

Contents

2.	Debugging and Evaluation	2-1
2.1.	Overview	2-1
2.2.	Instructions	2-1
2.2.1.	Debugging skills	2-1
2.2.2.	Software Quality Measurement	2-17
2.3.	Exercises	2-27
2.4.	Reference	2-28
2.5.	Appendix	2-29

2. Debugging and Evaluation

2.1. Overview

This Lab gives step-by-step instructions to perform a variety of debugging tasks and Software quality evaluation. Thought in this Lab the debugger target is ARMulator, but the skills can be applied to Multi-ICE/Angel with the ARM development board(s). The following instructions are based on the demonstration program that runs the **Dhrystone** test software, which is the same to that used in Lab 1. For details of the Dhrystone test program, please refer to the readme.txt file and the various source files in its subdirectory (e.g., [C:\Program Files\ARM\ADSV1_2\Examples\dhryansi\](#)).

Debugging skills you will learn

- Set breakpoints and watchpoints
- Locate, examine and change the contents of variables, registers and memory

The skills you will learn to evaluate software quality:

- Memory requirement of the program
- Profiling: Build up a picture of the percentage of time spent in each procedure.
- Evaluate software performance prior to implement on hardware

2.2. Instructions

2.2.1. Debugging skills

Initial setup

1. Make your working directory, e.g., C:\ARMSoC\Lab_02\
2. Copy all files in C:\Program Files\ARM\ADSV1_2\Examples\dhryansi\ to your working directory.
3. Double click on **dhryansi.mcp** in your working directory.
4. Make **dhryansi.mcp** and then Select **Project → Debug** (Ctrl + F5) to launch AXD.

- 4.1 A **Disassembly processor view** of the image is displayed and a blue arrow indicates the current execution point.
5. Select **Execute → Go** from the menu (or press F5, or from toolbar) to begin execution on the target processor.
 - 5.1 Execution stops at the beginning of function main(), where a breakpoint is set by default. A red disc indicates the line where a breakpoint is set.
 - 5.2 Also, a Source processor view of the relevant few lines of the relevant file is displayed. If it is not, right-click in the **Disassembly processor view** and select **Source** from the pop-up menu. Again, a red disc indicates the line where a breakpoint is set, and a blue arrow indicates the current execution point.
6. Select **Go** again to continue execution.
 - 6.1 You are prompted, in the **Console processor view**, for the number of iterations through the benchmark that you want to performe.
 - 6.2 **Enter 8000**. The program runs for a few seconds, displays some diagnostic messages, and shows the test results.
7. To repeat the execution of the program, select **Reload Current Image** from the **File menu** or toolbar shown in Figure 1, then repeat Steps 5 and 6.
 - 7.1 You do not have to open the **Source process view** again. Once opened, it remains displayed.

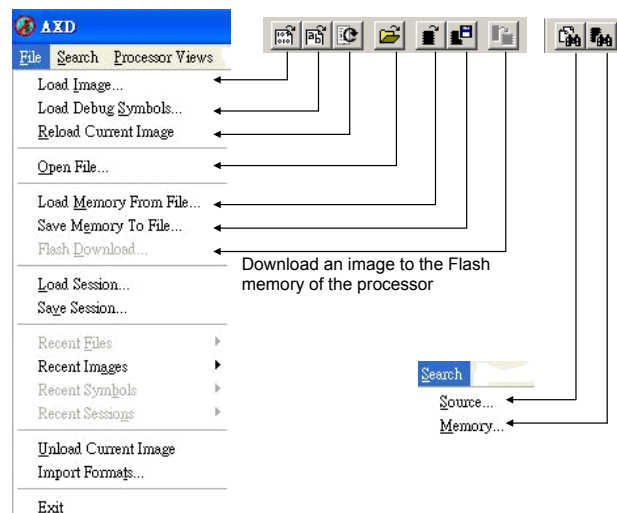


Figure 1. File and Search Menu, and their corresponding Toolbar.

Setting a breakpoint

1. Select **Reload Current Image**
2. Select **Go** to reach the first breakpoint, set by default at the beginning of function main() and indicated by a red dot. You can see the source file

dhry_1.c with a breakpoint and the current position indicated at line number 87.

3. Scroll down through the source file until line number 159 is visible. This is a call to **Proc_4()**. This process will be executed the number of times you specify.

3.1 Alternatively, you may use **Search in Source** by Search string set as “**Proc_4()**”, as shown in Figure 2.

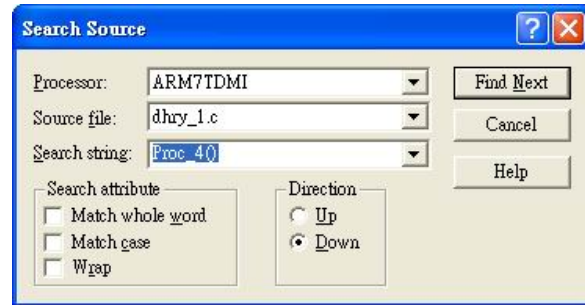


Figure 2. Search in Source.

