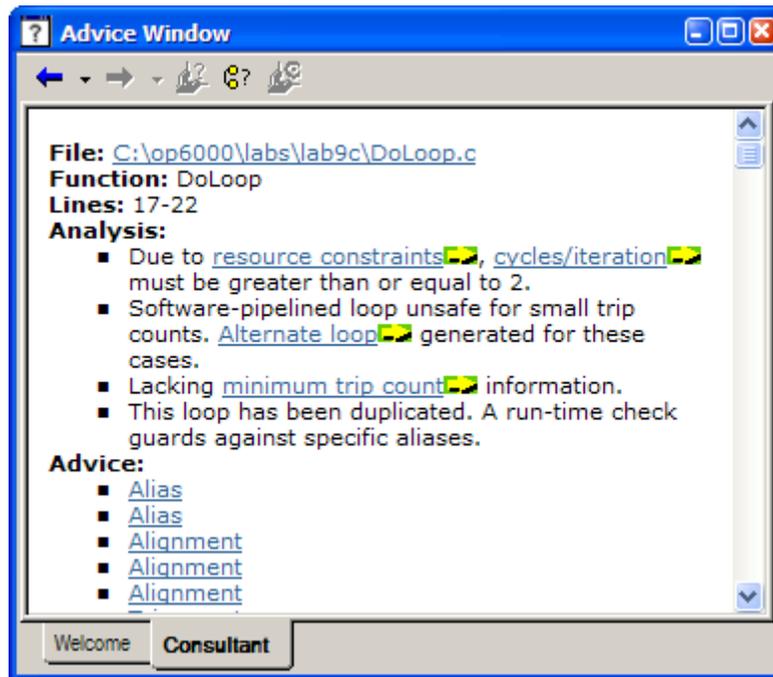
33. Double-click on the Advice Count cell in Profile Viewer for the other row.

The screenshot shows the 'Profile Viewer' window with the following table:

Address Range	Symbol Name	SLR	Estimated Cycle...	Estimated Start...	Advice Count	Advice List
Loop #1	DoLoop	17-22:DoLoop.c	11	0	5	(2) Alias (3) Alignment
Loop #2	DoLoop	17-22:DoLoop.c	2	9	6	(2) Alias (3) Alignment (1) Trip count

At the bottom of the window, there are three tabs: 'Profiler', 'Consultant', and 'CodeSizeTune'.

34. Look at the analysis of that loop. It has a similar note about a duplicated loop, but it presumes certain aliases are not present.



Further note the bullet about the alternate loop. This is a loop that is executed when a different runtime check for a minimum number of loop iterations has failed. This alternate loop is not software pipelined. For this reason, the alternate loop has no entry in the Profile Viewer.

This pseudo-code summarizes the relationship between the three variations of the loop.

```

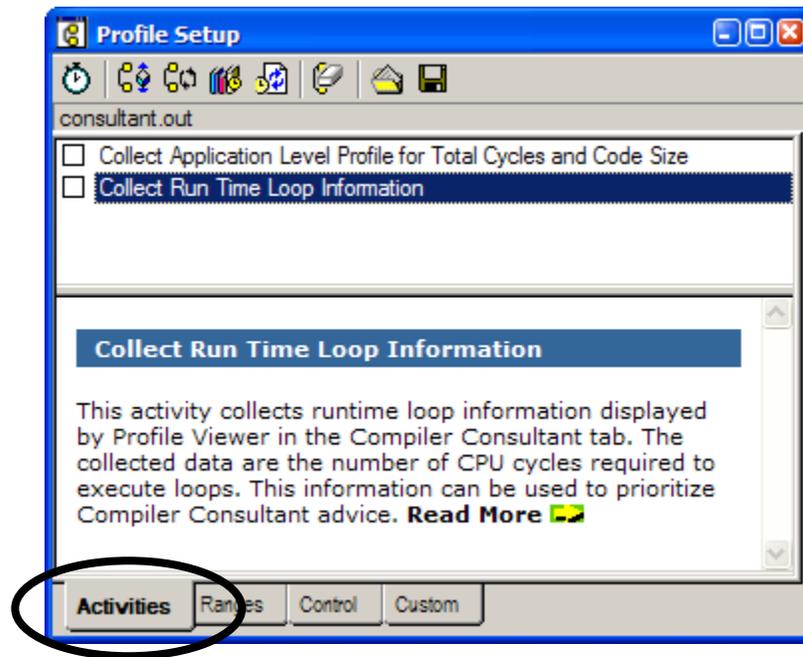
if (enough loop iterations) {
    if (aliases are present)
        loop #1
    else
        loop #2
}
else
    alternate loop // not in profile viewer

```

35. We can profile the code to see which loop versions get selected when the code is executed. From the **File** menu, choose **Load Program** to start the program load.
36. Browse to **C:\op6000\labs\lab9c\Debug_64** and double-click on **consultant.out** to load it.
37. From the **Profile** menu, choose **Setup**.

Profile → Setup

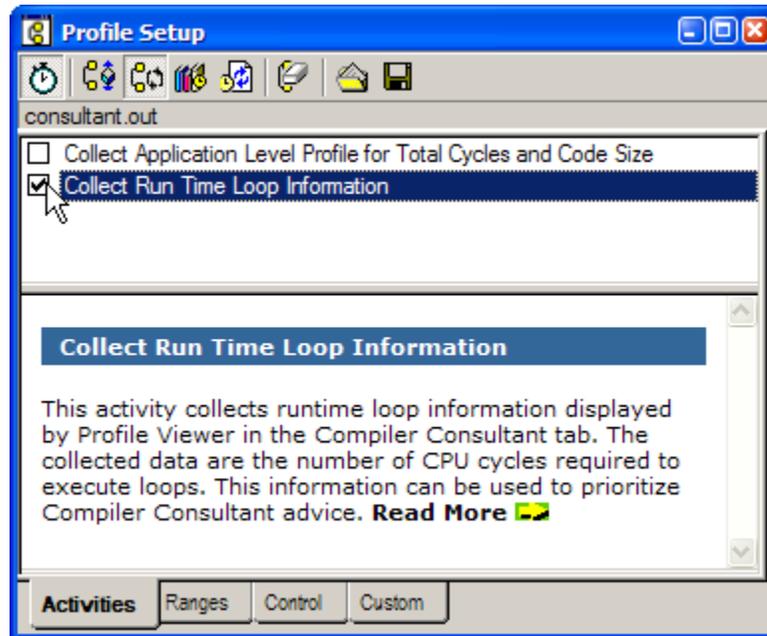
38. Select the **Activities** tab near the bottom of the window.



39. Check the icons at the top of the Profile Setup tool to determine if profiling is enabled. The

Enable/Disable Profiling icon  must be toggled on to enable profiling. Profiling may not be enabled if the program has started running or if it was explicitly disabled. Data will not be collected if profiling is not enabled.

40. Place a check next to **Collect Run Time Loop Information**.

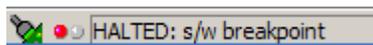


This activity will count the cumulative total of CPU cycles, ignoring system effects, executed in each version of the loop. System effects that it ignores include things like cache misses, memory bank conflicts, off-chip memory access latency, etc.

Note: Your execution platform may not support collection of the event cycle.CPU, and so will not show the activity Collect Run Time Loop Information. In that case, choose the activity **Profile All Functions and Loops for Cycles** instead. This activity also counts the cumulative total of the CPU cycles, but includes system affects. Further, in addition to profiling all versions of the loop, it profiles each function. In the sections which follow, use the column cycle.Total:Excl.Total in place of cycle.CPU:Excl.Total. For more information on runtime profiling events access, see the online help file.

41. Run the program by choosing **Debug**→**Run** or **clicking on the Run Icon** .

42. Wait until the program halts. One indicator is the message **HALTED** in the lower left corner:



43. In the **Profile Viewer**, click on the Consultant tab if it is not active, and hover the mouse over the column that starts with cycle.CPU to see that the full name of the column is "cycle.CPU,Excl.Total". You may have to scroll to the right to find this column.

The screenshot shows the Profile Viewer window with the Consultant tab selected. The table displays the following data:

Address Range	Symbol Name	Estimated Cycle...	Estimated Start...	Advice Count	Advice List	cycle.CPU,Excl.
0:0xac-0xd4	DoLoop	11	0	5	(2) Alias...	400
0:0x140-0x164	DoLoop	2	9	6	(2) Alias...	0

Note the loop with the 6 pieces of advice has 0 in the cycle.CPU:Excl.Total column. This means the loop for which the analysis states aliases are not present did not execute. The loop that did execute is the one for which aliases are presumed to be present. That is because the runtime check for aliases concluded that aliases may be present. In fact, there are no aliases among these pointers. The runtime check concluded that aliases may be present because it is too conservative; it was too cautious.

To clarify this point, please find the pseudo-code below with comments (in green) to indicate the actual path of execution.

```

if (enough loop iterations)
{
  /* executes because there are enough loop iterations */
  if (aliases are present)
  /* executes because test for aliases is too conservative */
  loop #1 else
  /* does not execute */
  loop #2 } else
  /* does not execute */
  alternate loop // not in profile viewer

```

44. The loops are presently ordered, from highest to lowest, by the number of cycles executed. This is unlikely to happen in a real application. Typically, information from many loops is present, and sorting the loops is a good way to determine which to focus on first. *You can sort on any column by double-clicking on the column header.*

Double click multiple times to sort the rows, toggling between ascending and descending order.

45. We will now focus on the executed loop that currently takes 400 cycles. As we implement the alias and trip count advice for this loop, the compiler will no longer generate the other versions of the loop, and they will no longer appear in the Profile Viewer.

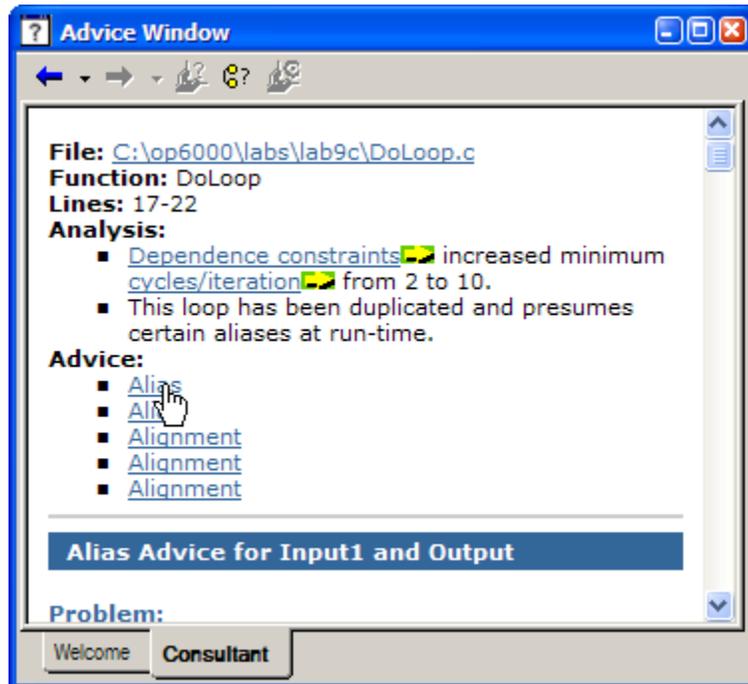
Double click on the **Advice Count** cell for the loop which takes 400 cycles.

There are five pieces of advice for the main loop, two Alias and three Alignment. We will first look at the Alias advice and solve it, before moving on to the Alignment advice.

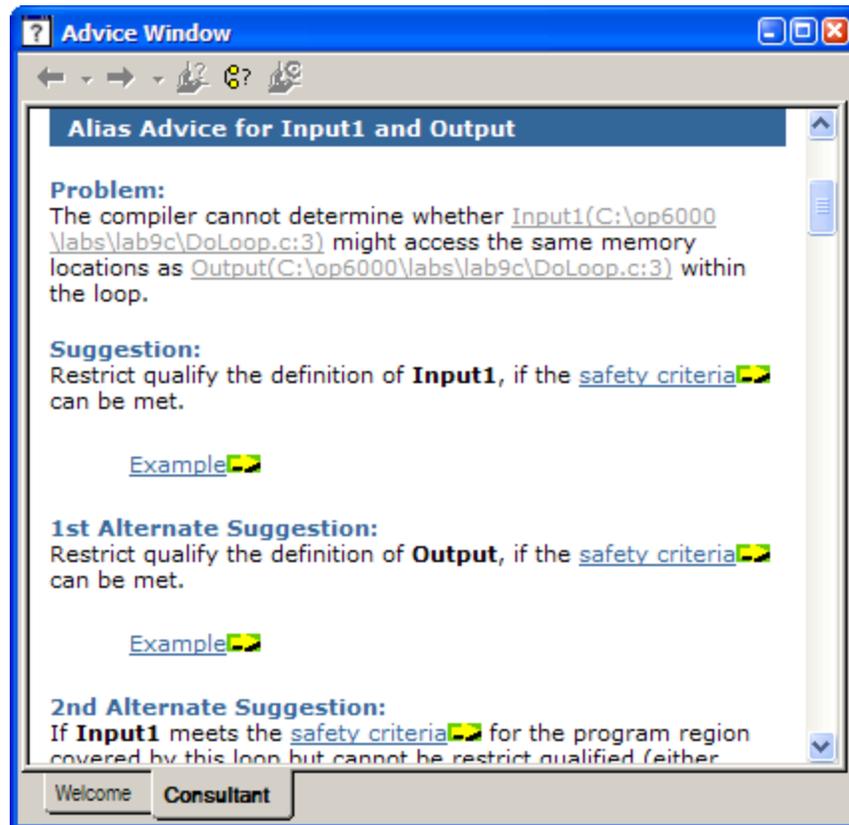
Alias Advice Implementation

In this lesson, you will analyze the alias advice given by the Compiler Consultant Tool, and implement it for the consultant.pjt project.

46. In the **Consultant** tab of the **Advice Window**, click on the link to the first piece of **Alias** advice.



The problem statement indicates the compiler cannot determine if two pointers may point to the same memory location, and therefore cannot apply more aggressive optimizations to the loop. The Consultant Tool offers several suggestions to solve the problem.

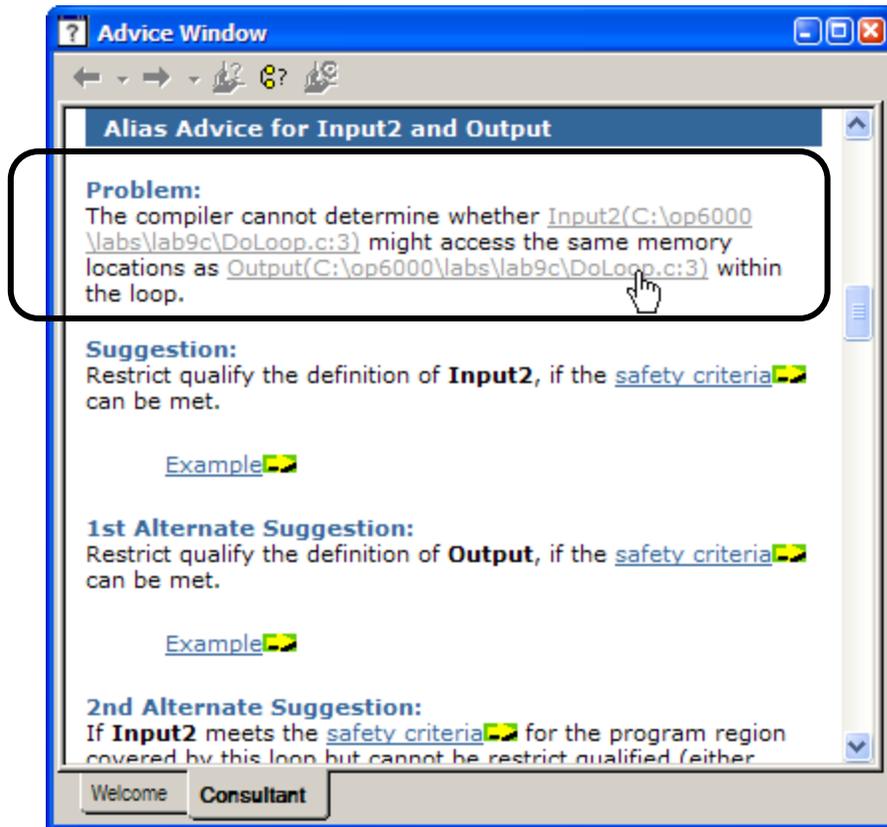


The primary suggestion tells us to use the restrict qualifier on the Input1 pointer if the safety criteria can be met. Inspect the call to the function DoLoop, and notice that the array passed in as the argument for Input1 is the global array also named Input1. This array does not overlap any of the arrays passed in for the other arguments. This means that the memory referenced by the Input1 variable cannot be referenced by any other variable in the function. Therefore, the Input1 variable meets the safety criteria required for correct use of the restrict keyword. Before we apply the primary suggestion, **let's take a look at the alternate suggestions.**

The first alternate suggestion indicates that using the restrict qualifier on the Output pointer could solve the problem. Finally, the second and third alternate suggestions indicate that restrict qualifying a local copy of the Output or Input1 pointer would also solve the problem.

47. Go back the top of the advice window and **click the link to the second piece of Alias advice.** This piece of alias advice is similar to the first, except it mentions pointers Output and Input2.
48. Because there are two pieces of alias advice, one for Output and Input1, and the other for Output and Input2, restrict qualifying Output resolves both problems.

Click on the link that says **Output** in the **Problem** section of the **Advice Window** to open the DoLoop.c file on the line that declares Output as an argument passed in to the DoLoop function.

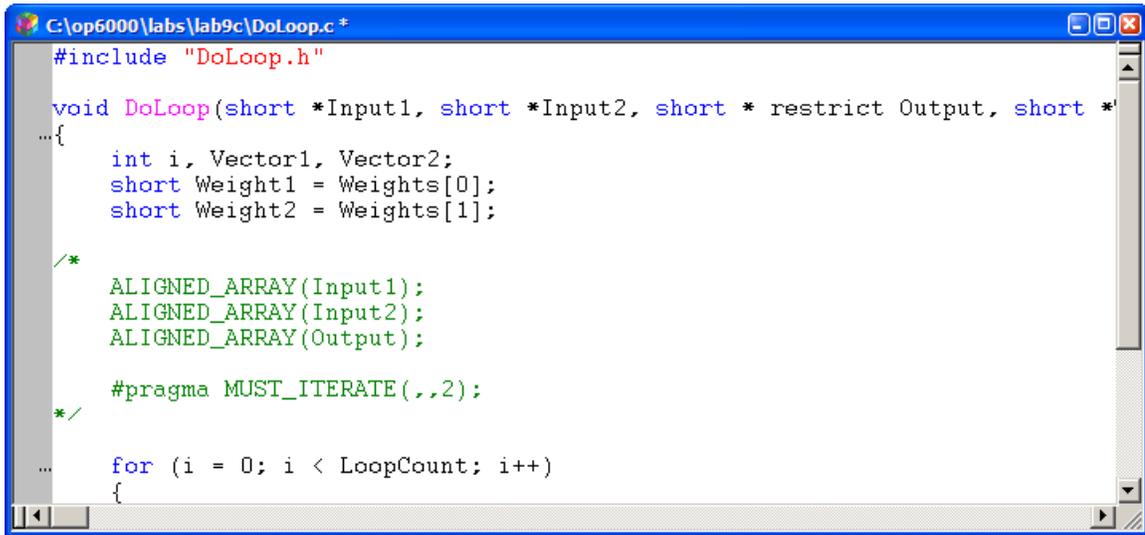


49. Modify the DoLoop.c to remove the C comment delimiters `/**/` around the restrict qualifier to the Output pointer parameter.

Before:

```
void DoLoop(short *Input1, short *Input2, short * /**restrict*/ Output,  
short *Weights, int LoopCount)
```

After:



```

C:\op6000\labs\lab9c\DoLoop.c *
#include "DoLoop.h"

void DoLoop(short *Input1, short *Input2, short * restrict Output, short *
...{
    int i, Vector1, Vector2;
    short Weight1 = Weights[0];
    short Weight2 = Weights[1];

    /*
    ALIGNED_ARRAY(Input1);
    ALIGNED_ARRAY(Input2);
    ALIGNED_ARRAY(Output);

    #pragma MUST_ITERATE(,,2);
    */

    for (i = 0; i < LoopCount; i++)
    {

```

50. Save the file with menu item **File**→**Save**, or **Control+S**.
51. In the **Project View**, open the Include folder and double-click on the **DoLoop.h** file.
52. Remove the C comment delimiters **/**/** around the restrict qualifier to the Output pointer parameter.

Before:

```

void DoLoop(short *Input1, short *Input2, short * /*restrict*/ Output,
short *Weights, int LoopCount);

```

After:

```

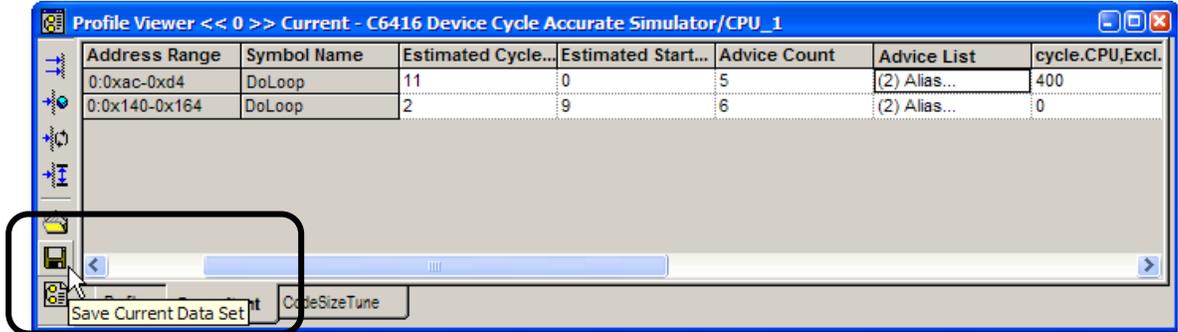
void DoLoop(short *Input1, short *Input2, short * restrict Output, short
*Weights, int LoopCount);

```

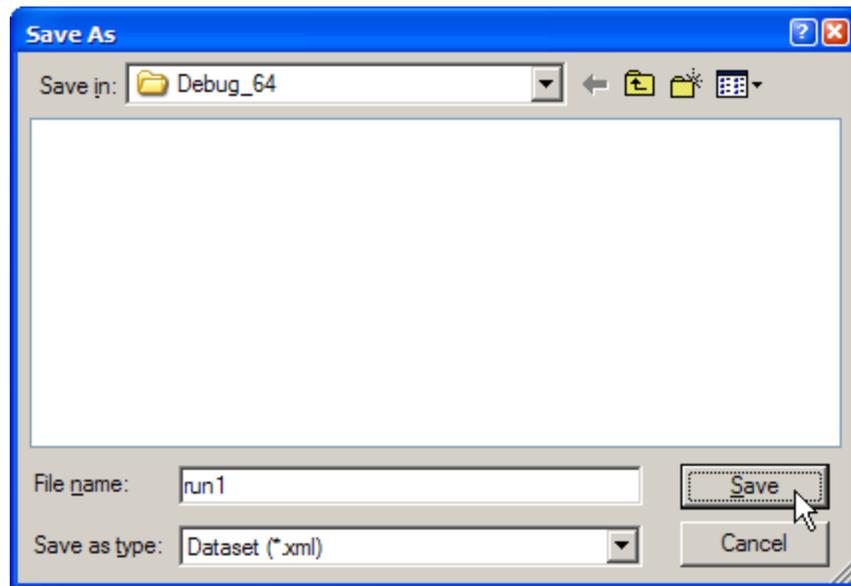
53. Save the file with menu item **File**→**Save**, or **Control+S**.

54. Source changes are complete, so it is time to build. However, building the project removes the cycle count Profile Viewer data, because it will be invalid. The following steps show how to save Profile Viewer data.

In the Profile Viewer toolbar on the left, click the icon  for **Save Current Data Set**.



55. Browse to **C:\op6000\labs\lab9c\Debug_64**.
56. Enter **run1** for the file name.
57. Select **Dataset (*.xml)** for the Save As type, and click **Save**.

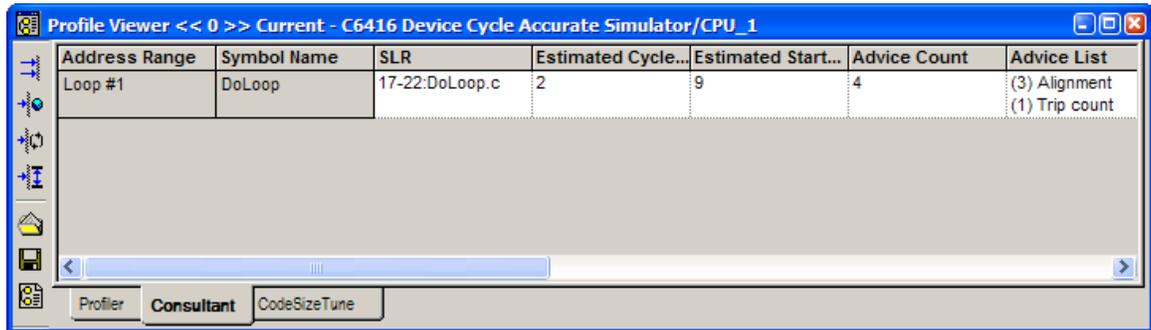


To see this data again later, you can use the Load Data Set  button in the Profile Viewer.

58. From the **Project** menu, choose **Build** or **click on the Build Icon** .

Note: The cycle count data remains in the Profiler tab of the Profile Viewer even after a build, and remains there until another round of profile data is collected.

59. In the Profile Viewer, click on the Consultant tab if it is not active, and **double-click on the Advice Count cell for DoLoop** to update the Advice Window for this loop. You can also increase the row height for the loop in the Profile Viewer again to see more information.



Address Range	Symbol Name	SLR	Estimated Cycle...	Estimated Start...	Advice Count	Advice List
Loop #1	DoLoop	17-22:DoLoop.c	2	9	4	(3) Alignment (1) Trip count

Notice that the Estimated Cycles Per Iteration for the loop has been reduced from 11 cycles to 2 cycles, while the Estimated Start Cycles has changed from zero cycles to 9 cycles.

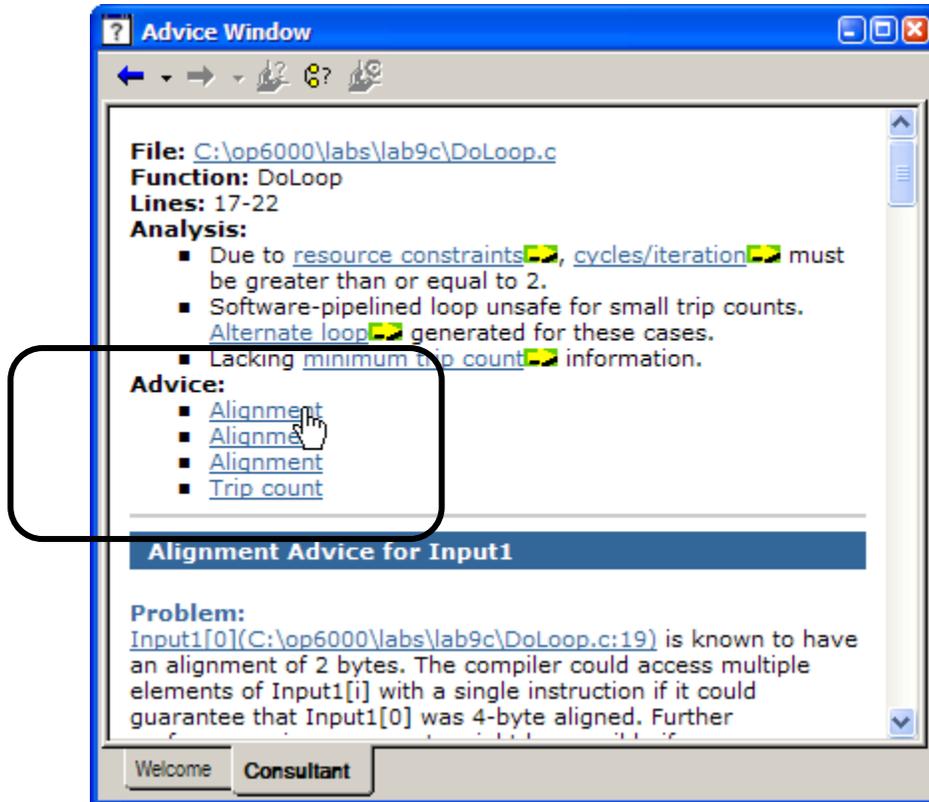
There are now four pieces of advice, three Alignment and one Trip Count. We will look at the Alignment advice first and solve it before moving on to the Trip Count advice.

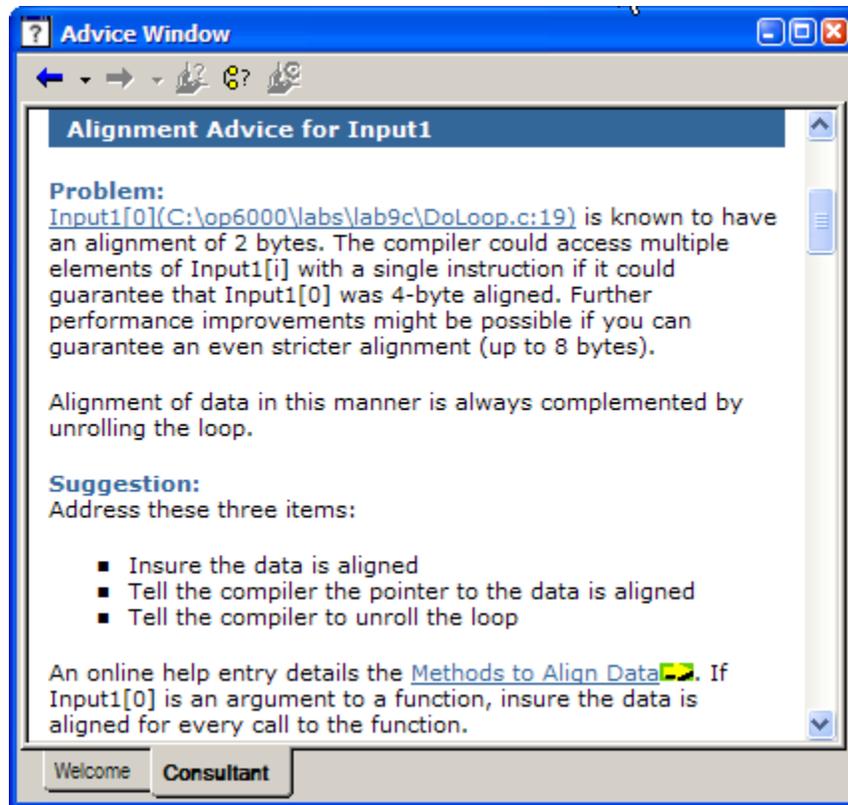
Alignment and Trip Count Advice Implementation

In this lesson, you will analyze the alignment and trip count advice given by the Consultant Tool, and implement it for the consultant.pjt project.

Alignment Advice

60. In the **Consultant** tab of the **Advice Window**, click on the link to the first piece of **Alignment** advice.





The problem statement indicates the compiler could access multiple elements of the `Input1` array if it could verify that the array was aligned on a four or eight byte boundary. The suggestion recommends addressing three items: aligning the data, indicating the pointers are aligned, and unrolling the loop.

- The help entry on aligning data notes that arrays are automatically aligned on an 8-byte boundary for C64xx, and a 4-byte boundary for C62xx and C67xx. The function `DoLoop` receives the base address of an array for all three pointer arguments. So, no further steps are required to align the data.
- For indicating the pointer holds the base address of an array, the help entry suggests a macro. So, we have the macro in the header file `DoLoop.h`, as it is a more comprehensive solution than the simple `_nassert` given in the advice.
- For unrolling the loop, the advice suggests the use of the `MUST_ITERATE` pragma. For this simple case, we could indicate the multiple of loop iterations is 40. However, 2 is a more realistic loop count multiple.

The purpose of the `MUST_ITERATE` pragma is to tell the compiler about properties of the trip count of the loop. The trip count is the number of times the loop iterates. The syntax is:

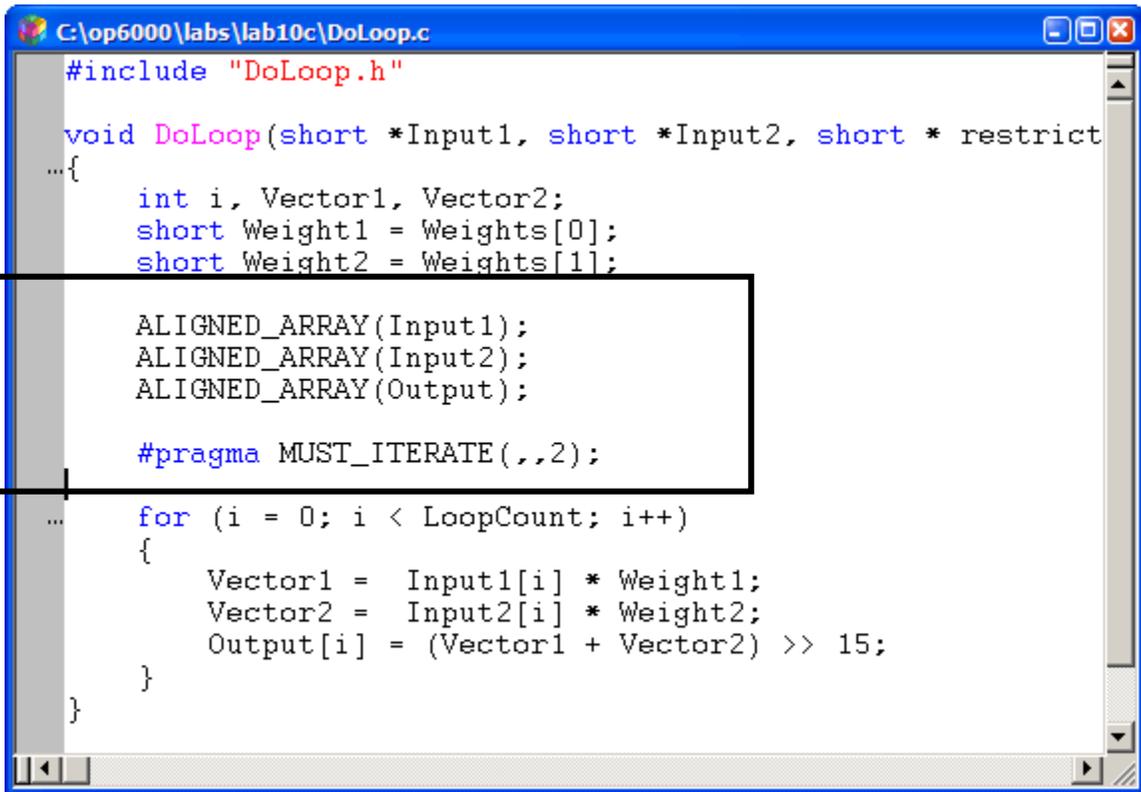
```
#pragma MUST_ITERATE([min, max, multiple]);
```

The arguments `min` and `max` are programmer-guaranteed minimum and maximum trip counts. The trip count of the loop must be evenly divisible by `multiple`. All arguments are optional. For information on the `MUST_ITERATE` argument, see the online help file.

The next two pieces of Alignment advice indicate the advice for Input1 is also applicable for the pointers Input2 and Output. This simple tutorial only calls DoLoop once. In a real application, every call to DoLoop must be inspected to insure that the base of an array is always passed to the pointers, and the loop iteration count is always a multiple of 2.

61. If the source file is not already open, click on the **source file name** (C:\op6000\labs\lab9c\DoLoop.c) at the top of the Advice Window to bring up the DoLoop.c file in the Editor.
62. DoLoop.c already contains all of the indicated changes as comments. Remove the comment delimiters (*/* */*) from the ALIGN_ARRAY macro and the MUST_ITERATE pragma.

Code After Changes:



```
C:\op6000\labs\lab10c\DoLoop.c
#include "DoLoop.h"

void DoLoop(short *Input1, short *Input2, short * restrict
...{
    int i, Vector1, Vector2;
    short Weight1 = Weights[0];
    short Weight2 = Weights[1];

    ALIGNED_ARRAY(Input1);
    ALIGNED_ARRAY(Input2);
    ALIGNED_ARRAY(Output);

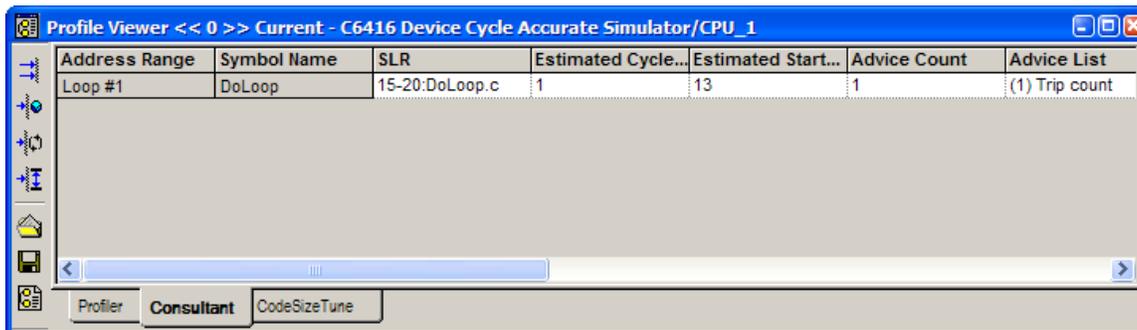
    #pragma MUST_ITERATE(,,2);

    ...
    for (i = 0; i < LoopCount; i++)
    {
        Vector1 = Input1[i] * Weight1;
        Vector2 = Input2[i] * Weight2;
        Output[i] = (Vector1 + Vector2) >> 15;
    }
}
```

63. Save the file with **File**→**Save** or **control+S**.

64. From the **Project** menu, choose **Build** or **click on the Build Icon** .

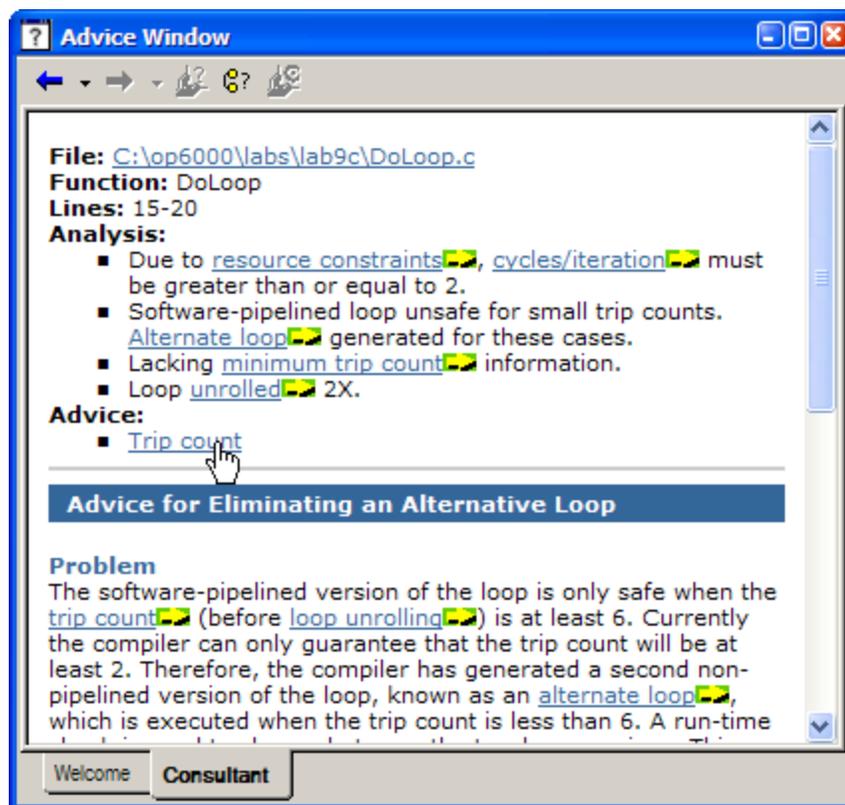
65. In the **Profile Viewer**, click on the Consultant tab if it is not active, and double-click on the DoLoop **Advice Count** column to update the Advice Window.

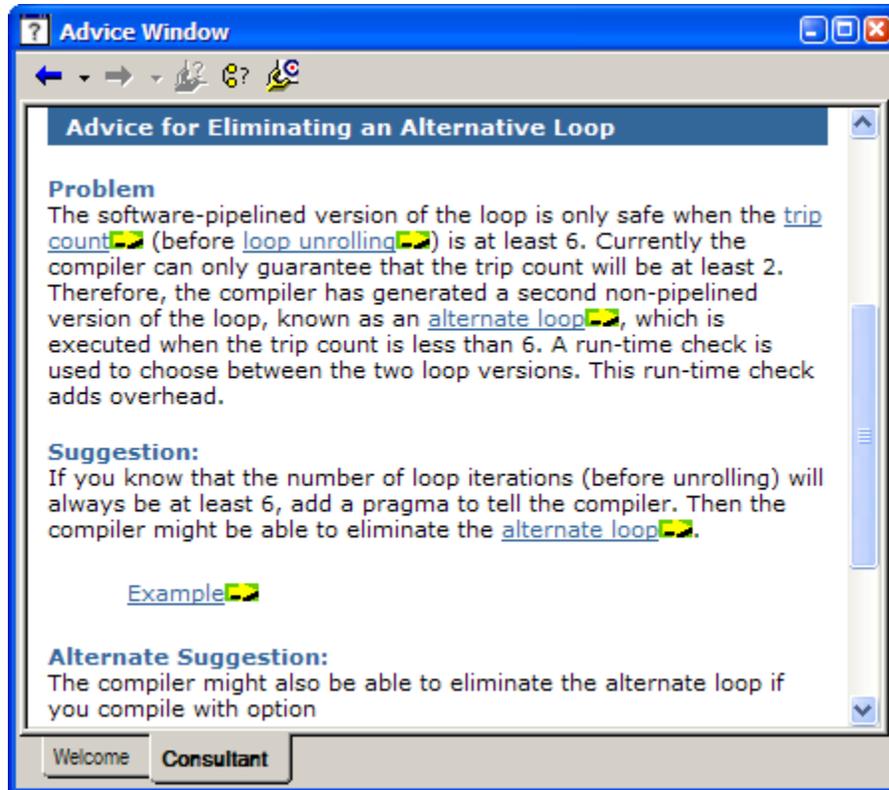


Notice that the Estimated Cycles Per Iteration for the loop improves to 1 cycle. The Estimated Start Cycles is now 13 cycles.

Trip Count

66. In the Advice Window, click on the link to the Trip Count advice.





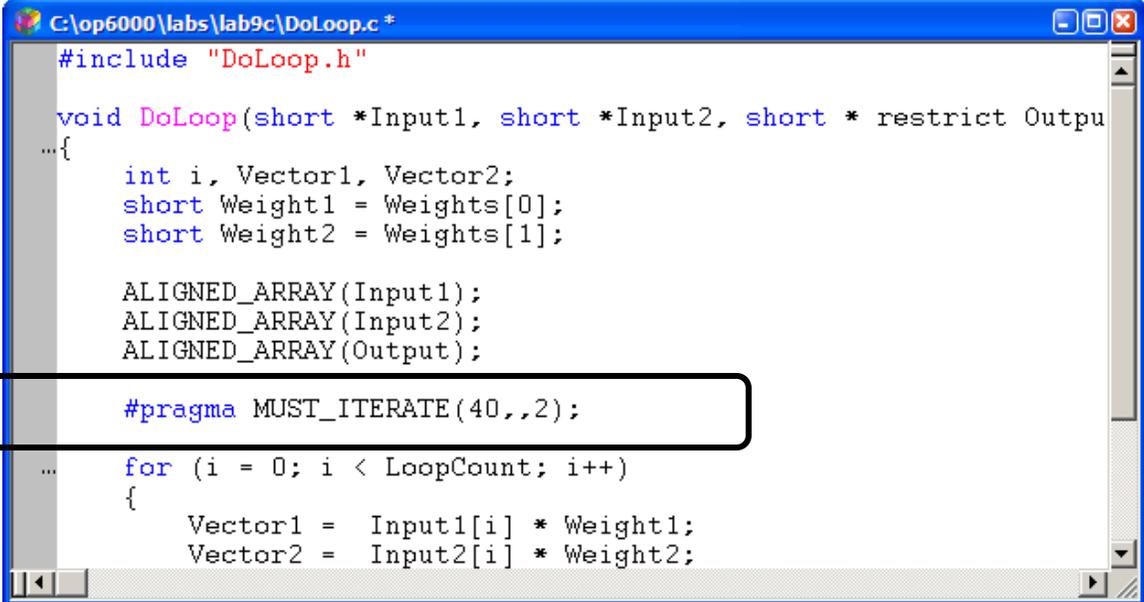
The problem statement indicates the compiler has no guarantee about the trip count of the loop. The primary suggestion suggests the use of the `MUST_ITERATE` pragma to indicate that the loop iterates at least 6 times. The alternate advice suggests using the `-mh` compiler option. In our case, we know that the loop must iterate 40 times, so we will apply the `MUST_ITERATE` pragma. In a real application, every call to `DoLoop` must be inspected to insure a loop count of at least 40.

67. If the source file is not already open, click on the **source file name** (`C:\op6000\labs\lab9c\DoLoop.c`) at the top of the Advice Window to bring up the `DoLoop.c` file in the Editor.

68. Add **40** as the first argument to the `MUST_ITERATE` pragma.

Before: `#pragma MUST_ITERATE(,2);`

After:



```

C:\op6000\labs\lab9c\DoLoop.c *
#include "DoLoop.h"

void DoLoop(short *Input1, short *Input2, short * restrict Output
...{
    int i, Vector1, Vector2;
    short Weight1 = Weights[0];
    short Weight2 = Weights[1];

    ALIGNED_ARRAY(Input1);
    ALIGNED_ARRAY(Input2);
    ALIGNED_ARRAY(Output);

    #pragma MUST_ITERATE(40,,2);

    ...
    for (i = 0; i < LoopCount; i++)
    {
        Vector1 = Input1[i] * Weight1;
        Vector2 = Input2[i] * Weight2;
    }
}

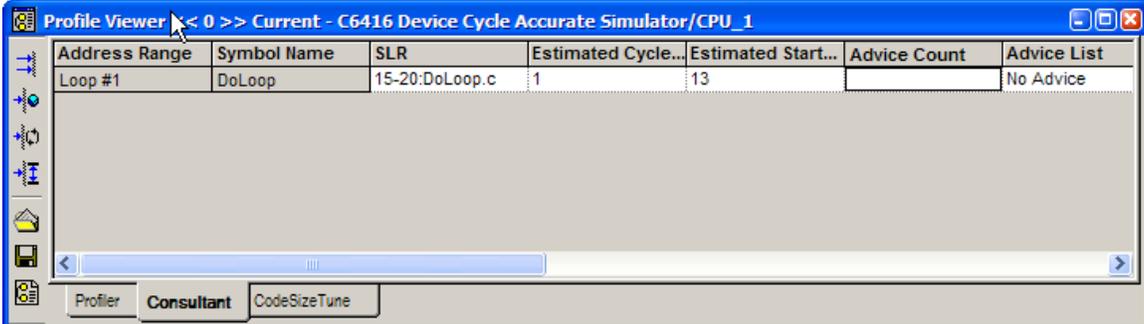
```

69. Save the file with **File**→**Save** or **control+S**.



70. From the **Project** menu, choose **Build** or **click on the Build Icon**.

71. In the **Profile Viewer**, click on the Consultant tab if it is not active, and double-click on the **DoLoop Advice Count** column to update the Advice Window.



Address Range	Symbol Name	SLR	Estimated Cycle...	Estimated Start...	Advice Count	Advice List
Loop #1	DoLoop	15-20:DoLoop.c	1	13	1	No Advice

The Estimated Cycles Per Iteration remains at 1 cycle, and the Estimated Start Cycles remains at 13.

But the alternate loop, for low trip counts, is eliminated. This saves a small amount of runtime overhead, and fair amount of code space.

The Advice Count is now 0, and we have reduced the total cycle time from an initial value of $64 * 40 = 2560$ cycles to $13 + (1 * 40) = 53$ cycles, or roughly a 48X improvement.

Throughout this lab we have been applying compiler optimization changes that not only affected performance, but also affected the code size. Code size versus performance is a trade-off that only the designer can make. To help you evaluate code size versus performance for the DoLoop function, you can use the CodeSizeTune tool, which we will discuss in the next chapter.

For more information, see the application note *Introduction to Compiler Consultant*.