4. **Right-click** on line 159 to position the cursor there and display the pop-up menu, and select **Toggle Breakpoint** (or left-click on the line and press **F9**, or double-click in the margin next to the line).

4.1 Another red dot indicates that you have set a second breakpoint.

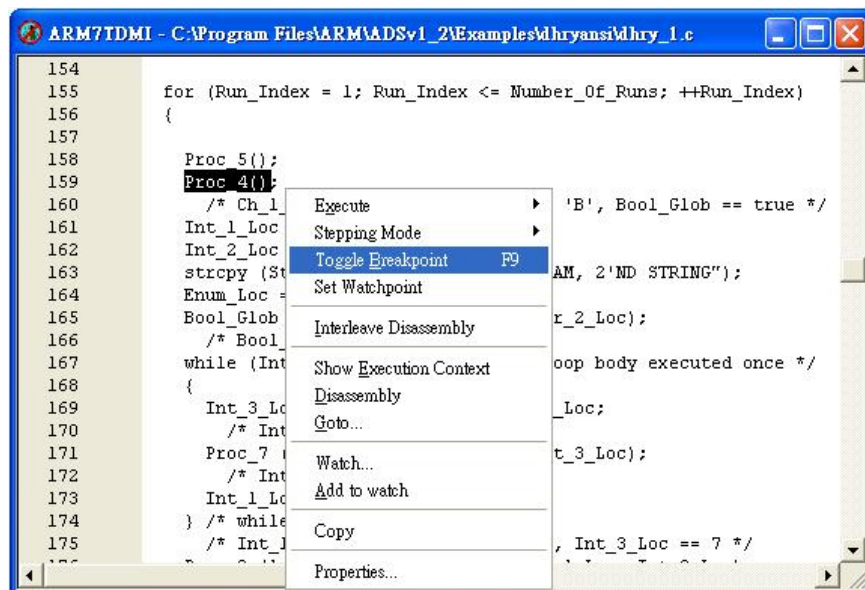


Figure 3. Set Breakpoint.

5. To edit the details of the new breakpoint, select **Breakpoints** from the **System Views** menu (Figure 4). The breakpoints panel is displayed (Figure 5).

- 5.1 Double-click on the line in the breakpoints panel that describes the new breakpoint, or right-click on it and select **Properties** (Figure 5), to display a **Breakpoint Properties** dialog.
- 5.2 Enter **750** in the **out** of field in the **Conditions box**, as shown in Figure 6. This is the number of times execution has to arrive at the breakpoint to trigger it.
- 5.3 Click OK.

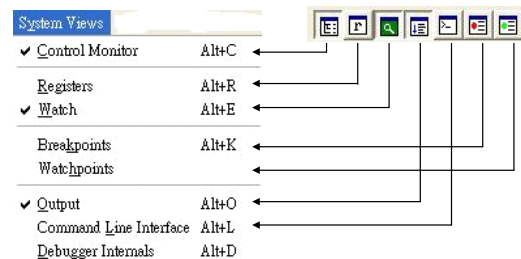


Figure 4. System View Menu and its corresponding Toolbar.

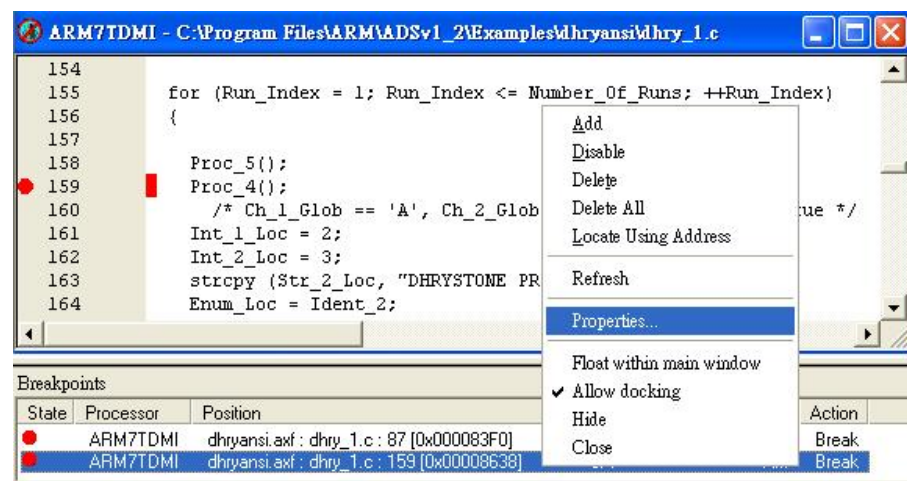


Figure 5. Breakpoints panel (the lower window).

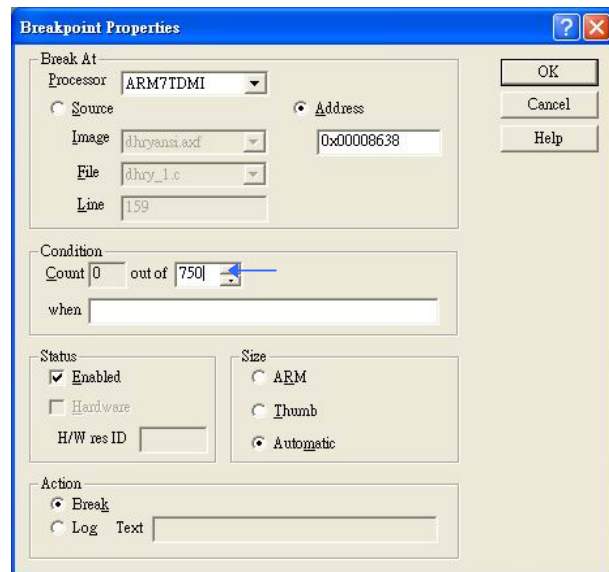


Figure 6. Setting breakpoint details.

6. Press **F5 (Go)** to resume execution, and enter the smaller number of 5000 this time for the number of iterations required.
 - 6.1 Execution stops when the **750th** time your new breakpoint is reached.
7. Select **Variables** from the **Processor Views menu** (Figure 7) to check progress. Reposition or resize the window if necessary.
 - 7.1 Click the **Local tab** and look for the **Run_Index** variable. Its value is shown as 0x2EE (hexadecimal).
 - 7.2 Right-click on the variable so that it is selected and a pop-up menu appears (Figure 8). Select **Format → Decimal** and the value is now displayed as 750 (decimal).

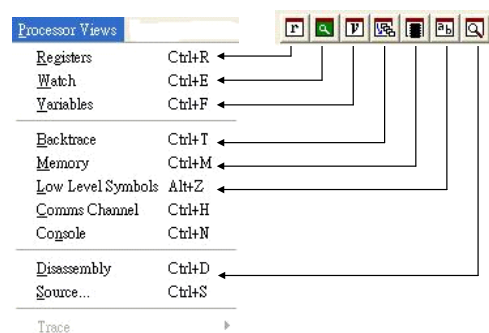


Figure 7. Processor View Menu and its corresponding Toolbar.

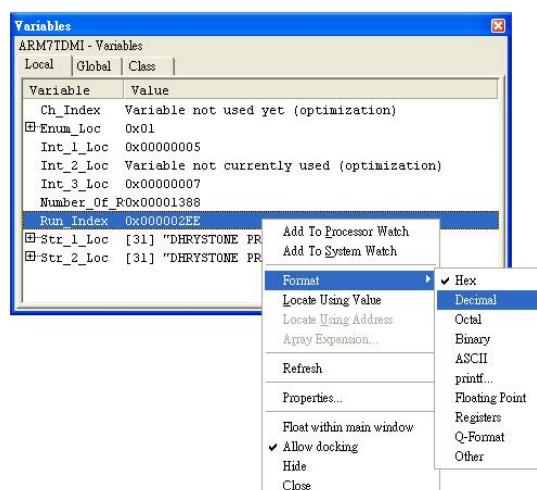


Figure 8. Pop-up menu for Local variable in Processor View.

8. Press **F5** to resume execution, and the value of the **Run_Index** local variable changes to **1500**. It is now (red) colored to show that its value has changed since the previous display.
9. Press **F5** repeatedly until the value of **Run_Index** reaches the highest multiple of 750 (1500→2250→3000→3750→4500) before exceeding your specified number of runs, then once more to allow the program to complete execution.
10. Close down the Breakpoints system view, either by right-clicking and selecting Close or by clicking on the Close button in the title bar if the view is not docked.

Setting a watchpoint

1. Select **Reload Current Image**
2. Select **Go** to reach the first breakpoint, set by default at the beginning of function main().
3. Select **Go** to continue execution.
4. When you are prompted for the number of iterations to execute, enter **770**. Execution continues until it reaches the breakpoint at line **159** for the 750th time. This is the breakpoint you defined in last section.
5. Select **Watchpoints** from the **System Views** menu/toolbar (Figure 4), right-click in the Watchpoints system view (Figure 9), and select **Add** to display the Watchpoint Properties dialog (Figure 10).
 - 5.1 Enter **Run_Index** in the **Item** field in the Watch box.
 - 5.2 Set the **out of field in the Conditions** box to a value of **6**. This is the number of times the watched value has to change to trigger the watchpoint action.