67

Change back to the ‘C67x Simulator

Now that we’re done with this lab, we need to reconfigure the CCS setup to use the ‘C67x Simulator for future labs. Launch CCS Setup:

File → Launch Setup

When setup opens, the “Import Configuration” window appears. Follow the procedure found at the start of Lab 4. These are the basic steps:

- Clear the previous system configuration (button on right).
- Scroll through the list to find *C67xx CPU Cycle Accurate Sim, Ltl Endian*. Select this item and click **ADD**
- Quit and save CCS Setup. When asked, allow CCS Setup to restart CCS for you.

67+

Change back to the ‘C672x Simulator

Now that we’re done with this lab, we need to reconfigure the CCS setup to use the ‘C64xp Simulator for future labs. Launch CCS Setup:

File → Launch Setup

When setup opens, the “Import Configuration” window appears. Follow the procedure found at the start of Lab 4. These are the basic steps:

- Clear the previous system configuration (button on right).
- Scroll through the list to find *C672x CPU Simulator, Ltl Endian*. Select this item and click **ADD**
- Quit and save CCS Setup. When asked, allow CCS Setup to restart CCS for you.

64xp

Change back to the ‘C64xp Simulator

Now that we’re done with this lab, we need to reconfigure the CCS setup to use the ‘C67x Simulator for future labs. Launch CCS Setup:

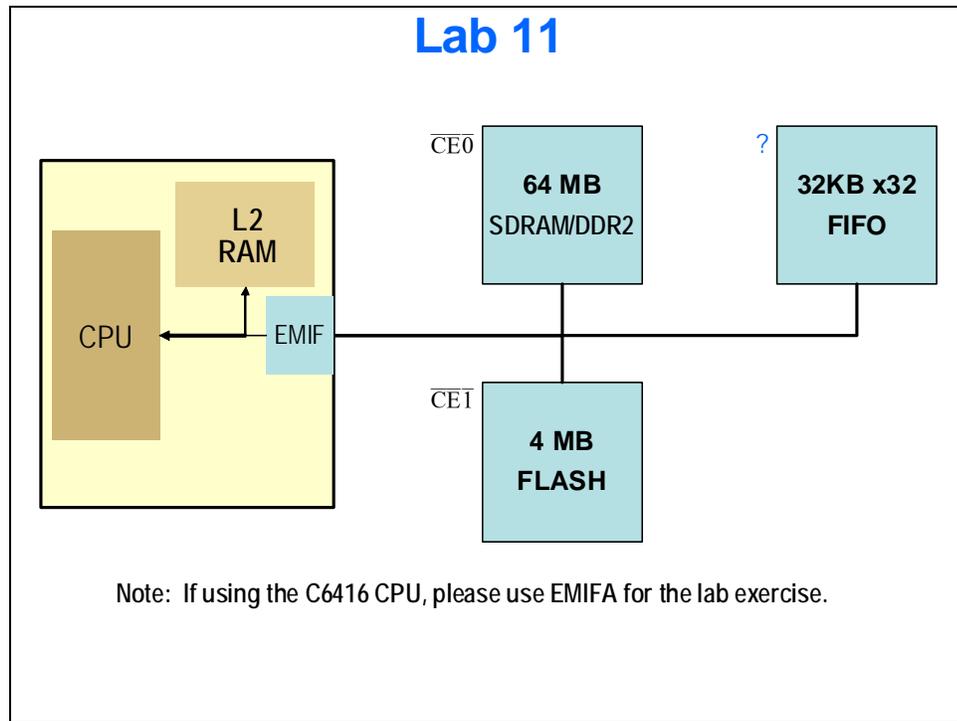
File → Launch Setup

When setup opens, the “Import Configuration” window appears. Follow the procedure found at the start of Lab 4. These are the basic steps:

- Clear the previous system configuration (button on right).
- Scroll through the list to find *C64xx CPU Cycle Accurate Sim, Ltl Endian*. Select this item and click **ADD**
- Quit and save CCS Setup. When asked, allow CCS Setup to restart CCS for you.

Lab 11

The goal of this lab is to get your program running on the system described by the following block diagram. This requires translating this block diagram into the Config Tool's MEM manager. Then use CCS to translate our object files into the final .OUT file.



Notice, three addresses have been left for you to determine. Using the appropriate memory map, you should be able to fill in these three missing addresses.

LAB11 Project Files

The files included in this project/lab.

LAB11.PJT	copy from LAB2 folder
MAIN.C	copy from LAB2 folder
LAB.TCF	copy from LAB2 folder, we'll edit this file
LABCFG.CMD	copy from LAB2 folder
PROFILER.INI	copy from LAB2 folder
DATA.H	provided for you

Change Simulator Model

1. Since this lab explores system events beyond just the CPU, we need to reconfigure CCS to use a different simulator model.

Note: At this time there is only one simulator for the C672x family. It is a CPU simulator which means that a flat memory model is assumed. In other words, the simulator does not comprehend the timing of the External Memory Interface. If you have been using the C672x for your lab exercises you will not need to change the simulator configuration for this lab. Cycle counts for external memory accesses will not be accurate but external memory can be simulated.

Also, the C6455 simulator is not currently working in the CCSv3.3, so we suggest that you switch over to the C6416 device simulator, as described below.

Launch CCS Setup, either from within CCS or using the desktop icon.

File → Launch Setup

2. When setup opens, follow the procedure found in Lab 2a. These are the basic steps:
 - Clear the previous system configuration with the **Remove All** button.
 - Use the *Family, Platform, Endianness* filters to locate one of these configurations
 - **C6416 Device Cycle Accurate Simulator, Little Endian**
 - **C6713 Device Cycle Accurate Simulator, Little Endian**
 - Select this appropriate entry and click **Add**
 - Save and Quit the CCS Setup.
3. Start CCS.

Define Your System's Memory Map

4. Using the diagram above, along with the memory maps (Student Notes pages 11-8 to 11-10), fill in the table for the processor you selected. You can choose to use either the C6713 or the C6416 devices.

Which device did you choose? _____

The TCF templates each name the L2 memory object differently:

- C64.TCF uses ISRAM
- C67.TCF uses IRAM

In this lab, you may see us refer to the L2 memory segment using either name. In each case, please substitute the name appropriate for your target.

Lab 11 - Memory Map C641x and C6713				
Memory	CE Space	Starting/Base Address	Length (hex bytes)	Type of Space (code/data/both)
L2 Internal Mem (ISRAM)		0x0000 0000		Both
SDRAM (64M Bytes)	$\overline{CE0}$		0x0400 0000	Both
Flash (4M Byte)	$\overline{CE1}$			
FIFO (32KB)			4	

Hint: From a hardware perspective, it is easier to design the target board if each of these memories are placed into separate memory regions.

Here's a little conversion aid:

Decimal	Hex
1K	400
16K	4000
32K	8000
64K	1 0000
256K	4 0000
1M	10 0000
4M	40 0000
16M	100 0000
64M	400 0000

Why is the FIFO only 4 bytes long? _____

Create project

5. Close any previous projects that may be open within CCS.
6. Using Windows Explorer, *COPY* the following files from **C:\op6000\labs\lab2** to **C:\op6000\labs\lab11**. Please make sure to *COPY* these files.

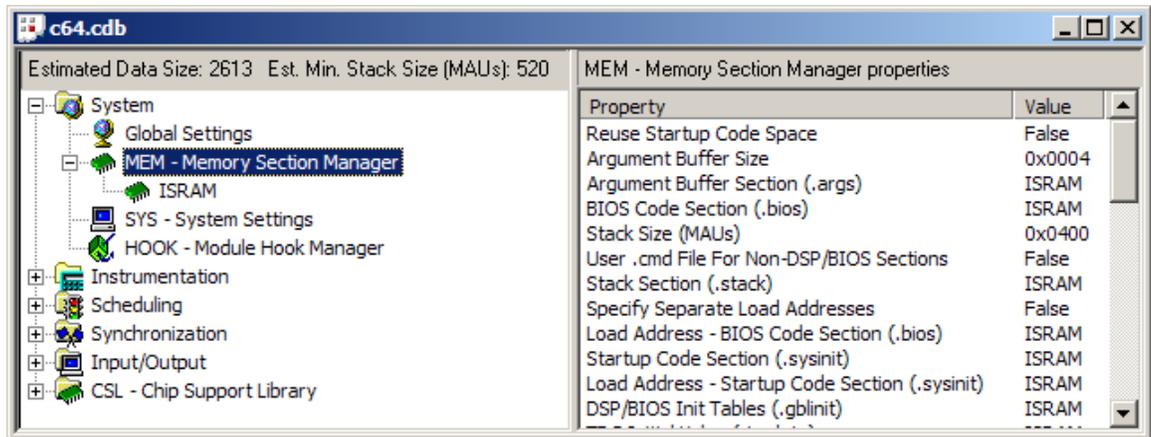
```
LAB2.PJT
MAIN.C
DATA.H
LAB.TCF
LABCFG.CMD
PROFILE.INI
```

Note: Please make sure to **copy** the above files correctly.

7. In Code Composer Studio, and open C:\op6000\labs\lab11\lab.pjt.

Modify TCF File

8. In the TCF window, expand the +**System** and +**MEM – Memory Section Manager**.



9. Before creating memory objects, **verify your answers:**

Lab 11 – C64/C67 Memory Map Solution				
Memory	CE Space	Starting/Base Address	Length (hex bytes)	Type of Space (code/data/both)
L2 Internal Mem C6416 (ISRAM)		0x0000 0000	0x0010 0000	Both
C6713 (ISRAM)			0x0004 0000	
SDRAM (64M Bytes)	$\overline{CE0}$	0x8000 0000	0x0400 0000	Both
Flash (4M Byte)	$\overline{CE1}$	0x9000 0000	0x0040 0000	Both
FIFO (32KB)	$\overline{CE2}$ or $\overline{CE3}$	0xA0000000 or 0xB000 0000	4	Data

- ◆ Why use all three different external \overline{CE} regions?
- ◆ Why is FIFO length only 4 bytes?
- ◆ What extra EMIF capability does the C6414/15/16 provide?

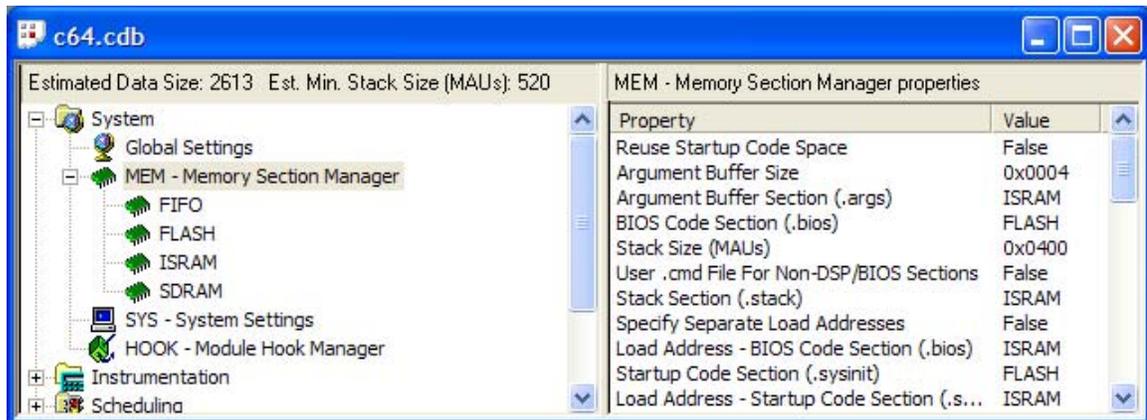
Note: If these results are different from yours, you may want to consult with the instructor before moving on.

10. **Create/Modify MEM objects** as needed using the tables from step #4.

Essentially, each row of the table should be converted into a MEM object.

Here's a few more guidelines/hints:

- Add a new MEM object by right-clicking on *MEM-Memory Section Manager* and choosing *Insert MEM*. (Hint: you need to add four MEM objects.)
- Modify a MEM object by right-clicking on the object and selecting *Properties*.
- Rename a MEM object by right-clicking on the object and selecting *Rename*.
- Disable heap for FIFO. For the others, don't worry about the heap property checkbox in your MEM objects; since we won't be using the heap in this lab.
- Notice that you can select whether *code* or *data* or *code/data* can be placed in a MEM object. This property field helps the Config Tool to perform some rudimentary validation when assigning DSP/BIOS objects to memory. In other words, it helps us to prevent mistakes.

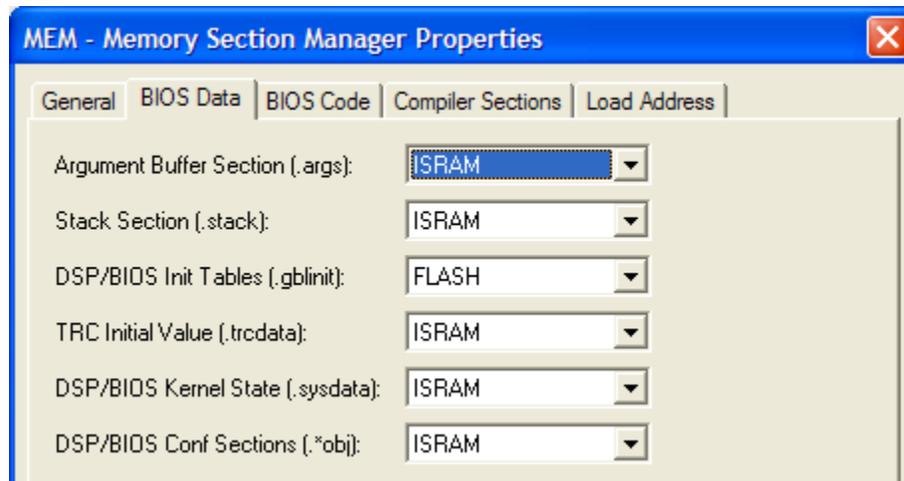


11. Place Sections into MEM objects.

- Right-click on *MEM - Memory Section Manager* and choose Properties
- Edit the dialogs as shown below

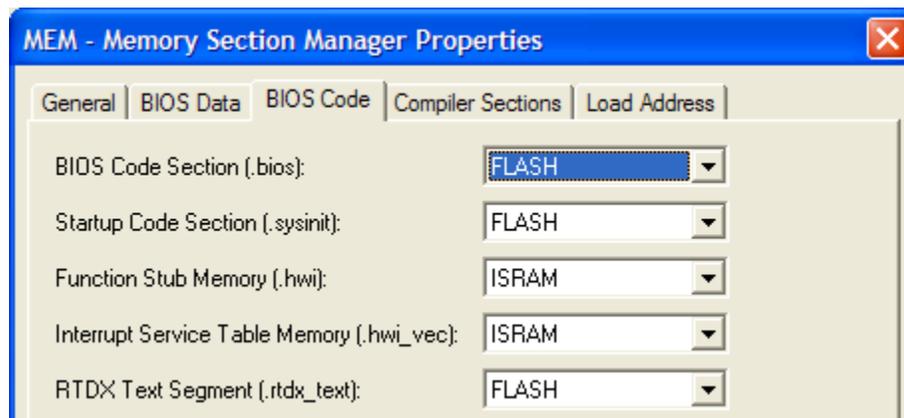
BIOS Data:

- Place the `.stack` into fast, on-chip data memory – as this is where local variables reside.
- The others are of no great concern, at this time.



BIOS Code:

- All of these should reside in *initialized* (ROM-like) memory.



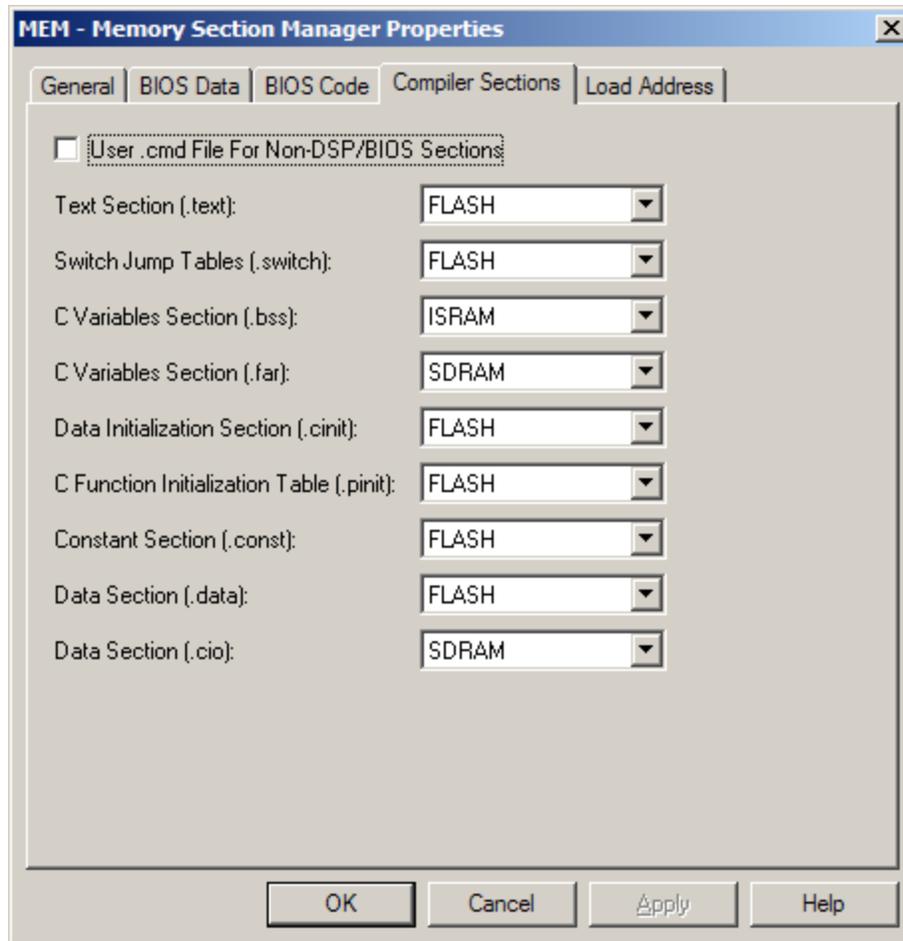
Note: If you place `.hwi_vec` into a section which does not include address 0 (e.g. if you place it into FLASH or SDRAM in our example), you should enable the *Generate RESET vector at address 0* option.

Generate RESET vector at address 0

This option is a property of *HWI - Hardware Interrupt Service Routine Manager* in the TCF file.

Compiler Sections:

- It's best if .bss can be located in fast, on-chip data memory – as this is where global and static variables reside.
- .CIO should be located in *uninitialized* (RAM-like) memory. Since it is not used for real-time data capture, it's suggested that you locate it in off-chip, bulk storage.
- .far is often located in off-chip RAM, unless there is additional space available in on-chip memory that it will fit into. This section is discussed further in the next chapter.
- The remaining sections should be placed into initialized (ROM-like) memory.

**Load Address:**

- Leave this advanced feature disabled.

12. Click OK to close the MEM properties dialog. Then save the TCF file
13. Close the TCF file.

Build / Load / Run

14. Build and load the program. (Build with the Debug configuration.)
15. Set a breakpoint at the end of main.
16. Run the program and verify the result.

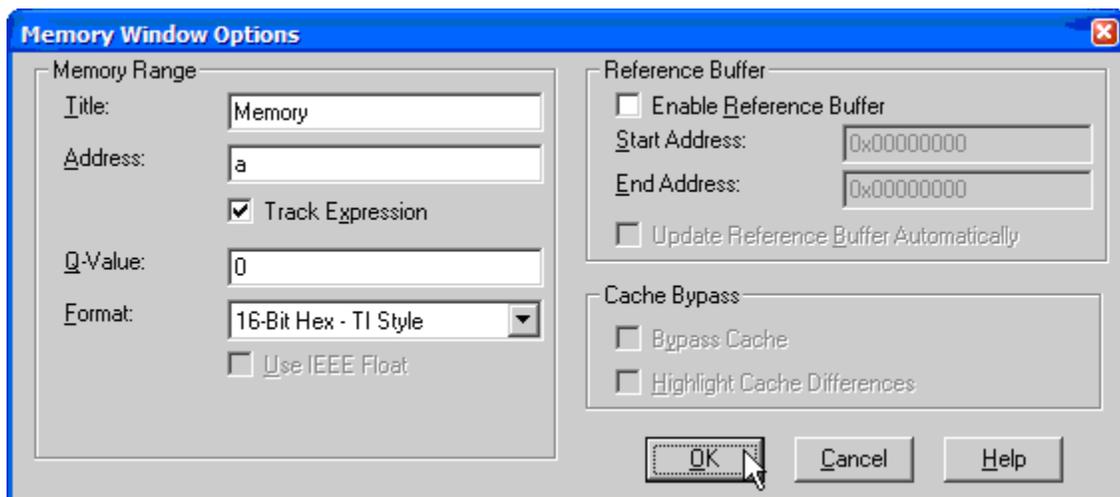
You can use the Watch window to view `y`. Alternatively, try hovering the cursor over `y` in the code window and its value should appear in a tool-tip fashion.

View Memory

Let's examine another great debugging tool, the Memory View window.

17. Start by selecting view memory and configuring it as shown.

View → Memory...



Where is location “a”? _____

Did you need to know this address before you could use the Memory window?

Thanks to symbolic debugging, all you need to know is the symbol (variable, array, etc.) that you're interested in.

18. If you increase the size of the resulting memory view, you'll notice you can see both `a` and `x`.
 - You can change the properties of a memory window by right-clicking it and choosing Properties.
 - Open multiple memory windows, giving them each a different title.
 - The ease by which you can choose the data size, makes it easy to view arrays with memory windows.
19. Close the **lab11.pjt** project.

Page left blank...

Lab 13

Lab 13 – Internal Mem. and Cache

- ◆ Lab 13a – Find Memory Bank Conflicts

Uses C6416 device simulator for this part

- ◆ Lab 13b – Compare the performance:

1. Code/Data Off-Chip with No Cache
2. Code/Data Off-Chip with Cache
3. Code/Data On-Chip

- ◆ Lab 13c – (Optional) Exploring CacheTune



Lab 13a – Removing Memory Bank Stalls

Getting Started

Setup CCS

1. Make sure CCS is configured to use the *C6416 Device Cycle Accurate Simulator, Little Endian*. If not, use CCS Setup to change the configured target to one of the above simulators. These simulators simulate L1P, L1D, L2, and various other peripherals including the EMIF.

67
64X+

Note: This lab is for C64x and C64x+ users only. The C67xx does not have the same memory architecture and its users don't need to worry about memory bank stalls in L1D. So, if you have been doing the labs for the C67xx, you can skip this lab and move on to Lab13b. If you have been doing the C67xx labs and you would still like to do this lab, ask your instructor where you can get the C64x or C64x+ files that you will need.

Creating a Project

2. Copy the following files from **C:\op6000\labs\lab7** to **C:\op6000\labs\lab13a**.

```
LAB7.PJT  
MAIN.C  
DOTP.SA  
LAB.TCF  
LABCFG.CMD  
PROFILER.INI  
DATA.H
```

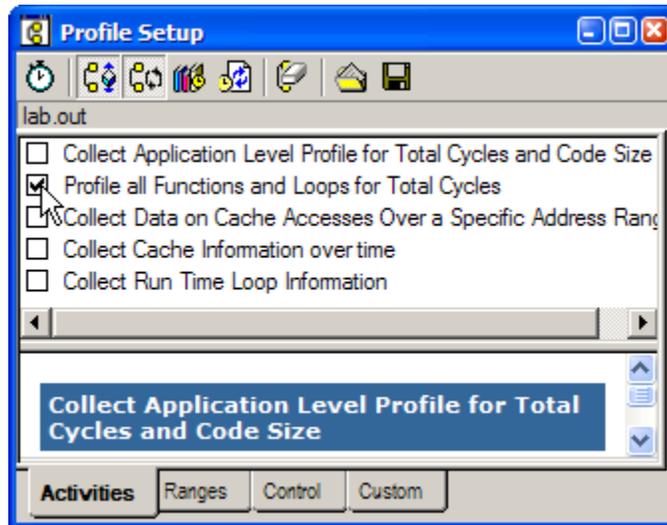
3. Use Windows Explorer to rename the **C:\op6000\labs\lab13\lab7.pjt** file to **lab13a.pjt** by right clicking on it and choosing rename.
4. Open **lab13a.pjt** inside CCS.

Build, Run, and Profile

5. Build your code using the *Debug* configuration.
6. Check the value of *y* to make sure that it is correct.
7. Rebuild your project using the *Release* configuration.
8. Re-enable the breakpoint at the end of *main()*.
9. Open the Profile Setup window.

Profile → Setup

10. Enable the *Profile all Functions and Loops for Total Cycles* option.



11. Click on the **Enable/Disable Profiling** icon  to **enable** profiling.
12. Open the Profile Viewer.

Profile → Viewer

13. Run your code.
14. How many cycles does the *dotp()* function take? _____

Finding Memory Bank Conflicts

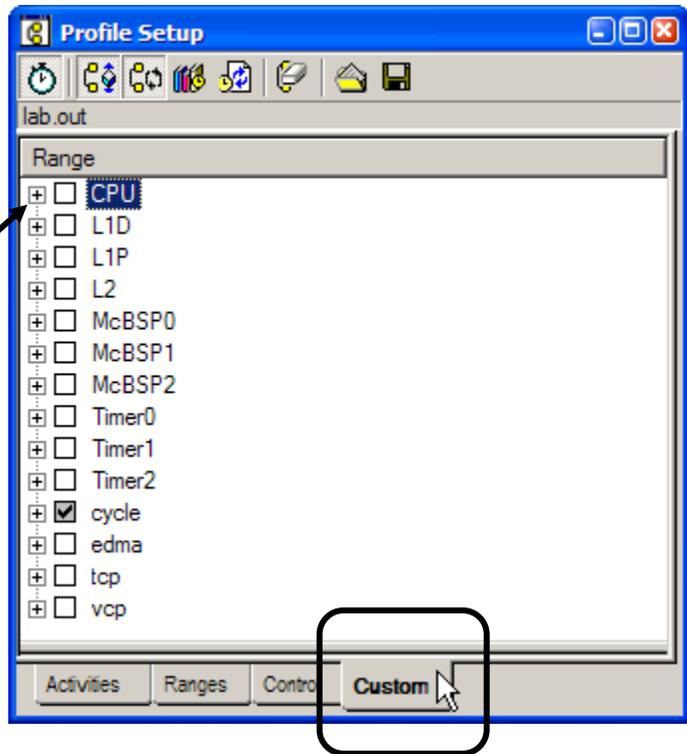
Are any of these cycles being wasted on Memory Bank Conflicts? Would you like to find out?

15. In order to change the Profile Setup to detect Memory Bank Stalls, we need to disable profiling.

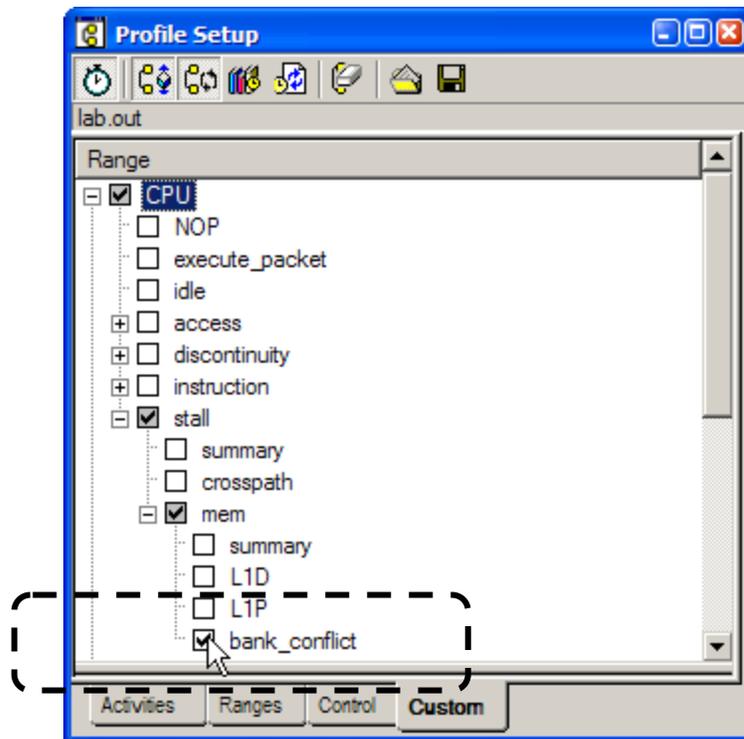
Disable profiling by clicking on the Enable/Disable Profiling icon .

16. Click on the custom tab in the Profile Setup window.

17. Click on the  next to **CPU** to expand it. Use the same technique to expand the **stall** category, and then the **mem** category.



18. Click in the box next to **bank_conflict** to enable it. This will tell the profiler to count all Memory Bank Conflicts.





19. Now that we've changed the Profile Setup, we can re-enable profiling with
20. Reset the CPU.

Debug → Reset CPU

21. Reload your program which was built with the *Release* configuration.

File → Reload Program

22. Run your code.
23. Take a look at the Profile Viewer. You should see two new columns that count the number of inclusive and exclusive Memory Bank Conflicts.

How many Memory Bank Conflicts are there in the *dotp()* function? _____

Address Range	Symbol Name	Symbo...	Acce...	cycle.Total: Incl...	cycle.Total: Excl...	CPU.stall.m...	CPU.stall.m...
0:0x1c0-0x1e0	c64cfg.s62:764:7...	function	0	0	0	0	0
0:0x1e0-0x200	c64cfg.s62:781:7...	function	0	0	0	0	0
0:0xce00-0xd008	c64cfg.s62:834:8...	function	1	6283	2617	0	0
0:0xd008-0xd068	c64cfg.s62:872:9...	function	0	0	0	0	0
0:0xa700-0xa86c	dotp	function	1	359	359	104	104
0:0xaf40-0xaf74	main	function	1	0	0	0	0
0:0xa720-0xa74c	dotp	loop	0	0	0	0	0
0:0xa7e0-0xa800	dotp	loop	121	0	0	0	0

Using Pragmas

We'll now use a pragma to get rid of these wasted cycles.

24. Open *main.c* for editing.
25. Find the global declarations for the *a* and *x* arrays.
26. Add a `DATA_MEM_BANK` pragma to make sure that *a* and *x* do not have any memory bank conflicts. If you need some help, try looking up the pragma in the help file.

Hint: You should make sure that the *a* and *x* arrays go into *different* memory banks.

Build, Run, and Profile

27. Reset the CPU.

Debug → **Reset CPU**

28. **Build** your code using the *Release* configuration.

29. **Load** your code if it doesn't load automatically.

30. **Run** your code.

31. **Profile** your project with the `DATA_MEM_BANK` pragma. Use the results to fill in the table below:

Dot-product Version	Cycles
Lab13a – Step 23 (Release options)	
Lab13a – Step 31 (Release options w/pragma)	

Did the `DATA_MEM_BANK` pragma have any effect on your code?

32. Save your project and close it.

End of Lab13a – Go on to Lab13b on page 13 - 7

Lab 13b – Using Cache Memory

This lab allows you to re-examine the modem code used in the CodeSize Tune lab, this time comparing the code execution time while varying where the code and data reside:

- Off-Chip Memory (with no caching)
- Off-Chip Memory (with L1 and L2 cache)
- On-Chip Memory (with L1 cache)

Note: This lab requires the use of a Device Simulator and since the C672x simulator assumes flat memory it cannot be used with this lab. If you have been doing C672x labs you should reconfigure CCS for the C6713 Device Cycle Accurate Simulator.

Part 1 – Code/Data off-chip with no cache

Setup CCS

1. Make sure CCS is configured to use the *C6416 Device Cycle Accurate Simulator, Little Endian*, or the *C6713 Device Cycle Accurate Simulator, Little Endian*. If not, use CCS Setup to change the configured target to the appropriate simulator. These simulators simulate L1P, L1D, L2, and various other peripherals including the EMIF.

Preparing for the Lab

2. We are going to use the memory configuration that you set up in Lab 11. Copy the following files from **C:\op6000\labs\lab11** to **C:\op6000\labs\lab13b**.

LAB.TCF

LABCFG.CMD

Note: If you used the C672x simulator for Lab 11 you should copy the two files above from the appropriate Lab 11 solutions directory for the simulator you are currently using. If you need help ask your instructor.

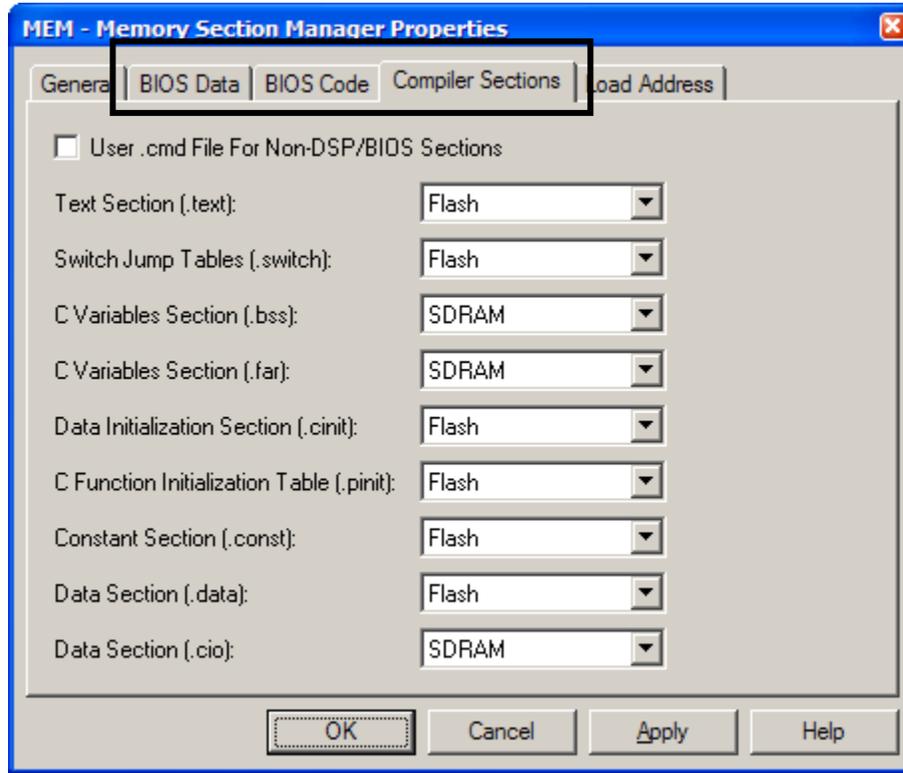
3. Open the project named **LAB13b_64.PJT** or **LAB13b_67.PJT** in the **C:\op6000\labs\lab13b** folder.

For your convenience, we have already created the project for you. After creating the project, we added our three source files, and told the linker to create a map file for us (-m option).

4. Open *LAB.TCF*.

5. Make sure all sections will be placed into off-chip memory except *.hwi* and *.hwi_vecs*.

Right-click on *MEM-Memory Section Manager*. Choose *Properties* and verify that every uninitialized section is placed into SDRAM. Leave the sections that are placed in FLASH in FLASH. Under the BIOS Code Tab, leave the *.hwi* and *.hwi_vecs* sections in ISRAM since these contain the reset vector. Here is how we placed our C compiler sections:



Note: Make sure to make the necessary change to the memory configuration under all three tabs: **BIOS Data, BIOS Code, and Compiler Sections**. Make sure to leave *.hwi* and *.hwi_vecs* in ISRAM.

6. Change the Build Configuration to *Release*.
7. Build and load your program.
8. Open the appropriate *lab.map* in either the **C:\op6000\labs\lab13b\Release_64** or **C:\op6000\labs\lab13b\Release_67** folder and verify all the code and data except *.hwi* and *.hwi_vecs* are in off-chip memory.

Counting Cycles

9. Set a breakpoint to stop execution.

Place the breakpoint at the second bracket from the **end** of the MODEMTX.C file.

```

/* For benchmarking purposes we will comment out the while loop and run
   a single execution of the modem data. */
// while(item)
// {

    /* get modem data, convert to constellation points, and add noise. */
    ReadNextData();

    /* run modem on new data */
    for( i = 0; i < BAUD_PER_LOOP; i++ )
    {
        ModemTransmitter(i, &(g_ModemData.OutputBuffer[i*SAMPLES_PER_BAUD]));
    }
    item += 1;
// }

```



Note: When a breakpoint is set on a C621x/C671x/C64x device, the corresponding cache line is invalidated. This means that cache overhead will occur every time through the loop. It is important to keep this in mind when profiling short sections of code with cache.

10. Run to main if you are not already there.

Debug → Go Main

11. Enable and display the clock.

Profile → Clock → View

Hint: To clear the *Clock* value, just double-click on the *Clock* window.

12. Run your program.

Your program should run from *main* and stop at the *breakpoint* (if you have been following the instructions). If you had not yet run to *main*, your cycle count will be larger since the processor will have counted all of the cycles while your system was initialized (.sysinit).

Hint: If you need to start running the code from the beginning, you should first perform a *GEL → Resets → Reset_and_EMIF_Setup*, then a *Debug → Restart*.

- Record the clock value in the first column of table at the end of this lab (page 13-15).
- Perform a GEL Reset and initialize the EMIF for the next part of the lab.

GEL → Resets → RESET_and_EMIF_Setup

64

or

GEL → Resets → ClearBreakPts_Reset_EMIFset

67

Note: This Reset is necessary to flush the cache and make sure that the EMIF is properly setup from one part of the lab to the next.

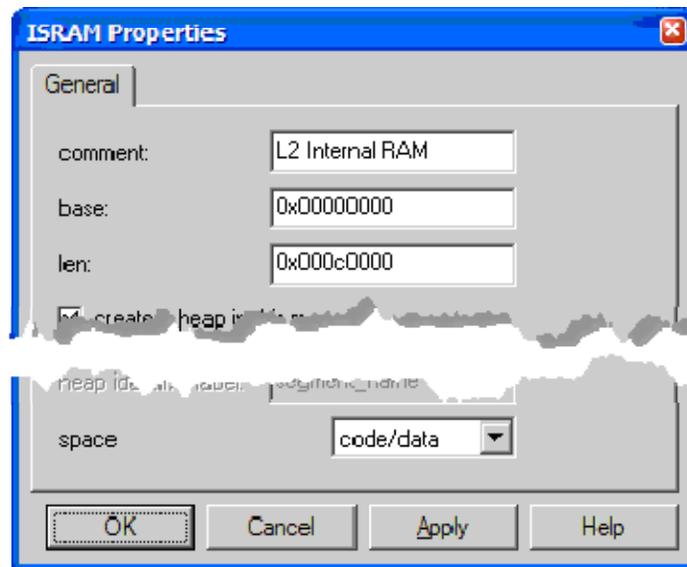
Part 2 – Code/Data (still) off-chip with Cache Enabled

Part 1 of this lab did not use cache at all. In fact, L2 cache is not only disabled, but the MARS bits for all of the external memory are disabled. In Part 2, we will turn on some L2 cache as well as enable caching of external memory.

- Open lab.tcf if it is not already open.

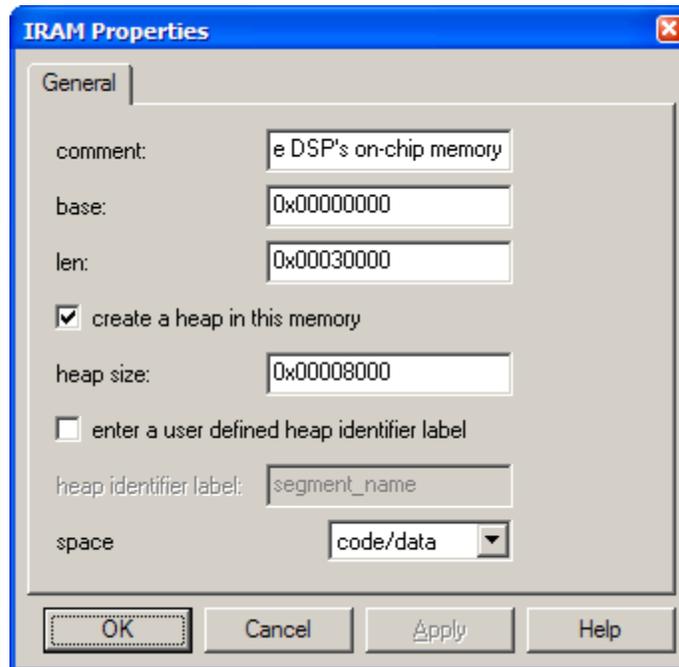
- Before we can turn on the L2 Cache, we need to make room for it by decreasing the size of the internal L2 SRAM. Change the properties of ISRAM so that it starts at 0x0 and is 768KB long. Here is what it should look like:

64



67

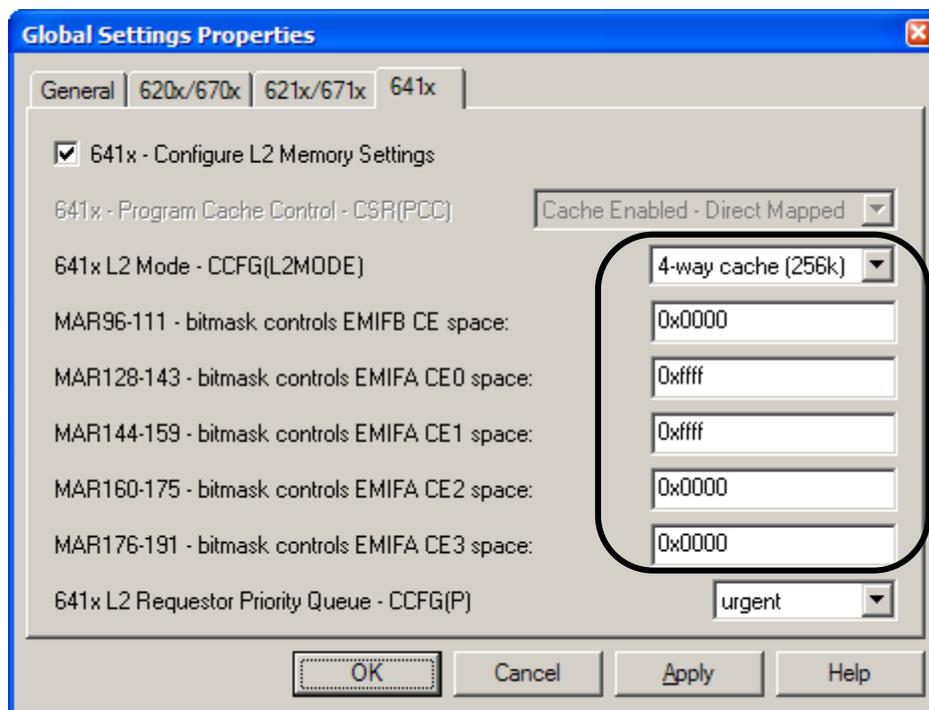
17. Before we can turn on the L2 Cache, we need to make room for it by decreasing the size of the internal L2 SRAM. Change the properties of IRAM so that it starts at 0x0 and is 192KB long. Here is what it should look like:



64

18. Enable L2 Cache. Make all external memory cacheable.

Open TCF file. Expand *System*. Right-click on *Global Settings*. Choose *Properties* and select the *641x* tab.



The L2 configuration is probably obvious. Rather than being set as addressable SRAM memory (4-way cache (0k)), it's now set to act as a 4-way cache using $\frac{1}{4}$ of the internal memory of a 'C6416.

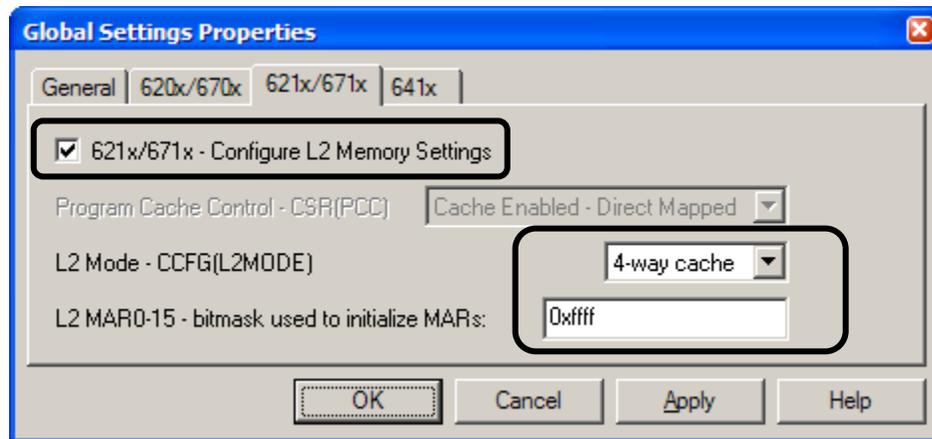
By changing the MAR lsb's to "1" (which is what happens when you put 0xFFFF into the MAR fields) you have told the cache and EMIF hardware that the external memory is allowed to be cached.

- Hint:** The **MARX-X – bitmask used to initialize MARs** TCF field name is misleading.
- These bit values are “enable” values (as are MAR lsb's), rather than “mask” bits.
 - Both L1 and L2 are affected by the MAR registers. When a region of memory is marked as uncacheable, it won't be found in either cache. (Note, though, that MARs do not affect L1P on the C64x+ devices.)
 - While the field name states “0-15”, I suspect that a value of “0x0001” would actually set the LSB of MAR0 (as opposed to MAR15).

19. **Enable L2 Cache. Make all external memory cacheable.**

67

Open TCF file. Expand *System*. Right-click on *Global Settings*. Choose *Properties* and select the *621x/671x* tab.



The L2 configuration is probably obvious. Rather than being set as addressable SRAM memory (SRAM), it's now set to act as a 4-way cache using $\frac{1}{4}$ of the internal memory of a 'C6713.

By changing the MAR lsb's to "1" (which is what happens when you put 0xFFFF into the MAR0-15 field) you have told the cache and EMIF hardware that the external memory is allowed to be cached.

Hint: The **MARX-X – bitmask used to initialize MARs** TCF field name is misleading.

- These bit values are "enable" values (as are MAR lsb's), rather than "mask" bits.
- Both L1 and L2 are affected by the MAR registers. When a region of memory is marked as uncacheable, it won't be found in either cache.
- While the field name states "0-15", I suspect that a value of "0x0001" would actually set the LSB of MAR0 (as opposed to MAR15).

Counting "Part 2" Cycles

20. **Rebuild and load your program.**
21. **Go to main if you are not taken there automatically.**
22. **Set a breakpoint in the MODEMTX.C file, as before. (see page 13-9, step 9) if it is not already there.**
23. **Clear the clock by double-clicking on it.**
24. **Run your program.** It should stop at the breakpoint.
25. **Record the clock value** in the *second* column of table at the end of this lab (page 13-15).

26. Perform a GEL Reset and initialize the EMIF for the next part of the lab.

GEL → Resets → RESET_and_EMIF_Setup

64

or

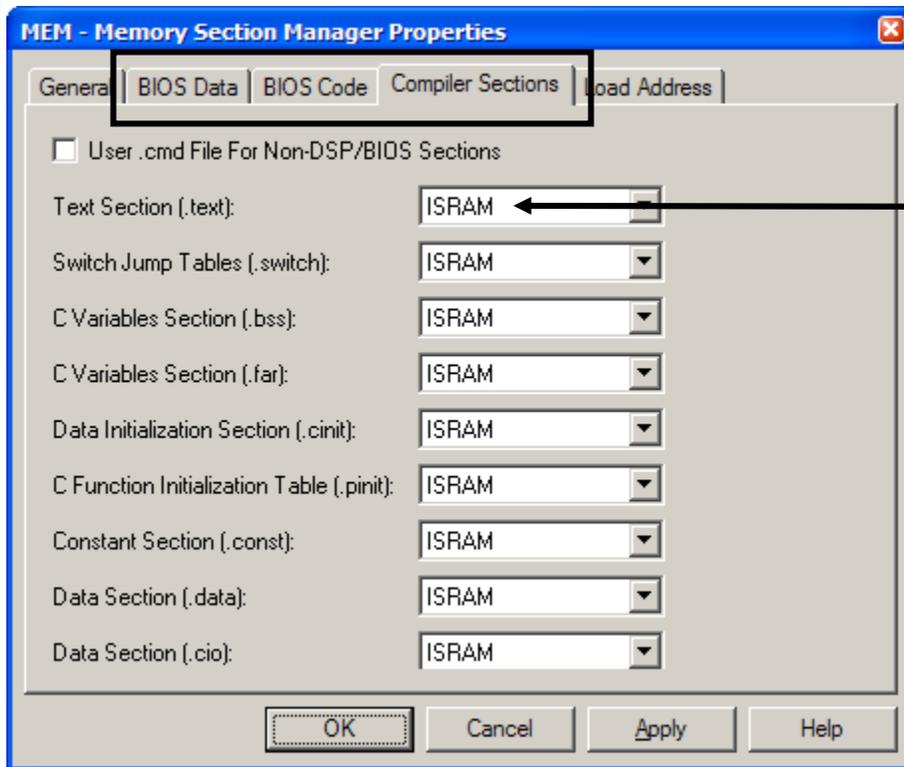
GEL → Resets → ClearBreakPts_Reset_EMIFset

67

Part 3 – Code/Data located on-chip with L1 cache

27. Open your lab.tcf file.

28. Open the Properties for the MEM – Memory Manager and place all of your software sections into the internal ISRAM segment. Make sure to change all three of the section tabs, BIOS Code, BIOS Data, and Compiler Sections.