5.3 Click the **OK** button. Take a look at the changes in **Watchpoints view**.

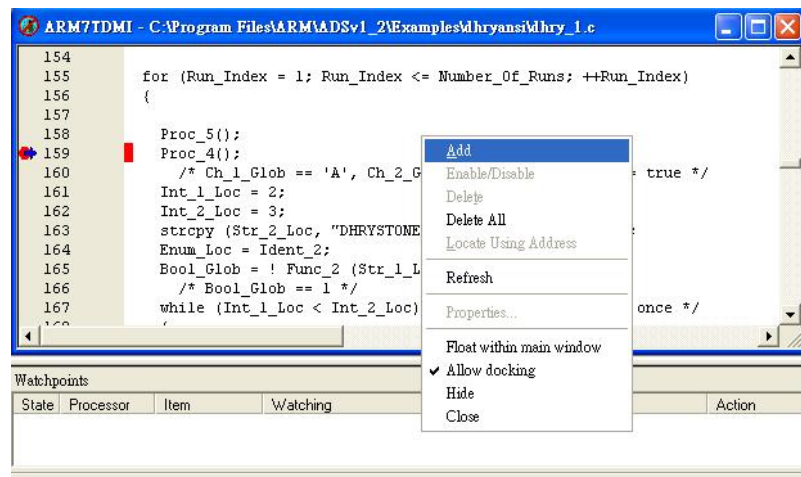


Figure 9. Watchpoints from the System Views.

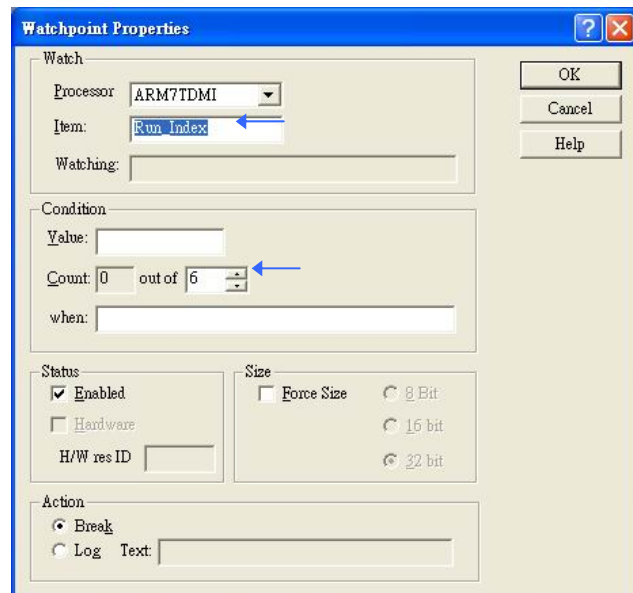


Figure 10. Watchpoint Properties dialog.

6. Show the **Run_Index** variable
 - 6.1 If the **Variables processor view** is not already displayed, select Variables from the Processor Views menu.
 - 6.2 Click the **Local** tab and look for the **Run_Index** variable. The value of **Run_Index** is currently **750**.
 - 6.3 If it is displayed in hexadecimal notation, right-click on the value and select **Format → Decimal** to change the display format to decimal.
7. Press **F5** to resume execution.

- 7.1 Soon the value of the **Run_Index** local variable changes to **756**. It is now displayed in red to show that its value has changed since the previous display. Execution stops.
8. Examine any displayed values, then press **F5** again to resume execution and perform **six** more runs.
 - 8.1 When the value of **Run_Index** becomes greater than the number of runs you specified (**770** at step four), the test results are displayed in console window and execution terminates.
9. Delete the watchpoint by right-clicking on its line in the **Watchpoints window** and selecting **Delete** from the pop-up menu, then close down the Watchpoints system view.

Examining the contents of variables

Two methods of examining the contents of variables are described in this section:

- Contents of variables (**variable processor view**): This method is simpler and shows only the contents of the specified variables.
- Addresses and contents of variables (**watch processor view**): This method shows the addresses of the variables as well as their contents.

1. Contents of variables

To examine the contents of variables as simply as possible, use the Variables processor view.

1. Select **Reload Current Image**
2. Select **Go** to reach the first breakpoint, set by default at the beginning of function main().
3. Select **Go** to continue execution.
4. When you are prompted for the number of runs to execute, enter **760**. Execution continues until it reaches the breakpoint at line **159** for the 750th time. This is the breakpoint you defined in Setting a breakpoint at the step of previous section.
5. If the **Variables processor view** is not already displayed, select Variables from the Processor Views menu. Reposition or resize the window if necessary. On the **Local** tabbed page, look for the **Run_Index** variable. Other variables that you can see include **Enum_Loc**, **Int_1_Loc**, **Int_2_Loc**, and **Int_3_Loc**.
 - 5.1 Right-click in the window, select **Properties... → Dec** and click **OK**. The display is now in decimal format and is similar to that shown in Figure 11.

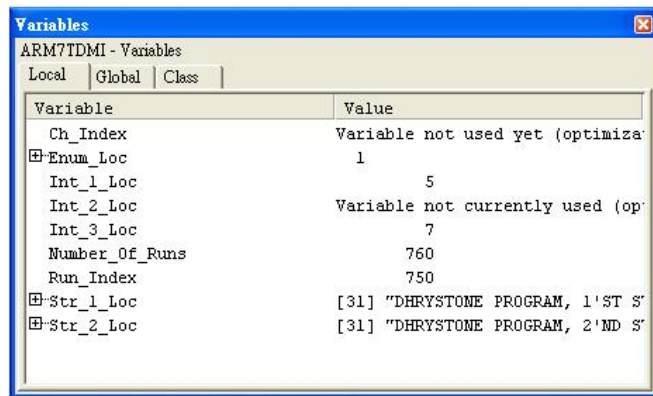


Figure 11. Examining the contents of variables.

6. Press **F10**. This is equivalent to selecting **Step** from the **Execute menu** (Figure 12). The program executes a single instruction and stops. Any values that have changed in the **Variables processor view** are displayed in red.

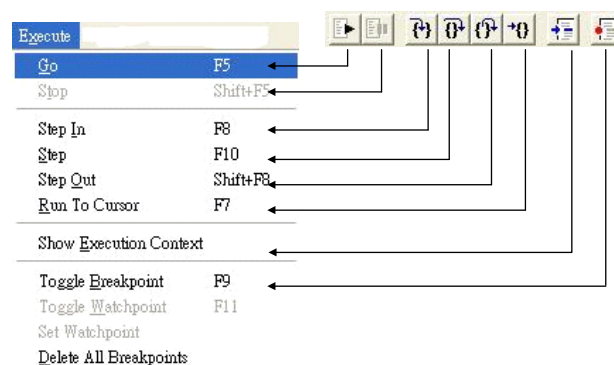


Figure 12. Execute Menu and its corresponding Toolbar.

7. Press **F10** repeatedly. As you execute the program, one instruction at a time, the values of several of the variables change. After you have allowed approximately **30** program instructions to execute, the value of **Run_Index** increases by **1**. The program has now completed one execution iteration of the Dhrystone test.
8. Explore the various display options available from the pop-up menu (Figure 8). Try some other settings in both the **Format** submenu and the **Default Display Options** dialog displayed when you select **Properties**....
9. Press **F5** to allow the program to complete its execution, then close down the **Variables processor view**.

II. Addresses and contents of variables

An alternative method of examining a variable is to use a **Watch processor view**. This allows you to see the memory address of the variable as well as its value.

1. Select **Reload Current Image**
2. Select **Go** to reach the first breakpoint, set by default at the beginning of function main().
3. Select **Go** to continue execution.
4. When you are prompted for the number of runs to execute, enter **760**. Execution continues until it reaches the breakpoint at line **159** for the 750th time.
5. Select **Watch** from the **Processor Views menu** (Figure 7) and reposition or resize the window if necessary. You can specify items to watch on several tabbed pages. In this example you examine a few variables using the first tabbed page only.
6. Right-click in the window, and select **Add Watch** from the pop-up menu (Figure 13). A **Watch dialog** appears, prompting you to enter an **expression**. For this example you enter some valid variable names, most of them preceded by an ampersand (&).
 - 6.1 Enter the first expression in the Expression field (as shown in Figure 14) by typing:
&Enum_Loc

-----Note-----

- **Enum_Loc** is a global variable, so it is stored in RAM at the address **&Enum_Loc**.
 - These names are **case-sensitive**.
 - You can also add a variable to the Watch view by selecting it in the source view using **right-clicking** and selecting **Watch**, as show in Figure 15. And then using the **Add Watch** pop-up menu command. ← Try this.
-