67
IRAM

29. Close the Memory Manager Properties dialog and then your TCF file.

Counting “Part 3” Cycles

30. **Rebuild and load your program.**
31. **Go to main if you are not already there.**
32. **Clear the clock.**
33. **Run your program.** It should stop at the breakpoint.
34. **Record the clock value** in the *third* column of Lab Results table below.

Lab Results

Code/Data Linked External L2 = SRAM (no L2 cache) MAR's = 0000 (L1 effectively disabled)	Code/Data Linked External L2 = 4-Way (L2 cache enabled) MAR's = 0xFFFF (L1/L2 eff. enabled)	Code/Data Linked Internal (ISRAM) L2 = ¾ SRAM / ¼ CACHE MAR's = 0xFFFF (L1 effectively enabled)

35. Which memory configuration is fastest? How did cache affect the results?

36. Was there a big advantage to linking your code and data into on-chip memory?

37. When might it make good sense (and cents) to link data on-chip?

End of Lab13b – Go on to Lab13c on page 13 - 16

Page left intentionally blank.

Lab 13c - CacheTune Introduction (Optional)

In Lab13a and Lab13b, we saw how dramatic an impact the memory and cache setup of an application can have on its performance. Once you have the cache setup the way that you want to, how do you further optimize the application so that it uses the cache intelligently? It sure would be nice to have a tool that would show us how our application is using the program and data caches and figure out where we should spend time trying to optimize.

The CacheTune tool provides graphical visualization of memory reference patterns for program execution over a set amount of time. All the memory accesses are color-coded by type. Various filters, panning, and zoom features facilitate quick drill-down to view specific areas. This visual/temporal view of cache accesses enables quick identification of problem areas, such as areas related to conflict, capacity, or compulsory misses. All of these features help the user to greatly improve the cache efficiency of the overall application.

Learning Objectives:

- Prepare an application to collect cache data
- Assess the cache overhead for the application
- Visualize the memory accesses in cache
- View the symbolic information
- Use the zooming features to view cache information
- Use Interference Grid, Interference Shadow, and Find Conflicting Symbols tools to identify object conflicts
- Eliminate the conflict misses in the program cache
- Measure the improvement

Application Objective

This tutorial introduces the basic features of CacheTune and takes you through a step-by-step process for analyzing and optimizing an application's cache performance using CacheTune. The demonstration example performs the following operation on matrices A and B and stores the result in C.

```
C = A <matrix operation> transpose (B)
```

It highlights the capacity misses in the data cache and conflict misses in the program cache. It also recommends cache optimization techniques to eliminate these misses.

Part 1 – Getting to Know the Application

Setup CCS

1. Make sure CCS is configured to use the *C6416 Device Cycle Accurate Simulator, Little Endian* or the *C6713 Device Cycle Accurate Simulator, Little Endian*. If not, use CCS Setup to change the configured target to the appropriate simulator. This simulator simulates L1P, L1D, L2, and various other peripherals including the EMIF.

Opening and Reviewing the Project

You begin this lesson by opening a project and examining the source code files used in the project. This lesson uses the `mat_oprn` demonstration application.

2. Start Code Composer Studio if it is not already running.
3. Open the `mat_oprn_1_64.pjt` or `mat_oprn_1_67.pjt` located in the `lab13c` folder.
4. Review the source code for this project.
 - Double-click on the `mat_xpose_oprn.c` program to open it. Examine the source code for this program. Function `mat_xpose_oprn` can be found in this file. It calls function `mat_oprn` to perform the operations on matrices A and B. Function `mat_oprn` operates on the *i*, *j*-th entry of matrix A and *j*, *i*-th entry of matrix B. The result is stored into the *i*, *j*-th entry of matrix C. Notice that matrices A and C are accessed in row major order whereas matrix B is accessed in a column major order. `CODE_SECTION` pragma is used to place the function in its own section.
 - Double-click on the `mat_oprn.c` program to open it. Examine the source code for this program. Function `mat_oprn` can be found in this file. It calls `calculate` function and performs several operations on the elements in the matrices and returns the results. `CODE_SECTION` pragma is used to place the function in its own section.
 - Double-click on the `mat_xpose_oprn_1.cmd` program to open it. Examine the source code for this program. Notice that, by a relative placement in the linker command file, the two functions: `mat_oprn` and `mat_xpose_oprn` are placed into the memory map such that they map to the same cache line in the level-one program cache.

The example is demonstrated within TMS320C64x™ (C64™) or the TMS320C67x™ (C67™) environment. The C64x™ DSP family has a dedicated level-one program cache (L1P) and data cache (L1D) of 16 Kbytes each. L1P is a direct-mapped cache with a 32 byte line size. L1D is a 2-way set associative cache. Each cache way is 8K bytes and the cache line is 64 bytes.

The C67x™ DSP family has a dedicated level-one program cache (L1P) and data cache (L1D) of 4 Kbytes each. L1P is a direct-mapped cache with a 64 byte line size. L1D is a 2-way set associative cache. Each cache way is 2K bytes and the cache line is 32 bytes.

This information is necessary to understand the cache behavior with this particular example.

Assessing the Cache Overhead

In this step, you will profile your application to assess the cache related overhead, using the profiling features of Code Composer Studio. The cache overhead can be assessed by measuring the cache stall cycles caused predominately by cache misses and L1D write buffer full occurrences. For more detailed discussion on cache overhead, please refer to the related CacheTune online help topic.

5. From the **Project** menu, click on **Rebuild All**. An output file (.out) must be generated and loaded before beginning the profiling process.
6. From the **File** menu, click on **Load Program**, and choose mat_oprn_1.out from the appropriate Release subfolder (Release_64 or Release_67). Click **Open**.
7. From the **Profile** menu, launch **Setup**. The Profile Setup opens on the right side of the screen and allows the setting of several options to customize application profiling. The Profile Setup window displays the activities tab by default, to allow users to select the profiling collection activities.

8. **Enable Profiling by clicking on the Profiling icon**  .

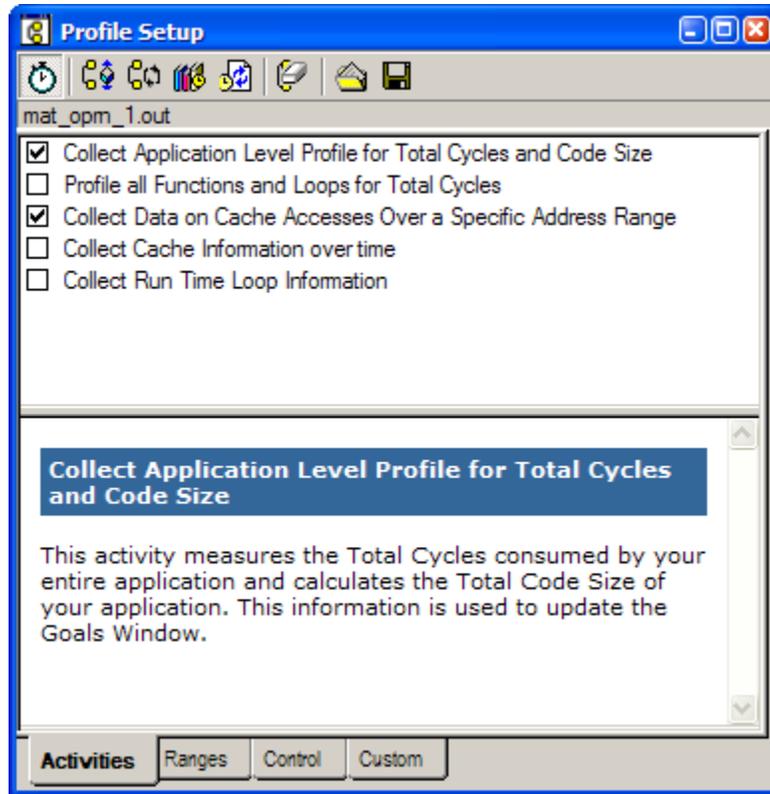
Profiling may not be enabled if the program has started running or if it was explicitly disabled. Data will not be collected if profiling is not enabled.

9. Click on the box beside **Collect Application Level Profile for Total Cycles and Code Size**.

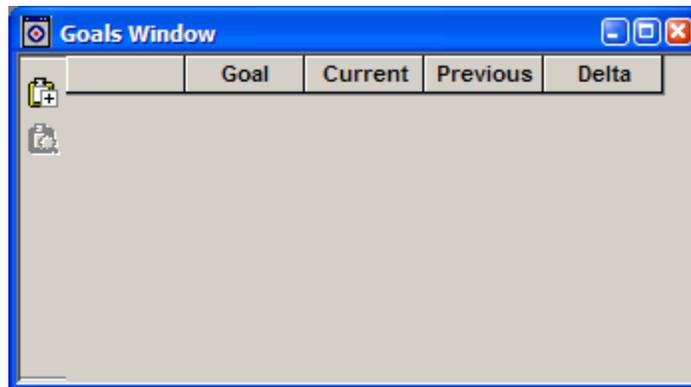
This activity measures the total cycles consumed by the entire application and calculates the total code size of the application. This information can be viewed later from Goals Window.

- Click on the box beside **Collect Data on Cache Accesses Over a Specific Address Range**.

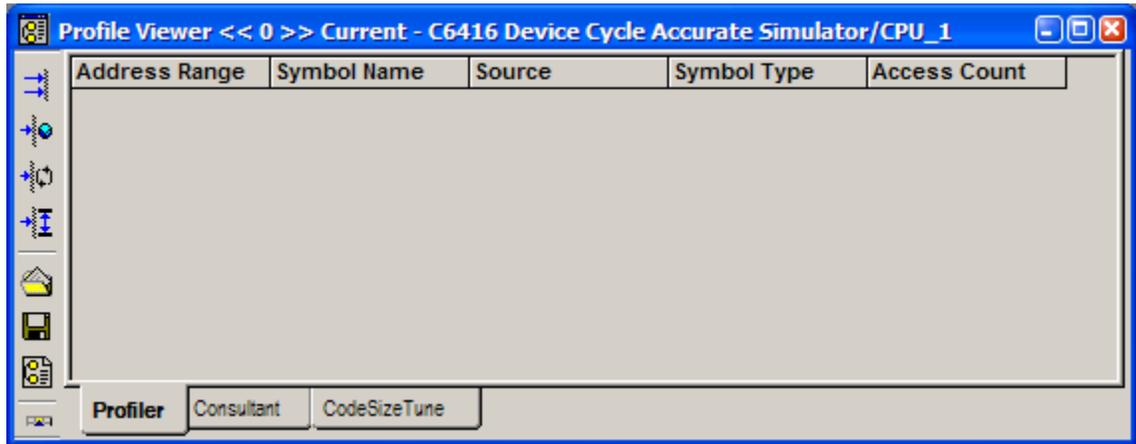
This activity tracks Program and Data Cache events, such as cache stall cycles, and the number of cache misses, during a profile run over functions and ranges of your choosing. You should now have something like this as your Profile Setup:



- Select the **Profile**→**Tuning**→**Goals** menu item to open **Goals Window**. Please see online help for more information about the Goals Window.



12. Select **Profile**→**Viewer** menu item to open **Profile Data Viewer**. Please see the online help for more information on the Profile Viewer.



13. Select the **Debug**→**Run** menu item or press **F5** to run the application.

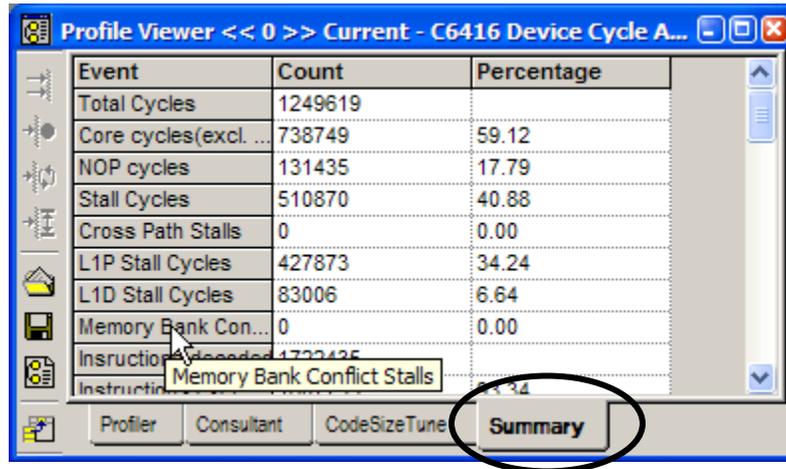
The program self-terminates, so it does not need an exit point to stop profile data collection. Please see the online help topics on Profile Setup for more information about exit points. Once the program terminates, you may view the cache accesses in the Profile Viewer.

14. Once the program has executed, initial data values will appear in the **Current** column of the **Goals** window. These values represent the Code Size and Cycle Total. For this tutorial, we will mainly focus on the cycle total and use the Goals Window to track tuning progress for improvements in the cycle total. The total cycle count is 1,249,619 for the C64 and 1,171,570 for the C67. Your numbers may vary.

	Goal	Current	Previous	Delta
Code Size		1504	-	-
Cycle Total		1249619	-	-

15. After the program finishes execution, a new Summary tab will be added in the Profile Viewer. The Summary tab displays the total counts and percentages of most available simulation events. Please refer to the Dashboard **Profile Viewer** online help for a detailed description of the displayed events for different targets.

In the Profile Viewer window, click on the **Summary** tab and look at the number for the following events. These are sample numbers, your numbers may vary.



Note: You may roll your mouse over the event cell to display the complete name.

Part 2 – Using CacheTune for Data

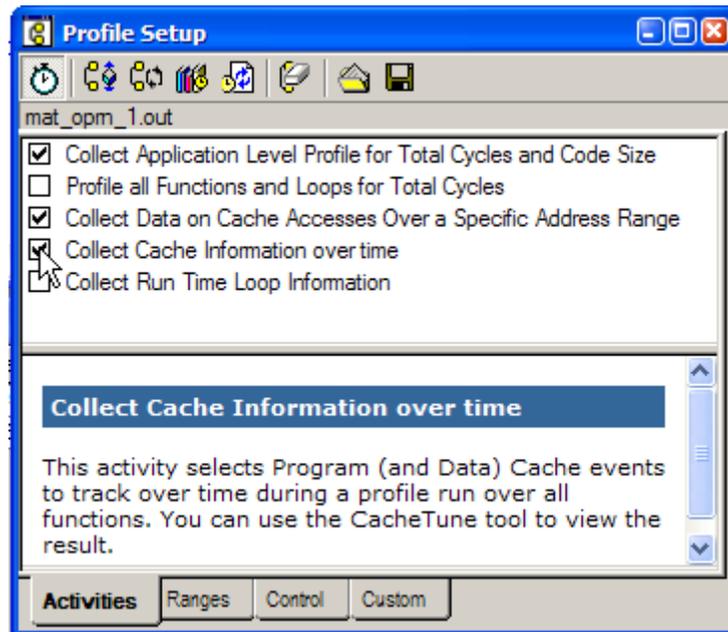
To collect data on cache accesses, use Profile Setup to select the appropriate activity by using the following steps:

16. From the **Profile** menu, launch **Tuning**→**CacheTune**. This selection launches the CacheTune tool in the main editor window.

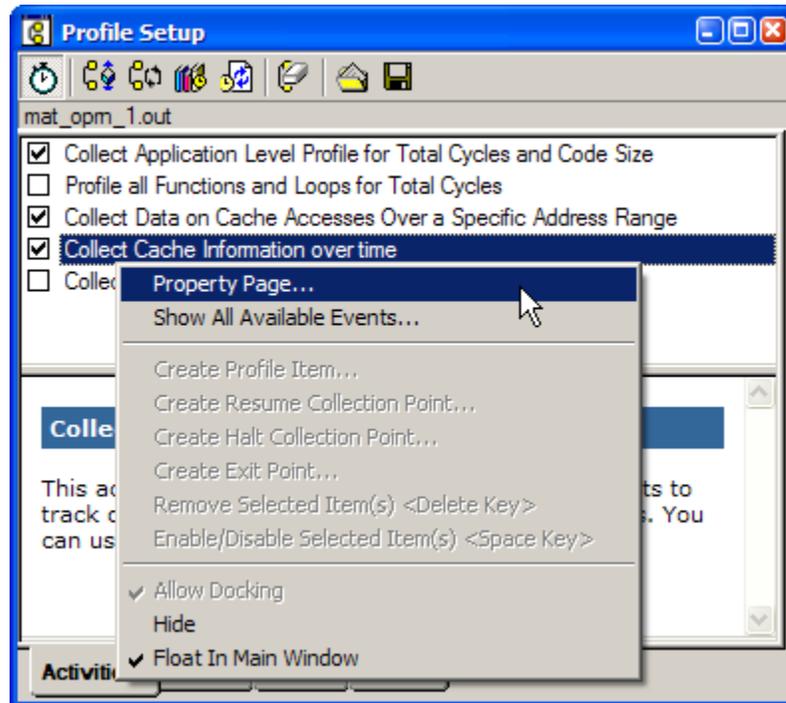
It also launches the advice window and activates the CacheTune tab. The page describes the tool, and provides suggestions for getting the most out of the tool. At this point, the program has not collected any cache data over time, so most of buttons on the toolbar are deactivated.

Note: You are directed to collect cache data by selecting the appropriate cache-related activity in Profile Setup.

17. From the **Profile Setup** window, select the activity **Collect Cache Information over time**. This activity selects cache events to track over time during a profile run over all memory addresses.

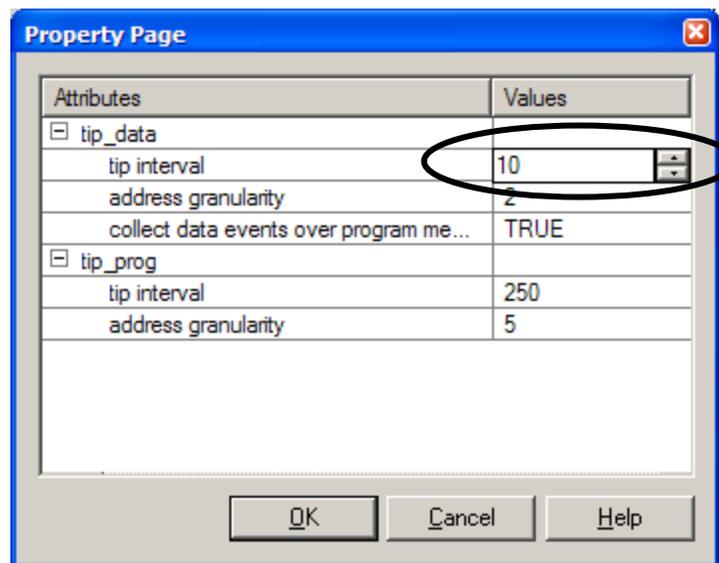


18. Right-click on the selected activity, and select the **Property Page** from the context menu.



This opens the Property Page, where you can modify the attributes of trace data, such as time interval profile (tip) interval and address granularity. The tip interval is the size in cycles of the time period over which the simulator summarizes the memory references before writing out a trace record. The address granularity for tip_data is 2 bytes, meaning that any access to a byte is visualized as if it were to a half word.

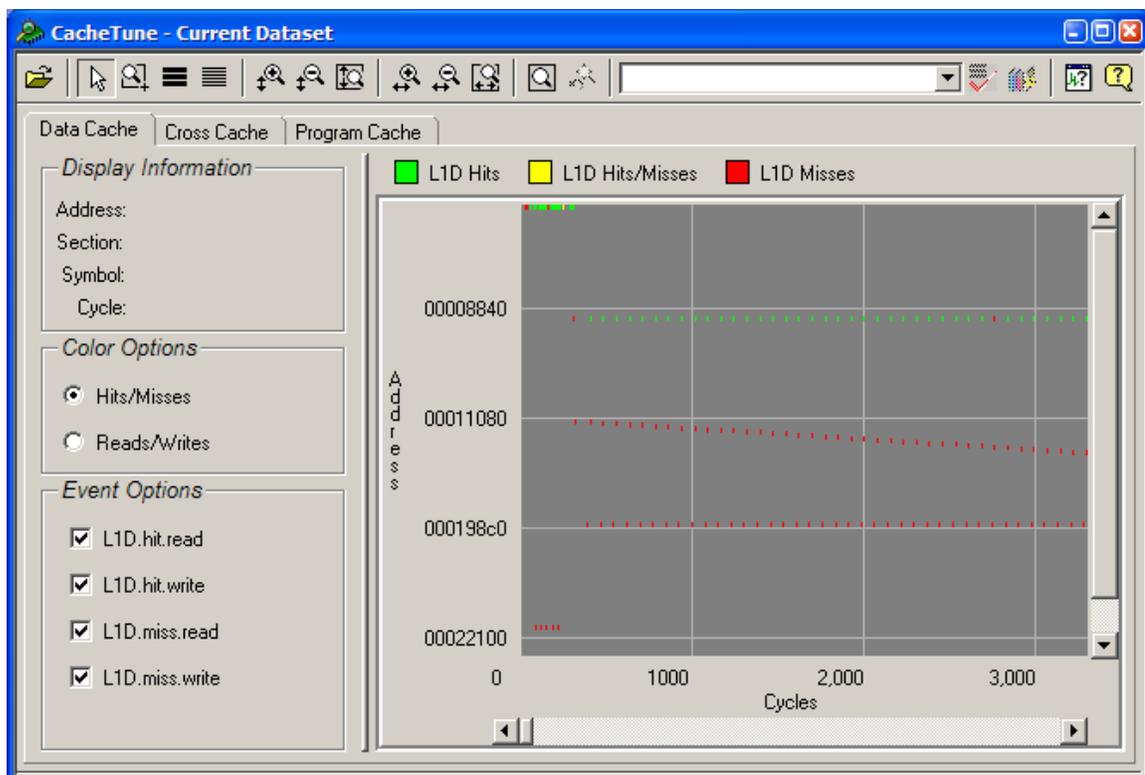
19. The default tip interval is 250 cycles for the program and data caches. Left-click in the first tip interval cell that is for tip_data to activate that cell. Once activated, change the cell value to 10, so you will have a larger resolution and, therefore, view more details on cache access for the data cache. Please refer to the related topic of CacheTune online help for more details.



20. Click **OK** to save changes and close the **Property Page**.
21. Before we run the program to collect data on cache accesses, we need to perform a CPU reset to clear the cache, as suggested by the General Advice Window for our second run of the program. From the **Debug** menu, choose **Reset CPU**.
22. Select **File**→**Reload Program** to reload the program.
23. Open the CacheTune window.

Profile → **Tuning** → **CacheTune**

24. Now we are ready to collect data on cache accesses. From the **Debug** menu, choose **Run**, or press the **F5** key. The program will run to completion. Once the program terminates, you may view the cache accesses from CacheTune.



Preparing to View Data Cache Accesses

In this part of the lab, you will use CacheTune to visualize and analyze the memory access patterns in the data cache and view the available symbolic information. In addition, you will learn to navigate within the tool and view a particular area using the zooming features. We will also discuss the optimization techniques used to eliminate the cache misses in the data cache.

After the program terminates, navigate to the CacheTune output window in your workspace to view memory accesses. By default, CacheTune displays the data cache tab. Click on the data cache tab title to display data cache advice in the advice window. If you have multiple windows open in the main CCS workspace, they may cover the CacheTune output graph window after profiling is completed. In this case, close some windows to view the output graph window.

Note: When tuning, always start with data cache, because optimizing for data cache performance may lead you to modify your program, thus changing the memory access pattern of the program cache.

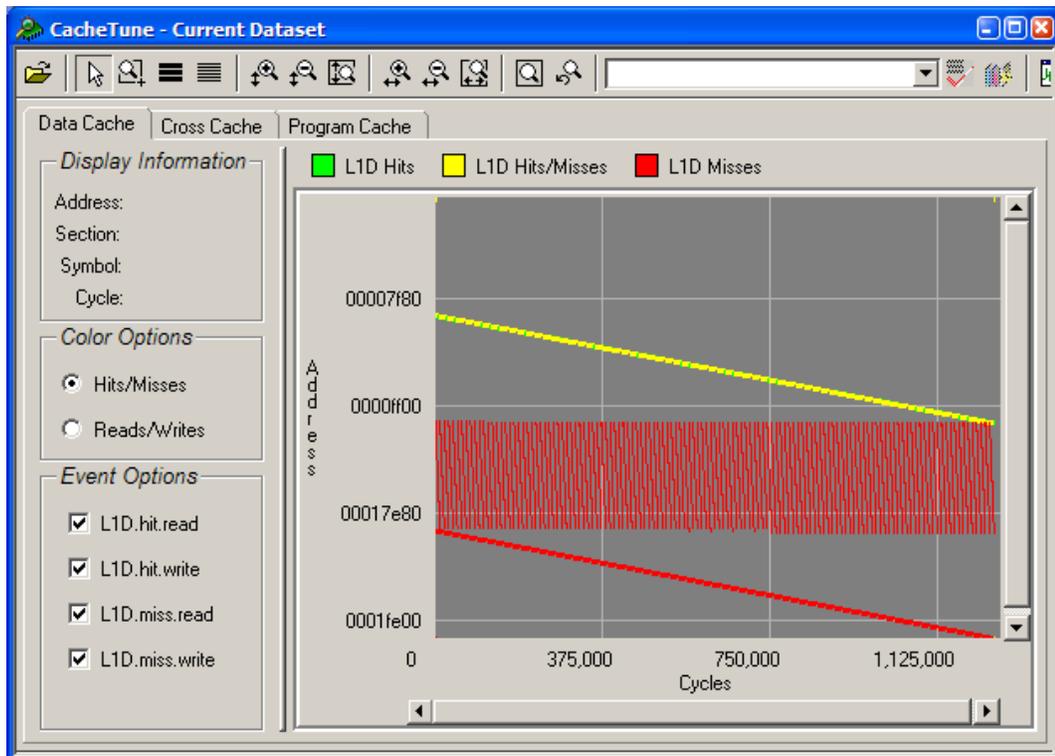
Viewing the Data Cache Accesses

CacheTune graphically shows memory access patterns over time, color-coding the accesses to distinguish cache hits from cache misses. Cache hits are shown as green pixels while cache misses are shown in red. The accessed addresses are plotted along the Y-axis, while access times are plotted along the X-axis.

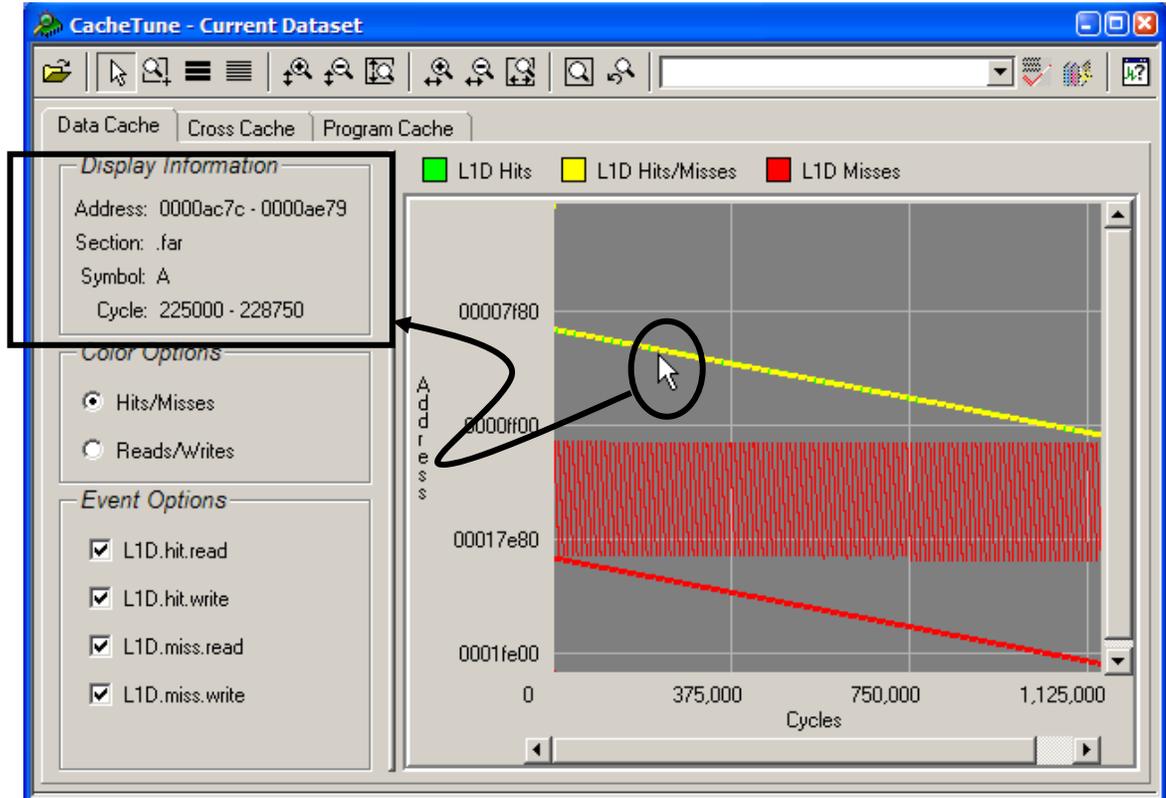
25. Adjust the CacheTune window to a suitable size. This view only shows a small part of the cache trace data (indicated by the horizontal and vertical scroll bar).

26. Click on the **Full Zoom**  button to view the entire trace.

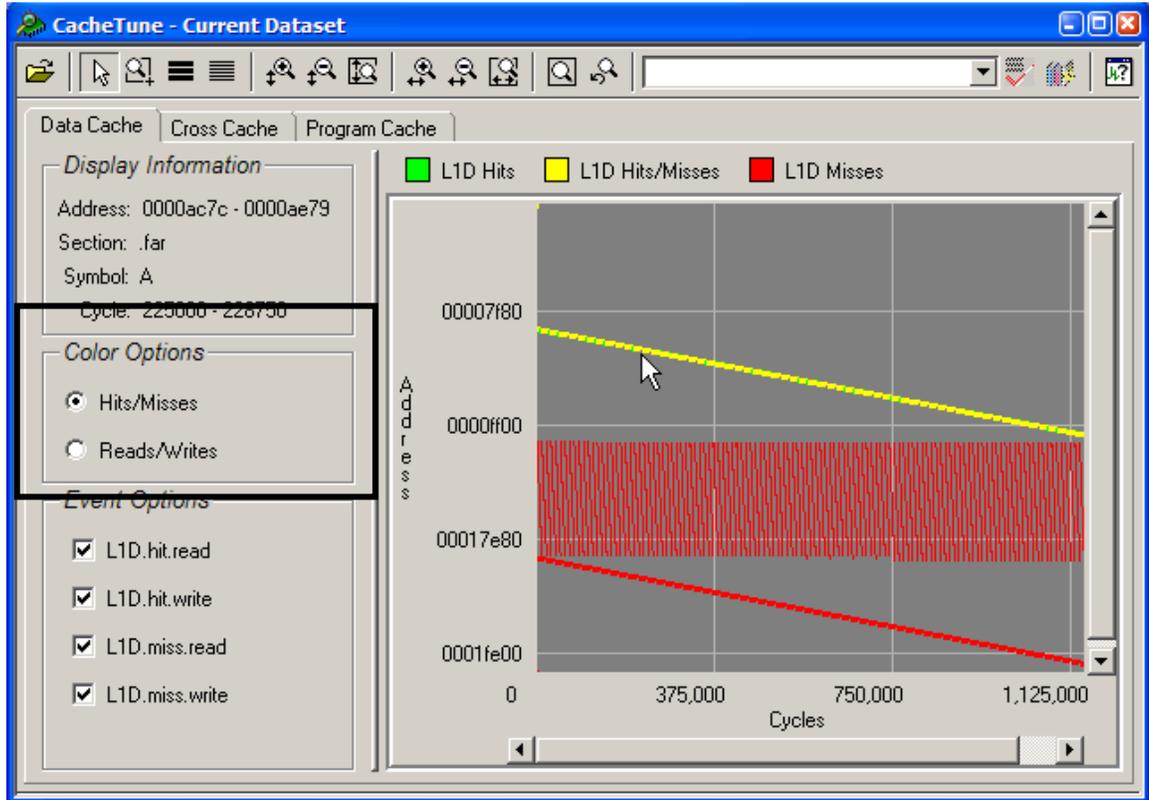
This provides an overview of memory access patterns and points out the hot spots where most cache misses occur. The graph should resemble the image below. There is a red bar in the middle of two inclined lines. The line on the top is partially yellow and green; the bottom is red. This indicates that both cache misses and hits occur when accessing the memory range associated with the upper line; the accesses to the middle and lower part of the memory addresses incur intensive cache misses.



27. Move the cursor across the display. The information display area (left side of the CacheTune window) updates with the current address range, section, symbol (if any), and cycle range corresponding to the pixel currently under the cursor. This provides the information of where (address) and when (cycle) cache events occur on which data (section and symbol) memory access. As observed from the display, cache events occur on memory accesses to matrices A, B and C through almost the entire program. The memory ranges are 0x9000 to 0x11000, 0x11000 to 0x19000, and 0x19000 to 0x21000 respectively. The address range matches the size of each matrix, which is $128 \times 128 \times 2 = 32768 = 0x8000$ bytes)

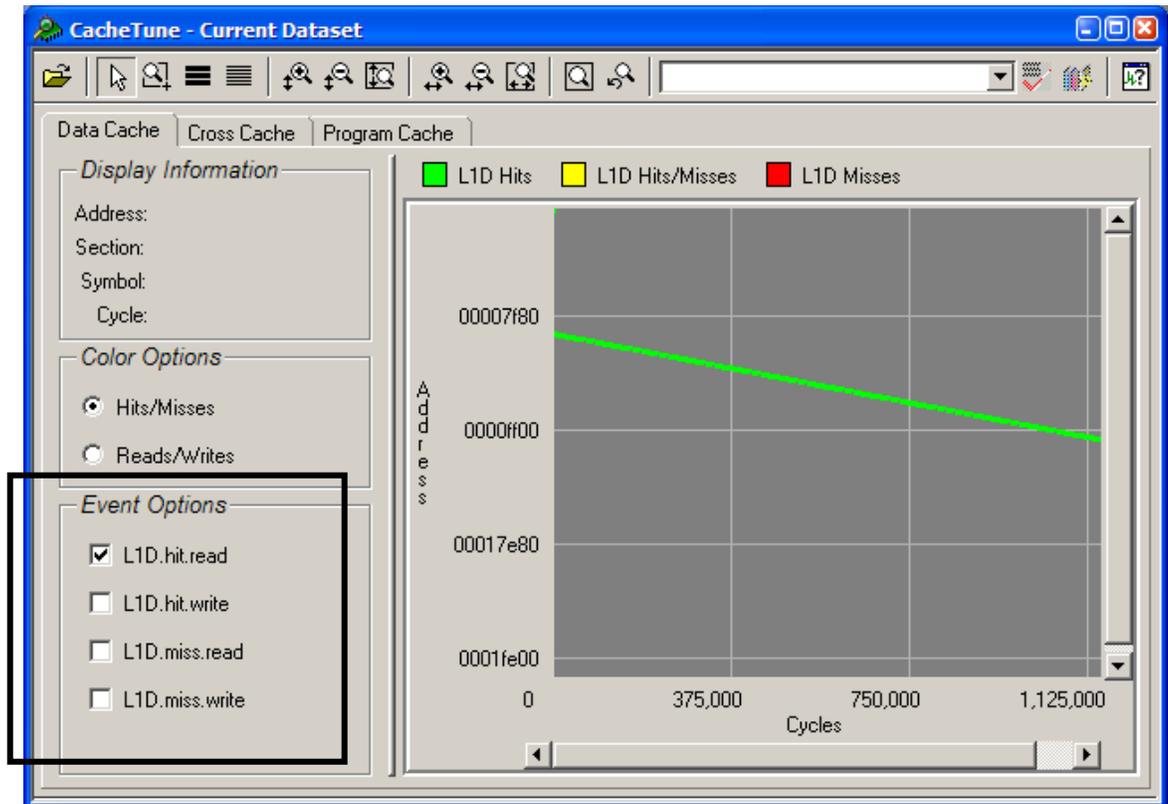


28. From the **Color Options** area on the left of the display, switch between the two different color schemes: **Hits/Misses** and **Reads/Writes**. One color scheme codes the memory references according to whether they were cache hits or cache misses, the other color scheme codes the references according to whether they were reads or writes. The area above the graph provides a legend for the colors used in the graph for various events. For example, with **Reads/Writes** color scheme, the cache reads color code in green pixels and cache writes in red. When a pixel is yellow, it indicates both a cache read and write occur. Notice that all accesses to A and B are cache reads and accesses to C are cache writes.



29. Reset the color scheme to **Hits/Misses**.
30. With the **Hits/Misses** color scheme on screen, try toggling some of the cache events on and off from the **Event Options** area. For instance, un-check other cache events and select **L1d.miss.read event** to see only the read misses events in the data cache.

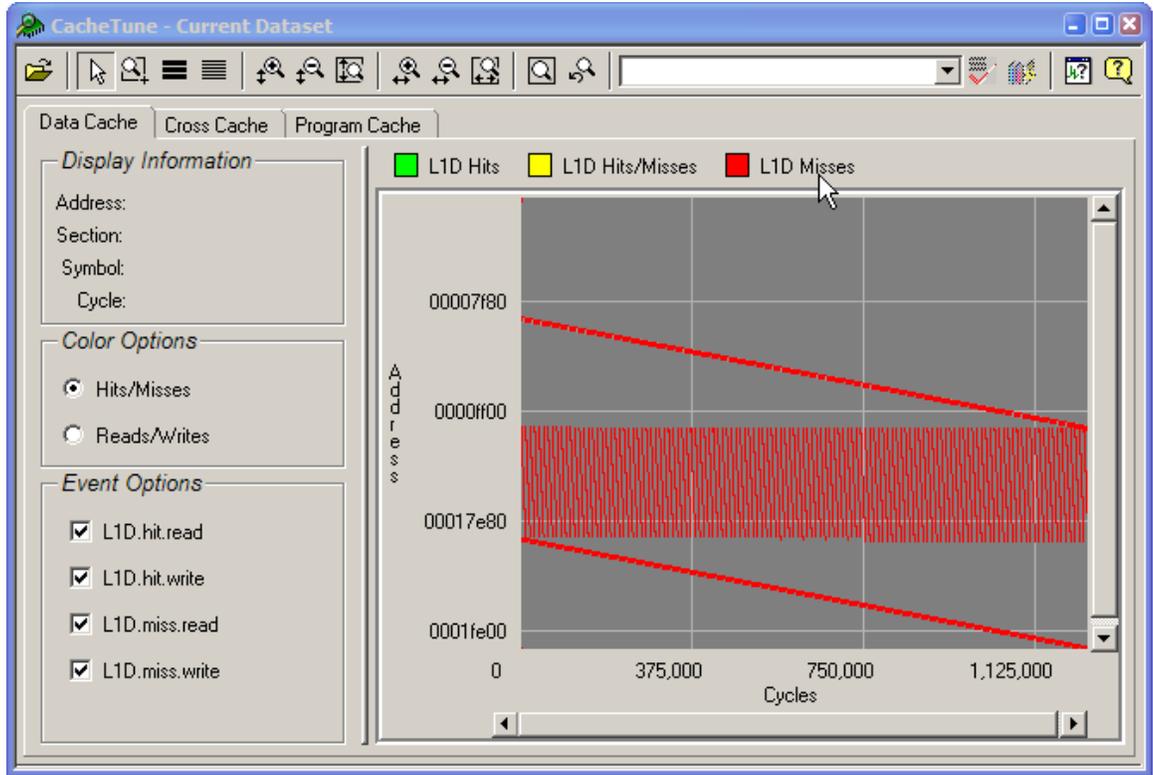
- Un-check the other cache events and select **L1D.hit.read** to see only the cache read hits events. Notice that there are no cache hits on accesses to matrices B and C, that is, all accesses to matrices B and C miss the cache.



With the help of features available from the tool, we quickly identify some basic access patterns: The upper part of the trace corresponds to the reading (load) of the input matrices A and B. Then the results are written (store) into matrix C. There are both cache misses and hits when accessing matrix A, whereas all the accesses to matrices B and C cause cache misses.

- Re-check all the **Event Options** boxes.

33. You can also view L1D Hits, L1D Hits/Misses, or L1D Misses by hovering over the color coded boxes at the top of the CacheTune Window, just below the tabs. For example, if you hover over the red box, you will see only the L1D Misses. It should look something like this, which shows all of the red information:

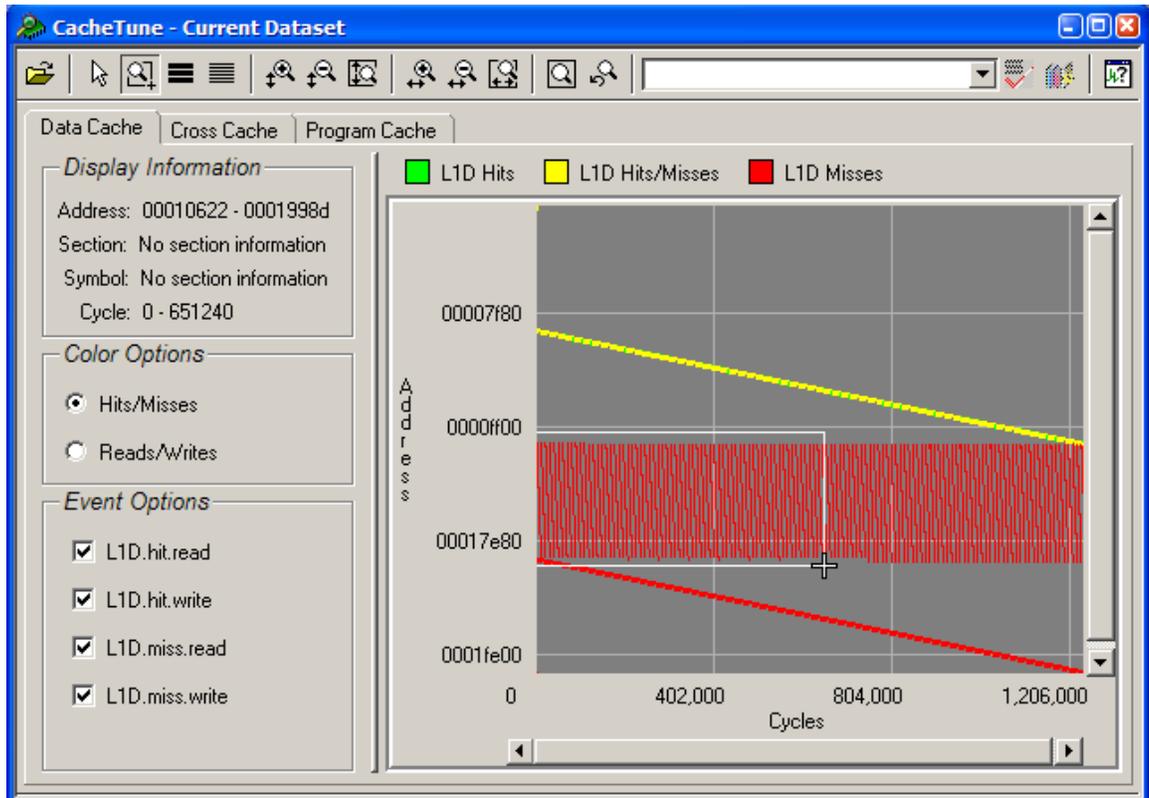


Since the hot-spot associated with most cache misses is accessing matrix B, the next step will take a closer look at it to analyze the code behavior and identify the causes of misses.

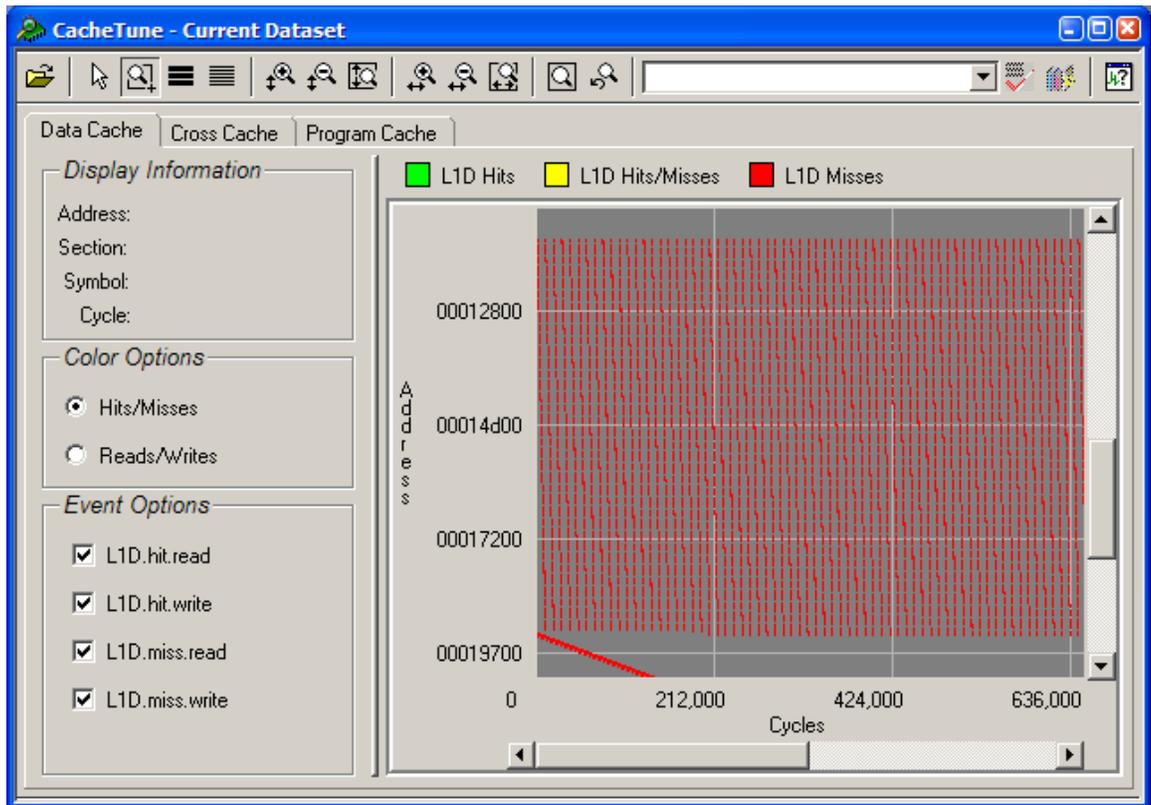
Zoom Features

CacheTune provides several zoom features to facilitate navigation, and allow the user to identify the cache miss problem areas.

34. Select the Zoom Area  button. The cursor will change from pointer mode to zoom mode. CacheTune has four different cursor modes: pointer, zoom, grid, and shadow. For more information, please see the online help topic on CacheTune.
35. Left-click to drag a rectangle over a partial area with respect to the accesses to matrix B, shown outlined in white below.