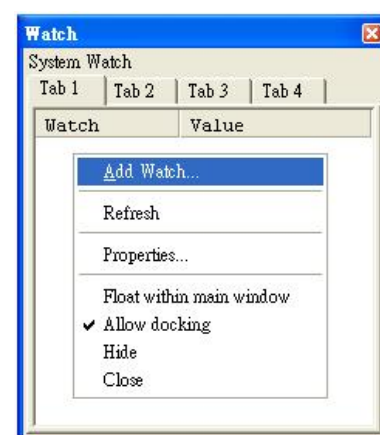
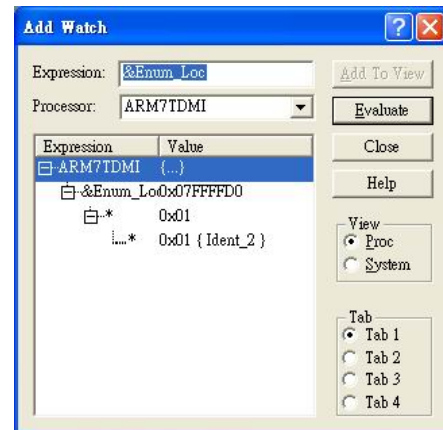
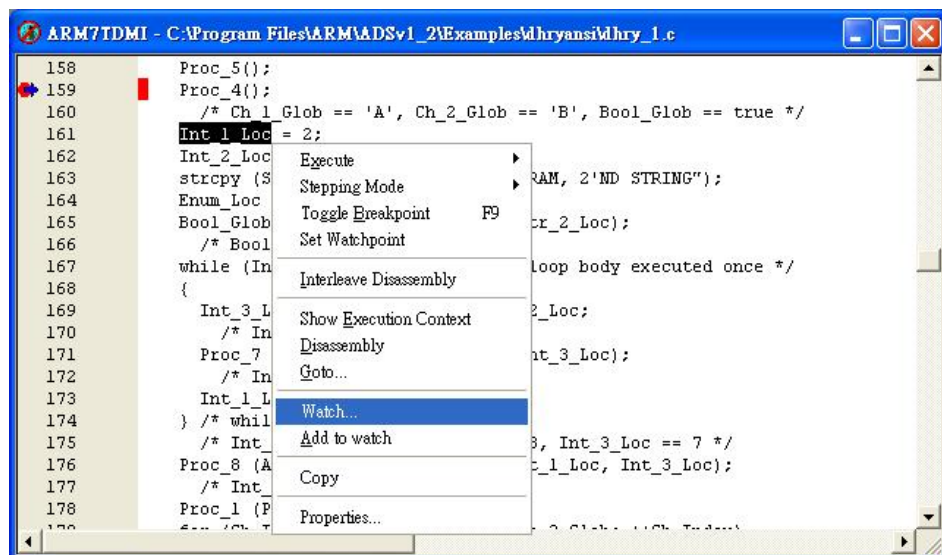


Figure 13. Add Watch to Processor Watch window.**Figure 14. Watch dialog.****Figure 15. Add a variable to the Watch view by selecting it in the source view.**

7. Press the **Enter** key or click on the **Evaluate** button. The expression you entered appears in the Expression column, and its value, being the address of the variable, appears in the Value column.
 - 7.1 Click on the + symbol to expand the display, and another line appears showing the contents of the variable in the Value column.
 - 7.2 Enter, in a similar way:
 - [&Int_1_Loc](#)
 - [&Int_3_Loc](#)
 - [Run_Index](#)
 - 7.3 Expand these lines. The result is shown in Figure 16.

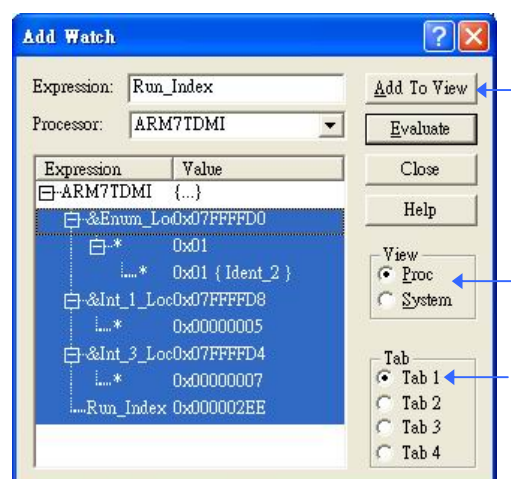


Figure 16. Specifying variables to watch.

8. Select all the lines you have entered, as shown in Figure 16, ensure that **Proc** is the selected View and **Tab1** the selected Tab, then click the **Add To View** button and the **Close** button.
9. The variables you have specified are now displayed in the **Watch processor view** (similar to that shown in Figure 17), and if you expand the lines you can see both the addresses and the contents of the variables.
 - 9.1 Point to the value displayed for the **Run_Index** variable and right-click to display the pop-up menu. Select **Format** → **Decimal** so that the value of **Run_Index** is displayed as a decimal number.

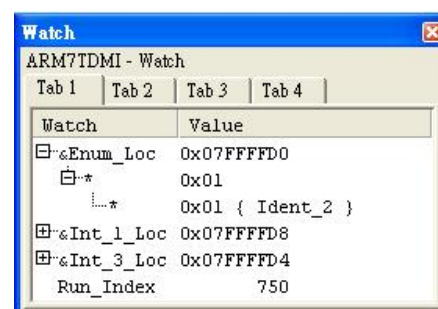


Figure 17. Watch processor view.

10. Press **F10**. This is equivalent to selecting Step from the Execute menu. The program executes a single instruction and stops. Any values that have changed in the Watch processor view are displayed in (red)color.
11. Press **F10** repeatedly. As you execute the program, one instruction at a time, the values of several of the variables change. After you have allowed approximately 30 program instructions to execute, the value of **Run_Index** increases by **1**. The program has now completed one further execution of the Dhrystone test.

12. Explore the various display options available from the pop-up menu. Try some other settings in both the **Format** submenu and the **Default** Display Options dialog displayed when you select **Properties**....
13. Press **F5** to allow the program to complete its execution, then close down the **Variables processor view**.

Examining the contents of registers and memory

1. Examining the contents of registers

To examine the contents of registers used by the currently loaded program:

1. Select **Reload Current Image**
2. Select **Go** to reach the first breakpoint, set by default at the beginning of function main().
3. Select **Registers** from the **Processor Views menu** (Figure 7) and reposition or resize the window if necessary.
 - 3.1 The registers are arranged in groups, with only the group names visible at first. Click on the **+** symbol of any group name to see the registers of that group displayed. An example is shown in Figure 18.

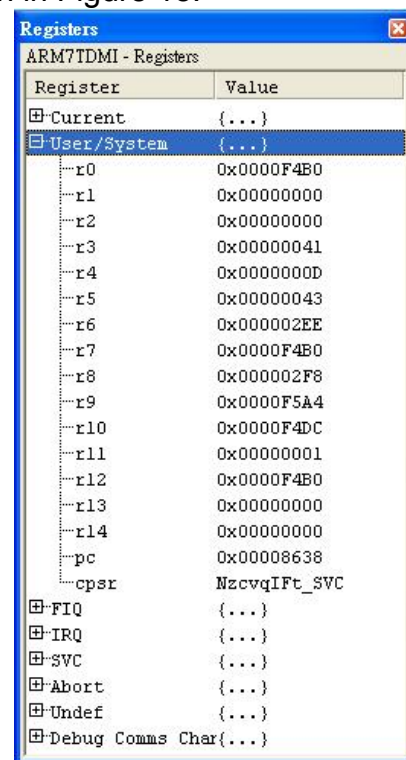


Figure 18. Examining contents of registers.

4. Press **F10**. This is equivalent to selecting Step from the Execute menu/toolbar (Figure 12). The program executes a single instruction and

stops. Any values that have changed in the Registers processor view are displayed in red.

5. Press **F10** a few more times. As you execute the program, one instruction at a time, you can see the values of some registers change.
 - 5.1 You soon reach the point when you are prompted, in the Console processor view, for the number of runs to perform. A very small number (e.g., **300**) is sufficient this time.
6. Explore the format options available from the **Registers processor view** pop-up menu.
 - 6.1 If you position the mouse pointer on a selectable line, right click will select the line. You can change the display format of the selected lines only.
 - 6.2 You can select multiple lines by holding down the **Shift** or **Ctrl** keys while you click on the relevant lines, in the usual way.
 - 6.3 If you select **Add to System** from the pop-up menu, the currently selected register is added to the **Registers system view** window. This is particularly useful when your target has **multiple processors** and you want to examine the contents of some registers of each processor. Collecting the registers of interest into a single Registers system view avoids having to display many separate processor views.
 - 6.4 You can also select **Add Register** from the pop-up menu of the **Registers system view**. This allows you to select registers from any processor to add to those being displayed in the **Registers processor view**.
7. Press **F5** to allow the program to complete its execution, then close down the Registers processor view.

II. Examining the contents of memory

To examine the contents of memory used by the currently loaded program:

1. Select **Reload Current Image**
2. Select **Go** to reach the first breakpoint, set by default at the beginning of function main().
3. Select **Go** to continue execution.
4. When you are prompted for the number of runs to execute, enter **760**. Execution continues until it reaches the breakpoint at line **159** for the 750th time.
5. Select **Memory** from the **Processor Views menu** (Figure 7).
 - 5.1 Addresses and contents of variables in Figure 19 shows that addresses of interest are in the region of **0x07FFFFD0**, so set the **Start address** value to, say, **0x07FFFF00**.

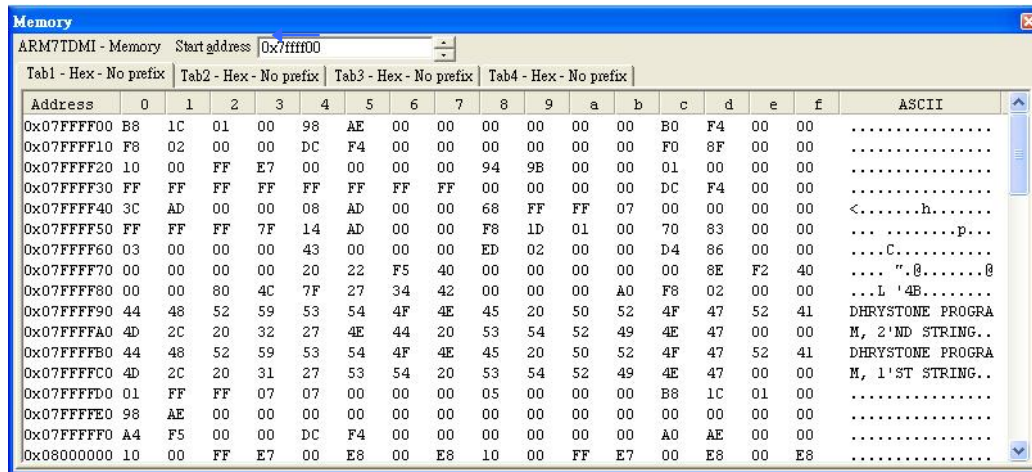


Figure 19. Examining contents of memory.