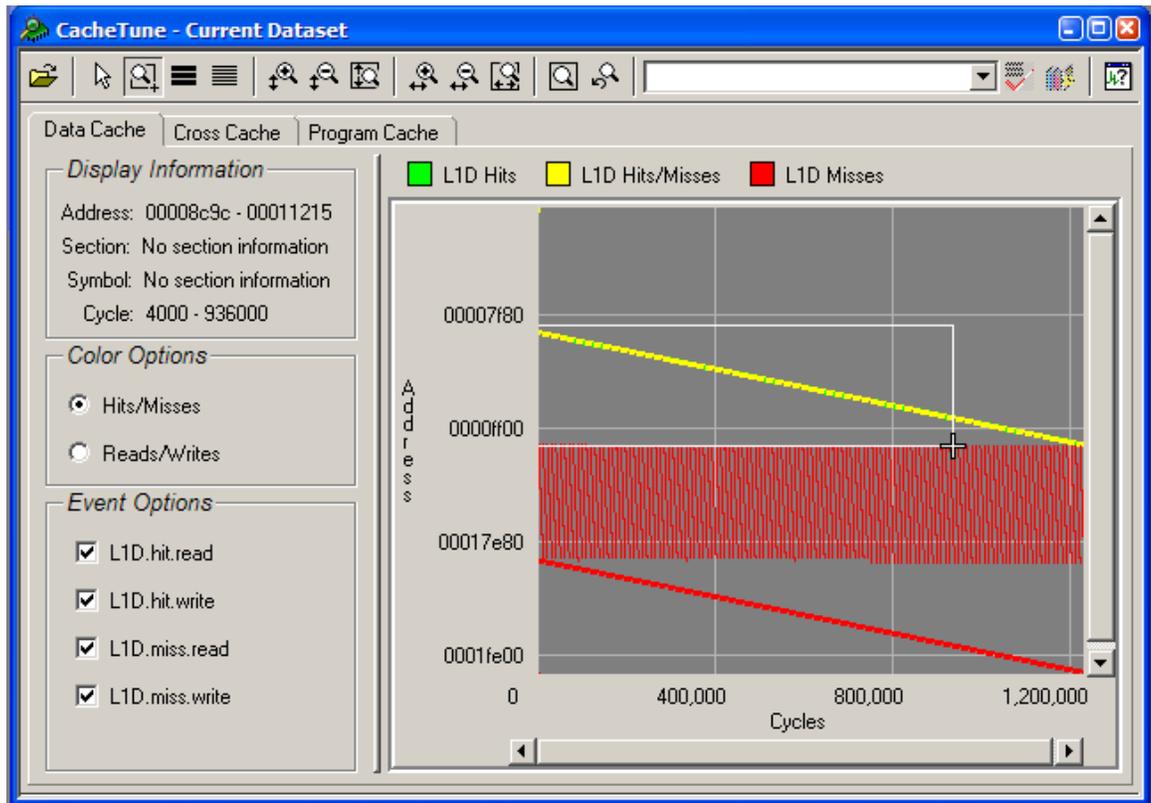
36. The resulting display should resemble the following image, showing a much more detailed picture of the memory access pattern.



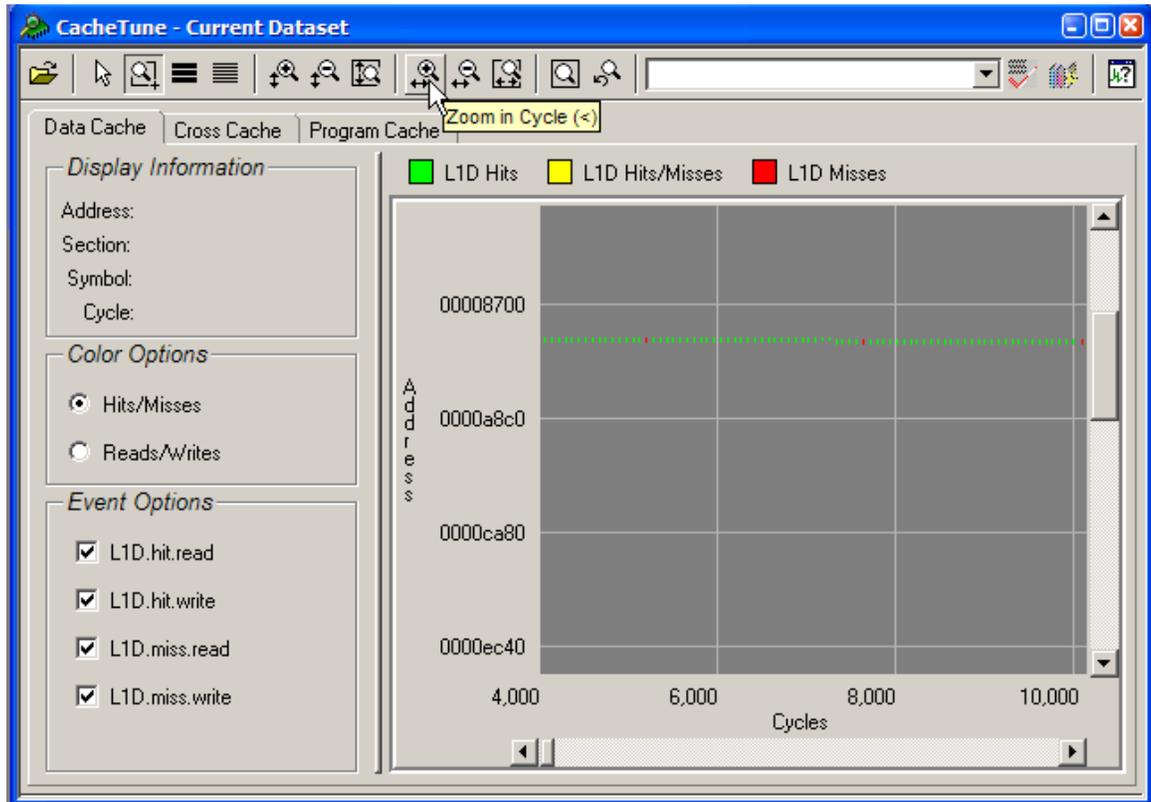
Notice the red bar is composed of multiple red vertical lines because matrix B is accessed by column, that is, the n th element of every row is accessed consecutively. The accesses to each column correspond to each vertical red line in the memory range of matrix B on the display. Every two consecutive accesses within a single column are with a stride of $0x100$ bytes in memory range. Because the stride is larger than the cache line size of L1D (64 bytes for C64 and 32 bytes for C67), the accesses cannot take advantage of spatial locality, thus causing cache misses. Furthermore, as the size of matrix B is at least twice as large as the L1 data cache size, the rows of the matrix B that are already in the cache will be replaced by the rows that are brought into the cache later. This causes cache misses when previous allocated rows are accessed again. By going through the entire dimension of the large matrix, the cache cannot hold the data that are referenced within particular time period and hence cause capacity misses.

37. Click on the Undo-Zoom  button to reset the display.

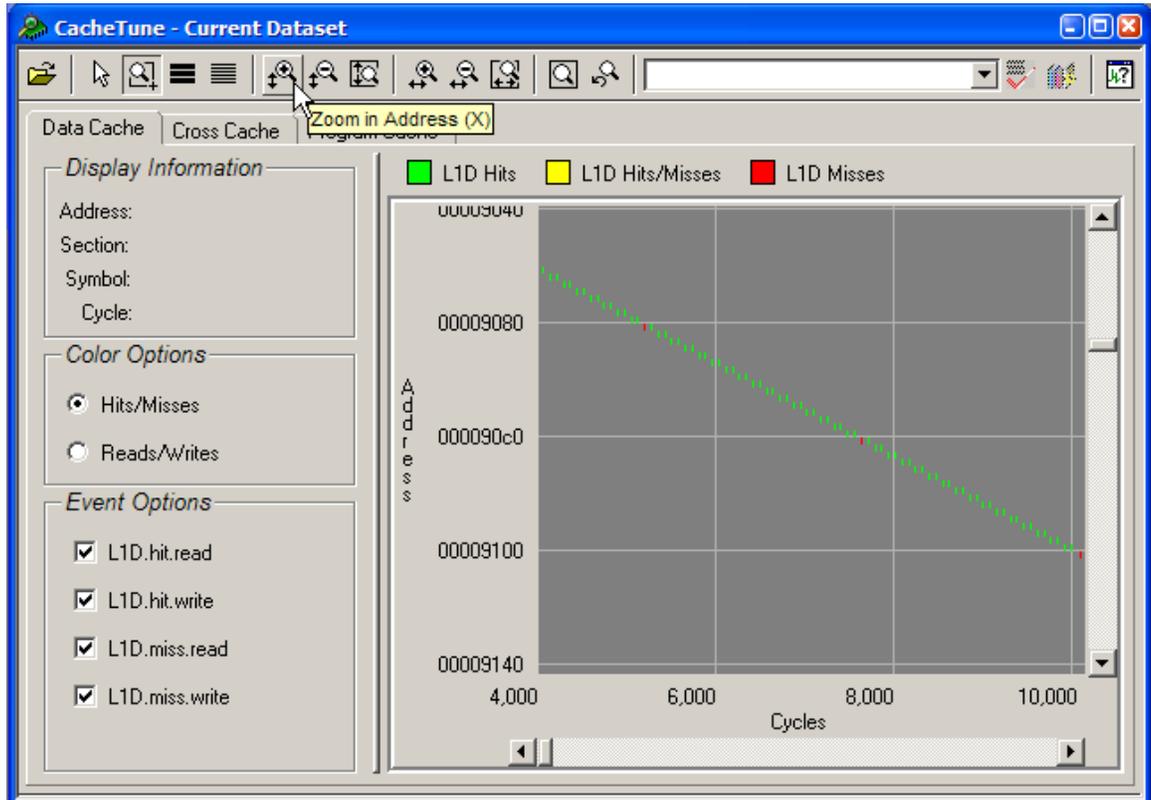
38. Left-click to drag a rectangle over a partial area with respect to the accesses to matrix A as shown outlined in white below.



39. Click on the Zoom-in Cycles  button to zoom to the cycles (X-axis) only. Repeat this until the graph resembles the image below.



40. Click on the Zoom-in  zoom to the addresses (Y-axis) only. Repeat this process until the graph resembles the image below.



The program makes a large series of sequential accesses to matrix A. The access pattern of matrix A is demonstrated in the above image. When a cache miss occurs, 32 matrix elements, which are 64 bytes of data (size of cache line in L1D), will be brought into the cache. The C67 brings in 16 elements at a time. Since elements in each row of matrix are accessed consecutively, there are 31 (or 15 for the C67) consecutive cache hits after one cache miss as can be observed. The cache misses here are mostly compulsory misses.

The memory access pattern of matrix C is almost identical with matrix A, except matrix C is accessed by store rather than load operations and all misses are store misses. As we know from the profiling results from lesson 1, there are no write buffer full related stalls for this application. This means the write misses when accessing matrix C do not stall the CPU.

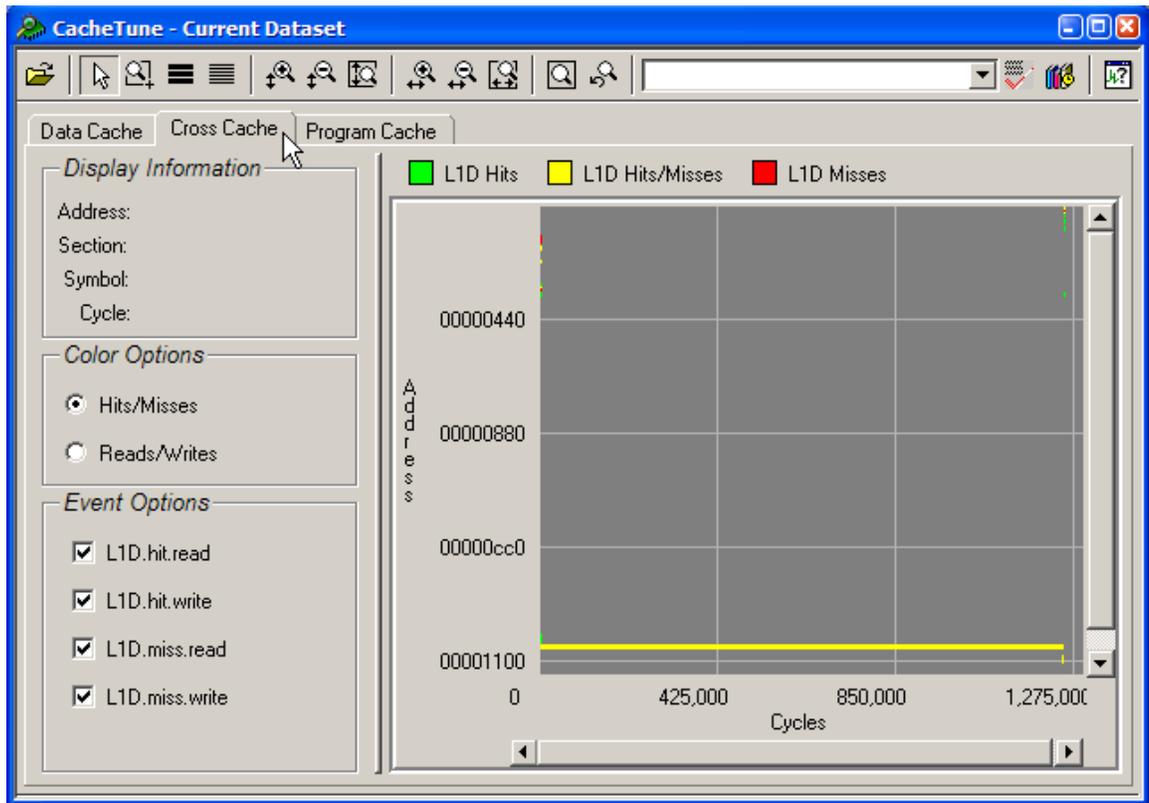
Now we know that the capacity misses occurred when accessing matrix B during load operations. This matches the third misses' scenario discussed from CacheTune Data Cache Advice Window. As suggested, these misses can be eliminated by dividing the data into several sets and processing one at a time. However, we have no knowledge of which part of the code references the matrix. For our example, which is a small and simple application, the particular matrix is only referenced within one function. But for some applications, particular data could be accessed by multiple functions. The information contained within the Cross Cache will help us at this point.

Part 3 - Cross Cache

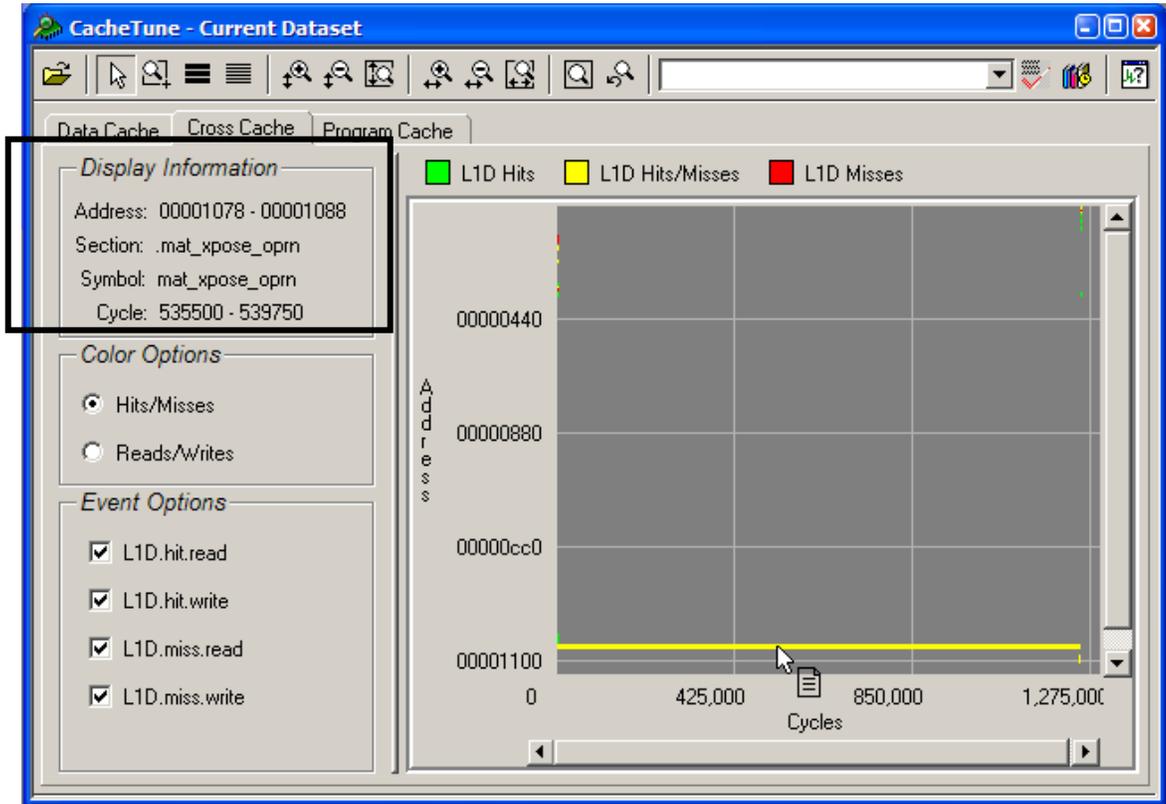
The cross cache view in CacheTune encodes a cross-reference of program memory accesses with data cache accesses. The addresses are the same as for the program cache display, but the events displayed are with respect to the data memory reads and writes (loads and stores) contained within those instructions and whether those loads and stores hit or missed in the data cache (L1D).

Cross Cache can help to analyze the data memory references by the program. For example, we can tell which functions access the data memory, and whether the accesses miss the data cache or not.

41. Switch to the cross cache view by clicking on the **Cross Cache** tab located below the CacheTune toolbar. Each tab maintains its own toolbar state, mode, selected symbol, etc. The advice will change to cross cache advice.
42. Press the **Full Zoom button to view the entire trace**. Notice there is one horizontal bar across the display, which represents the code executed repeatedly.



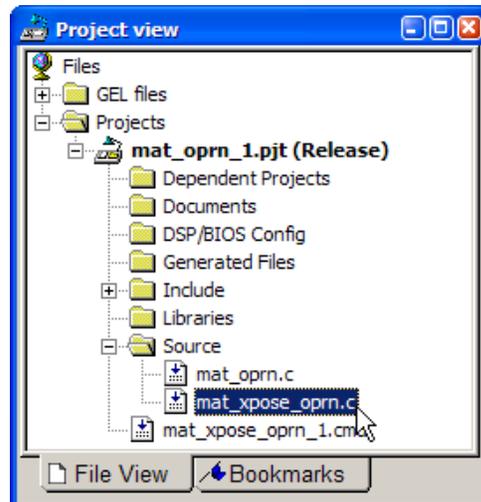
- Place the cursor on top of the bars. The function containing the code then can be easily identified from the Information Display Area. The memory accesses are only associated within function `mat_xpose_oprn`.



Optimizing C Code

Based on the analysis above, most cache misses on L1D are capacity misses due to suboptimal data usage, that is, the algorithm does not reuse the data when they are still in cache. This type of miss can be reduced by restructuring the data in order to work on smaller blocks of data at a time.

- From **Project View** window, double-click on `mat_xpose_oprn.c` program to open it.



45. Modify the `mat_xpose_oprn` function as the source code shown below. Please include the appropriate tab spaces.

```
void mat_xpose_oprn(void)
{
    short i,j,ii,jj;

    /* Matrices are accessed in parts and the parts are reused */
    for (ii = 0; ii < MAT_ORDER; ii += MAT_PART_SIZE) {
        for (jj = 0; jj < MAT_ORDER; jj += MAT_PART_SIZE) {
            for (i = ii; i < MAT_PART_SIZE + ii; i++) {
                for (j = jj; j < MAT_PART_SIZE + jj; j++){
                    C[i][j] = mat_oprn(A[i][j],B[j][i]);
                }
            }
        }
    }
}

main() { mat_xpose_oprn(); }
```

The function now works on smaller pieces of the matrix at a time. This reduces the size of the working set and improves temporal locality.

46. Click on the Incremental Build  icon to rebuild the modified program.
47. Choose **Debug**→**Reset CPU** to clear the cache.
48. Choose **File**→**Reload Program** to reload the program.

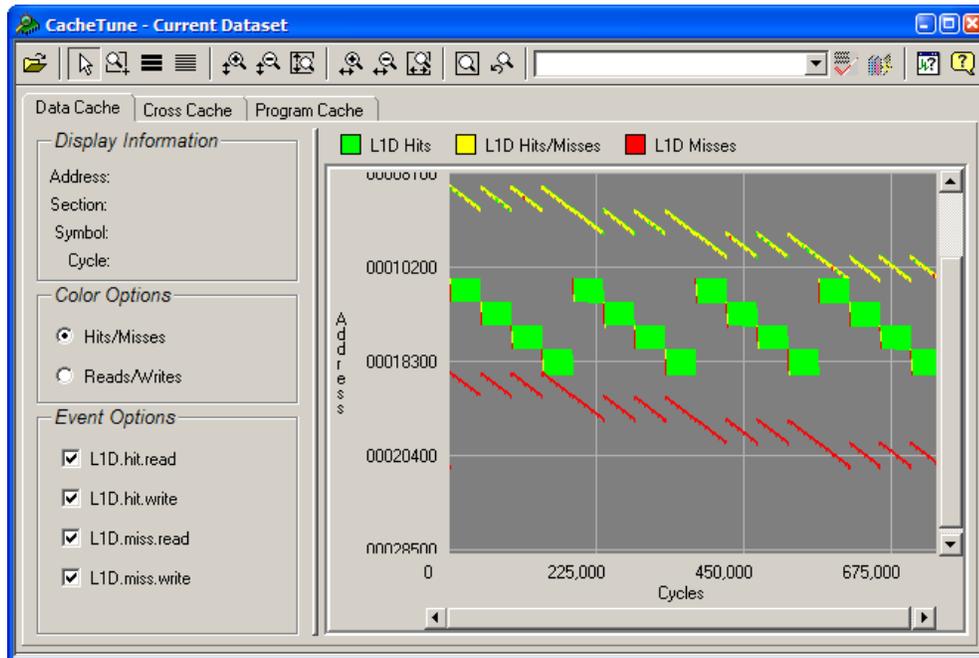
Measuring Cache Usage Improvement

We can now visualize the new data cache access pattern with the CacheTune tool and use Goals Window and Profile Viewer to check the improvement.

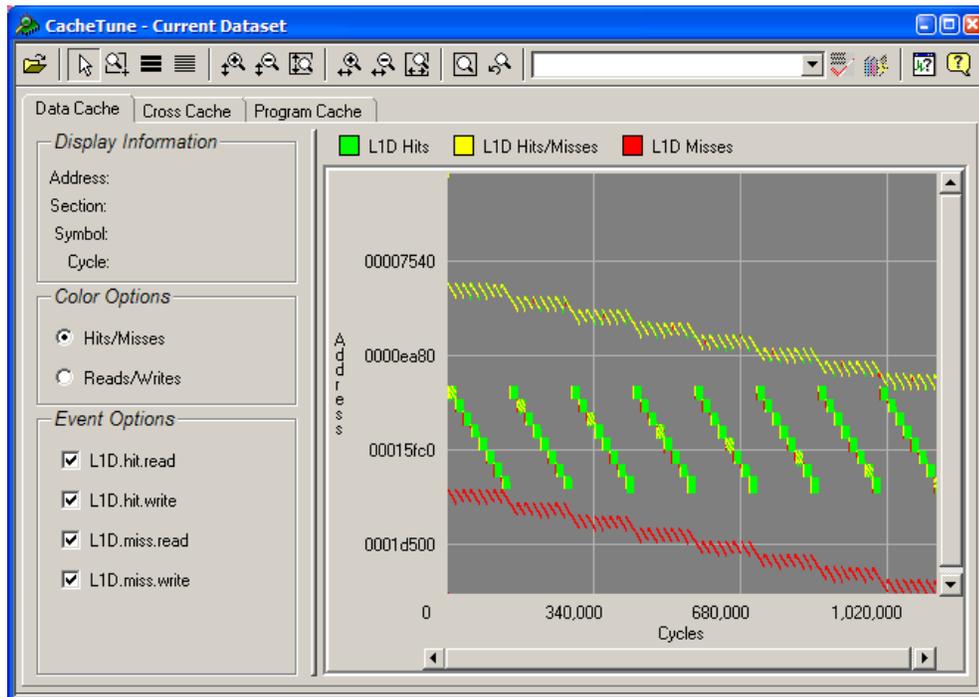
49. Press **F5** key to run the program. Once program finishes execution, the CacheTune output window, Goals Window, and Profile Viewer will update to display the new data.
50. Click on the **DataCache** tab of the CacheTune output window.

51. Click on the **Full Zoom**  button to view the entire trace. Most read misses now are compulsory misses. Also notice that only a smaller block of matrix is accessed within each particular time frame.

Here is the result for the C64:

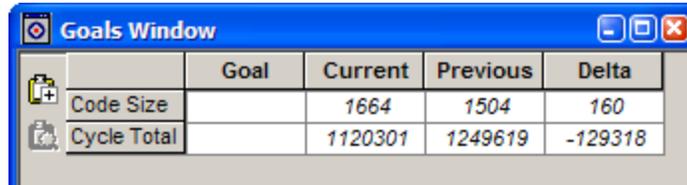


And, here is the result for the C67:



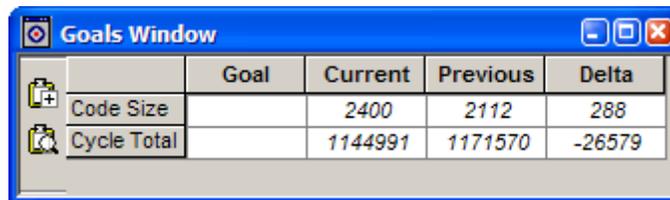
52. Check the data in **Goals Window**. The data from the last execution of the program moves to the Previous column and the difference between values is displayed in the **Delta** column. Notice that the overall performance of the program improves 129,318 cycles after optimizing the data cache. Your numbers may vary.

Here are the results for the C64:



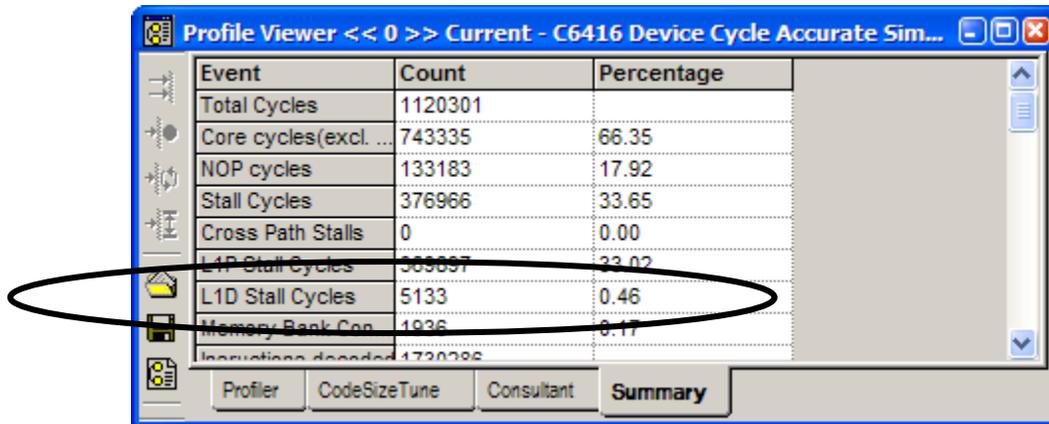
	Goal	Current	Previous	Delta
Code Size		1664	1504	160
Cycle Total		1120301	1249619	-129318

For the C67:



	Goal	Current	Previous	Delta
Code Size		2400	2112	288
Cycle Total		1144991	1171570	-26579

53. In **Profile Viewer**, check the data for the L1D Stall Cycles row in the Summary tab. Now the count for the C64 is 5,133 cycles, and the percentage is 0.46%, both of which are dramatically improved. Your numbers may vary. These stalls are caused by the compulsory misses when the array elements are first accessed.



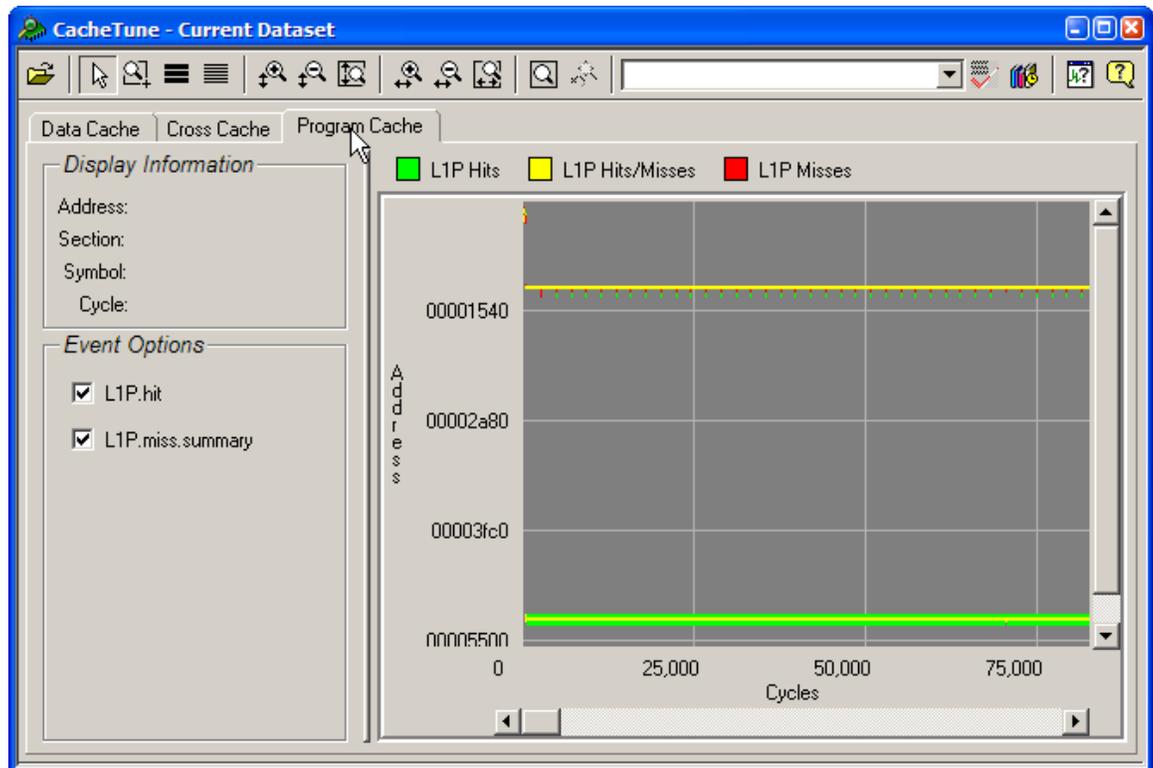
Event	Count	Percentage
Total Cycles	1120301	
Core cycles(excl. ...	743335	66.35
NOP cycles	133183	17.92
Stall Cycles	376966	33.65
Cross Path Stalls	0	0.00
L1P Stall Cycles	363897	33.02
L1D Stall Cycles	5133	0.46
Memory Bank Con	1936	0.17
Instruction decoder	172028	

Since there are still a high number of cache misses in the program cache, we need to investigate the program memory access pattern and improve program cache utilization in the next lesson.

Part 4 - Program Cache Overview

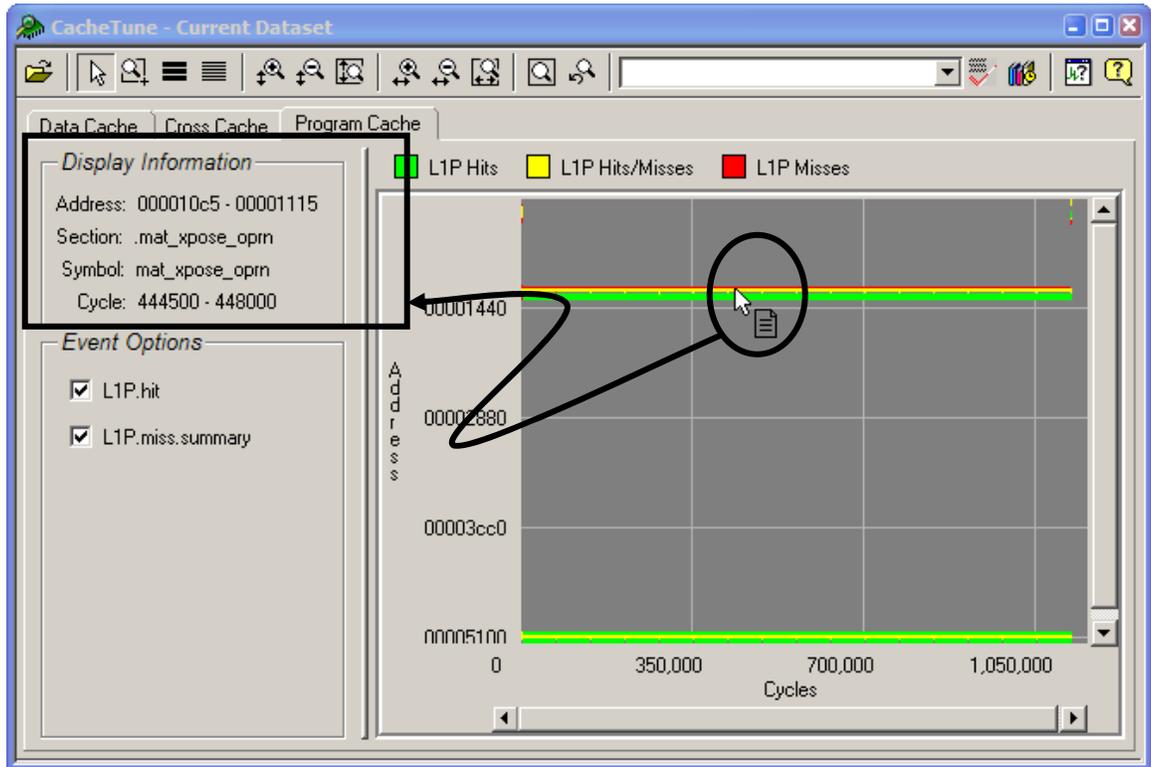
The CacheTune tool is also valuable for improving program memory performance in cache-based devices. The CacheTune display easily highlights conflict misses in the program cache when two or more functions map to the same cache location. In this exercise, you will identify these kinds of conflict misses and learn to use the Interference Grid, Interference Shadow, and Find Conflicting Symbols Tool to locate the conflicting functions. We will also discuss the optimization techniques used to eliminate these misses.

- Switch to the program cache view by clicking on the Program Cache tab located below the CacheTune toolbar. This also displays the program cache advice in the advice window.

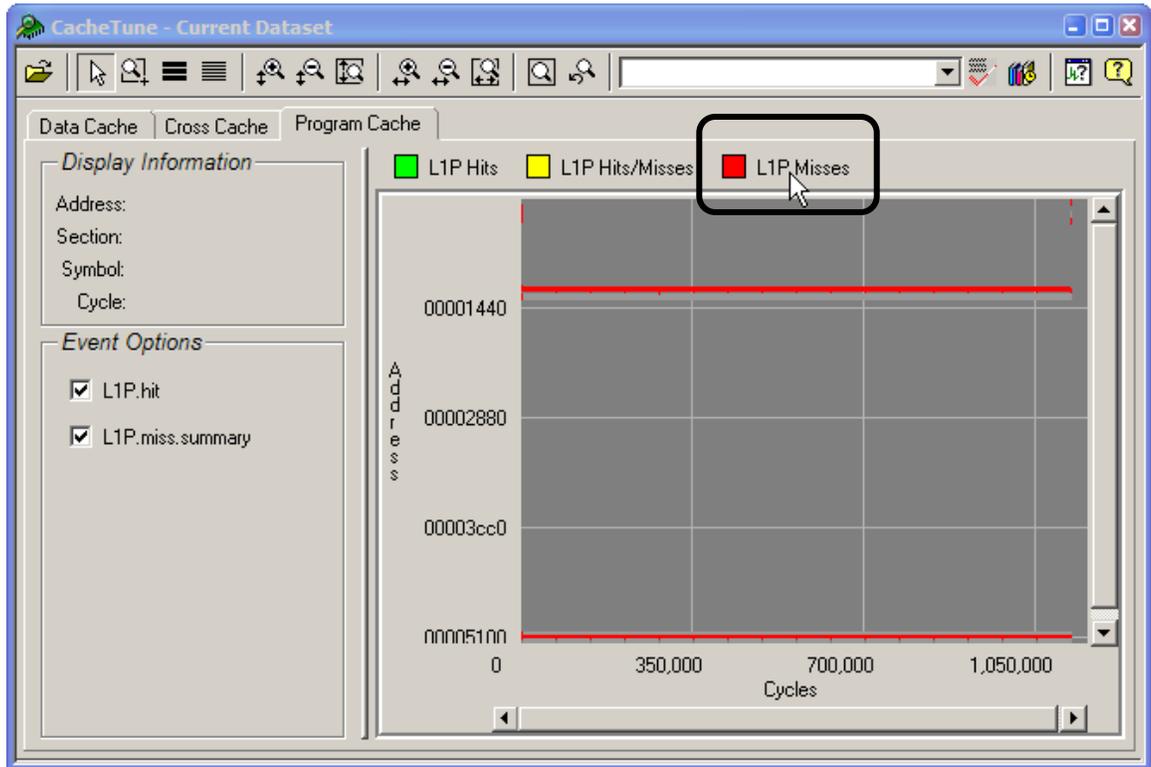


55. Press the Full zoom  button to view the entire trace.

Note again how the function names change in the Symbol display as you move the cursor up and down over the graphical display area. Notice the horizontal bars across the display. Each bar represents a piece of code that is executed repeatedly at short time intervals. Moving the cursor over the bars identifies the functions that contain the code.



56. Roll your mouse over the **L1P Misses** legend above the graph to view cache misses events only, other events are grayed out.



Notice that there are two red bars along their horizontal extent, signifying that the same pieces of code miss repeatedly in the cache within short periods of time. Cache conflicts cause this type of repetitive misses.

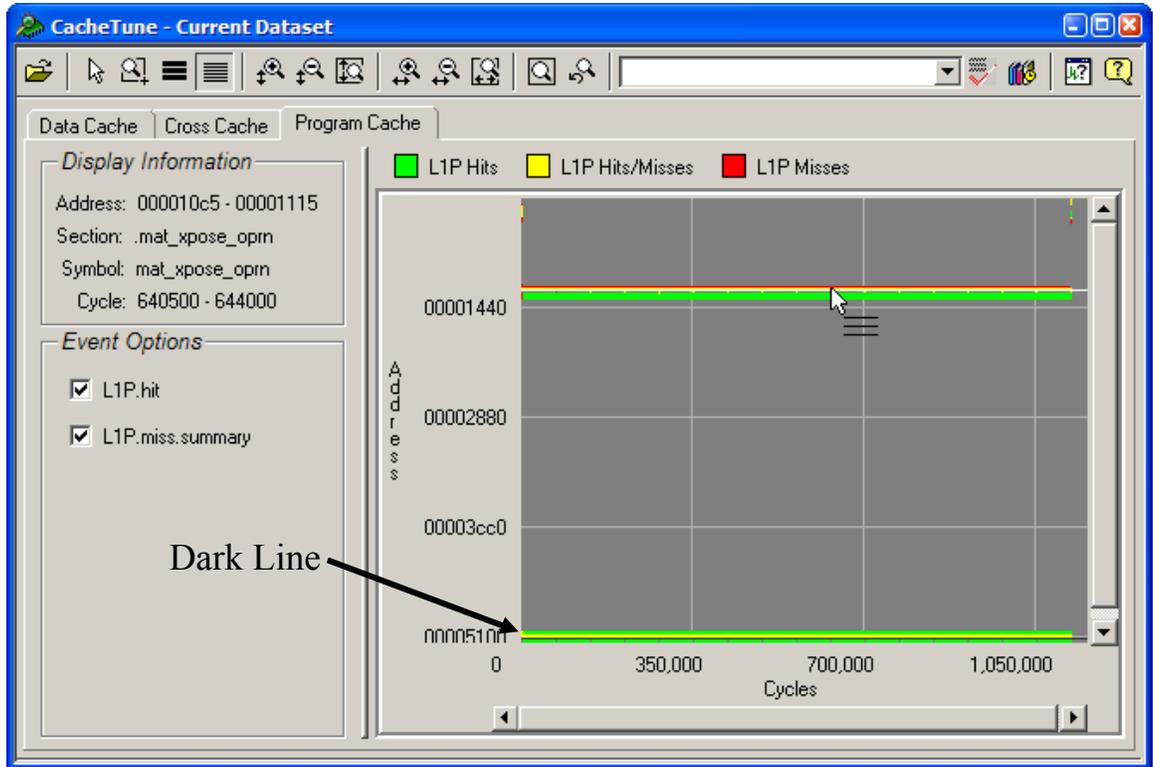
CacheTune has three tools to help identify which objects (functions in this case) create such conflicts in the cache. These tools are called the Interference Grid, the Interference Shadow, and the Find Conflicting Symbols tools.

Using the Interference Grid

The Interference Grid draws a line at every address in the display that interferes with the selected address in the cache.

57. Press the Interference Grid  button.

58. Select the origin of the grid at an address within the memory range of function `mat_xpose_oprn` (represented by the top horizontal bar). A dark grid will be drawn at any conflicted memory address, which is every other 16K byte address (size of L1P cache) for a direct-mapped L1P on the C6416. Notice a dark line is drawn on top of the lower horizontal bar.



59. Right-click on the graph to clear the interference lines.

Using the Interference Shadow

The Interference Shadow draws a shadow at every address range in the display that interferes in the cache with the selected symbol.

Note: This feature only works for global symbols that have a size associated with them.

The shadow mode has three different cursors:

- The cursor with two solid bars indicates that it is over a symbol. You can left click to select the shadow.



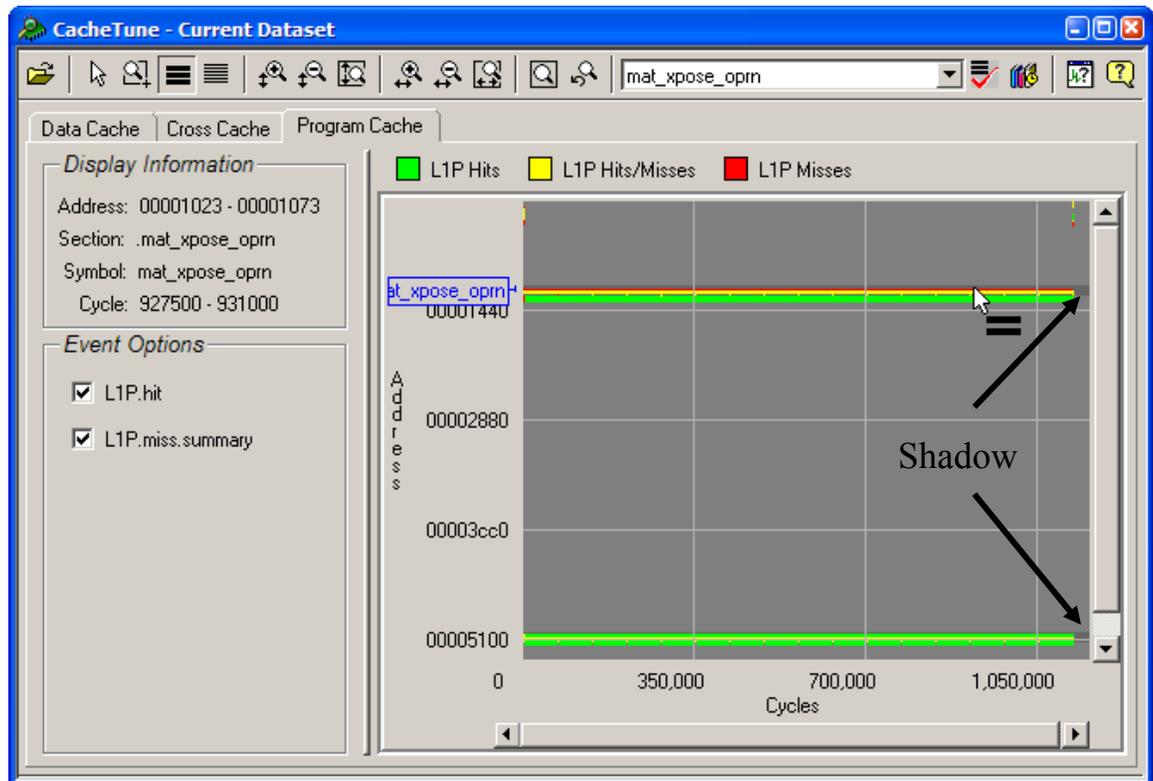
- The cursor with two empty bars indicates that no symbols are available under the cursor.



- The cursor with large filled in box that indicates the cursor is over a symbol conflicting with all of the cache block (larger than cache size). The user can still click this symbol to select it (red name in address area) but the shadow will not be drawn, as it would fill the entire graph.

60. Press the Interference Shadow  button.

61. Place the cursor at an address within the memory range of function `mat_xpose_oprn` (represented by the top horizontal bar) and left click. The resulting display will show dark bands across the graphical display area to indicate the address ranges that interfere with this function in the cache.

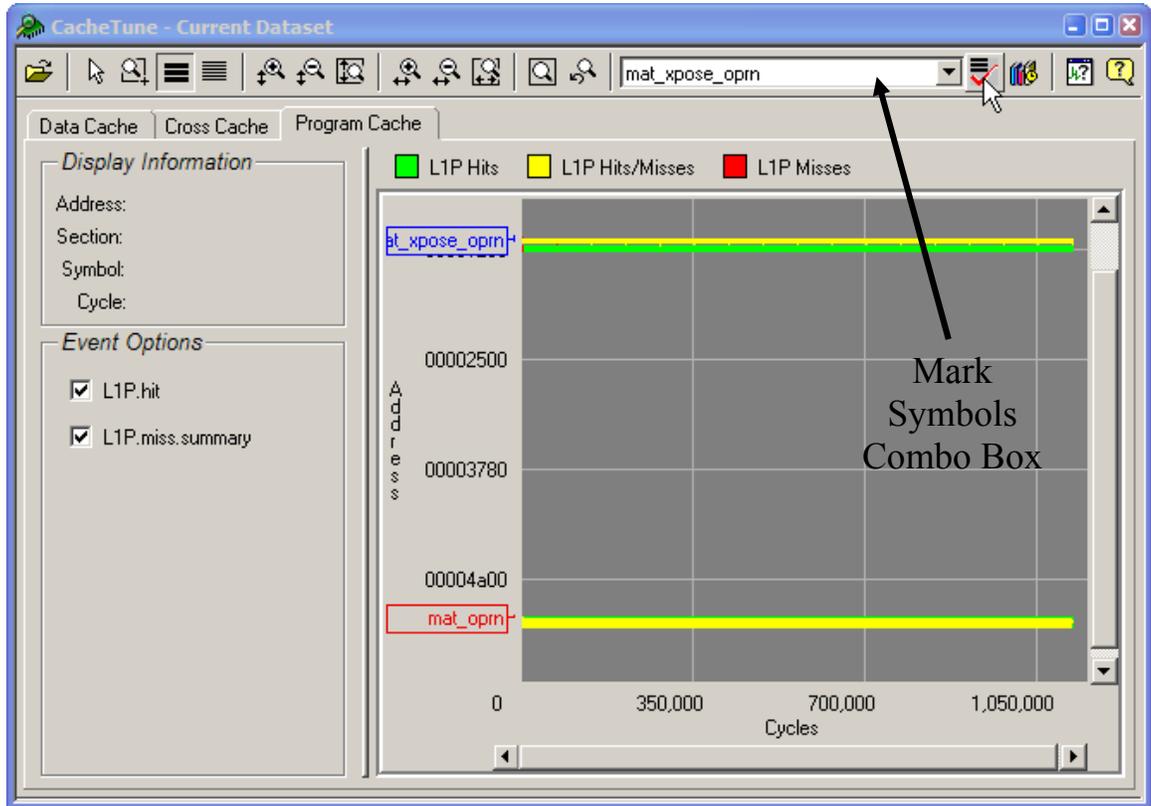


62. Right-click on the graph to clear the interference lines.

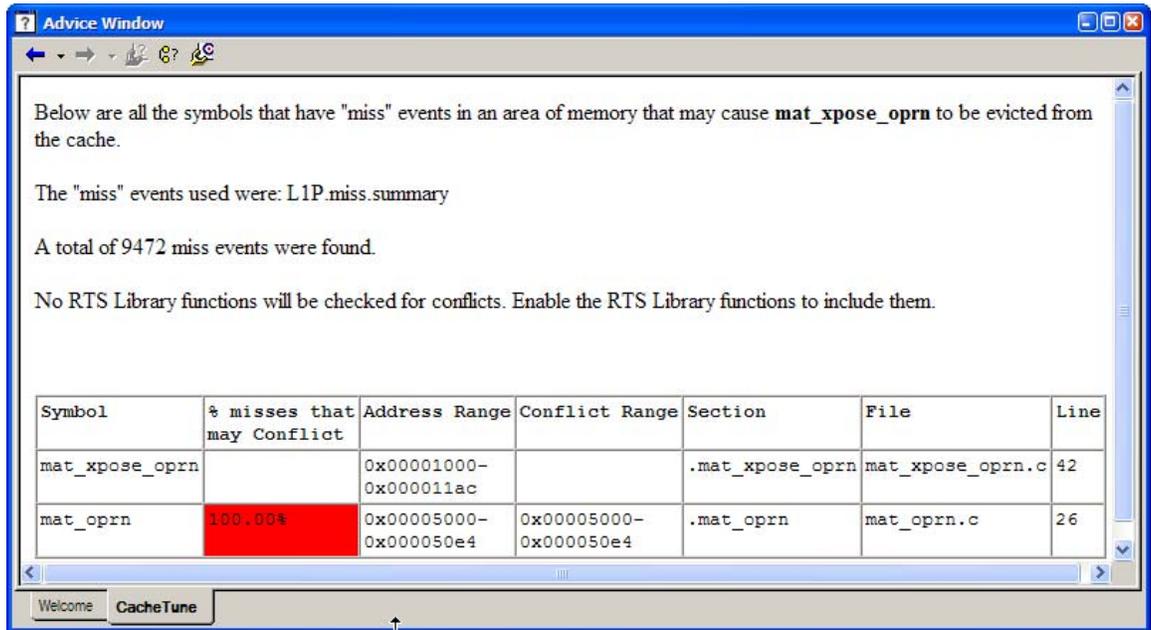
Find Conflicting Symbols

The Interference Shadow tool also enables the Find Conflicting Symbols button . Find Conflicting Symbols displays a list of the symbols and graphs that conflict in memory with the selected symbol.

63. Click on the Find Conflicting Symbols button . This displays all cache conflicting symbols with a red box around them in the Address Section.

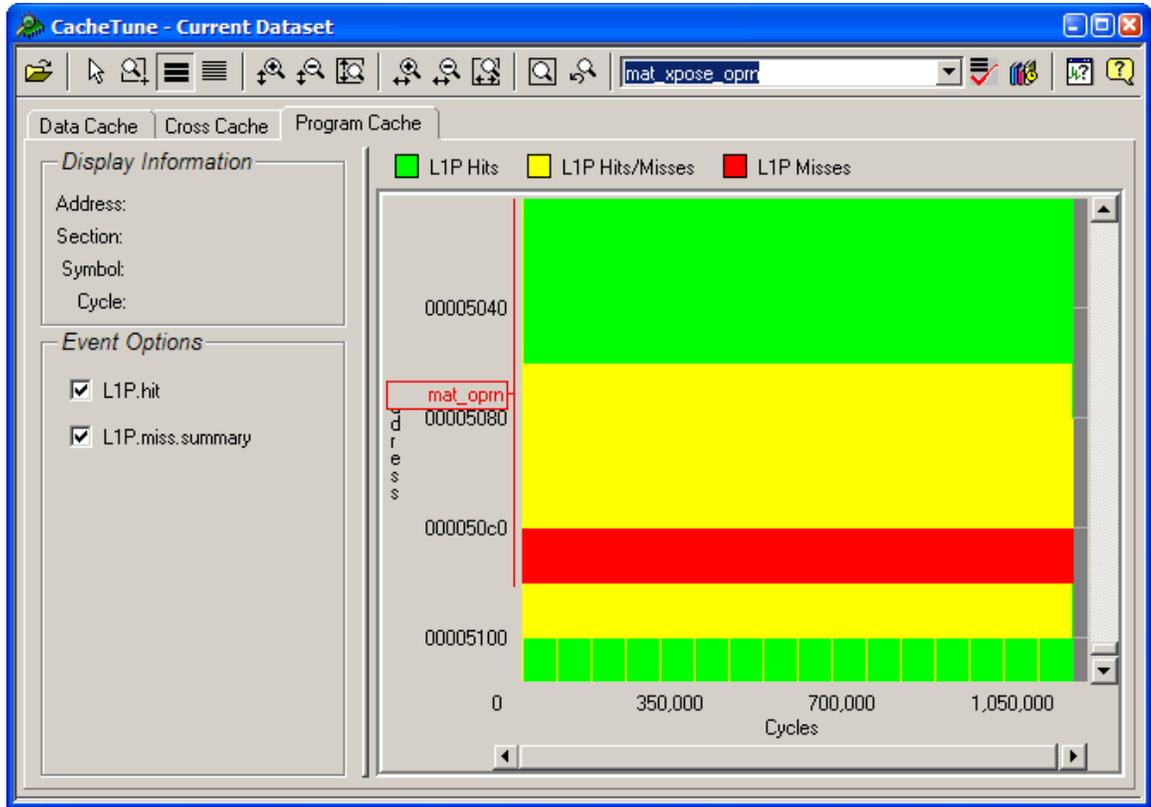


Also, the CacheTune tab of Advice Window displays text information on the conflicting symbols, such as the symbol's name, the conflict range, etc.



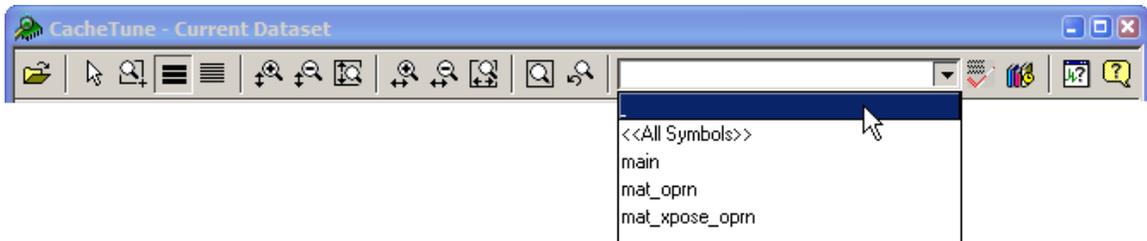
Note: If the **Find Conflicting Symbols** button is grayed out, you can also select the particular symbol from the **Mark Symbol Combo** box. Once a symbol is selected, it enables the **Find Conflicting Symbols** button and displays the symbol name in the Address Section.

64. The conflicting symbols displayed are based on memory placement, they might or might not cause cache conflict misses with selected symbol. For example, the conflicting symbols might not be referenced during the same program run. You might also want examine CacheTune graph for cache misses. Double-click on a conflicting symbol. CacheTune will zoom in to display a more detailed view of that symbol.



The function "mat_xpose_oprn" interferes in the cache with the function mat_oprn. When they execute back to back, they evict each other from the cache, and, subsequently, miss the next time they execute.

65. Select the first empty item from the **Mark Symbols Combo** box. This removes all the displayed symbol names in the Address Section.



The Program Cache patterns of this example matches the first miss scenario discussed in the Program Cache Advice Window. The advice for eliminating the misses is to allocate the conflicting functions contiguously in memory.

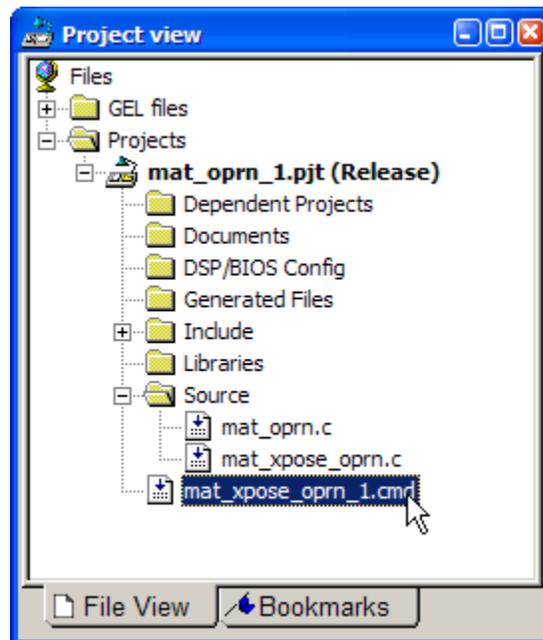
Eliminating the Cache Conflicts

Knowing the cache misses are conflict misses due to two functions mapped to the same cache lines, we can place the affected functions in different memory locations so that they do not overlap in cache. To achieve this efficiently, place the code of the two functions contiguously in memory.

Since the functions that execute when the interference occurs occupy less than 16KB of memory (the C64x program cache size, 4KB for the C67x), allocating them contiguously will help eliminate any conflicts.

The two functions: `mat_xpose_oprn` and `mat_oprn` that require contiguous placement have been assigned individual sections by using the `pragma CODE_SECTION` before the definition of the functions. We can simply edit the linker command file to allocate them congruently.

66. From the Project View window, double-click on `mat_xpose_oprn_1.cmd` to open the program.



67. Modify the linker command file supplied with the original source with the changes shown below.

```

MEMORY
{
    MEM0:          o = 00000000h    l = 00001000h
    MEM1:          o = 00001000h    l = 00004000h
    MEM2:          o = 00005000h    l = 00004000h
    ISRAM0:        o = 00009000h    l = 00037000h
    ISRAM1:        o = 00040000h    l = 000c0000h
    CEO:           o = 80000000h    l = 01000000h
}

SECTIONS
{
    .text:          >          MEM0

    .mat_xpose_oprn >          MEM1
    .mat_oprn       >          MEM1

    .stack          >          MEM0
    .cinit          >          MEM0
    .data           >          MEM0
    .bss            >          MEM0
    .const         >          MEM0
    .far            >          ISRAM0
    .external       >          CEO
}

```

Note: For details regarding the linker command file, see the *Assembly Language Tools Users' Guide (SPRU186)*.

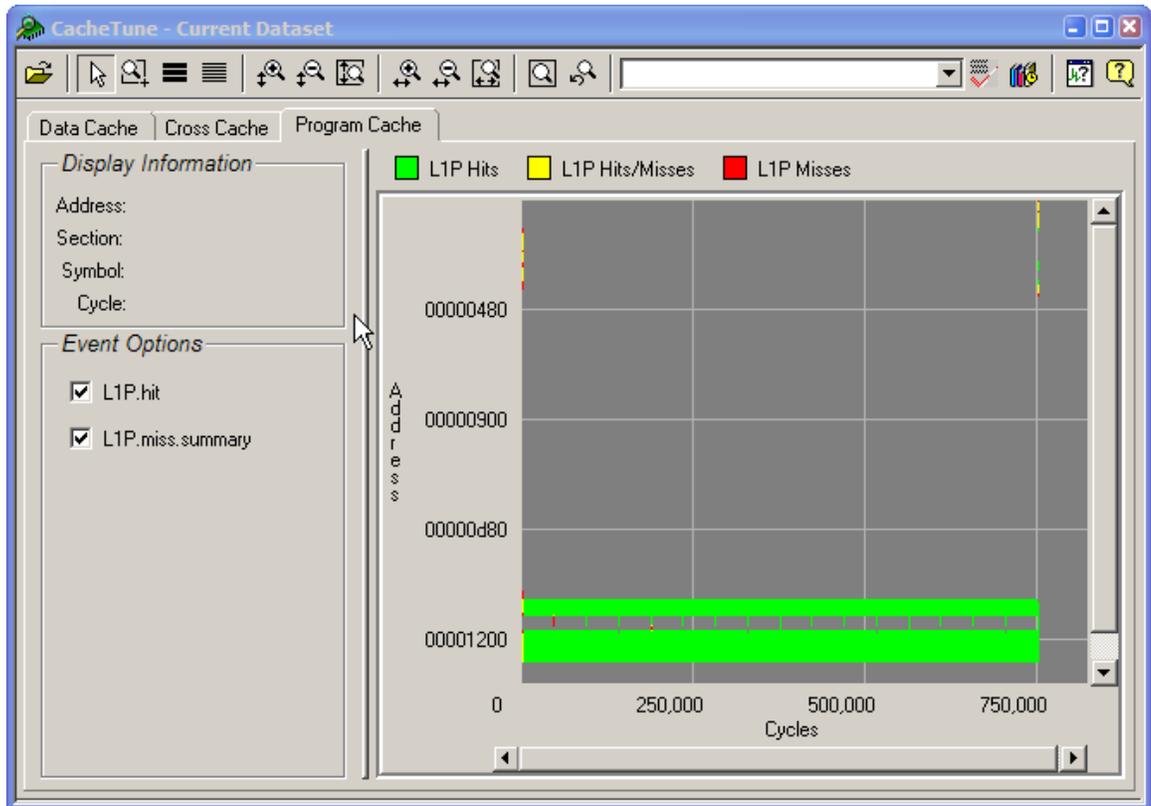
68. Click on the Incremental Build  button. As only the linker command file is modified, this will only re-link the program.
69. Choose **Debug**→**Reset CPU** to clear the cache.
70. Choose **File**→**Reload Program** to reload the program.

Measuring the Improvement

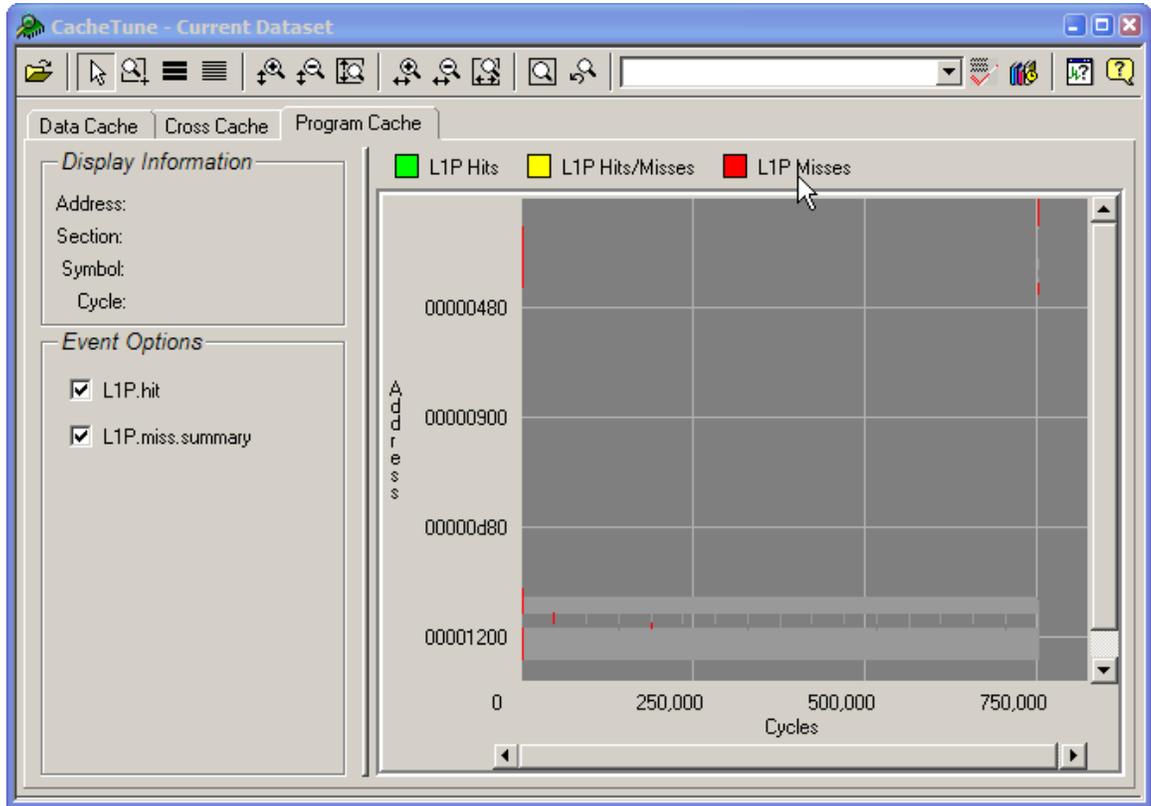
To see the result of applying this modification, repeat the steps from the first lesson.

71. From the **Debug** menu, select **Run**, and run the program to completion. Once program finishes execution, the CacheTune output window, Goals Window and Profile Viewer will update to display the new data.
72. Click on the **Program Cache** tab in the CacheTune window.

73. Click on the **Full Zoom** button view the entire trace.



74. Roll your mouse over the **L1P Misses** legend above the graph to view cache misses events only. The graph should resemble the image below. Notice that red horizontal bars are eliminated and most of cache references are cache hits. The cache misses that are left are compulsory misses, and cannot be eliminated.



75. Check the data in **Goals Window**. The data from the last execution of the program moves to the **Previous** column and the difference between values is displayed in the **Delta** column. Notice that the overall performance of the program improves 369,740 cycles for the C64x and 409,060 cycles for the C67x after optimizing the program cache.

Here is the goals window for the C64:

	Goal	Current	Previous	Delta
Code Size		1664	1664	0
Cycle Total		750561	1120301	-369740

Here is the goals window for the C67:

	Goal	Current	Previous	Delta
Code Size		2400	2400	0
Cycle Total		735931	1144991	-409060

76. In **Profile Viewer**, check the data for the L1P Stall Cycles in the Summary tab. Now the count is 157 cycles for the C64 and 138 for the C67. The percentage for each is 0.02%, both of which are dramatically improved. Your numbers may vary. These stalls are caused by the compulsory misses when the functions are first accessed.

Event	Count	Percentage
Total Cycles	750561	
Core cycles(excl. ...	743335	99.04
NOP cycles	133183	17.92
Stall Cycles	7226	0.96
Cross Path Stalls	0	0.00
L1P Stall Cycles	157	0.02
L1D Stall Cycles	5133	0.68
Memory Bank Con...	1936	0.26
Instructions decod...	172028	

End of Lab13c

Lab Debrief

While your results may differ slightly from these, they should be similar.

Lab14a Results – Memory Bank Conflicts

Dot-product Version	Cycles
Lab15a – Step 20 (Release options)	365*
Lab15a – Step 28 (Release options w/pragma)	282

* Includes 104 Memory Bank Conflict stalls as detected by the simulator



Lab 14b Results

	P/D: Off-Chip MAR's = 0x0000 L2 Cache = SRAM (off)	P/D: Off-Chip MAR's = 0xFFFF L2 Cache = 4-WAY (on)	P/D: On-Chip MAR's = xxxx L2 = 1/4 Cache & 3/4 SRAM
C64	7,418,733* cycles	263,617 cycles	3,091 cycles
C67	1,240,004* cycles	1,162,345 cycles	4,150 cycles

35. Which configuration is fastest? How did cache affect the results?

On-chip is fastest, though turning on the cache made a huge difference!

36. Was there an advantage to linking your code and data into on-chip memory?

Yes.

37. When might it make good sense (and cents) to link data on-chip?

As discussed earlier, when the EDMA is moving data from/to a peripheral.



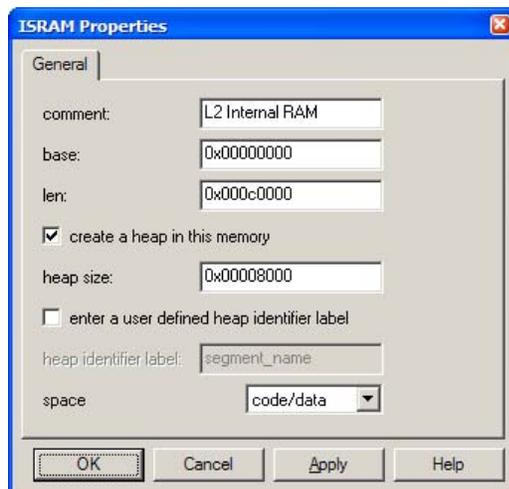
*** This large delta is caused by different External Memory Configurations.**

Lab 14b Results

16 & 17. The base was set to 0x0 since that's where L2 begins.

◆ Why did we set the length to 0xC0000 or 0x30000?

3/4 L2 is 768K bytes or 192Kb



This page intentionally left blank.

iUniversal Lab Exercise

Lab Topics

iUniversal Lab Exercise	14-1
<i>Lab 14a – Creating a Universal Algo</i>	<i>14-2</i>
Running the GenCodecPkg wizard	14-2
Import Codec Library Project into CCSv4	14-5
Customizing the Code to Fit Your Algorithm	14-6
Customizing the Code to Fit Your Algorithm	14-7
Create a CCS Project for our Test Algorithm	14-11
Setup the new RTSC Config Project	14-15
Setup the Algorithm Test Project	14-17
Run and Debug Algorithm	14-18
<i>Test Application Code</i>	<i>14-19</i>
<i>Lab 14b – Creating a "server" for your algorithm</i>	<i>14-23</i>
 Copy the necessary files into the VM shared folder	14-23
 File management in Linux	14-23
 Make the DSP Server	14-24
 Build and Test the app and algo	14-24

Lab 14a – Creating a Universal Algo

Running the GenCodecPkg wizard

1. Start CCSv4.
2. Invoke *GenCodecPkg* wizard.

Option 1: From CCSv4

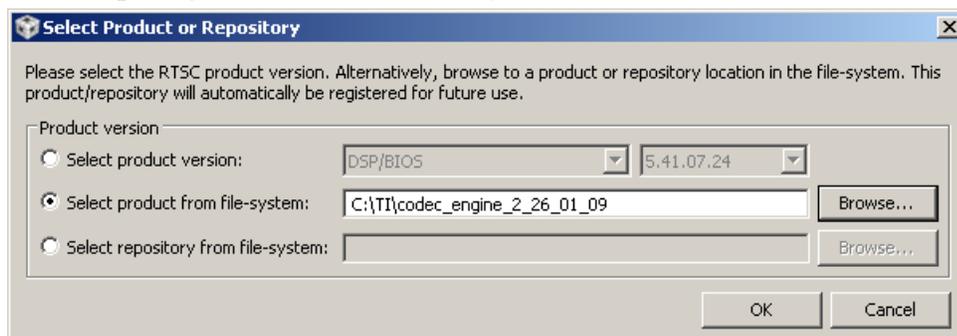
Add your Codec Engine installation directory to the "Tool Discovery Path". Go to

Window | Preferences | CCS | RTSC

and verify the package `codec_engine_2_26_01_09` shows up in the window. If so, check the box next to it and exit the preferences.

If it's not there:

- a. Add this package from the *C:/TI* directory.



- b. Close and save the dialog.
- c. Restart CCS. (At this point, it's the easiest way to restart workspace).
- d. Then reopen the same dialog to check the checkbox for the codec engine, then close the dialog.

At this point, you will have a new "Tools | Codec Engine Tools | GenCodecPkg" menu item that will launch the wizard. Go ahead and startup the wizard.

Option 2: Create a simple makefile to invoke the wizard

```
# Paths to required dependencies
XDC_INSTALL_DIR := C:/ti/xdctools/xdctools_3_20_03_63
CE_INSTALL_DIR  := C:/ti/codec_engine/codec_engine_2_26_01_09
XDAIS_INSTALL_DIR := $(CE_INSTALL_DIR)/cetools

# Create an XDCPATH variable
export XDCPATH = $(CE_INSTALL_DIR)/packages;$(XDAIS_INSTALL_DIR)/packages

.PHONY : gencodecpkg
gencodecpkg:
    $(XDC_INSTALL_DIR)/xs ti.sdo.ce.wizards.gencodecpkg
```

From the command-line, start the wizard with: `make gencodecpkg`

Try
this
option

3. Generate IUNIVERSAL starterware – GenCodecPkg page 1.



- Click 3rd option, "I want to create an algorithm from scratch".
- The XDAIS directory should already point to your XDAIS directory (frequently the $\$(CE_INSTALL_DIR)/cetools$ directory if you're not using a SDK).
- Click Next.

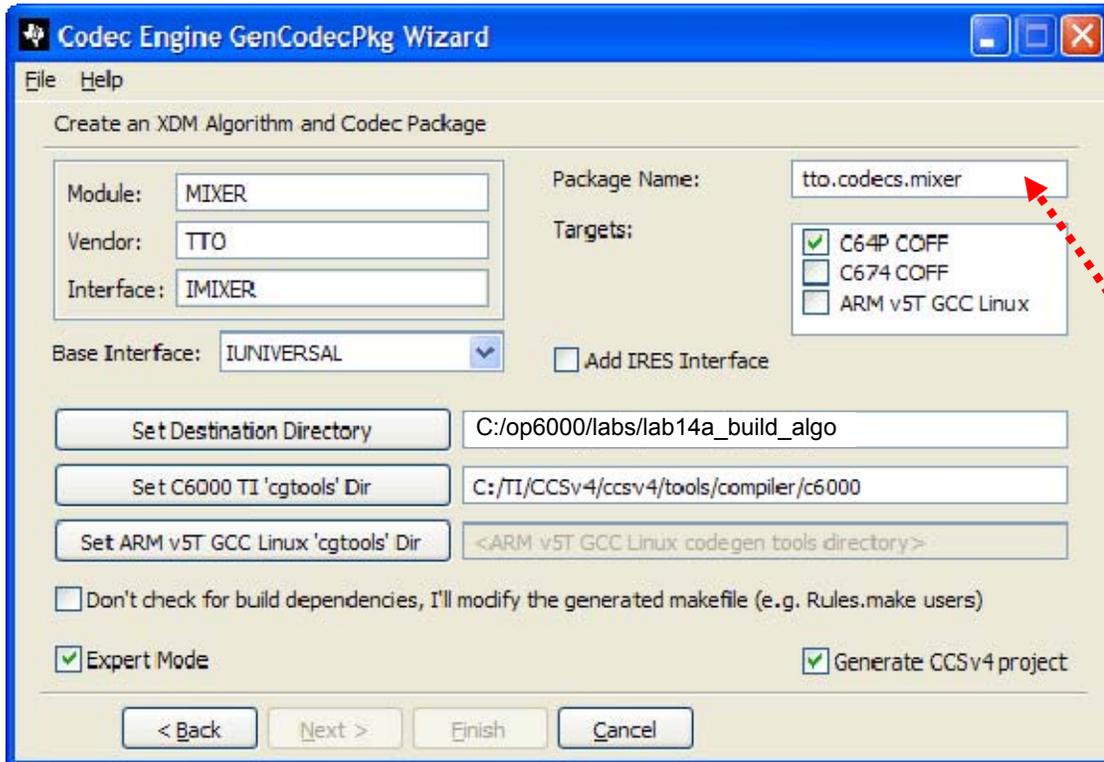
Sidenote

When downloading the Codec Engine (stand-alone) from TI, you can choose either the standard or *lite* versions.

The standard version contains an extra *cetools* directory which includes a number of additional packages which CE depends on – such as XDAIS.

The SDK team has chosen to install the lite version; then they install all the other packages at the root level of the SDK directory.

4. Generate IUNIVERSAL starterware – GenCodecPkg page 2.

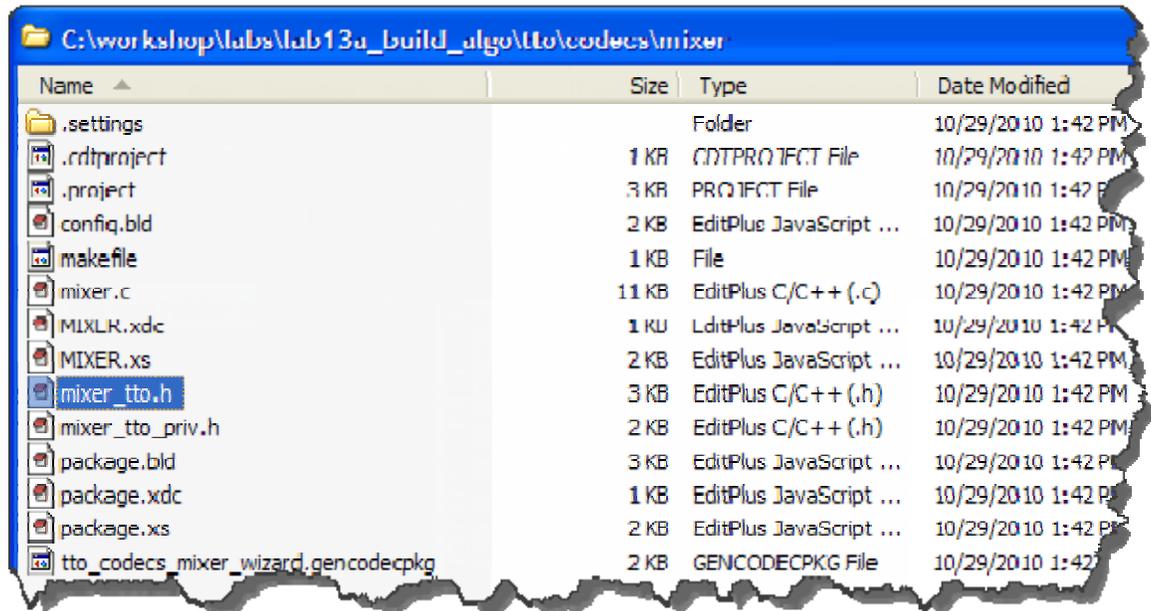


- a. In the "**Module**" field enter the name of the algorithm you're creating: MIXER
- b. In the "**Vendor**" field enter your company name: TTO
- c. Under "**Base Interface**" choose: IUNIVERSAL
- d. We want to build for C64x+ devices, so for "**Target**" choose: C64P COFF
- e. The **Destination Directory** is where the generated files will be placed. Click the button and create (and then select) the directory shown above.
- f. The **C6000 TI 'cgtools' Directory** should point to your TI DSP compiler (the root of the compiler installation, just above above the "bin" directory). When invoking the wizard from within CCSv4, this is probably already set for you.
- g. Check "**Expert Mode**" and change the "**Package Name**" to match that above. (While not required, we wanted to organize our files in this way.)
- h. Make sure that "**Generate CCSv4 project**" is checked.
- i. Click Finish to generate the starter files.

Common Mistake

Make sure you change the Package Name, so that your directory paths are named like those in this lab write-up.

5. View wizard's generated output files.



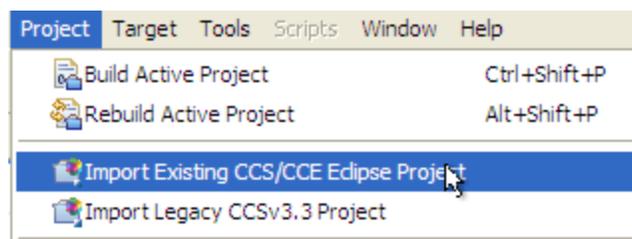
The generated files will reside at:

C:\op6000\labs\lab14a_build_algo\tto\codex\mixer

< repository >
< package name >

Import Codec Library Project into CCSv4

6. Open CCSv4 and “Import Existing Project”.



Import the codec wizard project we just created in:

C:\op6000\labs\lab14a_build_algo

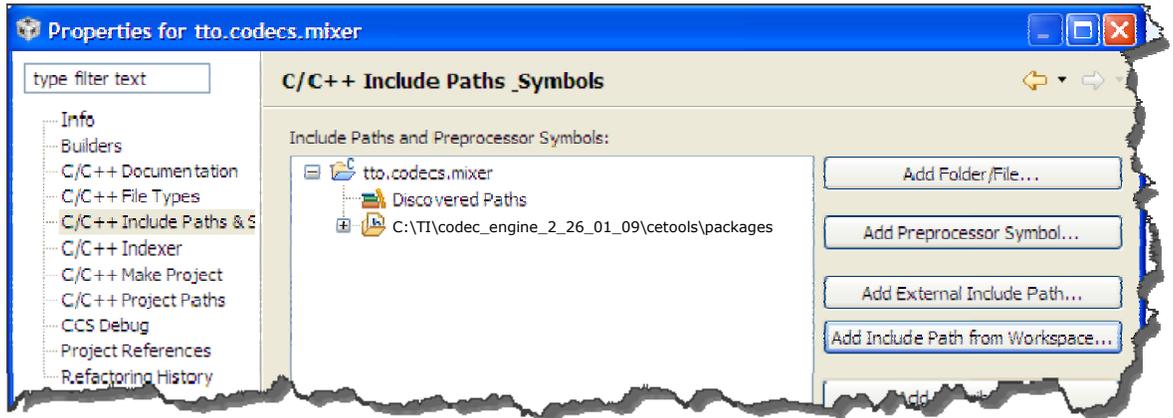
7. Select the project you just imported and build it.

Before making any changes, you should be able to build the generated package as is. It's worth verifying the generated files build correctly.

Select Project | Right-Click | Build Project

8. Setup code completion for CCSv4.

A good reason to use CCS is to take advantage of features like code completion. Later, when debugging, if you press <Ctrl> <Space> on your keyboard you can invoke the Eclipse code completion feature, which will help to finish typing the names of variables, functions, etc. You will likely want CCS to have the capability of auto-completing names/fields related to XDAIS's IALG libraries.



To set this up, do the following:

- Right-click on your project in the "C/C++ **Projects**" window and select Properties.
- Click on "C/C++ **Include Paths & Symbols**".
- Click "**Add External Include Path**".
- Click "**Browse**" and navigate to the appropriate source directory. Specifically for IALG definitions, you need to point to <xdais>\packages.

C:\TI\codec_engine_2_26_01_09\cetools\packages

Note

The XDAIS path shown in this step assumes that you have installed the full version of the Codec Engine, which includes a number of other libraries in the "cetools" directory. If you happened to install the "lite" version of the Codec Engine, then you would need to have installed the XDAIS library separately – and would want to use that path instead.

Customizing the Code to Fit Your Algorithm

We have chosen to implement a 2-channel mixer algorithm. This algo takes two buffers, weights them each with their own gain value, then adds the buffers together.

Here is the data we will need to pass to our algo's `_process()` function:

- Two **InBufs**
- One **OutBufs**
- Two Additional elements for **InArgs**:
 - gain0
 - gain1

Before

```

tto.codecs.mixer [Active]
├── Includes
├── lib
├── package
├── mixer_tto_priv.h
├── mixer_tto.h
│   ├── ti/xdais/dm/iuniversal.h
│   ├── ti/xdais/ialg.h
│   ├── # MIXER_TTO_IMIXER_
│   ├── IMIXER_PARAMS
│   ├── MIXER_TTO_IALG
│   ├── MIXER_TTO_IMIXER
│   ├── IMIXER_DynamicParams
│   └── IMIXER_DynamicParams
│       ├── base
│       ├── IMIXER_InArgs
│       └── IMIXER_InArgs
│           ├── base
│           ├── IMIXER_OutArgs
│           └── IMIXER_OutArgs
│               ├── base
│               ├── IMIXER_Params
│               └── IMIXER_Params
│                   ├── base
│                   ├── IMIXER_Status
│                   └── IMIXER_Status
│                       ├── base
│                       └── mixer.c
                    
```

After

```

tto.codecs.mixer [Active]
├── Includes
├── lib
├── package
├── mixer_tto_priv.h
├── mixer_tto.h
│   ├── ti/xdais/dm/iuniversal.h
│   ├── ti/xdais/ialg.h
│   ├── # MIXER_TTO_IMIXER_
│   ├── IMIXER_PARAMS
│   ├── MIXER_TTO_IALG
│   ├── MIXER_TTO_IMIXER
│   ├── IMIXER_DynamicParams
│   └── IMIXER_DynamicParams
│       ├── base
│       ├── IMIXER_InArgs
│       └── IMIXER_InArgs
│           ├── base
│           ├── gain0
│           ├── gain1
│           └── IMIXER_OutArgs
│               ├── IMIXER_OutArgs
│               ├── base
│               ├── IMIXER_Params
│               └── IMIXER_Params
│                   ├── base
│                   ├── IMIXER_Status
│                   └── IMIXER_Status
│                       ├── base
│                       └── mixer.c
                    
```

First, we'll add two new elements to **IMIXER_InArgs**

Then we'll implement the `_process()` call in **mixer.c**

9. Modify the algo’s data structures.

Since the buffers descriptors can take a variable number of buffers, we don’t really need to modify the wizard’s output for them. We will have to add the gain variables to our code, though.

Please refer to the “[Getting Started with IUNIVERSAL](#)” wiki page – Step 4 : Customizing the Code to Fit Your Algorithm. (We provided this in PDF format in the `lab14_starter_files` folder.)

Some hints on what to edit when following the Getting Started with IUNIVERSAL:

`mixer_tto_priv.h`

- e. Edits to `<module>_<vendor>_priv.h`:
No changes needed...

`mixer_tto.h`

- f. Edits to `<module>_<vendor>.h` (in our case: `mixer_tto.h`):
 - Define commands for use in the `control()` function.
No changes needed...
 - Extend the `I<MODULE>_Params` as needed.
No changes needed...

- Extend the `I<MODULE>_InArgs` structure as needed.

```
typedef struct IMIXER_InArgs {  
    /* This must be the first field */  
    IUNIVERSAL_InArgs base;  
  
    XDAS_Int16 gain0;  
    XDAS_Int16 gain1;
```

Type in By Hand

- Extend the `I<MODULE>_OutArgs` structure as needed.
No changes needed...
- Extend the `I<MODULE>_DynamicParams` structure as need.
No changes needed...
- Extend the `I<MODULE>_Status` structure as necessary.
No changes needed...

mixer.c Edits to <module>.c

No changes needed

- const I<MODULE>_Params I<MODULE>_PARAMS
- <MODULE>_<VENDOR>_alloc()
- <MODULE>_<VENDOR>_free()
- <MODULE>_<VENDOR>_initObj()
- <MODULE>_<VENDOR>_process()

Cut & Paste

Edit `_process()` call

This is where the actual algorithm goes! Add your algorithm code here.

Note, we provided the entire `process()` function code for you to cut/paste to help minimize typos.

```

/*
 * ===== MIXER_TTO_process =====
 */

/* ARGSUSED - this line tells the TI compiler not to warn about unused args. */
XDAS_Int32 MIXER_TTO_process(IUNIVERSAL_Handle h,
    XDML_BufDesc *inBufs, XDML_BufDesc *outBufs, XDML_BufDesc *inOutBufs,
    IUNIVERSAL_InArgs *universalInArgs,
    IUNIVERSAL_OutArgs *universalOutArgs)
{
    XDAS_Int32 numInBytes, i;
    XDAS_Int16 *pIn0, *pIn1, *pOut, gain0, gain1;

    /* Local casted variables to ease operating on our extended fields */
    IMIXER_InArgs *inArgs = (IMIXER_InArgs *)universalInArgs;
    IMIXER_OutArgs *outArgs = (IMIXER_OutArgs *)universalOutArgs;

    /*
     * Note that the rest of this function will be algorithm-specific. In
     * the initial generated implementation, this process() function simply
     * copies the first inBuf to the first outBuf. But you should modify
     * this to suit your algorithm's needs.
     */

    /*
     * Validate arguments. You should add your own algorithm-specific
     * parameter validation and potentially algorithm-specific return values.
     */
    if ((inArgs->base.size != sizeof(*inArgs)) ||
        (outArgs->base.size != sizeof(*outArgs))) {
        outArgs->base.extendedError = XDM_UNSUPPORTEDPARAM;

        return (IUNIVERSAL_EUNSUPPORTED);
    }

    /* validate that there's 2 inBufs and 1 outBuf */
    if ((inBufs->numBufs != 2) || (outBufs->numBufs != 1)) {
        outArgs->base.extendedError = XDM_UNSUPPORTEDPARAM;

        return (IUNIVERSAL_EFAIL);
    }

    /* validate that buffer sizes are the same */
    if (inBufs->descs[0].bufSize != inBufs->descs[1].bufSize)
        return IUNIVERSAL_EFAIL;
    if (inBufs->descs[0].bufSize != outBufs->descs[0].bufSize)
        return IUNIVERSAL_EFAIL;
    
```

↳ Continued on next page

```

// DO IT!
pIn0 = (XDAS_Int16*)inBufs->descs[0].buf;
pIn1 = (XDAS_Int16*)inBufs->descs[1].buf;
pOut = (XDAS_Int16*)outBufs->descs[0].buf;
gain0 = inArgs->gain0;
gain1 = inArgs->gain1;
numInBytes = inBufs->descs[0].bufSize;

for(i=0; i<numInBytes/2; i++)
{
    pOut[i] = (pIn0[i]*(XDAS_Int32)gain0 + pIn1[i]*(XDAS_Int32)gain1) >> 15;
}

/* report how we accessed the input buffers */
inBufs->descs[0].accessMask = 0;
XDM_SETACCESSMODE_READ(inBufs->descs[0].accessMask);
inBufs->descs[1].accessMask = 0;
XDM_SETACCESSMODE_READ(inBufs->descs[1].accessMask);

/* report how we accessed the output buffer */
outBufs->descs[0].accessMask = 0;
XDM_SETACCESSMODE_WRITE(outBufs->descs[0].accessMask);

/*
 * Fill out the rest of the outArgs struct, including any extended
 * outArgs your algorithm has defined.
 */
outArgs->base.extendedError = 0;

return (IUNIVERSAL_EOK);
}

```

IMPORTANT

For the purpose of cache coherence it's important to tell the framework about how the CPU has accessed the buffers.

You must tell the application if you have read from or written to any of the buffers. This is achieved by using the `accessMask` fields associated with each and every buffer.

- `<MODULE>_<VENDOR>_control()`
 - o By default this function already supports the required command `XDM_GETVERSION`.
 - o There is an `#ifdef 0` that you can get rid of to add handling for any commands that you defined in `<module>_<vendor>.h`, (i.e. `mixer_tto.h`).

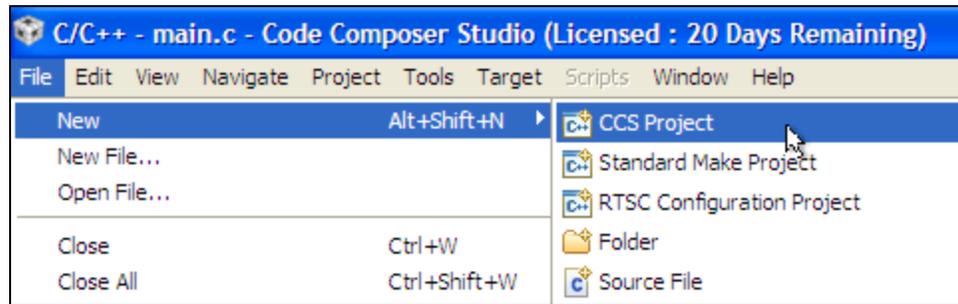
10. Build your modified algorithm to verify it's free from C language errors.

Use CCSv4 to build the algorithm. Keep debugging the algorithm until it builds correctly.

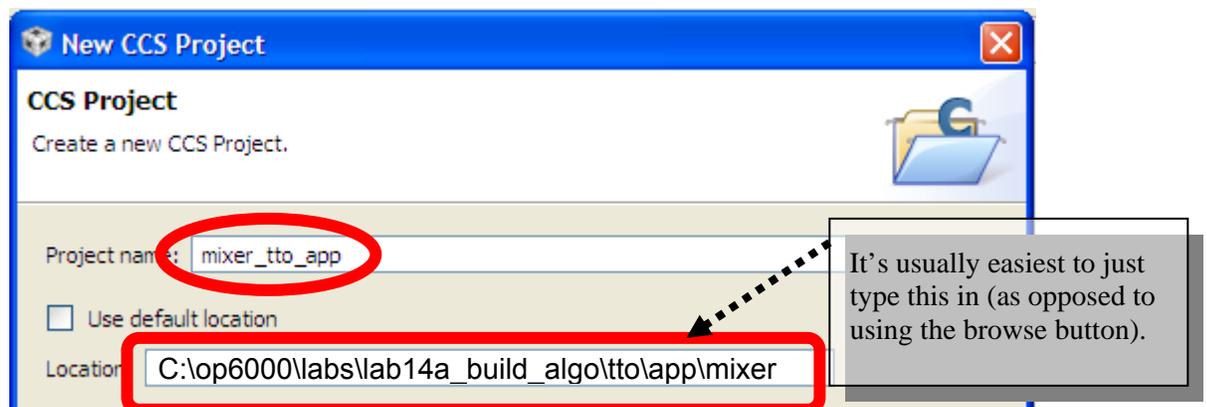
In the next part, we will test the algorithm to verify it is logically correct. For now, we just want to make sure we have not introduced any C errors.

Create a CCS Project for our Test Algorithm

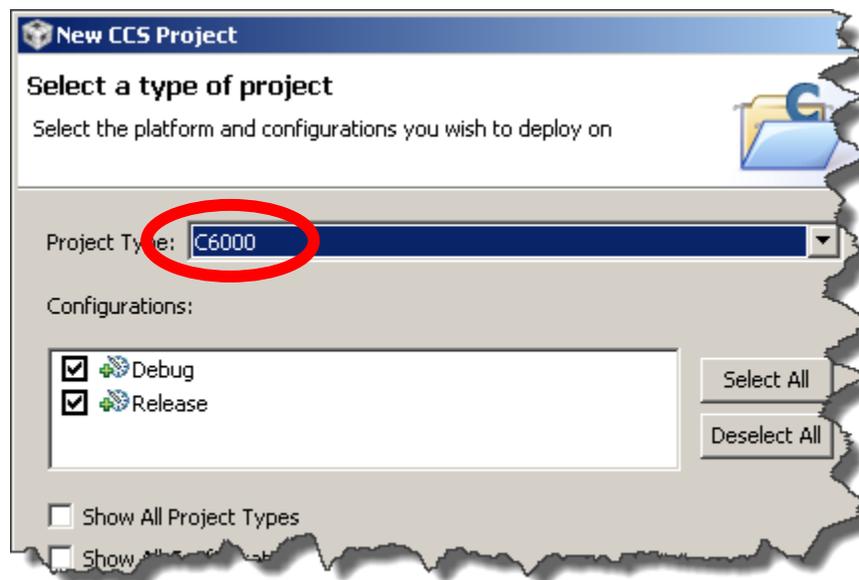
11. Create a new CCS Project.



12. Name and locate the project as shown below.

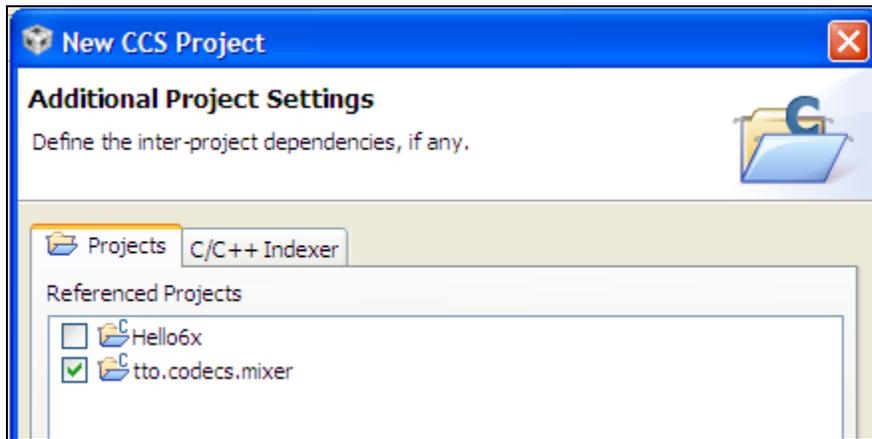


13. Choose the C6000 compiler target.

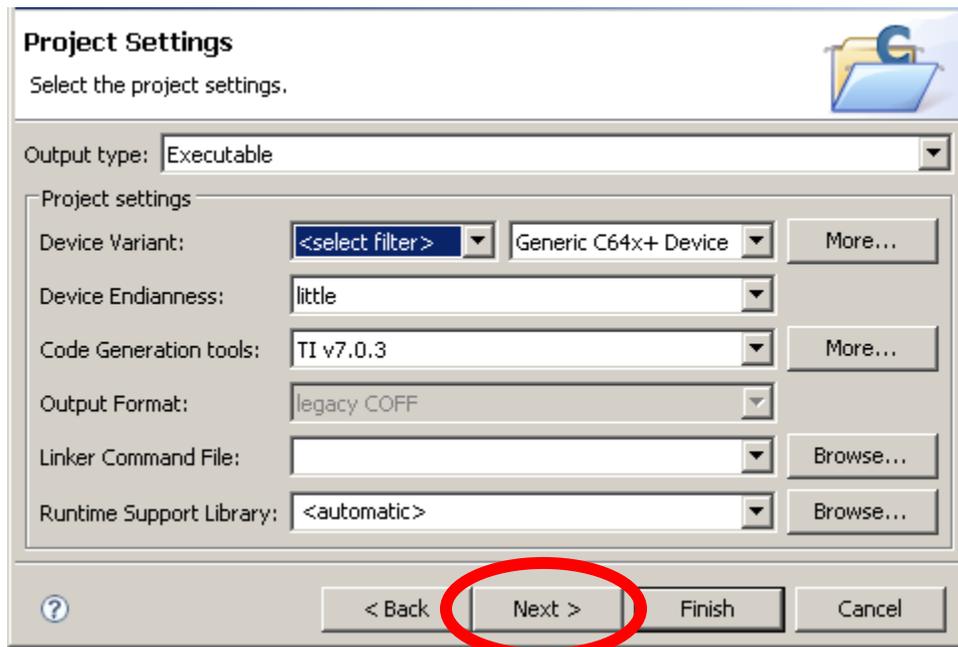


14. Select your codec project as a *dependent* project.

This makes it easy to debug code from both projects at the same time.



15. Set the project settings for running on the C64x+ DSP.



Choose:

- g. Generic C64x+ Device
- h. Little endian
- i. The default TI DSP compiler

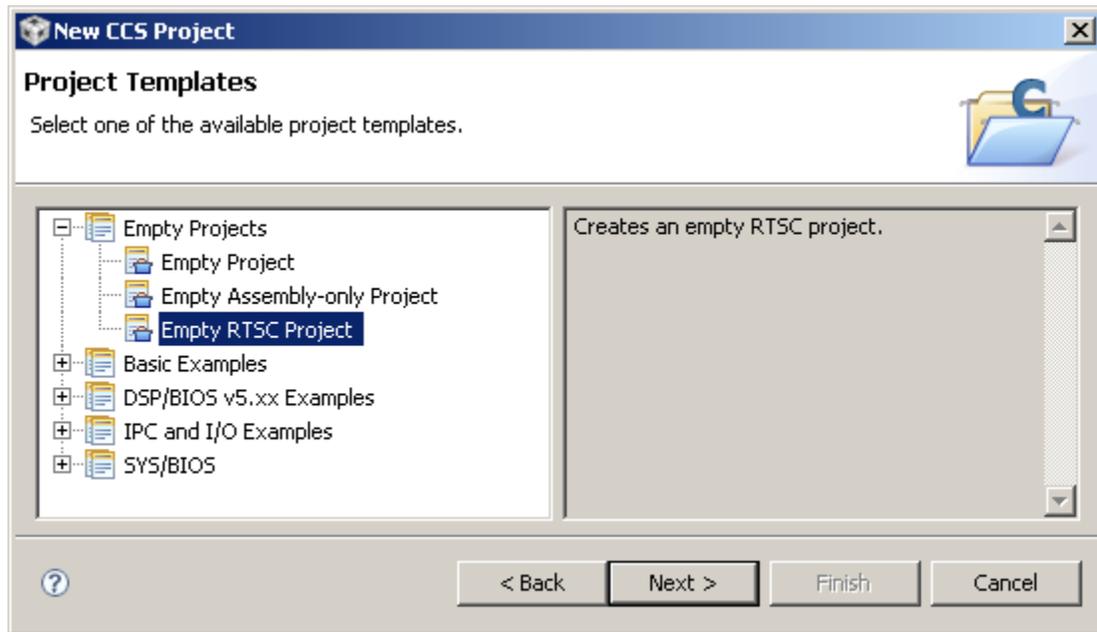
and then hit **Next**.

Avoid Common Mistake

Make sure you click NEXT!

16. Create an *Empty RTSC Project*.

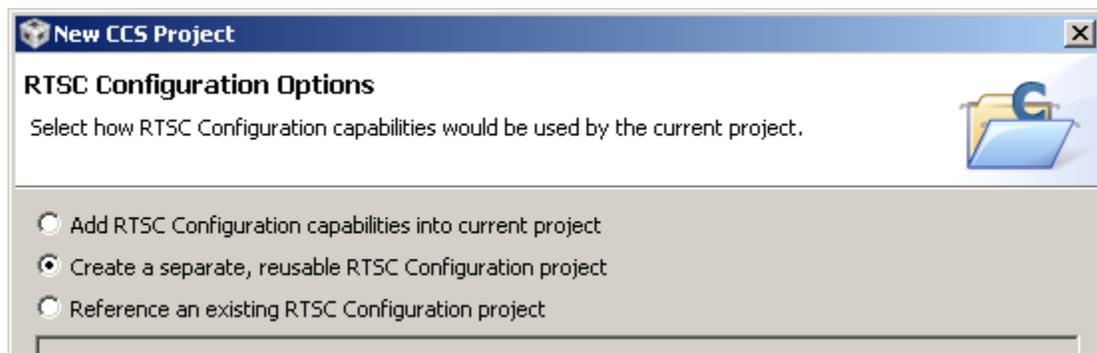
We want to create a RTSC project so we can make use of our new RTSC packaged algorithm. So, choose *Empty RTSC Project* and click *Next*.



17. Create a “separate, reusable RTSC Configuration project”.

Based on the “**Empty RTSC Project**” template from the previous dialog box, CCS will ask if you already have a RTSC config project or want to create a new one. In our case, we’ll be creating a new one.

While we could just add the RTSC config info to our current project (item 1 below), we have chosen to create a stand-alone RTSC config project.

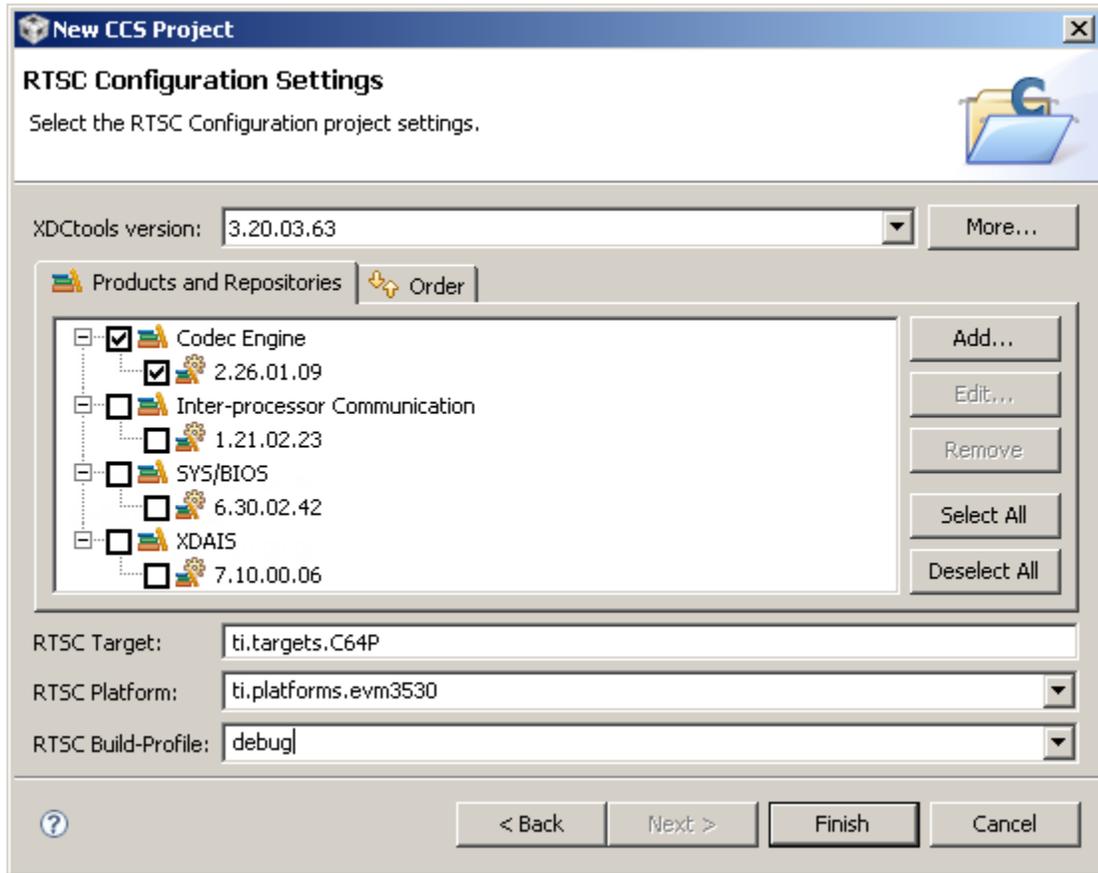


When you select this option, CCS will create a new CCS project (of type RTSC Config Project). This means you’ll actually have 3 CCS projects now:

- j. Algorithm
- k. Test application
- l. RTSC configuration

The RTSC configuration project is the way CCS can run Configuro. This is similar to adding the Configuro step to our makefile in previous labs.

18. Choose the CCSv4 installed packages that will be included with the project.



Fill-out the dialog as shown above, then click Finish.

Note: we'll add more packages in our `codec.cfg` file in an upcoming step.

19. You should now see three projects:



Setup the new RTSC Config Project

20. Add the CE and BIOS configuration files to your new RTSC config project.

The Configuro allows us to consume RTSC packages that are specified in a `.cfg` file.

In CCS, we will need to add our `.cfg` file to the RTSC configuration project. In this way, it will end up doing the same thing as our Configuro command performed in our standard make file.

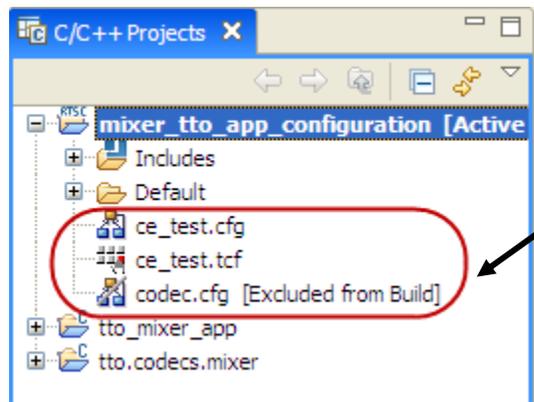
You can right-click on the `mixer_tto_app_configuration` project, select `Add File to Project...`, and add the following files from the Lab14a starter files:

```
ce_test.cfg
cd_test.tcf
codec.cfg
```

} from C:\op6000\labs\lab14a_starter_files\config

At the end of this step, your project should look something like:

Important !!!



Make sure that "codec.cfg" is the file which is excluded from the build.

You can do this by right-clicking on the individual files and selecting whether to include/exclude it from the build process.

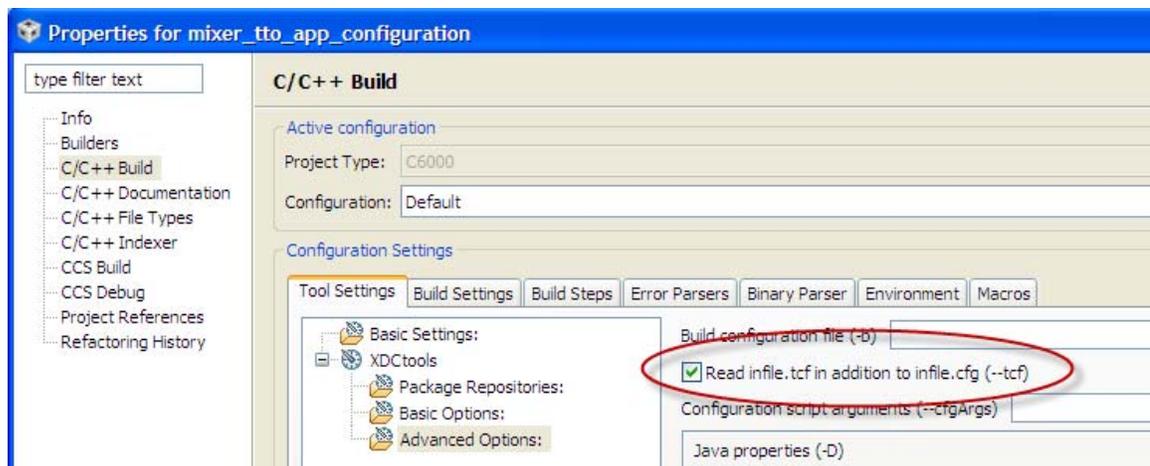
The reason we're excluding `codec.cfg` is that it's actually included (similar to `#include`) into `ce_test.cfg`. The XDC tools require the `.cfg` and `.tcf` files to be named the same.

21. Change [Excluded from Build] – as shown above.

So many folks miss this little item, we decided to call it out specifically.

22. Tell the config project to read both `.tcf` and `.cfg` files.

Right-click and open the properties of the `mixer_tto_app_configuration` project and check the box under the: `C/C++ Build | Advanced Options`:

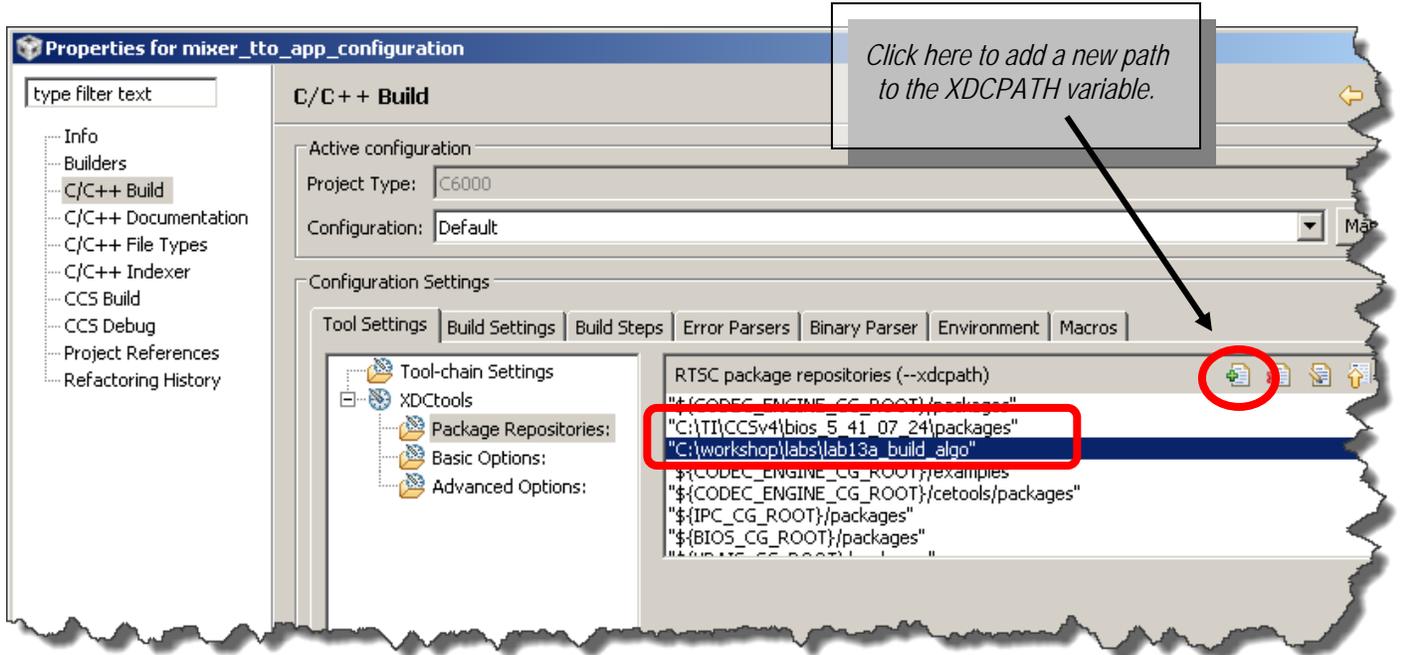


23. While you have the properties open, we need to add to the search path (i.e. XDCPATH).

Our config project needs two more paths added to its XDCPATH search list:

```
C:\TI\CCSv4\bios_5_41_07_24\packages  
C:\op6000\labs\lab14a_build_algo
```

Without adding the lab14a path, the config project wouldn't be able to find the algorithm we've just created.



Setup the Algorithm Test Project

The test project allows us to run and test our algorithm. Without this project, we wouldn't know if the algorithm really gives the correct answer. We know it builds, but we also want to know it works logically, too.

24. Add the test program – main.c.

We have written a simple test application that will use the UNIVERSAL VISA calls to test your algorithm.

Right-click *mix_tto_app* and select **Add Files to Project...**

Add the following file to your project:

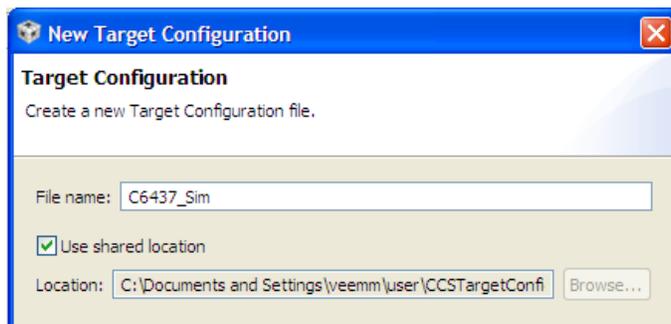
C:\op6000\labs\lab14a_starter_files

25. Create a Target Configuration File for the C64x+ Simulator.

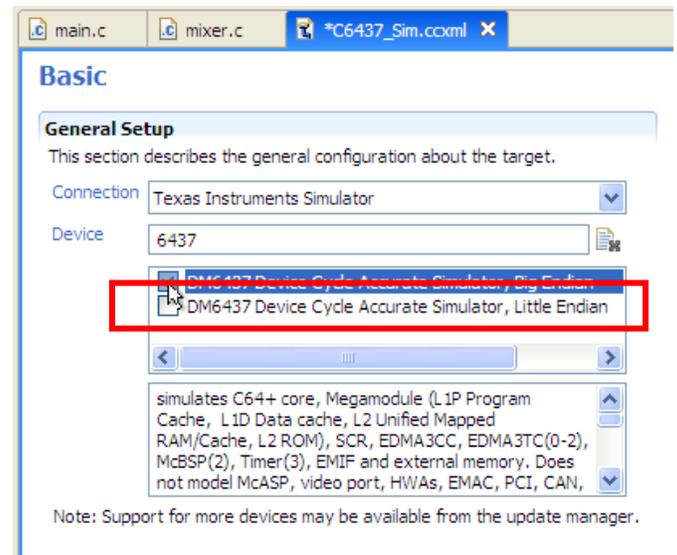
CCSv4 uses *Target Configuration Files* to specify which target you want to run (i.e. debug) your code on. (In CCSv3.3, you were required to run CCS Setup which set the target for all projects withing CCS. CCSv4 allows each project to define a different target it will run on, which is very nice.)

New | Target Configuration Files

Name the file something like C6437_Sim and let's just choose to use the shared location. (The shared location just means that every project in the workspace can use this Target Config file).



Pick the *DM6437 Device Cycle Accurate Simulator, Little Endian*.



26. Build the test application.

You shouldn't have any errors to fix ... our fingers are crossed.

Run and Debug Algorithm

27. Start a debug session.

Click the debug icon to start a debug session and download the program to the simulator. Also, CCSv4 will change to a debug perspective (i.e. window layout).



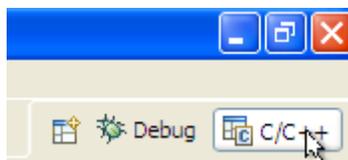
28. Set breakpoints on the 4 VISA functions in our `main()` function.

```
MIXER_create()
MIXER_process()
MIXER_control()
UNIVERSAL_delete()
```

29. Set breakpoints in the algorithm, too.

One of the big conveniences of debugging our test application – along with the algorithm library package as a dependent project – is the ability to set breakpoints right in the algo itself.

Click on the C/C++ perspective button to go back into the project/editing window layout.

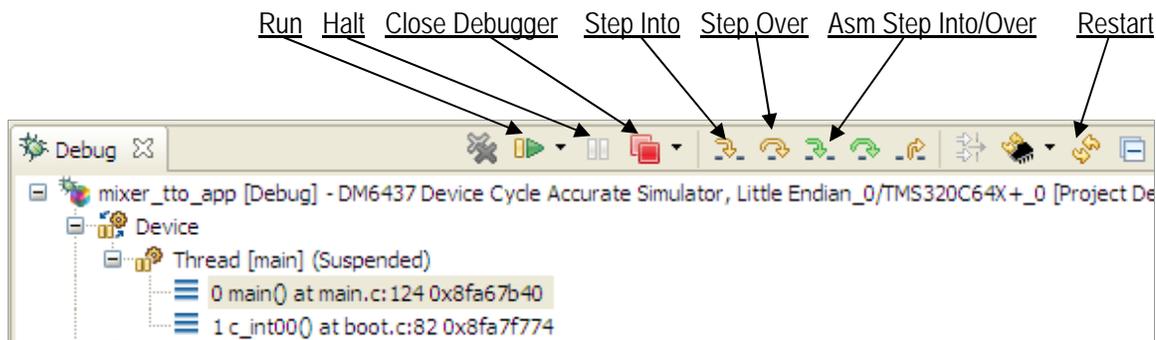


Navigate to the `mixer.c` file (in the `tto.codecs.mixer` project) and set a breakpoint inside the `MIXER_TTO_process()` function. For example, maybe set a breakpoint on the line:

```
if ((inArgs->base.size != sizeof(*inArgs)) ||
```

30. Return to the Debug perspective and run/step thru the code.

Some hints when running the debugger.



Note, you may see an error after the `MIXER_control()` call returns. At this point, the string return is not working properly. Sorry, we'll figure it out later ... or you can take on the challenge.

31. When you're finished testing the code, close the debugger.

You can also close CCSv4 now, too — if you want to.

Test Application Code

```

//=====
//
//   FILE NAME : main.c
//
//   ALGORITHM : MIXER
//
//   VENDOR    : TTO
//
//   TARGET DSP: C64x+
//
//   PURPOSE   : This file contains code to test the MIXER algorithm.
//
//   Nextxen Algorithm Wizard Version 1.00.00 Auto-Generated Component
//
//   Creation Date: Mon - 17 January 2011
//   Creation Time: 12:51 AM
//
//=====

#include <xdc/std.h>
// which includes: stdarg.h, stddef.h

#include <ti/sdo/ce/CERuntime.h>
#include <ti/sdo/ce/osal/Memory.h>           // Contiguous memory alloc functions
#include <ti/sdo/ce/universal/universal.h>
// which includes: iuniversal.h, xdm.h, ialg.h, xdas.h, Engine.h, visa.h, skel.h

#include <stdint.h>                        // used for definitions like uint32_t, ...
#include <string.h>                        // used for memset/memcpy commands

// Note1: Make sure you have added this algorithms repository as a -i compiler option;
// that is, make sure the path that contains the folder "tto" is provided as a -i include
// path. The compiler will concatenate the -i path, along with the specified in the <> to
// find the actual header file.
// Note2: The 'mixer_tto.h' header also includes: imyalg.h

#include <tto/codecs/mixer/mixer_tto.h>

//=====
//
//   Prototypes
//
//=====
int main ( void );
void setup_IMIXER_buffers ( void );
void error_check ( Uint32 event, XDAS_Int32 val );

//=====
//
//   Global/Static Variables
//
//   Note: We chose to make most of the variables and arrays 'static local'
//         to make debugging easier for our simple codec engine test example.
//         By declaring them this way, they remain in scope during the entire
//         program.
//
//=====
#define PROGRAM_NAME      "universal_test"
#define ENGINE_NAME      "myEngine"
#define ALGO_NAME         "mixer"

static String             sProgName      = PROGRAM_NAME;
static String             sEngineName    = ENGINE_NAME;
static String             sAlgoName      = ALGO_NAME;

static Engine_Handle      hEngine        = NULL;
static UNIVERSAL_Handle  hUniversal     = NULL;

```

```

static IMIXER_Params      myParams;

#define IN_BUFFER_SIZE 20
#define OUT_BUFFER_SIZE 20
#define INOUT_BUFFER_SIZE 16
#define STATUS_BUFFER_SIZE 16

static XDAS_Int8         *in0;           // [IN_BUFFER_SIZE];
static XDAS_Int8         *in1;           // [IN_BUFFER_SIZE];
static XDAS_Int8         *out0;          // [OUT_BUFFER_SIZE];
static XDAS_Int8         *status0;       // [STATUS_BUFFER_SIZE];

static XDM1_BufDesc      myInBufs;
static XDM1_BufDesc      myOutBufs;
static XDM1_BufDesc      myInOutBufs;
static IMIXER_InArgs     myInArgs;
static IMIXER_OutArgs    myOutArgs;
static IMIXER_DynamicParams myDynParams;
static IMIXER_Status     myStatus;

#define RETURN_ERROR 0
#define RETURN_SUCCESS 1

static unsigned int      funcReturn = RETURN_SUCCESS;

static XDAS_Int32        rStatus = 0;

// =====
// MIXER_ Function Macros
//
// The following three macros make it easier to call the algorithm's create, process,
// and control methods. They provide recasting of the functions and arguments from
// the MIXER algorithm, to the UNIVERSAL API. This is needed since the Codec Engine
// framework implements the common API, which provides portability and ease-of-use.
//
#define MIXER_create(hEngine, sAlgoName, Params)
    UNIVERSAL_create(hEngine, sAlgoName, (IUNIVERSAL_Params *)&Params)
#define MIXER_process(hUniversal, InBufs, OutBufs, InOutBufs, InArgs, OutArgs)
    UNIVERSAL_process(hUniversal,
        &InBufs,
        &OutBufs,
        &InOutBufs,
        (IUNIVERSAL_InArgs *)&InArgs,
        (IUNIVERSAL_OutArgs *)&OutArgs )
#define MIXER_control(hEngine, eCmdId, DynParams, Status)
    UNIVERSAL_control(hEngine,
        eCmdId,
        (UNIVERSAL_DynamicParams *)&DynParams,
        (UNIVERSAL_Status *)&Status )

//=====
//
// Functions
//
//=====

int main(void)
{
    // ==== Open the Codec Engine =====

    CERuntime_init();

    hEngine = Engine_open(sEngineName, NULL, NULL);
    error_check(TEST_ENGINE_OPEN, (XDAS_Int32) hEngine);

```

```

// ==== Create an Algo Instance =====
// Initialize the params used for the create call
myParams.base.size = sizeof( IMIXER_Params );

//The MIXER_create() function creates an instance of our algorithm; you
// can call the generic UNIVERSAL_create() function, but you would need to
// correctly cast the parameters. The iMIXER.h file defines macros which
// simplify the _create, _process, and _control function calls.
hUniversal = MIXER_create(hEngine, sAlgoName, myParams);

error_check(TEST_ALGO_CREATE, (XDAS_Int32) hUniversal);

// ==== Run the Algorithm =====
setup_IMIXER_buffers();

// Default values were applied; please change if you want to select other values.
myInArgs.base.size = sizeof(IMIXER_InArgs);
myInArgs.gain0 = 0x3fff;
myInArgs.gain1 = 0x3fff;

//IMIXER_OutArgs was not extended, so no additional values must be set in myOutArgs.
myOutArgs.base.size = sizeof(IMIXER_OutArgs);

rStatus = MIXER_process(hUniversal, myInBufs, myOutBufs, myInOutBufs, myInArgs, myOutArgs);

error_check(TEST_ALGO_PROCESS,rStatus);

// ==== Call Algo control function =====
//IMIXER_DynamicParams was not extended, so no additional values must be set
myDynParams.base.size = sizeof(IMIXER_DynamicParams);
myStatus.base.size = sizeof(IMIXER_Status);

rStatus = MIXER_control(hUniversal, XDM_GETVERSION, myDynParams, myStatus);

if (!rStatus)
{
    printf("Program '%s': Algo '%s' control call succeeded\n",sProgName, sAlgoName);

    printf("\tAlg version:  %s\n", (rStatus == UNIVERSAL_EOK ?
        ((char *)myStatus.base.data.descs[0].buf) : "[unknown]"));
}
else
{
    fprintf(stderr, "Program '%s': ERROR: Algo '%s' control call failed;
        rStatus=0x%x\n", sProgName, sAlgoName, (unsigned int) rStatus);
}

// ==== Delete and Close the Algo Instance =====
UNIVERSAL_delete(hUniversal);

Engine_close (hEngine);

// ==== Return from Main Function =====
#ifdef _DEBUG_
    printf("Program '%s': Function main() now exiting.\n", sProgName);
#endif

    //while(1);

    return(funcReturn);
}

// =====

```

```
// Buffer setup
// =====

void setup_IMIXER_buffers(void)
{

    // ==== ALLOCATE BUFFERS =====
    //
    // - Buffers are allocated with the Codec Engine's contigAlloc function
    // - On ARM, this fxn alloc's memory from the CMEM driver, which is req'd
    //   when passing data to the DSP. A contiguous allocation is made when
    //   run on the DSP, but this maps to a simple MEM allocation.
    // - This prevents a failure on architectures like OMAP3530 which provide
    //   an MMU in the DSP's memory path.

    in0    = Memory_contigAlloc( IN_BUFFER_SIZE, 8 );
    in1    = Memory_contigAlloc( IN_BUFFER_SIZE, 8 );
    out0   = Memory_contigAlloc( OUT_BUFFER_SIZE, 8 );
    status0 = Memory_contigAlloc( STATUS_BUFFER_SIZE, 8 );

    // ==== INITIALIZE BUFFERS =====
    //
    // - We chose to use a simple data set for testing our algorithm.
    // - In "real" life you would want to enhance this test program with test
    //   data that validates your algorithm.

    memset( in0,    0xA, IN_BUFFER_SIZE );
    memset( in1,    0xB, IN_BUFFER_SIZE );
    memset( out0,   0x0, OUT_BUFFER_SIZE );

    // === Setup buffer descriptors for calls used in the MIXER_process()
    //      and MIXER_control() functions

    myInBufs.numBufs          = 2;
    myInBufs.descs[0].bufSize = IN_BUFFER_SIZE;
    myInBufs.descs[0].buf     = (XDAS_Int8 *) in0;
    myInBufs.descs[1].bufSize = IN_BUFFER_SIZE;
    myInBufs.descs[1].buf     = (XDAS_Int8 *) in1;

    myOutBufs.numBufs        = 1;
    myOutBufs.descs[0].bufSize = OUT_BUFFER_SIZE;
    myOutBufs.descs[0].buf   = (XDAS_Int8 *) out0;

    myInOutBufs.numBufs     = 0;

    myStatus.base.data.numBufs = 1;
    myStatus.base.data.descs[0].bufSize = STATUS_BUFFER_SIZE;
    myStatus.base.data.descs[0].buf   = (XDAS_Int8 *) status0;

}
}
```

Lab 14b - Creating a "server" for your algorithm

Now that you've proven your algorithm works on a single CPU DSP system, we can wrap our algorithm into a server and call it from Linux on the ARM.

Copy the necessary files into the VM shared folder

1. Create the folder `lab14b_run_algo` in `vm_images\shared`.

— Create the following directory.

```
C:\vm_images\shared\lab14b_run_algo
```

We can't implement these steps in the Optimization Workshop, since they depend upon us having an ARM+DSP processor with a Linux Build environment.

If you're interested in trying this out, we suggest you go access the Linux VMware image, lab files, and instructions from the following workshop:

OMAP™/DaVinci™/Sitara™/Integra™ System Integration using Linux Workshop

http://processors.wiki.ti.com/index.php/OMAP%E2%84%A2/DaVinci%E2%84%A2_System_Integration_using_Linux_Workshop

3. Create the server folder.

```
C:\vm_images\shared\lab14b_run_algo\server
```

4. Copy the algorithm into the `lab14b_run_algo` folder.

— Copy the whole algorithm (`tto.codecs.mixer`) directory to the `lab14b_run_algo` folder.

```
C:\vm_images\shared\lab14b_run_algo\tto\codecs\mixer
```

File management in Linux

5. Copy the entire `lab14b_run_algo` folder into `~/labs`.

— Open up your VMware Ubuntu Linux and copy the entire `lab14b_run_algo` folder from the VMware shared folder into your `/home/user/labs` directory.

6. Copy the makefile from `lab14server` and edit `XDCPATH`.

Copy the makefile from your `lab14server` folder into the `lab14b_build_algo_server` folder

— Verify that the your current lab path is included in the `XDCPATH` correctly.

7. Copy the makefiles from `lab14a_build_server/app` to `lab14b_build_algo/app` and edit them.

— After copying the two makefiles: (1) verify that your new codec package is on the `XDCPATH`; and, (2) remove the OSD file from the `INSTALL_OSD_IMAGE` variable.

```
INSTALL_OSD_IMAGE := ../osdfiles/ti_rgb24_640x80.bmp
```

DM6446 Labs:

You need to do the same edits as in steps 6-8, but yours will look a little different than what is shown here.

Also your files would be in the "workshops" folder, not the "labs" folder.

~~8. Copy the `app_cfg.cfg` from `lab14a_build_server/app` to `Lab14b_run_algo/app`.~~

~~— Again, we need to only make one small edit. Change the engine name from “`encodedecode`” to “`myEngine`” which is the name used in our `main.c` file.~~

~~Make the DSP Server~~

~~9. Switch to the `Lab14b_run_algo/server` folder and run the DSP Server using the wizard.~~

~~— Follow the steps from `lab12a_build_server` to create a new server using the “mixer” codec.~~

~~— Use the same platform and server package name as we’ve done in the past, but remember to use the repo location: `/home/user/labs/lab14a`~~

~~10. Build the server.~~

~~Build and Test the app and algo~~

~~11. Switch to the `lab14b_build_algo/app` directory and build/install the test application.~~

```
make debug install
```

~~12. Switch over to Tera Term and run the application.~~

~~— If you’ve reset the EVM, make sure you run `loadmodules.sh` before running the application. You should see a `printf` statement output as the program completes each VISA call. (You can review the `main.c` code to find these `printf`’s in the test app—or add more if you’d like to trace things more closely..)~~

Lab 14 – Creating an Algorithm (xDAIS)

Building a (xDAIS compliant) software component

In this lab exercise, we build a C6000 library which contains a xDAIS algorithm. This is accomplished using the xDAIS Component Wizard, along with the CCS library-type project. Additionally, a test project is created to help validate our new xDAIS algorithm.

This lab consists of 5 basic parts:

- Using the Wizard to create the algorithm interface and files
- Building the algorithm’s library as a CCS project
- Building a test application for our library
- Adding the code from the provided sinewave algo to our library project
- Testing the final, working library

Given:

A sinewave algorithm is provided with this lab exercise. This code will be used so that we don’t have to write the sinewave routine from scratch. (If you have previously attended the C6000 Integration Workshop, you may recognize this sine algorithm which was used in those lab exercises.) The two files that are provided are:

- sine.h
- sine.c

To make the lab easier, you may want to refer to these two files which are printed in the appendix.

Lab 14 – Creating an Algorithm (xDAIS).....	14-1
<i>Initial CCS Setup.....</i>	<i>14-2</i>
<i>TI xDAIS Component Wizard.....</i>	<i>14-3</i>
<i>Create Sine Library.....</i>	<i>14-13</i>
Setup Library for Publishing.....	14-16
<i>Build test application for new software component.....</i>	<i>14-18</i>
Create Sinewave Test Project	14-18
Setup the Heap for Dynamic Memory	14-20
Add/Edit Files to the SineTest Project.....	14-21
Build and Fix the Test Program	14-23
<i>Add “algorithm code” to our software component.....</i>	<i>14-25</i>
Completing the Sine Algorithm Functions	14-25
Complete the TIO_SINE_process() Function.....	14-26
Complete the Sine Initialization Function.....	14-27
Add final definitions, includes, and prototypes	14-29
<i>Test completed algorithm component</i>	<i>14-30</i>
<i>Lab 14 – Appendix</i>	<i>14-32</i>
sine.h.....	14-32
sine.c	14-33

Initial CCS Setup

Before beginning the lab exercise we need to configure CCS for the correct simulator. For this exercise we will be using one of the CPU Cycle Accurate Simulators. Using the Setup CCS icon select one of the following simulator configurations:

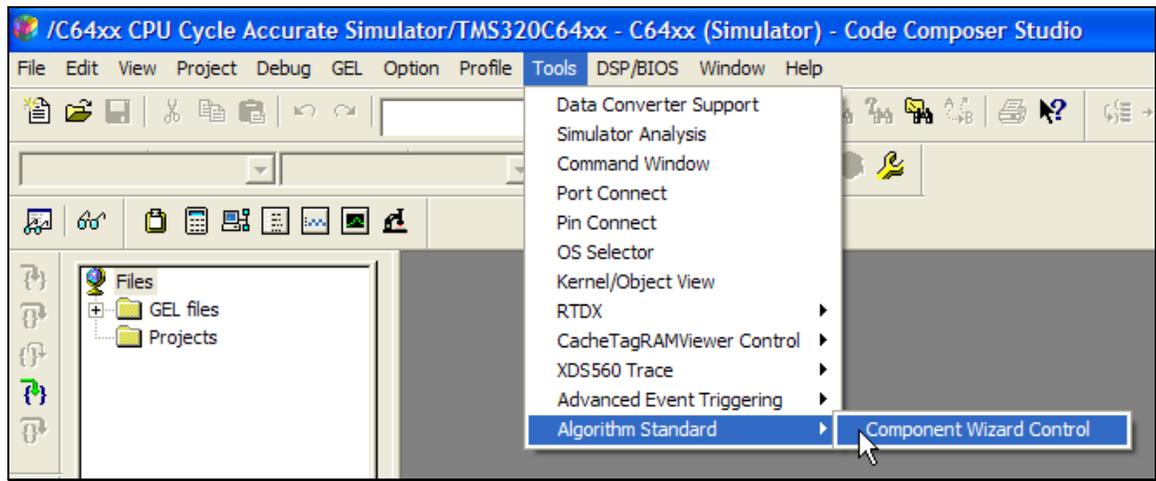
- C64xx CPU Cycle Accurate Simulator, Little Endian
- C64+ CPU Cycle Accurate Simulator, Little Endian
- C67xx CPU Cycle Accurate Simulator, Little Endian
- C672x CPU Simulator, Little Endian

The illustrations which accompany the instructions for the lab exercise are based on the C64xx CPU Cycle Accurate Simulator. Your view may be slightly different depending on your choice of processor but the variations will be minor.

Note: This is a fairly long lab exercise and, as you can expect, each of the basic parts build on the previous one. It is particularly important to not make early errors that will propagate through the rest of the lab exercise. Check each step carefully to avoid problems at the final testing phase. Be very careful when specifying path names as this lab exercise makes use of multiple and dependent projects and it is relatively easy to generate file path errors.

TI xDAIS Component Wizard

1. Start the Component Wizard

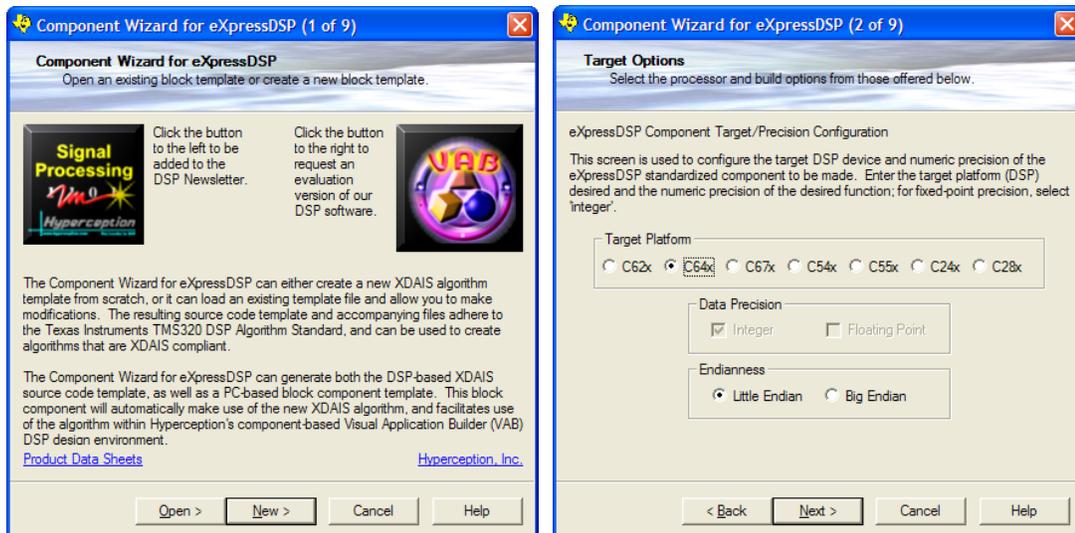


If the Component Wizard is not on the tools menu, then it probably hasn't been installed yet. Please ask your instructor for help with installing it.

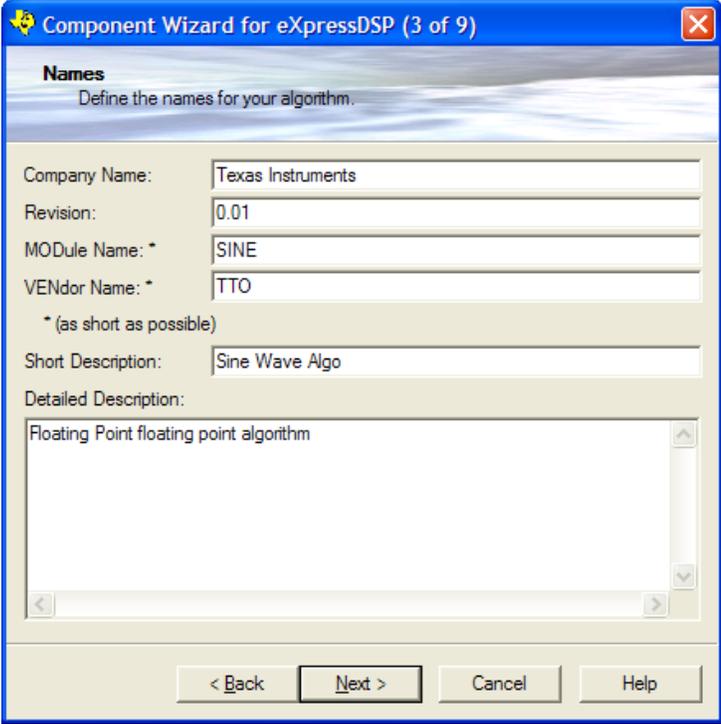
2. Component Wizard Steps 1 & 2.

Step 1: Is just an informational step. This component wizard tool was originally designed by Hyperception, a TI 3rd party (who was later purchased by National Instruments). They created the tool to build DSP blocks for their VAB development tool as well as xDAIS components for TI.

Step 2: Asks you to select constraints for your environment. The following items do not alter the xDAIS coding, but again were more important for the VAB tool. In any case, it is good to answer the three simple questions. (Note, for C64x+ systems, we recommend you choose C64x as the Target.)



3. Wizard Step 3 - Vendor/Algorithm information.



Component Wizard for eXpressDSP (3 of 9)

Names
Define the names for your algorithm.

Company Name: Texas Instruments

Revision: 0.01

MODule Name: * SINE

VENdor Name: * TTO
* (as short as possible)

Short Description: Sine Wave Algo

Detailed Description:
Floating Point floating point algorithm

< Back Next > Cancel Help

To prevent in function and variable name collisions, the xDAIS algorithm standard requires specific naming conventions. Step 3 of the wizard asks for your algorithm and company names so that it can build a component that meets the xDAIS specifications.

We recommend that you enter a company name of your choice; but more importantly, create a short Vendor name that represents your company – or team. (We chose TTO, which represents TI’s Technical Training Organization.)

above, enter the following data:

Company Name

Revision

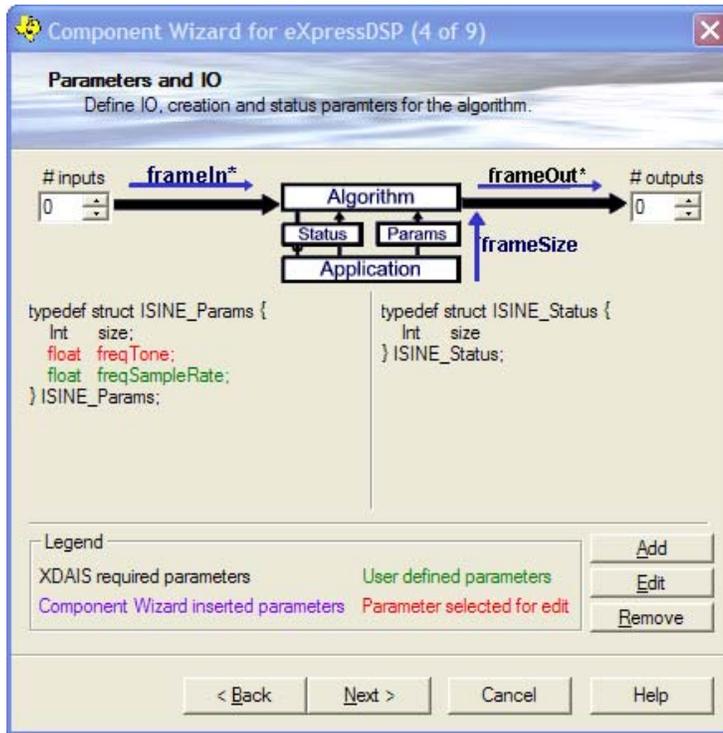
Module Name *(we recommend you use “SINE”)*

Vendor Name

Descriptions

As mentioned in the paragraph

4. Wizard Step 4 – Creation/Status parameters.



_Params

xDAIS algorithms can specify creation (i.e. constructor) parameters. These are defined in a structure called:

```
I<AlgoName>_Params
```

Thus, for our SINE algorithm the structure is:

```
ISINE_Params
```

The component tool allows you to add whatever parameters you deem necessary for your algorithm. As you add them, it graphically builds the structure for you ... and sets the required *size* parameter for you.

_Status

While only briefly mentioned in our xDAIS discussion, you may remember that a status structure has also been defined that lets the algorithm designer to define a set of runtime status parameters. These can be accessed by the application at runtime.

Dialog-box Diagram

The diagram in this dialog box correctly shows that Params are a read-only path from the application to the algorithm; also, it shows that Status can be read and/or write properties.

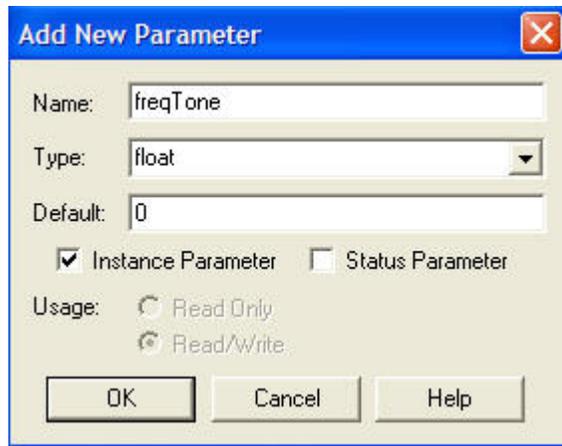
While the frameIn*/frameOut* items in the diagram are not wrong, we suggest you set them to 0 (as shown above). This mechanism was convenient for VAB users as a way to pass input and output data to/from their algorithms. Most users today, though, prefer to pass input/output data via their processing function call. You can see this reflected in how the xDM functions were defined – which we modeled our lab examples after. (You will see this in a later lab step.)

Setup our _Params structure

As shown in the graphic above, add the two parameters to our _Params structure which will define the sinewave tone our algorithm will create:

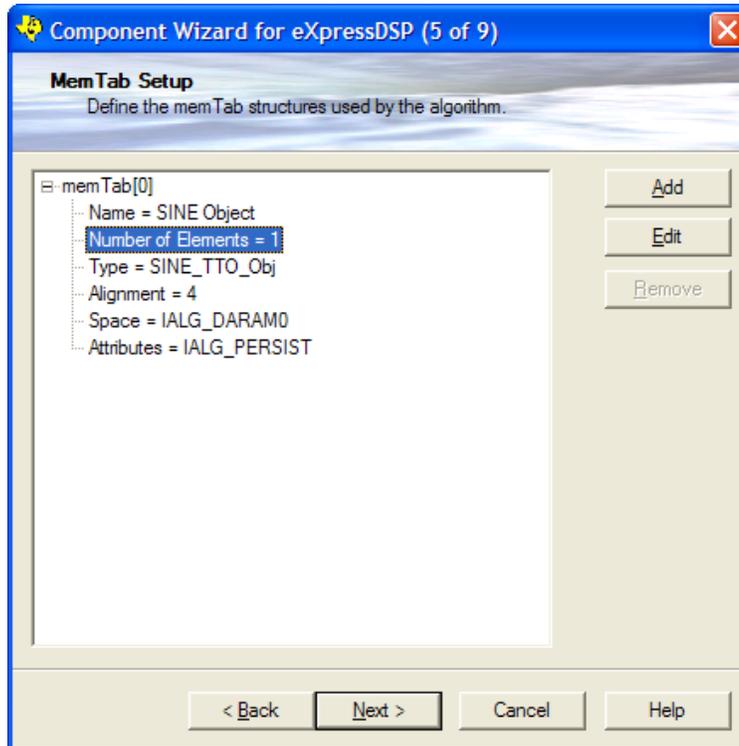
```
float freqTone
float freqSampleRate
```

When you select the Add button you see a dialog like this:



When filling out this panel and the one for `freqSampleRate` be sure that the *Status Parameter* box is not checked. We will not be using this feature in the lab exercise.

5. Wizard Step 5 – Define your algorithm’s memory needs



As discussed earlier, xDAIS requires all memory allocations be left to the application. This gives the end-user of the algorithm full control of their memory-map.

The algorithms memory requirements are provided to the application when it calls the algorithms `_alloc()` function.

The `_alloc()` function does this by filling-in a memory table structure (i.e. `memTab`).

By entering the memory needs of our algorithm into the wizard (Step 5), we can avoid work and let the wizard write the `_alloc()`

function for us.

One “problem” with choosing SINE as our example algorithm, is that it’s almost too easy. Besides the single buffer that will be passed in during the `_process()` function call, this algorithm does not require any extra blocks of memory to perform its functions.

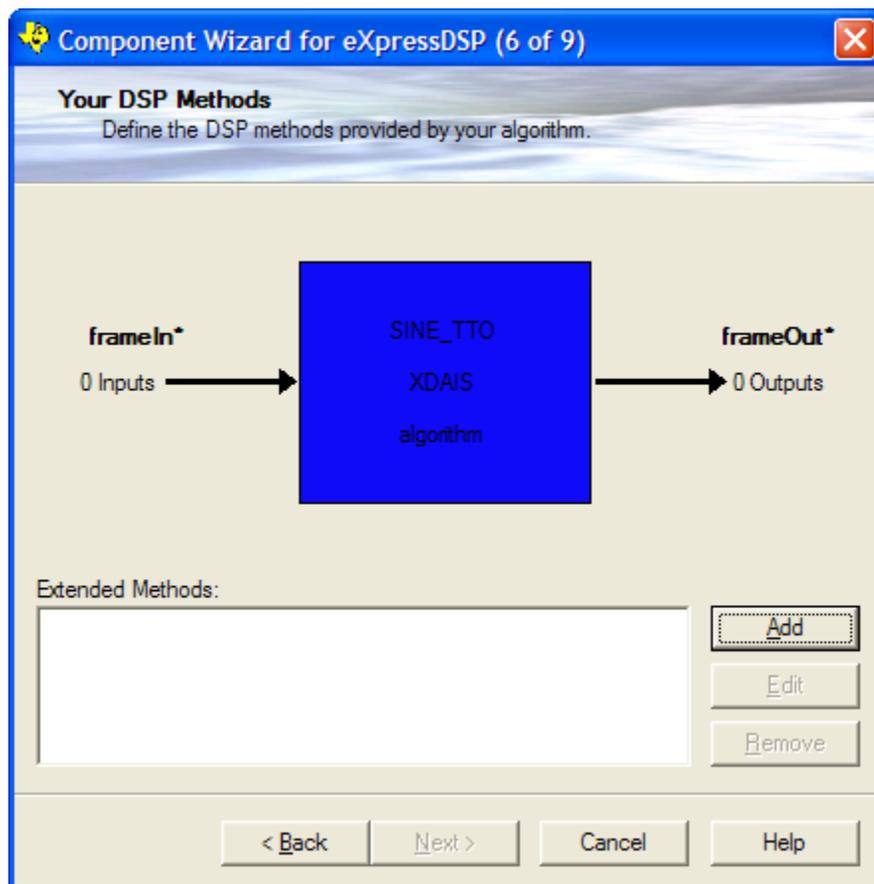
That is, this algo only needs the one required block of memory to hold the Instance Object. (Using our C++ class analogy, you might think of this as the class object, where all the class properties are stored.)

You may notice that we only changed one item versus the defaults provided by the wizard: we changed the alignment from 1 to 4. Actually, this isn’t required, it was just something we changed to have something to edit.

If your algorithm required more memory, such as in the FIR example used during the lecture, you could click the add button to define additional blocks of memory.

6. Wizard Step 6 - Define the algorithm's processing (i.e. DSP) functions

Next, we are asked to define the *Extended Methods*; in other words, our DSP functions.



We want to add our “block fill” function to the algorithm. To add the function, click the **Add** button, and follow the dialogs which ask for the function name, prototype, and parameters:

```
XDAS_Void SINE_TTO_process(ISINE_Handle handle, XDAS_Int16 * buf, XDAS_Int32 len)
```

The next page demonstrates the various dialogs used to configure these items.

Add the process function with its three parameters. Note, the first parameter (handle) is already entered for you automatically. Be careful to select the correct parameter types.

Function Prototype Wizard

Function Declaration

Function Name:

Return Type:

Function Parameters:

XDAS_Void SINE_TTO_process(ISINE_Handle handle, XDAS_Int32 * param, XDAS_Int32 * param);

< Back Next > Cancel Help

Function Prototype Wizard

Parameter Definition for function SINE_TTO_process

Parameter Name:

Parameter Type:

Parameter Number: 1 of 2

XDAS_Void SINE_TTO_process(ISINE_Handle handle, XDAS_Int16 * buf, XDAS_Int32 * param);

< Back Next > Cancel Help

Function Prototype Wizard

Parameter Definition for function SINE_TTO_process

Parameter Name:

Parameter Type:

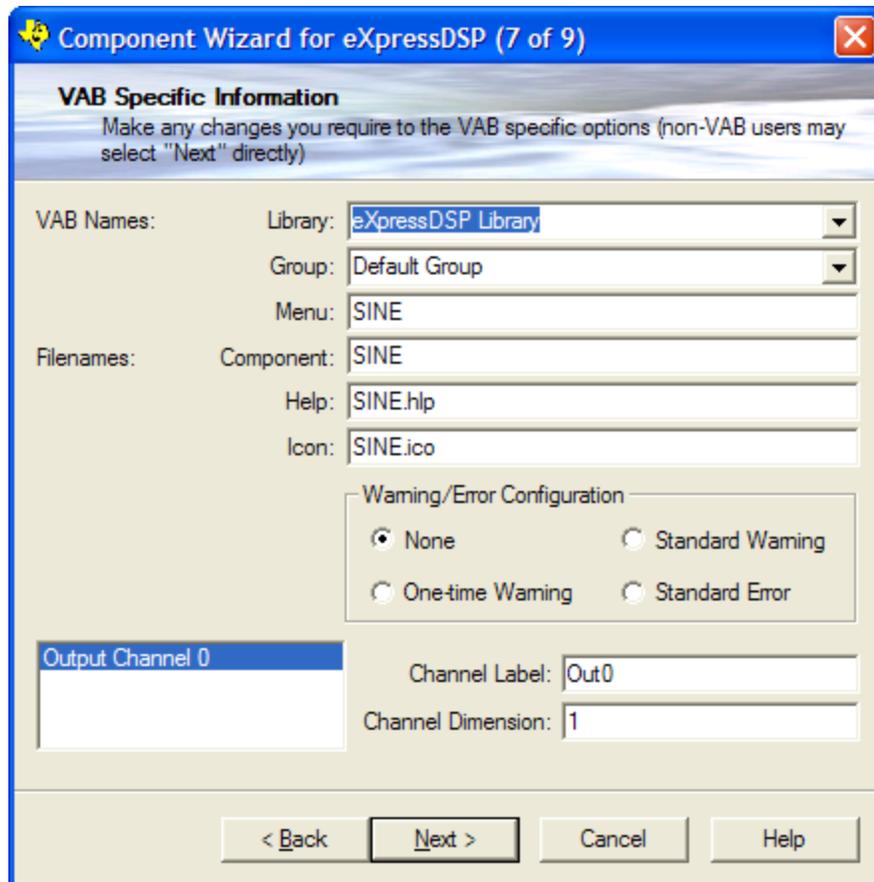
Parameter Number: 2 of 2

XDAS_Void SINE_TTO_process(ISINE_Handle handle, XDAS_Int16 * buf, XDAS_Int32 len);

< Back Finish Cancel Help

7. Skip step 7 of the Wizard.

This step was used to define the PC block used in Hyperceptions Visual Application Builder environment. This is not needed when creating a xDAIS algorithm.



The screenshot shows the "Component Wizard for eXpressDSP (7 of 9)" dialog box. The title bar includes a yellow lightning bolt icon and a close button. The main title is "VAB Specific Information" with a subtitle: "Make any changes you require to the VAB specific options (non-VAB users may select 'Next' directly)".

The dialog is divided into several sections:

- VAB Names:** Library: eXpressDSP Library (dropdown), Group: Default Group (dropdown), Menu: SINE (text field).
- Filename:** Component: SINE (text field), Help: SINE.hlp (text field), Icon: SINE.ico (text field).
- Warning/Error Configuration:** A group box containing four radio buttons: None, Standard Warning, One-time Warning, and Standard Error.
- Output Channel 0:** A list box containing "Output Channel 0". To its right are text fields for Channel Label: Out0 and Channel Dimension: 1.

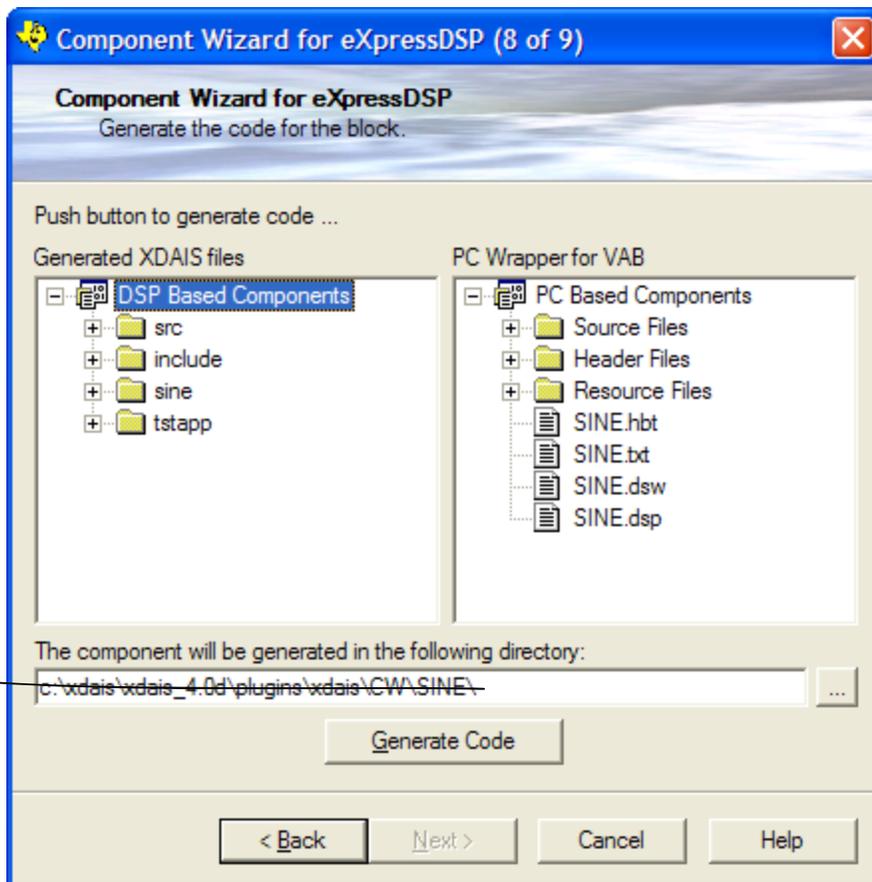
At the bottom, there are four buttons: "< Back", "Next >", "Cancel", and "Help".

8. Wizard step 8 provides you with a “Generate Code” button. Let’s use it!

This wizard dialog shows the files that will be built. While it will generate files for both xDAIS and VAB, we’ll only end up using the xDAIS files as we continue through this exercise.

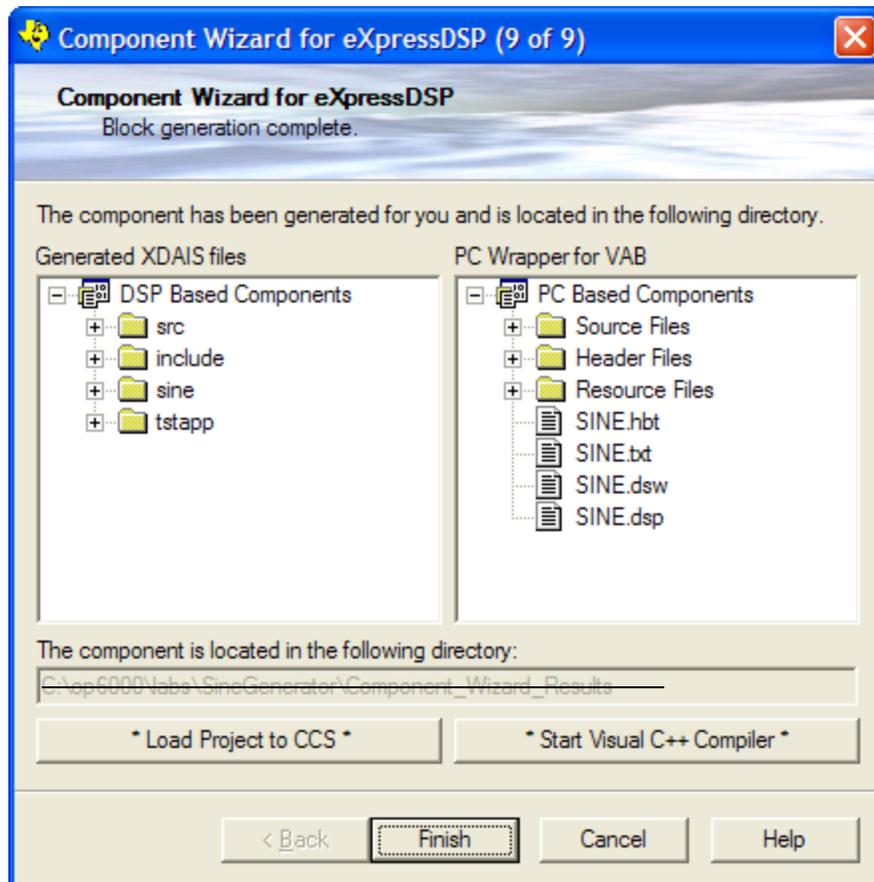
Before we click the Generate Code button, first we need to change the location where the files will be written to. Change the appropriate textbox to:

C:\op6000\labs\lab14\Sine_WizardFiles



9. Skip Wizard step 9 to “load” project to CCS.

We will open up CCS and create a new project later in this lab. For now, just go ahead and click **Finish**. Just click **OK** when the Reminder panel pops up.



10. Zip up output of component wizard (to refer back to original if needed).

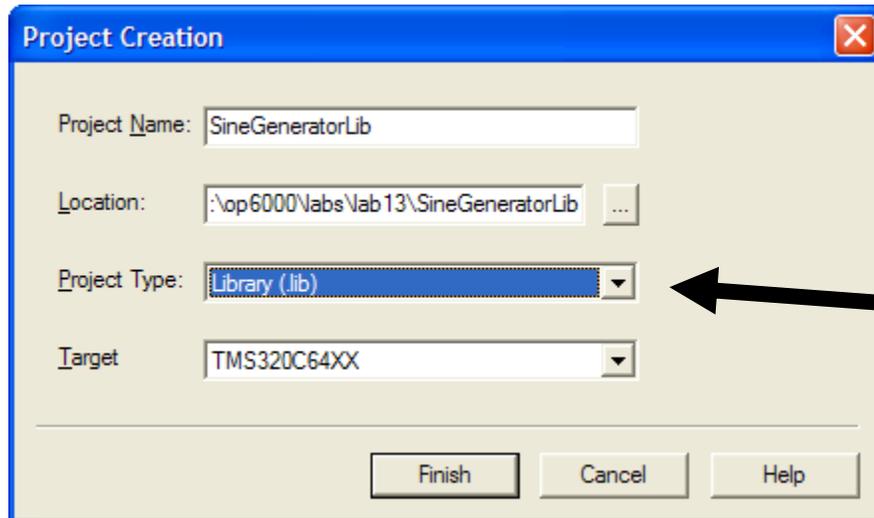
In Windows Explorer, find the folder created by the Component Wizard and use WinZip to create a ZIP archive of the folder and all its subdirectories. This will provide us with a backup copy of the Wizard’s output, should we accidentally delete or corrupt a file as we use them in the following steps.

Create Sine Library

11. Create a “Library” CCS project: SineGeneratorLib.PJT

Create a new CCS project in the the following folder:

```
C:\op6000\labs\lab14\SineGeneratorLib
```



12. Move \include & \source folders from the wizard’s output folder into our library’s project folder.

Using Windows Explorer, move from:

```
C:\op6000\labs\lab14\Sine_WizardFiles\include
```

to

```
C:\op6000\labs\lab14\SineGeneratorLib\include
```

and from:

```
C:\op6000\labs\lab14\Sine_WizardFiles\src
```

to

```
C:\op6000\labs\lab14\SineGeneratorLib\src
```

13. Add both sine_tto_ixxx.c files from “source” folder to the SineGeneratorLib project.

These files describe our algorithms interface, hence they have “ialg” in their filenames.

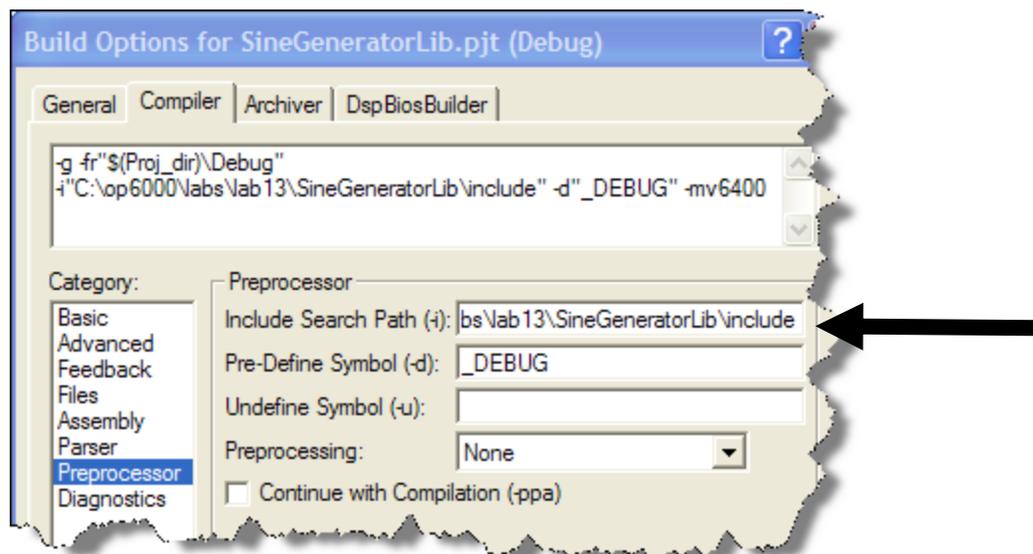
These interface files were autogenerated by the algorithm description we provided when following the component wizard steps.

```
C:\op6000\labs\lab14\SineGeneratorLib\src\sine_tto_ialg.c
C:\op6000\labs\lab14\SineGeneratorLib\src\sine_tto_ialgvt.c
```

14. Add a reference to the “include” folder to project (-i compiler option).

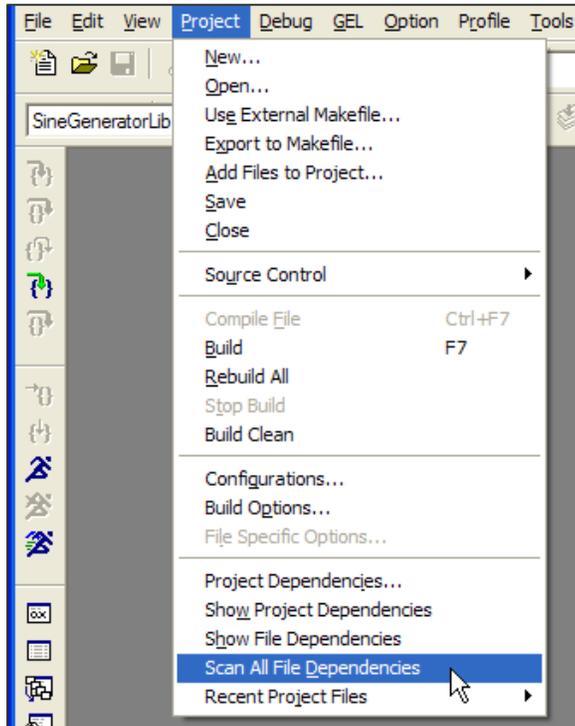
This will allow the compiler to find the header files referenced by the source files we just added to our project. To do this, open the build options dialog and add the following to the “Include Search Path” entry for the Compiler’s Preprocessor.

```
C:\op6000\labs\lab14\SineGeneratorLib\include
```

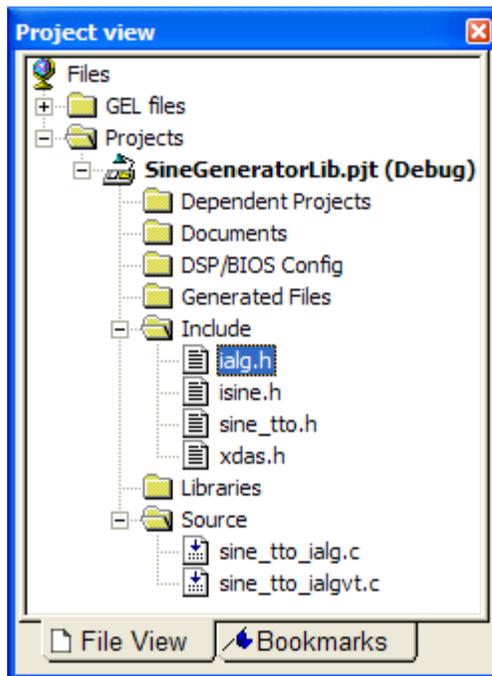


Close the compiler build option dialog once the reference has been added.

15. Scan for project dependencies.



This step is not required since it is done automatically whenever you hit the build button, but it is interesting to look at the results of the scan.



Once the scan has been completed, the Project View should now appear like this. Here we see that the scan not only picked up the header files created by the wizard (*isine.h*, *sine_tto.h*), but also two xDAIS header files provided by TI (*ialg.h*, *xdas.h*) which were referenced by our sine header files. The TI provided header files provide reference information related to the xDAIS standard.

If you right-click on *ialg.h* and select its “properties”, you will see that it was actually found within the CCS studio installation directory.

16. Build your library project.

Building your project should generate about five warnings, but no errors. The warnings should tell you that an item was created but never used. This is OK because we still need to add the actual sinewave functionality to our xDAIS algorithm; we will do this later in the lab exercise after testing the library interface we just created (starting on page 14-25).

Setup Library for Publishing

17. Create a “Files_For_Release” folder.

When handing out your library, you need more than just the .lib files, therefore, let’s setup the appropriate folders and files. Create the following three folders:

```
C:\op6000\labs\lab14\SineGeneratorLib\Files_For_Release
C:\op6000\labs\lab14\SineGeneratorLib\Files_For_Release\lib
C:\op6000\labs\lab14\SineGeneratorLib\Files_For_Release\user_src
```

If you wanted to ship source code along with your library you might also want to include a \src directory.

We chose to create a directory called \user_src to distinguish it from the actual library source code; \user_src will contain .c files that describe our algorithms interface and which are required to be delivered with a xDAIS algorithm – whether or not you want to publish the source code of the library itself.

18. Add header files to “Files for Release” folder and correct the search reference to them.

Move the include folder from our project into the Files_For_Release folder. The end-user of our library will also need these files.

```
Move: C:\op6000\labs\lab13\SineGeneratorLib\include
To:    C:\op6000\labs\lab13\SineGeneratorLib\Files_For_Release\include
```

Make sure you also change the “**Include Search Path**” in the project to reflect the new directory structure. (Hint, if you don’t remember where to make this change, look back at step 14.)

Before going onto the next step, try re-building to make sure we haven’t accidentally introduced an error.

19. Move isine.c from “src” to “Files_For_Release\user_src”.

You must provide default values for any creation parameters when publishing algorithms. The wizard created them for us in the file isine.c. We just need to move this file to our release folder.

```
Move: C:\op6000\labs\lab13\SineGeneratorLib\src\isine.c
To:
C:\op6000\labs\lab13\SineGeneratorLib\Files_For_Release\user_src\isine.c
```

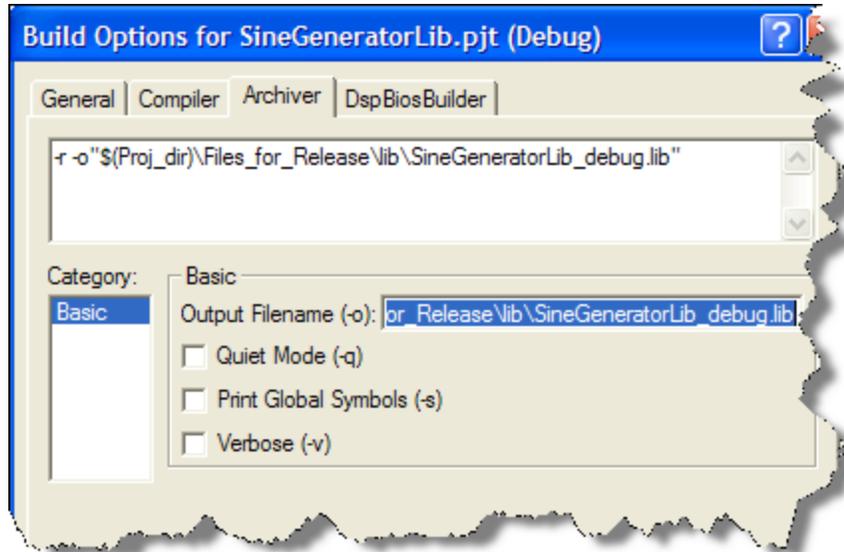
The isine.c file contains initialization parameters for our sine generator. We need to make a couple of changes so that the file contains the correct values. Locate the SINE_PARAMS section at the bottom of the file and use the CCS editor to make the following changes to the structure:

```
ISINE_Params ISINE_PARAMS = {
    sizeof(ISINE_Params),
    200, /* freqTone */
    48000, /* freqSampleRate */
};
```

20. Redirect your output libraries to the \lib folder.

Open the build options and change the output locations (and filenames) to the \lib folder.

```
$(Proj_dir)\Files_For_Release\lib\SineGeneratorLib_debug.lib
```



From this point forward, whenever you build the library, the resulting .lib files will be put into the \Files_For_Release\lib directory.

21. Perform another build, just to make sure we haven't introduced any errors in our pjt.

You might also want to double-check if the lib file was created in the \lib folder. Also, you probably received another five warnings or so but we will fix them later. The important thing now is that the build needs to complete without any errors.

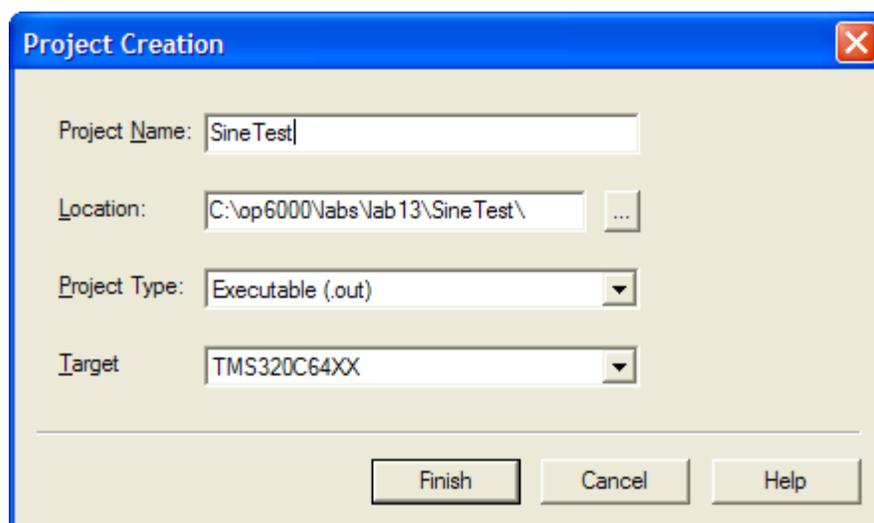
Build test application for new software component

Create Sinewave Test Project

22. Create “Executable” CCS project: SineTest.pjt

Problem with building only a library is that there isn’t a way to test its functionality. To this end, we are going to create a second project which will contain the code to run and test our library. (Note, you should leave the SineGeneratorLib project open, we are just going to create a second project.)

Create the project: `C:\op6000\labs\lab14\sinetest\sinetest.pjt`



After clicking finish, notice that your new project is listed in bold in the Project View window, which indicates it is now the active project. (You can switch which project is active by right-clicking on the project and setting it active.)

23. Add SineGeneratorLib project as a “dependent” project to SineTest.pjt.

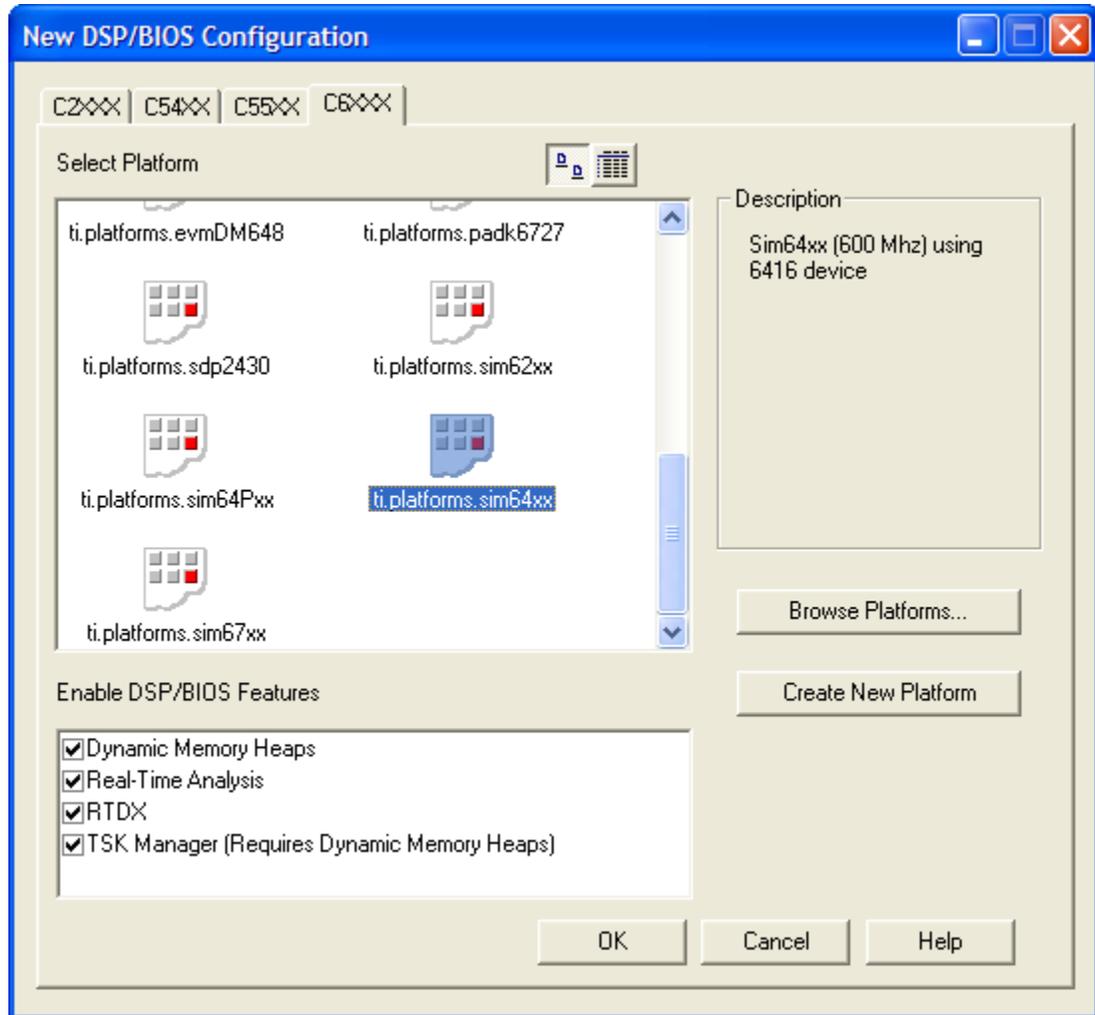
Since our SineTest project is intended to test the SineGeneratorLib library, we would usually need to build the library, then add it to our SineTest project. Making sure the library is up-to-date and then re-building the test can be tedious. A more convenient feature is available in CCS Studio; we can just make the SineTest project dependent upon our library project and both will be built everytime we build SineTest.

You can add dependent project by using right-click and browsing for the project, though it’s much easier to:

Drag SineGeneratorLib onto the SineTest’s Dependent Projects folder

24. Create a BIOS .tcf file and add it (and its .cmd) file to your project.

Create a new DSP/BIOS Configuration using the appropriate ti.platform file (our example uses the ti.platforms.sim64xx file). As a reminder, this is done with **File** → **New** → **DSP/BIOS Configuration**.

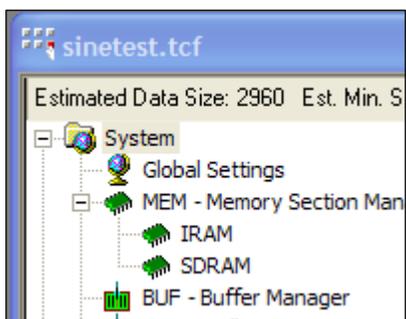


You should use one of the following ti.platform files depending on which simulator you decided to use:

- ti.platforms.sim64xx, for the C64xx CPU Cycle Accurate Simulator
- ti.platforms.sim64Pxx, for the C64+ CPU Cycle Accurate Simulator
- ti.platforms.sim67xx, for the C67xx CPU Cycle Accurate Simulator
- ti.platforms.padk6727, for the C672x CPU Simulator

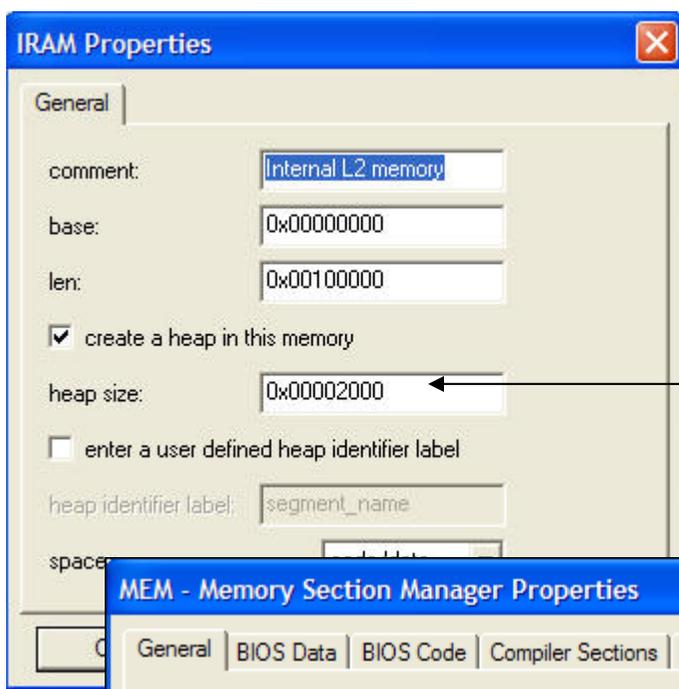
Setup the Heap for Dynamic Memory

25. Create a heap and add malloc support in your .tcf configuration.

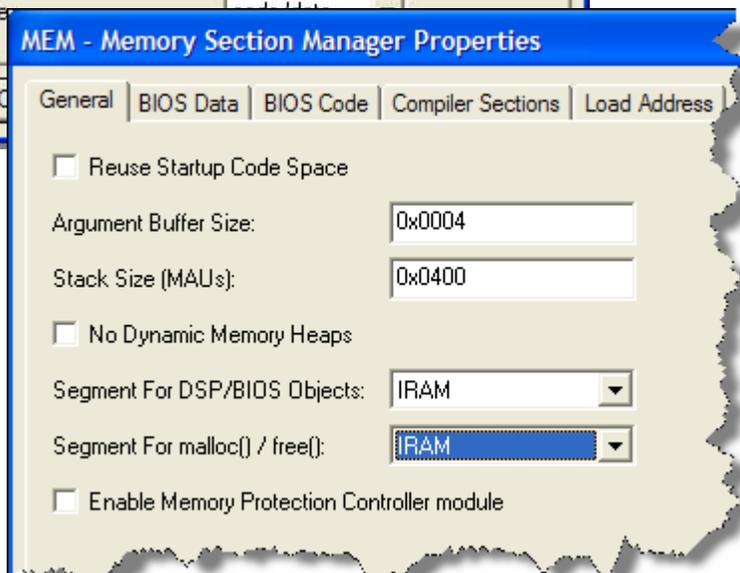


After creating the .tcf file, open up the IRAM memory object ...

(Note, your 'platform' may have additional MEM objects.)



... and turn on the heap. Set the **Heap Size** to 0x2000 bytes.



Select the IRAM segment heap for both BIOS and malloc/free.

Finally, save the .tcf file with **File** → **Save As**. Before saving the file be sure to verify that the path is set to **c:\op6000\labs\lab13\SineTest**.

Add/Edit Files to the SineTest Project

26. Move the three test/framework files created by the wizard into your SineTest project folder.

Add these three files to the SineTest project folder:

```
C:\op6000\labs\lab14\Sine_WizardFiles\tstapp\main.c
C:\op6000\labs\lab14\Sine_WizardFiles\sine\sine.c
C:\op6000\labs\lab14\Sine_WizardFiles\sine\sine.h
```

27. Now, add the two C files we just copied into the SineTest project ... plus three more.

```
C:\op6000\labs\lab14\SineTest\main.c
C:\op6000\labs\lab14\SineTest\sine.c
```

Also add the following three files to your project:

```
C:\op6000\labs\lab14\SineGeneratorLib\Files_For_Release\user_src\isine.c
C:\op6000\labs\lab13\SineTest\sinetest.tcf
C:\op6000\labs\lab13\SineTest\sinetestcfg.cmd
```

28. Delete unnecessary code from main.c in the SineTest.pjt.

We need to edit out a few extraneous code created by the xDAIS wizard.

Delete the following two function calls. These are not needed as the initialization code is already present in our file.

```
SINE_init();
SINE_exit();
```

Additionally, you can **delete** the following comment lines since we do not plan to test the “control” function in this lab. (We will test the “process” function, though, so leave that comment in the code.)

```
/* Call the control function to modify parameters as the algorithm runs
   SINE_Status status;

// get current status
   SINE_control(pPtr->handle, SINE_GETSTATUS, &status);

// modify/check status parameters before setting the new status info
   SINE_control(pPtr->handle, SINE_SETSTATUS, &status); */
```

Note: In this exercise we are using the simple test functions provided by the wizard; in a real program, though, you might prefer to use the Codec Engine or the DSKT2 framework.

29. Add code to main.c (in SineTest.pjt) to setup the buffers needed to test our algorithm.

Declares

```
//
// ===== Declarations =====
//
#define BUFFSIZE 512
```

Global variable

```
//
// ===== Global Variables =====
//
short gBuffer[BUFFSIZE];
```

for{} loop to main.c to clear the target buffer

```
int i;

for (i=0; i < BUFFSIZE; i++ ){           // zero out buffers
    gBuffer[i] = 0;
}
```

Your main.c file should now look something like (*we shaded the original code below*):

```
//
// ===== Include files =====
//
#include <std.h>
#include "sine.h"
#include "sine_tto.h"

//
// ===== Declarations =====
//
#define BUFFSIZE 512

//
// ===== Global Variables =====
//
short gBuffer[BUFFSIZE];           // buffer for sinewave data

void main(){
    SINE_Handle  handle;
    ISINE_Fxns   fxns;
    SINE_Params  params;
    int i;

    for (i=0; i < BUFFSIZE; i++ ){           // zero out buffers
        gBuffer[i] = 0;
    }

    fxns = SINE_TTO_ISINE;
    params = SINE_PARAMS;

    if((handle = SINE_create(&fxns, &params)) != NULL){
        ...
    }
}
```

Build and Fix the Test Program

30. Try building the project after verifying that the correct processor is selected in Build Options.

While the SineGeneratorLib project should build without errors, the SineTest project should fail. It fails when trying to find some header files referenced by our test code.

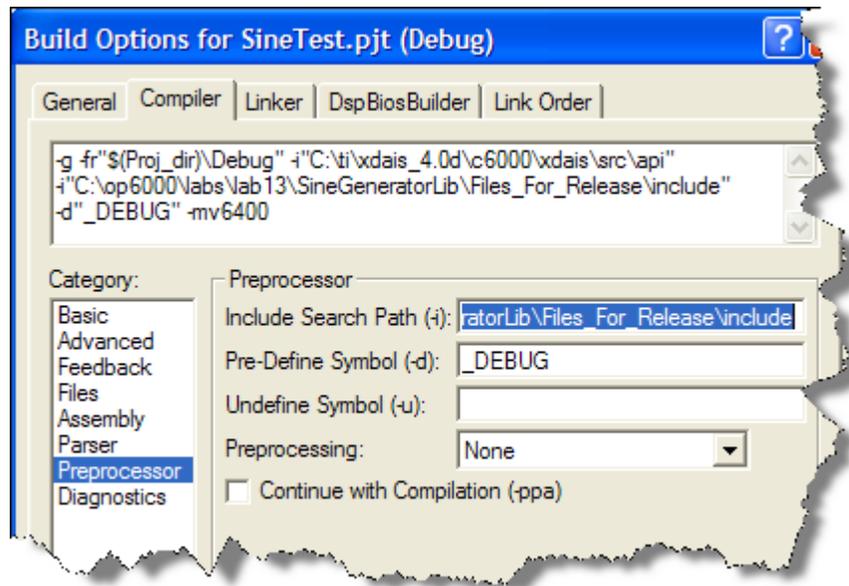
31. Add directories to the compiler's Include Search Path.

We need to add these references made by the test code:

- C:\op6000\labs\lab13\SineGeneratorLib\Files_For_Release\include **isine.h/isine_tto.h** are the algorithm's interface description files
- C:\ti\xdais_4.0d\c6000\xdais\src\api **alg.c/alg.h** are used by the test code generated by the wizard. These files are installed as part of the xDAIS library.

You can either type in the `-i` option in the advanced box, or type the “;” separated path into the *Compiler*→*Preprocessor*→*Include Search Path* box:

```
C:\ti\xdais_4.0d\c6000\xdais\src\api;  
C:\op6000\labs\lab13\SineGeneratorLib\Files_For_Release\include
```



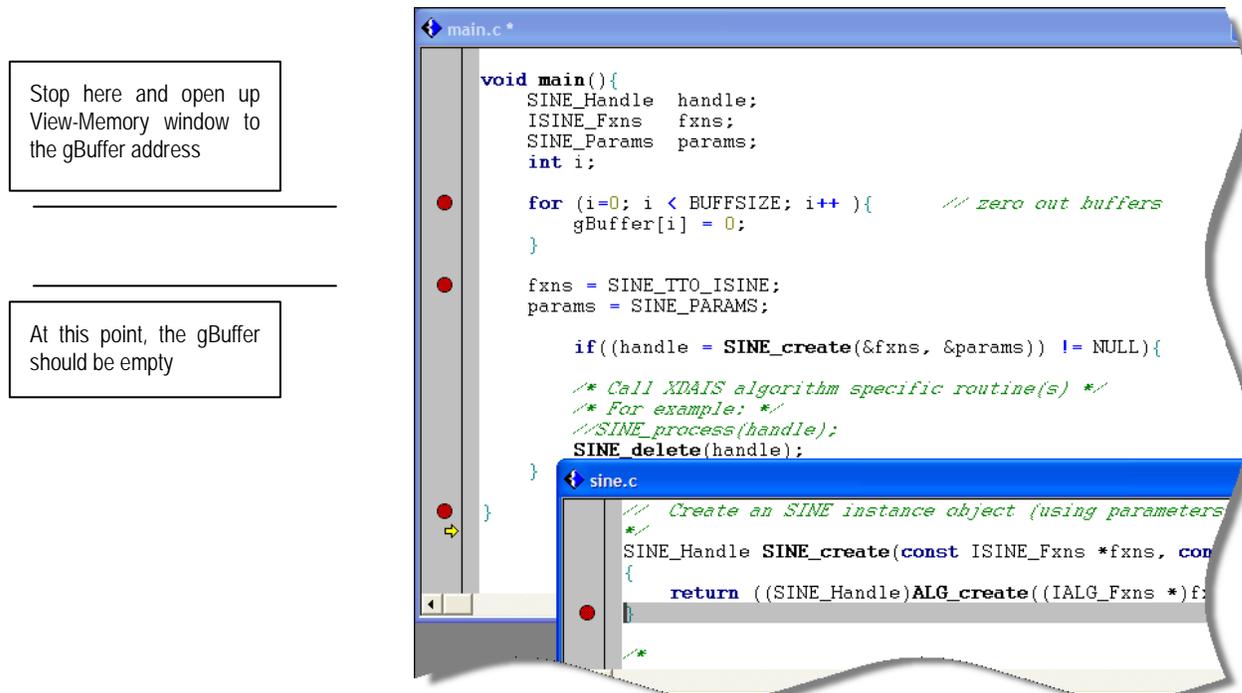
32. Add the correct xDAIS alg library to your project.

This is a library referenced in the last step. Use one of the two following libraries depending on which processor you selected for the lab exercise.

```
For the C64 and C64+ add: c:\ti\xdais_4.0d\c6000\xdais\lib\api.L64  
For the C67 and C672x add: c:\ti\xdais_4.0d\c6000\xdais\lib\api.L67
```

33. Build and run the test program

At this point your project should build without errors. Once it has been loaded into the simulator, try running it. Here's some example breakpoints we used to check out our program.



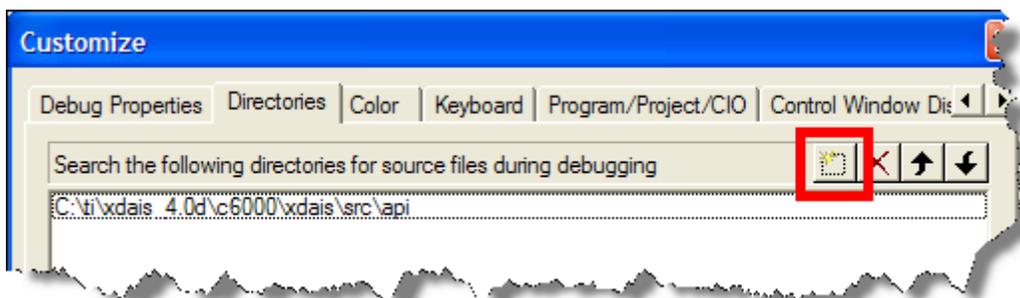
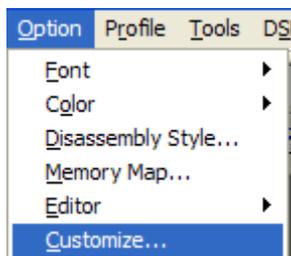
Did the sine algorithm produce any results? (Actually, it doesn't produce anything, does it?) At this point, we've only built the algorithm interface; next we need to add the actual implementation.

34. Single-stepping ... telling CCStudio where to find source files that aren't in the project.

Notice that you can step through the code, even into the dependent library project. If you try this – and step into the ALG_create() function – CCStudio may ask where to find the ALG_ functions source code. If you want to step through the source code, you should navigate to:

C:\ti\xdaish_4.0d\c6000\xdaish\src\api

Another alternative to navigating to files over-and-over again is to tell CCStudio where to find them.



Add “algorithm code” to our software component

To complete this next part of the lab, we’ll ‘steal’ code from the provided sine wave algorithm.

```
C:\op6000\labs\workshop_examples\sine
```

We could have wrapped the dot-product code we have used throughout the workshop, but we wanted to try working with something a little more complex. (The sinewave algo isn’t much more complex, but it does require some additional elements; most notably, it needs to be initialized before it can run.

This is the sinewave algorithm used in the “C6416/C6713 Integration Workshop”. This is not an optimized sinewave algorithm – actually, far from it. But, it does produce a steady, stable sinewave which is useful for this exercise.

For your convenience, we have printed out a copy of this algorithm’s code in the appendix to this workshop. You should be able to get what you need from the files in the folder referenced above, but look in the appendix if it reading through the code helps you better understand what the algorithm is doing.

Completing the Sine Algorithm Functions

As discussed in the preceding chapter, xDAIS algorithms must inherit a set of functions to assist with creating, running, and deleting instances of an algorithm. The functions created for our algo are defined in a table, usually called the “virtual table” or vTab.

35. Examine the vTab created by the wizard.

Open `sine_tto_ialgvt.c` in the SineGeneratorLib project and examine the table definition:

```
#define IALGFXNS \
    &SINE_TTO_IALG,          /* module ID          */ \
    NULL,                   /* activate   (NULL => not supported) */ \
    SINE_TTO_alloc,        /* algAlloc          */ \
    SINE_TTO_control,     /* control   (NULL => not supported) */ \
    NULL,                   /* deactivate (NULL => not supported) */ \
    SINE_TTO_free,        /* free            */ \
    SINE_TTO_initObj,     /* init            */ \
    NULL,                   /* moved   (NULL => not supported) */ \
    NULL                    /* numAlloc  (NULL => IALG_DEFMEMRECS) */ \
```

Notice that the functions which were not implemented are defined by NULL. For example, since our algorithm does not use scratch memory, neither *activate* nor *deactivate* were defined.

As an example, check out the `SINE_TTO_alloc` function created by the wizard. You will find it in the `sine_tto_ialg.c` file (which is another source file in our library project).

36. Adding “class” variables to our sine instance object.

Look through the sine_tto_ialg.c file until you find the definition of our instance object:

```
typedef struct SINE_TTO_Obj {
    IALG_Obj      alg;    /* MUST be first field of all SINE objs */
    float         freqTone;
    float         freqSampRate;

    /* TODO: add custom fields here */
} SINE_TTO_Obj;
```

You might remember the two items listed in this structure are the two creation parameters defined when working through the wizard. The wizard automatically adds all creation parameters to our instance object.

Compare this structure to the object used in the IW6000 workshop. You will find that definition in the workshop examples folder:

```
C:\op6000\labs\workshop_examples\sine\iw6000_sine.h
```

You should see that the object definition for the iw6000_sine.h contains a few more variables. We didn’t add all of these as creation parameters (because they are only needed internal to our algorithm), therefore we will have to add them now.

Copy and paste the additional variables into our sine_tto_ialg.c object definition.

Complete the TTO_SINE_process() Function

37. Locate the TTO_SINE_process() function in tto_sine_ialg.c.

38. So it won’t confuse us some day in the future, remove the comments:

```
/*
int index;
for(index=0; index<SINE->framesizeOut0; index++){
    *(ptrOut0+index) = *(ptrIn0+index);
}
*/
```

39. Add sine generation loop to the _process() function.

We want to copy the SINE_blockFill() function from iw6000_sine.c into our process function. The code is very simple:

```
int i = 0;

for (i = 0; i < len; i++) {
    buf[i] = sineGen(SINE);
}
```

Since our objects are named slightly differently, we just changed the variable reference from “sineObj” to “SINE” in the above code.

40. Add the sine generation function to our tto_sine_ialg.c file.

Since our `_process()` function called another function, we need to include it in our file. Copy and paste the `SineGen()` function from `iw6000_sine.c` to the end of `tto_sine_ialg.c`.

Notice, the one change we made to the code after copying it: we changed the data type of `sineObj` from "SINE_Obj" to "SINE_TTO_Obj".

```
// ===== sineGen =====
// Generate a single sine wave value
//
static short sineGen(SINE_TTO_Obj *sineObj)
{
    float result;

    if (sineObj->count > 0) {
        sineObj->count = sineObj->count - 1;
    }
    else {
        sineObj->a = sineObj->aInitVal;
        sineObj->b = sineObj->bInitVal;
        sineObj->y0 = sineObj->y0InitVal;
        sineObj->y1 = sineObj->y1InitVal;
        sineObj->y2 = sineObj->y2InitVal;
        sineObj->count = sineObj->countInitVal;
    }

    sineObj->y0 = (sineObj->a * sineObj->y1) + (sineObj->b * sineObj->y2);
    sineObj->y2 = sineObj->y1;
    sineObj->y1 = sineObj->y0;

    // To scale full 16-bit range we would multiply y[0]
    // by 32768 using a number slightly less than this
    // (such as 28000) helps to prevent overflow.
    result = sineObj->y0 * 28000;

    // We recast the result to a short value upon returning it
    // since the D/A converter is programmed to accept 16-bit
    // signed values.
    return((short)result);
}
```

41. Add the prototype for SineGen() to the tto_sine_ialg.c file.**42. Build to verify there are no errors.**

You may still have a warning, though, indicating a variable was declared but never used.

Complete the Sine Initialization Function**43. Locate the TTO_SINE_initObj() function in tto_sine_ialg.c.**

Make sure you are working with the `_initObj()` function (not the `_init()` function).

44. Create the local variable “rad”.

The variable rad is used by the init routine to setup the coefficients needed to produce the specified tone.

```
float rad = 0;
```

45. Copy and paste most of the init code from the iw6000_sine.c file.

Since the tone and sample-rate variables are already set, we just need to copy the remaining code into our TIO_SINE_initObj() function.

```
/* TODO: Implement any additional algInit desired */
rad = sineObj->freqTone / sineObj->freqSampleRate;
rad = rad * 360.0;
rad = degreesToRadiansF(rad);

sineObj->a = 2 * cosf(rad);
sineObj->b = -1;
sineObj->y0 = 0;
sineObj->y1 = sinf(rad);
sineObj->y2 = 0;
sineObj->count = sineObj->freqTone * sineObj->freqSampleRate;

sineObj->aInitVal = sineObj->a;
sineObj->bInitVal = sineObj->b;
sineObj->y0InitVal = sineObj->y0;
sineObj->y1InitVal = sineObj->y1;
sineObj->y2InitVal = sineObj->y2;
sineObj->countInitVal = sineObj->count;
```

46. Search and replace for sineObj.

Since the iw6000_sine.c used “sineObj” as the object name and our algo uses “SINE”, we need to use search and replace to fix this. Be careful, though, since the use of “sineObj” is legal in the SineGen() function. (We highlighted the places needing changes in the above snippet of code.)

47. Add some code to load the default sine generator parameters

```
/* TODO: Implement any additional algInit desired */

SINE->freqTone = params->freqTone; // Load default parameters
SINE->freqSampleRate = params->freqSampleRate; // Load default parameters

rad = SINE->freqTone / SINE->freqSampleRate;
rad = rad * 360.0;
rad = degreesToRadiansF(rad);

SINE->a = 2 * cosf(rad);
SINE->b = -1;
SINE->y0 = 0;
SINE->y1 = sinf(rad);
SINE->y2 = 0;
SINE->count = SINE->freqTone * SINE->freqSampleRate;
```

Add final definitions, includes, and prototypes

Before we can build our library we just need to finalize some additional details required by our `_initObj()` function.

48. Add a reference to the standard C language `math.h`.

This is used by our initialization code.

```
#include <math.h>
```

49. Add the definition for `PI`.

Also used by our `initObj` function.

```
// ===== Definitions =====  
#define PI 3.1415927
```

50. Add the degrees to radians conversion function.

Append this function to the end of the file. It is used by the `initObj` function, too.

```
// ===== degreesToRadiansF =====  
// Converts a floating point number from degrees to radians  
static float degreesToRadiansF(float d)  
{  
    return(d * PI / 180);  
}
```

Don't forget to add the prototype for this function towards the beginning of this file!

51. Build `SineGenerator` (to see if it builds).

We'll test it's functionality in the next section.

Test completed algorithm component

52. If it is not active, make SineTest.pjt active.

53. Build and run program. Check the values in gBuffer at the end of main.

Did it work? If not, why didn't it work?

If you followed our directions explicitly, then you need to add one more item to your test program. We actually need to call the process function. It needs to be uncommented and two arguments added.

```
/* Call XDAIS algorithm specific routine(s)
 * For example:
 *   SINE_process(handle);
 */SINE_process(handle, gBuffer, BUFFSIZE);
```

54. Build the test app again and verify that you get the correct data

55. Now it is time to clean up those build warnings that we've been getting.

There are three lines of source code that we need to comment in order to have a clean build. All this source is located in the sine_tto_ialg.c file.

Locate the **SINE_TTO_control** function and comment the following two lines:

```
// ISINE_Status *sPtr = (ISINE_Status *)status;
// SINE_TTO_Obj *SINE = (Void *)handle;
```

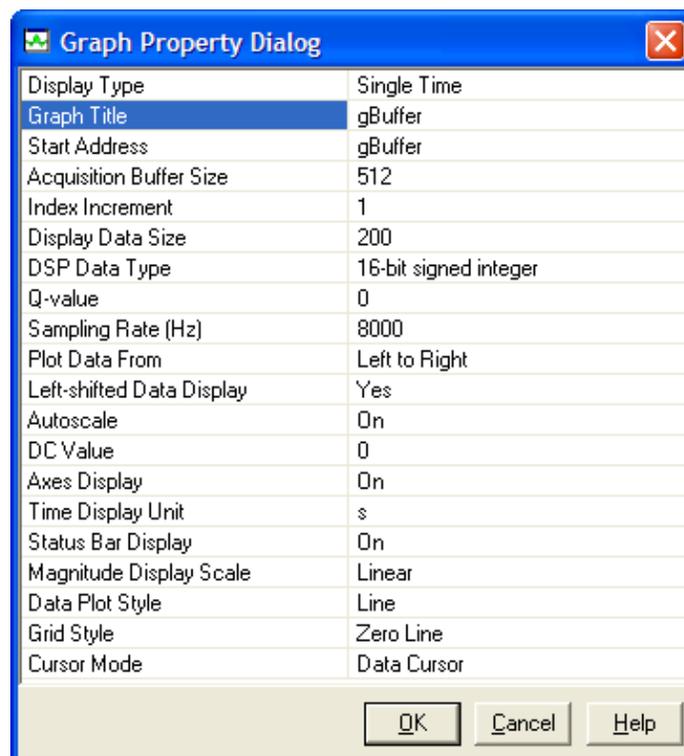
The last line to be commented is in the **SINE_TTO_free** function call:

```
// SINE_TTO_Obj *SINE = (Void
*)handle;
```

Build the project one more time and verify that there are no warnings.

56. To really verify our results, try graphing gBuffer[].

As a reminder, the Graph Property Dialog is accessible by going to: **View** → **Graph** → **Time\Frequency**. Complete the dialog as shown on the right and execute the program with breakpoints set at the same locations that they were back in **Step 33**.



Final 1000 Words

The screenshot displays the Code Composer Studio interface for a TMS320C6416 simulator. The main window shows the source code for `main.c`, which implements a sine wave generation algorithm. The code includes a buffer initialization loop, function pointers for the algorithm, and a call to `SINE_create` to generate the sine wave. A plot window titled `gBuffer` shows the resulting sine wave, with the x-axis representing time and the y-axis representing amplitude. The plot shows a single cycle of a sine wave with a peak amplitude of approximately 2.8e+4 and a trough of approximately -2.8e+4. The x-axis has markers at 0, 0.0125, and 0.0249. The status bar at the bottom indicates that the program has halted at a software breakpoint.

```

short gBuffer[BUFSIZE];

void main(){
    SINE_Handle  handle;
    ISINE_Fxns  fxns;
    SINE_Params  params;
    int  i;

    for (i=0; i < BUFSIZE; i++){ // zero out buffers
        gBuffer[i] = 0;
    }

    fxns = SINE_TTO_ISINE;
    params = SINE_PARAMS;

    if((handle = SINE_create(&fxns, &params)) != NULL){
        /* Call XDAIS algorithm specific routine(s)
        * For example:
        * SINE_process(handle);
        */
        SINE_process(handle, gBuffer, BUFSIZE);

        SINE_delete(handle);
    }
}
    
```

Address	Value	Value	Value
0x00010794	27846	27913	
0x00010798	27961	27990	
0x0001079C	27999	27989	
0x000107A0	27961	27913	
0x000107A4	27846	27759	
0x000107A8	27654	27530	
0x000107AC	27387	27225	
0x000107B0	27045	26846	
0x000107B4	26628	26393	
0x000107B8	26139	25867	
0x000107BC	25578	25271	
0x000107C0	24947	24605	
0x000107C4	24247	23872	
0x000107C8	23481	23074	
0x000107CC	22651	22212	
0x000107D0	21758	21289	
0x000107D4	20806	20309	
0x000107D8	19797	19272	
0x000107DC	18734	18183	
0x000107E0	17619	17043	
0x000107E4	16456	15857	
0x000107E8	15248	14628	
0x000107EC	13998	13358	
0x000107F0	12709	12052	
0x000107F4	11386	10713	
0x000107F8	10032	9344	
0x000107FC	8650	7950	
0x00010800	7245	6534	
0x00010804	5819	5100	
0x00010808	4378	3652	
0x0001080C	2924	2194	
0x00010810	1463	730	
0x00010814	-2	-735	

Build Complete,
0 Errors, 0 Warnings, 0 Remarks.

Build

HALTED: s/w breakpoint

Lab 14 – Appendix

sine.h

```
/*
 * ===== sine.h =====
 * This file contains prototypes for all functions
 * contained in sine.c
 */
#ifndef SINE_Obj
typedef struct {
    float freqTone;
    float freqSampleRate;
    float a;
    float b;
    float y0;
    float y1;
    float y2;
    float count;
    float aInitVal;
    float bInitVal;
    float y0InitVal;
    float y1InitVal;
    float y2InitVal;
    float countInitVal;
} SINE_Obj;
#endif

void copyData(short *inbuf, short *outbuf ,int length);
void SINE_init(SINE_Obj *sineObj, float freqTone, float freqSampleRate);
void SINE_blockFill(SINE_Obj *myObj, short *buf, int len);
void SINE_addPacked(SINE_Obj *myObj, short *inbuf, int length);
void SINE_add(SINE_Obj *myObj, short *inbuf, int length);
```

sine.c

```

// ===== sine.c =====
// The coefficient A and the three initial values
// generate a 200Hz tone (sine wave) when running
// at a sample rate of 48KHz.
//
// Even though the calculations are done in floating
// point, this function returns a short value since
// this is what's needed by a 16-bit codec (DAC).

// ===== Includes =====
#include "iw6000_sine.h"
#include <std.h>
#include <math.h>

// ===== Definitions =====
#define PI 3.1415927

// ===== Prototypes =====
void SINE_init(SINE_Obj *sineObj, float freqTone, float freqSampleRate);
void SINE_blockFill(SINE_Obj *sineObj, short *buf, int len);
void SINE_addPacked(SINE_Obj *sineObj, short *inbuf, int length);
void SINE_add(SINE_Obj *sineObj, short *inbuf, int length);
static short sineGen(SINE_Obj *sineObj);
static float degreesToRadiansF(float d);
void copyData(short *inbuf, short *outbuf ,int length );

// ===== Globals =====

// ===== SINE_init =====
// Initializes the sine wave generation algorithm
void SINE_init(SINE_Obj *sineObj, float freqTone, float freqSampleRate)
{
    float rad = 0;

    if(freqTone == NULL)
        sineObj->freqTone = 200;
    else
        sineObj->freqTone = freqTone;

    if(freqSampleRate == NULL)
        sineObj->freqSampleRate = 48 * 1000;
    else
        sineObj->freqSampleRate = freqSampleRate;

    rad = sineObj->freqTone / sineObj->freqSampleRate;
    rad = rad * 360.0;
    rad = degreesToRadiansF(rad);

    sineObj->a = 2 * cosf(rad);
    sineObj->b = -1;
    sineObj->y0 = 0;
    sineObj->y1 = sinf(rad);
    sineObj->y2 = 0;
    sineObj->count = sineObj->freqTone * sineObj->freqSampleRate;

    sineObj->aInitVal = sineObj->a;
    sineObj->bInitVal = sineObj->b;
    sineObj->y0InitVal = sineObj->y0;
    sineObj->y1InitVal = sineObj->y1;
    sineObj->y2InitVal = sineObj->y2;
    sineObj->countInitVal = sineObj->count;
}

```

```

}

//      ===== SINE_blockFill =====
// Generate a block of sine data using sineGen
void SINE_blockFill(SINE_Obj *sineObj, short *buf, int len)
{
    int i = 0;

    for (i = 0; i < len; i++) {
        buf[i] = sineGen(sineObj);
    }
}

//      ===== SINE_addPacked =====
// add the sine wave to the indicated buffer of packed
// left/right data
// divide the sine wave signal by 8 and add it
void SINE_addPacked(SINE_Obj *sineObj, short *inbuf, int length)
{
    int i = 0;
    static short temp;

    for (i = 0; i < length; i+=2) {
        temp = sineGen(sineObj);
        inbuf[i] = (inbuf[i]) + (temp>>4);
        inbuf[i+1] = (inbuf[i+1]) + (temp>>4);
    }
}

// ===== SINE_add =====
// add the sine wave to the indicated buffer
void SINE_add(SINE_Obj *sineObj, short *inbuf, int length)
{
    int i = 0;
    short temp;

    for (i = 0; i < length; i++) {
        temp = sineGen(sineObj);
        inbuf[i] = (inbuf[i]) + (temp>>4);
    }
}

//      ===== sineGen =====
// Generate a single sine wave value
static short sineGen(SINE_Obj *sineObj)
{
    float result;

    if (sineObj->count > 0) {
        sineObj->count = sineObj->count - 1;
    }
    else {
        sineObj->a = sineObj->aInitVal;
        sineObj->b = sineObj->bInitVal;
        sineObj->y0 = sineObj->y0InitVal;
        sineObj->y1 = sineObj->y1InitVal;
        sineObj->y2 = sineObj->y2InitVal;
        sineObj->count = sineObj->countInitVal;
    }

    sineObj->y0 = (sineObj->a * sineObj->y1) + (sineObj->b * sineObj->y2);
    sineObj->y2 = sineObj->y1;
    sineObj->y1 = sineObj->y0;
}

```

```
// To scale full 16-bit range we would multiply y[0]
// by 32768 using a number slightly less than this
// (such as 28000) helps to prevent overflow.
result = sineObj->y0 * 28000;

// We recast the result to a short value upon returning it
// since the D/A converter is programmed to accept 16-bit
// signed values.
return((short)result);
}

// ===== degreesToRadiansF =====
// Converts a floating point number from degrees to radians
static float degreesToRadiansF(float d)
{
    return(d * PI / 180);
}

// ===== copyData =====
// copy data from one buffer to the other.
void copyData(short *inbuf, short *outbuf ,int length )
{
    int i = 0;

    for (i = 0; i < length; i++) {
        outbuf[i] = inbuf[i];
    }
}
```

This page intentionally left blank.

Lab and Exercise Solutions

Lab and Exercise Solutions	1
Chapter 1	1
Chapter 3	5

Chapter 1

Module 1 Exam

1. Functional Units

(5 pts) a. How many can perform an ADD? Name them.

six; .L1, .L2, .D1, .D2, .S1, .S2

(5 pts) b. Which support memory loads/stores?

.M .S  **.D** .L

2. C6000 Memories and Busing

(5 pts) a. What is the advantage of L1 memory?

Fastest memory on the device (single-cycle; zero waitstates)

(5 pts) b. What does the EDMA3 stand for ... and do?

Enhanced DMA version 3 – transfer blocks of memory

(5 pts) c. What is an SCR master?

A CPU or peripheral that can initiate a data transfer across the Switched Central Resource (SCR) (aka TeraNet)

3. Conditional Code

(10 pts) a. Which registers can be used as cond'l registers?

A1, A2, B0, B1, B2 (*C64x allows A0, too)

(10 pts) b. Which instructions can be conditional?

All of them

4. Performance

(5 pts) a. What is the 'C6201 instruction cycle time?

5 ns (C6201 can run up to 200 MHz)

(10 pts) b. How can the 'C6201 execute 1600 MIPs?

1600 MIPs = 8 instructions (units) x 200 MHz

How many MIPs can the 'C6678 execute?

80,000 MIPs* (8 instructions x 1.25 GHz x 8 core's)

* not counting the affect of Packed Data Processing (like ADD4)

4c. Performance

(10 pts) c. How many 16-bit MMACs (millions of MACs) can the 'C6201 perform?

400 MMACs (two .M units x 200 MHz)

How about C674x or C66x cores?

CPU Core (example device)	C64x (C6416)	C64x+ (C6416)	C674x (C6A8168)	C66x (C6678)
MMACS (16x16)	4,000	10,000	12,000	40,000
# .M units	2	2	x 2	2
# MACS per .M per cycle	x 2	x 4	x 4	x 16
Speed	x 1 GHz	x 1.2 GHz	x 1.5 GHz	x 1.25 GHz
Total	= 4000	= 10000	= 12000	= 40000

5a - 5b. Coding Problems

(5 pts) a. Move contents of A0 → A1

```

MV    A0, A1
or    ADD  A0, 0, A1
or    AND  A0, -1, A1
or    OR   A0, 0, A1

```

(5 pts) b. Clear register A5

```

ZERO  A5
or    SUB  A5, A5, A5
or    MPY  A5, 0, A5
or    CLR  A5, 0, 31, A5
or    MVK  0, A5
or    XOR  A5, A5, A5
or    AND  A5, 0, A5

```

5c - 5e. Coding Problems

(5 pts) c. $A2 = A0^2 + A1$

```
MPY.M1    A0, A0, A2
ADD.L1    A2, A1, A2
```

(10 pts) d. If $(B1 \neq 0)$ then $B2 = B5 * B6$

```
[B1] MPY.M2    B5, B6, B2
```

(5 pts) e. Load an unsigned constant (0x19ABC) into register A6.

see next two slides

5e. Solutions that Don't Work

```
mvk 0x19abc, a6 → mvk 0x9abc, a6
```

A6	F	F	F	F	9	A	B	C
----	---	---	---	---	---	---	---	---

```
mvkl 0x9abc, a6 → mvkl 0x00009abc, a6
mvkh 1, a6      mvkh 00000001, a6
```

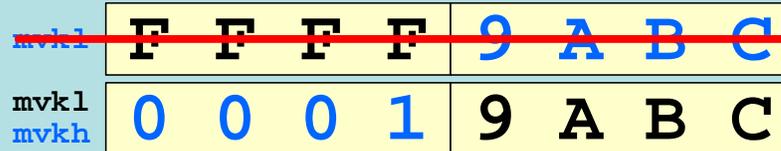
mvkl	F	F	F	F	9	A	B	C
mvkl mvkh	0	0	0	0	9	A	B	C

5e. Solutions that Work

```

mvkl 0x00019abc, a6 → mvkl 0x00009abc, a6
mvkh 0x00019abc, a6   mvkh 0x00019abc, a6

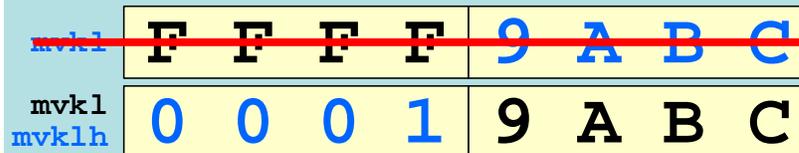
```



```

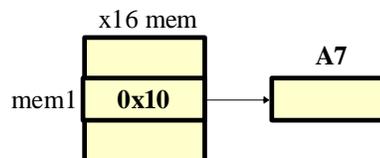
mvkl 0x9abc, a6 → mvkl 0x00009abc, a6
mvklh 1, a6     mvklh 00000001, a6

```



5f. Coding problems

- (10 pts) f. Load A7 with contents of mem1 and post-increment the selected pointer.



```

load_mem1:  MVKL  .S1    mem1,  A6
            MVKH  .S1    mem1,  A6
            LDH   .D1    *A6++, A7

```

Chapter 3

1. Pipeline

(15 pts) a. Name and describe the three primary pipeline stages.

Program-Fetch, Decode, Execute

Program-Fetch: PG, PS, PW, PR
4 phases required to read FP from memory

Decode: DP, DC
2 phases required to route instruction to functional unit and decode it

Execute: E1 - E10
Execution of actual instr in functional-unit

1. Pipeline

(10 pts) b. Why did we make program fetches (PF pipeline stage) four cycles?

Otherwise, memories wouldn't be fast enough for the C6000.

(10 pts) c. Why are pipeline stages of a load (E1-E5) similar to the phases of a program fetch (PG-DP)?

They are both memory read operations.

(15 pts) 2. What is a fetch packet? execute packet?

FP - group of eight instr all fetched together

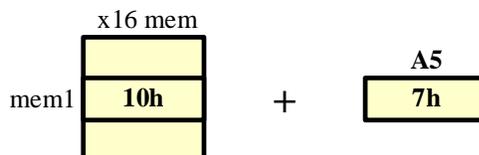
EP - group of parallel instructions which all execute together

3. Insert NOPs as needed:

(30 pts) **start:** LDH .D1 *A0, A1
 NOP 4
 MPY .M1 A1, A2, A3
 ; No NOP needed
 ADD .L1 A1, A2, A4
 LDB .D1 *A5, A6
 NOP 3
 LDW .D1 *A7, A8
 ; No NOP needed
 MPY .M1 A6, A9, A10
 NOP
 ADD .L1 A9, A10, A11
 B .S1 start
 NOP 5

4. Coding

(20 pts) Add the contents of mem1 to A5.



```
load_mem1:  MVKL  .S1  mem1,  A7
            MVKH  .S1  mem1,  A7
            LDH   .D1  *A7,   A7
            NOP   4
add_mem1:   ADD   .L1  A7,   A5, A5
```

Register Allocation Table

Instruction / Notes	A register file	#	#	B register file	#	Instruction / Notes
		A0	B0			
		A1	B1			
		A2	B2			
	<i>Struct</i>	A3	B3	<i>Address</i>		
	<i>(1)</i> <i>Value</i>	A4	B4	<i>(2)</i>		
		A5	B5			
	<i>(3)</i>	A6	B6	<i>(4)</i>		
		A7	B7			
	<i>(5)</i>	A8	B8	<i>(6)</i>		
		A9	B9			
	<i>(7)</i>	A10	B10	<i>(8)</i>		
		A11	B11			
	<i>(9)</i>	A12	B12	<i>(10)</i>		
		A13	B13			
		A14	B14	<i>*DP</i>		
		A15	B15	<i>*Stack</i>		

Registers A1, A2, B0-B2 can be used as conditional registers for all instructions.

Registers A4-A7, B4-B7 can be used for circular addressing pointers.

If modified, registers A10-A15, B10-B15 must be saved by assembly-called subroutine.

Other references refer to the register usage by C Compiler.

Register Allocation Table

Instruction / Notes	A register file	#	#	B register file	#	Instruction / Notes
		A0	B0			
		A1	B1			
		A2	B2			
	<i>Struct</i>	A3	B3	<i>Address</i>		
	<i>(1)</i> <i>Value</i>	A4	B4	<i>(2)</i>		
		A5	B5			
	<i>(3)</i>	A6	B6	<i>(4)</i>		
		A7	B7			
	<i>(5)</i>	A8	B8	<i>(6)</i>		
		A9	B9			
	<i>(7)</i>	A10	B10	<i>(8)</i>		
		A11	B11			
	<i>(9)</i>	A12	B12	<i>(10)</i>		
		A13	B13			
		A14	B14	<i>*DP</i>		
		A15	B15	<i>*Stack</i>		

Registers A1, A2, B0-B2 can be used as conditional registers for all instructions.

Registers A4-A7, B4-B7 can be used for circular addressing pointers.

If modified, registers A10-A15, B10-B15 must be saved by assembly-called subroutine.

Other references refer to the register usage by C Compiler.

Scheduling Worksheet

Cycle																				
.L1																				
.L2																				
.M1																				
.M2																				
D1																				
.D2																				
.S1																				
.S2																				
Cycle																				
.L1																				
.L2																				
.M1																				
.M2																				
D1																				
.D2																				
.S1																				
.S2																				

Scheduling Worksheet

Cycle																				
.L1																				
.L2																				
.M1																				
.M2																				
D1																				
.D2																				
.S1																				
.S2																				
Cycle																				
.L1																				
.L2																				
.M1																				
.M2																				
D1																				
.D2																				
.S1																				
.S2																				

Appendix B - Circular Buffering

Appendix B consists of two parts:

1. Application Note
2. Optional set of slides

Page left intentionally blank

TMS320C6000 Non-Power of 2 Circular Buffers

*Eric Wilbur
Yao-Ting Cheng*

Digital Signal Processing Solutions

ABSTRACT

The TMS320C6000™ circular buffer block size must be specified as a power of 2. However, some filters require coefficient/data table sizes which are not a power of 2. This application report shows an easy work around to implement the circular buffer with any block size.

Contents

1	Design Problem	1
2	Solution	1
	2.1 Procedure	2

List of Figures

Figure 1.	Pointers to the Beginning of the Circular Buffer	2
Figure 2.	Pointer Increment and Reset	3
Figure 3.	Pointer Increment and Reset	3
Figure 4.	Pointer Increment and Reset	4

List of Examples

Example 1.	Code Listing	5
------------	--------------------	---

TMS320C6000 is a trademark of Texas Instruments.

1 Design Problem

Eight C6x registers (A4~A7, B4~B7) can be used for circular addressing. The block size (or buffer size) must be specified as a power of 2 (i.e. 2ⁿ). However, some filters require coefficient/data table sizes which are not a power of 2. So, how then can you implement these algorithms on the C6x? One method to solve the problem is provided below.

2 Solution

Let's start with an example of an FIR filter with 6 16-bit coefficients/samples. The code to implement this solution is given at the end of the document.

$$Y = \sum_{i=1}^6 X_i \cdot C_i \tag{1}$$

This solution does waste some RAM (the 2ⁿ blk size minus the actual table size, in this case: 8-6 = 2 locations), but without any speed penalties. The basic principle is that you create a "hole" of unused samples that circulates through memory. The benefit is that once you set up your tables correctly, the code is very straightforward.

2.1 Procedure

1. Arrange the input samples in incrementing order such as X1,X2,...,X6 and the coefficients in descending order such as C6,C5,...,C1.
2. Set the block size of the circular buffer to the 2^n size greater than the actual table size, select circular register and initialize AMR. In our example, the block size will be 16 bytes or 8 shorts (16-bit). A4 will be used in circular mode (as the sample pointer), therefore AMR= 0x00030001.
3. Initialize the circular buffer pointer A4 to the top of the input samples (i.e. A4=&X1).
4. Initialize another pointer (using linear mode), to the top of the coefficients. In our example B4=&C1.

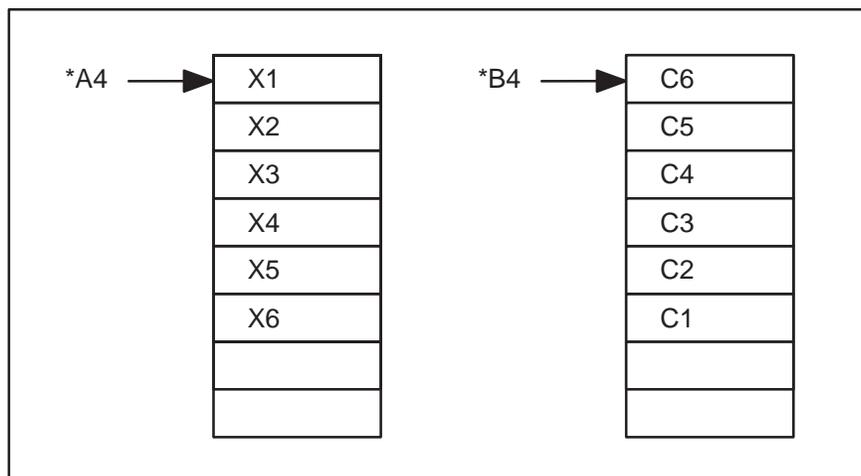


Figure 1. Pointers to the Beginning of the Circular Buffer

5. Set the loop counter to be the number of coefficients minus one. So, B0=5. This might look like a problem because your loop will only execute 5 times instead of 6. However, see step 7 for the solution.
6. Load the value from *A4++ and *+B4[B0] to registers and perform a MAC. As you can see, the count value is being used as a pre-offset to the base of the coefficient table. Therefore, the code will access C1 first, then C2, etc.
7. In the MAC loop, the counter should be decremented, looping a total of 6 times. However, please note that the SUB instruction should occur one cycle AFTER the branch or the code will only loop 5 times. This method allows the code to use B0 (the count value) as a pre-offset for the coefficients. If you are pipelining your algorithm, the same concept applies.

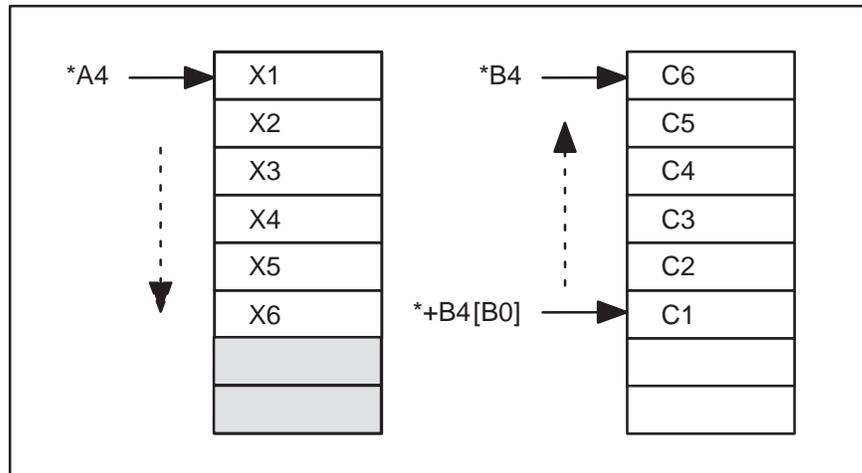


Figure 2. Pointer Increment and Reset

8. Get a new input sample. After the MAC operation, the circular register (e.g. A4) is pointing to the next “empty” location to be filled. When loading a new value into the sample table, if you post increment by $(2^n - \text{actual table size} + 1)$ (in our example this would be $8 - 6 + 1 = 3$), the pointer (A4) will be set to the new “top” of the buffer (the oldest sample).
9. Reset the circular pointer to point to the oldest sample. This can be done separately or as part of loading the new sample (as described in step 8).

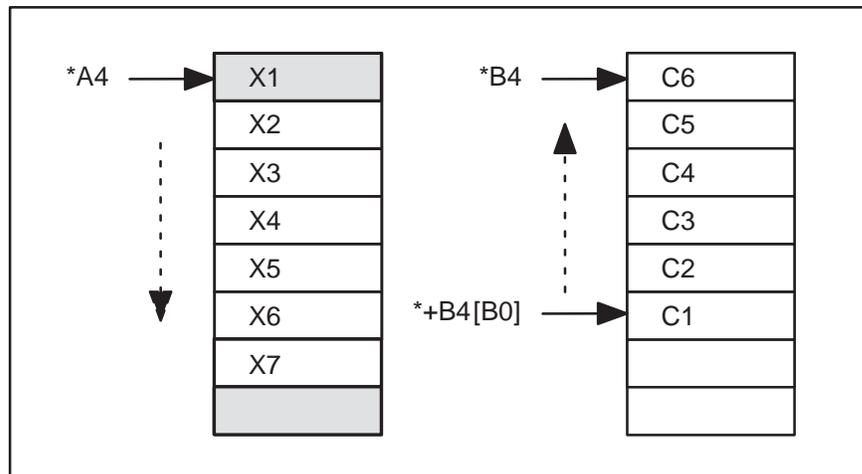


Figure 3. Pointer Increment and Reset

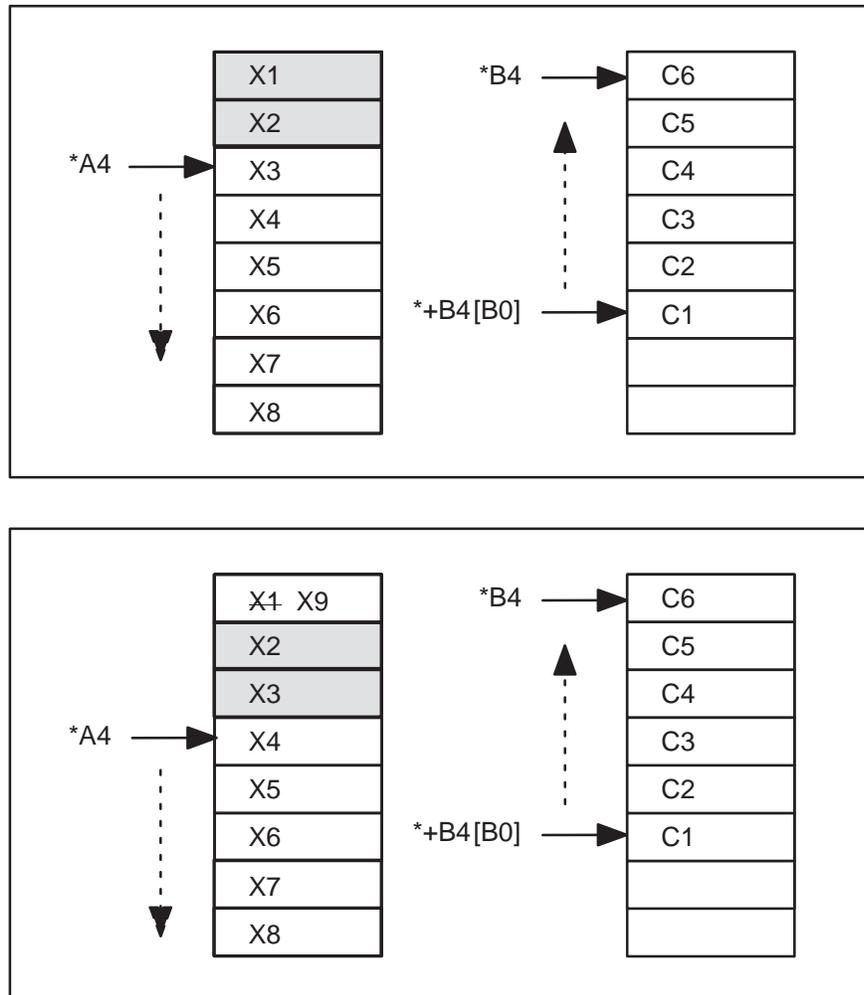


Figure 4. Pointer Increment and Reset

10. Go to step 5. As you can see in the diagram above, a “hole” of unused samples is created inside the buffer. Let’s say X9 is the newest sample. The filter will run from X4 through X8 and then X9. In this case, X2 and X3 locations were NOT used. As the filter continues, this “hole” of size 2 moves in a circular fashion. So, when X10 comes in, it overwrites X2, the filter runs from X5–X10 and X3/X4 locations are not used, etc. There is no wasted space in the coefficient table.
11. Note: This example does not include any software pipelining. However, the same concept can be applied to a s/w-pipelined loop. The goal was to provide a methodology for dealing with an algorithm using a block size which is not a power of 2.

Example 1. Code Listing

```

        .title  "fir.asm"           ; non 2^n Circular Buffer
        .def    init
half    .equ    2                   ; half is defined as 2 bytes
        .data
coeff   .short  6,5,4,3,2,1        ; FIR filter coefficients
initsmp .short  1,2,3,4,5,6        ; input samples
new_smp .short  7,8,9,0xa         ; new input samples
        .bss   sample,8*half,8*half
        .bss   output,1*half,1*half
        .text
init_AMR:
        MVK    .S2    0x0001,B0    ; setup cir. buf AMR=0001_0003h
        MVKLNH .S2    0x0003,B0    ; setup cir. buf AMR=0001_0003h
        MVC    .S2    B0,AMR      ; blk size is 16 bytes, pointer is A4
init_buffer: ; assume the variable has been initialized
init_pointers:
        MVK    .S2    new_smp,B7   ; init pointers
        MVKH   .S2    new_smp,B7
        MVK    .S2    coeff,B4
        MVKH   .S2    coeff,B4
        MVK    .S1    output,A9
        MVKH   .S1    output,A9
        MVK    .S1    sample,A4
        MVKH   .S1    sample,A4
fir:    ZERO    .L1    A7
        MVK    .S2    5,B0
loop:   LDH     .D1    *A4++,A5      ; load samples
        LDH     .D2    *+B4[B0],B5  ; load coefficients
        NOP
        MPY    .M1x   A5,B5,A6
        NOP
        ADD    .L1    A7,A6,A7
        [B0] B   .S1    loop
        [B0] SUB .L2    B0,1,B0
        NOP
        4
get_new_smp:
        LDH     .D2    *B7++,A8     ; get new sample
        STH     .D1    A7,*A9       ; store the result to output
        NOP
        3
        STH     .D1    A8,*A4--[5] ; store new sample to buffer
                                         ; and init circular pointer
        B       .S1    fir
        NOP
        5
        B       .S1    $
        NOP
        5
    
```

IMPORTANT NOTICE

Texas Instruments and its subsidiaries (TI) reserve the right to make changes to their products or to discontinue any product or service without notice, and advise customers to obtain the latest version of relevant information to verify, before placing orders, that information being relied on is current and complete. All products are sold subject to the terms and conditions of sale supplied at the time of order acknowledgment, including those pertaining to warranty, patent infringement, and limitation of liability.

TI warrants performance of its semiconductor products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are utilized to the extent TI deems necessary to support this warranty. Specific testing of all parameters of each device is not necessarily performed, except those mandated by government requirements.

Customers are responsible for their applications using TI components.

In order to minimize risks associated with the customer's applications, adequate design and operating safeguards must be provided by the customer to minimize inherent or procedural hazards.

TI assumes no liability for applications assistance or customer product design. TI does not warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such semiconductor products or services might be or are used. TI's publication of information regarding any third party's products or services does not constitute TI's approval, warranty or endorsement thereof.

Circular Buffering Example

Circular Buffer Using Non-2ⁿ Size

FIR Algorithm:

Sample size = 6

Buffer size = 8 (power of 2)

coeff		smp1	x16 (le)
a5		x(1)	
a4		x(2)	
a3		x(3)	
a2		x(4)	
a1		x(5)	
a0		x(6)	

“The Bubble in Memory”

Step 1: setup AMR and ptr to smp1

Step 2: Calculate 6-term result

Step 3: Next sample

Step 4: repeat steps 2-3



Non 2ⁿ Buffer Example

DECLARATIONS

SETUP

AMR
A4 = smp1
B4 = coeff
B0 = count

ALGORITHM

A4 (circular)
B4 (+offset using tapsize)

NEXT SAMPLE

A4 reset to “top” of buffer



```

tapsize .equ 6 ;# taps (count)
bufsize .equ 8 * 2 ;pwr 2 > tapsize
.sect "firCoeff"
coeff .short a5, a4, a3, a2, a1, a0
smp1 .usect "firBuf", bufsize

.sect ".text"
setup: MVK 0001h, B1 ;AMR=00030001
MVKLH 0003h, B1
MVC B1, AMR
MVKL smp1, A4
MVKH smp1, A4
MVKL coeff, B4
MVKH coeff, B4

FIR: ZERO A7
MVK tapsize-1, B0
loop: LDH *A4++, A5
LDH *+B4[B0], B5
NOP 4
MPY A5, B5, A6
NOP
ADD A6, A7, A7
[B0] B loop
[B0] SUB B0,1,B0
NOP 4
next: LDH *new_smp1_ptr, A8
NOP 4
STH A8, *A4--[tapsize-1]
B FIR
NOP 5

```


Non 2ⁿ Buffer Example

coeff

B4 →	a5
	a4
	a3
	a2
	a1
	a0

smpl

A4 →	x(0)
	x(1)
	x(2)
	x(3)
	x(4)
	x(5)

x16 (le)

```

tapsize .set 6           ;# taps (count)
bufsize .set 8 * 2       ;pwr 2 > tapsize

FIR:     ZERO    A7
        MVK     tapsize-1, B0
loop:    LDH     *A4++, A5
        LDH     *+B4[B0], B5
        NOP     4
        MPY     A5, B5, A6
        NOP
        ADD     A6, A7, A7
[B0]     B       loop
        SUB     B0,1,B0
        NOP     4
    
```



Non 2ⁿ Buffer Example

coeff

B4 →	a5
	a4
	a3
	a2
	a1
	a0

smpl

A4 →	x(0)
	x(1)
	x(2)
	x(3)
	x(4)
	x(5)

x16 (le)

```

tapsize .set 6           ;# taps (count)
bufsize .set 8 * 2       ;pwr 2 > tapsize

FIR:     ZERO    A7
        MVK     tapsize-1, B0
loop:    LDH     *A4++, A5
        LDH     *+B4[B0], B5
        NOP     4
        MPY     A5, B5, A6
        NOP
        ADD     A6, A7, A7
[B0]     B       loop
        SUB     B0,1,B0
        NOP     4
    
```



Non 2ⁿ Buffer Example

coeff

a5
a4
a3
a2
a1
a0

B4 →

smpl

x(0)
x(1)
x(2)
x(3)
x(4)
x(5)

A4 →

x16 (le)

```

tapsize .set 6           ;# taps (count)
bufsize .set 8 * 2       ;pwr 2 > tapsize

FIR:     ZERO    A7
        MVK     tapsize-1, B0
loop:    LDH     *A4++, A5
        LDH     *+B4[B0], B5
        NOP     4
        MPY     A5, B5, A6
        NOP
        ADD     A6, A7, A7
[B0]     B       loop
        SUB     B0,1,B0
        NOP     4
    
```



Non 2ⁿ Buffer Example

coeff

a5
a4
a3
a2
a1
a0

B4 →

smpl

x(8)
x(3)
x(4)
x(5)
x(6)
x(7)

A4 →

x16 (le)

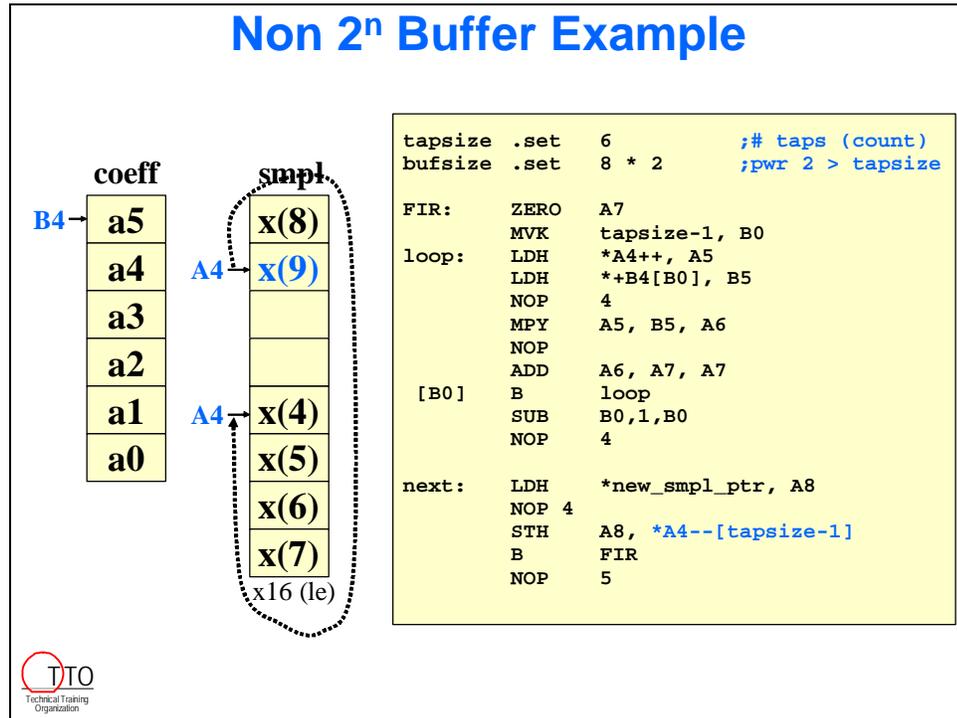
```

tapsize .set 6           ;# taps (count)
bufsize .set 8 * 2       ;pwr 2 > tapsize

FIR:     ZERO    A7
        MVK     tapsize-1, B0
loop:    LDH     *A4++, A5
        LDH     *+B4[B0], B5
        NOP     4
        MPY     A5, B5, A6
        NOP
        ADD     A6, A7, A7
[B0]     B       loop
        SUB     B0,1,B0
        NOP     4

next:    LDH     *new_smpl_ptr, A8
        NOP     4
        STH     A8, *A4--[tapsize-1]
        B       FIR
        NOP     5
    
```





Circular Addressing in Linear Assembly

Circular Addressing (.circ)

.circ x/B6, y/A7 ...

- ◆ **.circ** provides two effects:
 - ◆ maps symbol to register (same as .map)
 - ◆ circular declaration
- ◆ Register symbol must be mapped to a circular register
 - ◆ A4, A5, A6, A7
 - ◆ B4, B5, B6, B7
- ◆ Only valid inside a procedure (.cproc)
- ◆ Does not insert code for setting up circular addressing, it only maps the register to be used (as shown on next slide)



Circular Addressing in Linear Assembly

```
main.c
#define    tapsize 6        ;# taps (count)
#define    bufsize 8 * 2    ;pwr 2 > tapsize

#pragma DATA_SECTION(coeff, ".data")
short coeff[tapsize] = {a5,a4,a3,a2,a1,a0};

#pragma DATA_ALIGN(smpl, bufsize)
#pragma DATA_SECTION(smpl, "firBuf")
short smpl[bufsize];
```

```
fir.sa
        .sect    "code"
        .global  _coeff, _smpl
fCirc: .cproc   taps, bsize
        .reg     aData, cReg
        .circ    sReg/A4

setup:  MVKL     0001h, aData    ;AMR=00030001
        MVKLNH  0003h, aData
        MVC     aData, AMR
        MVKL    _smpl, sReg
        MVKH    _smpl, sReg
        MVKL    _coeff, cReg
        MVKH    _coeff, cReg
```