- Press **F10** (or **Step** from the Execute menu). The program executes a single instruction and stops. Any values that have changed in the **Memory processor view** are displayed in red.
- Press **F10** a few more times. As you execute the program, one instruction at a time, you can see the values stored in several of the memory addresses change.
- Explore the format options available in the **Memory processor view** pop-up menu. Size settings appear both on the pop-up menu and in the dialog displayed when you select **Properties...** from the pop-up menu.

III. Locating and changing values and verifying changes

To locate a value (of a variable or string, for example) in memory and change it:

- Select **Reload Current Image**
- Select **Go** to reach the first breakpoint, set by default at the beginning of function main().
- Select **Memory** from the **Search menu** (Figure 1). A window shown in Figure 20 appears.
 - Enter **2'ND** in the **Search** for field, set the **In range** and **to** addresses to **0x0** and **0xFFFF**,
 - Select **ASCII** for the **Search string type**, and click the **Find** button. A **Memory processor view** opens and shows the contents of an area of memory, with the string you specified highlighted. Reposition, resize and/or adjust the resolution of the window if necessary.
To see a display similar to that in Figure 21, You might have to right-click in the window to display the pop-up menu and set **Size** to **8 bit** and **Format** to **Hex - No prefix**.
 - Click **Cancel** to close Search Memory Window.

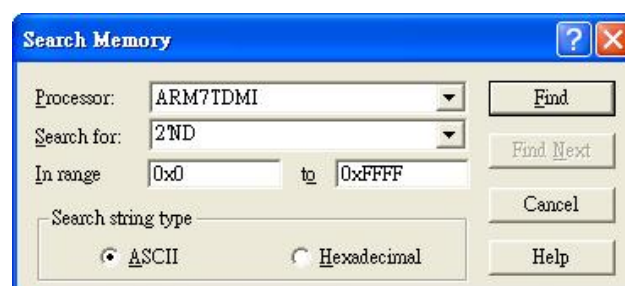


Figure 20. Search for a string in memory.

Address	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f	ASCII
0x000089D0	45	20	50	52	4F	47	52	41	4D	2C	20	32	27	4E	44	20	E PROGRAM, 2'ND
0x000089E0	53	54	52	49	4E	47	00	00	44	48	52	59	53	54	4F	4E	STRING..DHRTSTON
0x000089F0	45	20	50	52	4F	47	52	41	4D	2C	20	33	27	52	44	20	E PROGRAM, 3'RD
0x00008A00	53	54	52	49	4E	47	00	00	45	78	65	63	75	74	69	6F	STRING..Executio
0x00008A10	6F	20	65	6F	6A	73	0A	00	46	69	6F	61	6C	20	76	61	n ends Final va

Figure 21. Search for string in memory (2'ND).

4. In **Memory processor view**, the four hexadecimal values highlighted are **32 27 4E 44**.
 - 4.1 An example of entering a **hexadecimal value**: Double-click on the value **32** and type **0x4E** and press **Enter**. The corresponding change in ASCII column will be **"2'ND"** → **"N'ND"**.
 - 4.2 An example of entering an **ASCII value**: Double-click on the value **27** and type **"o"** (a double quote followed by a lowercase letter o) and press **Enter**. The corresponding change in ASCII column will be **"N'ND"** → **"NoND"**.
 - 4.3 An example of entering a **decimal value**: Double-click on the value **4E** (the one before **44**) and type **46** and press **Enter**. The corresponding change in ASCII column will be **"NoND"** → **"No.D"**.
 - 4.4 An example of entering an **octal value**: Double-click on the value **44** and type **o62** and press **Enter**. The corresponding change in ASCII column will be **"No.D"** → **"No.2"**.
5. Press **F5** (or **Go**) to continue execution, and enter a value of, say, **100** when you are prompted in the Console processor view for the number of runs to perform.

When the program displays its messages in the **Console Window** after completing its tests you can see that one of the lines that in earlier examples included the text **2'ND STRING** now has **No.2 STRING** instead because of the change you made.

6. Close AXD and CodeWarrior

2.2.2. Software Quality Measurement

Memory requirement of the program

1. Double click on **dhryansi.mcp**, and then **Make dhryansi.mcp**
 - 1.1 A compiling and linking status window would appear to indicate making progress.
 - 1.2 After finishing compiling and linking, the **Errors and Warnings window** would appear, as shown in Figure 22.

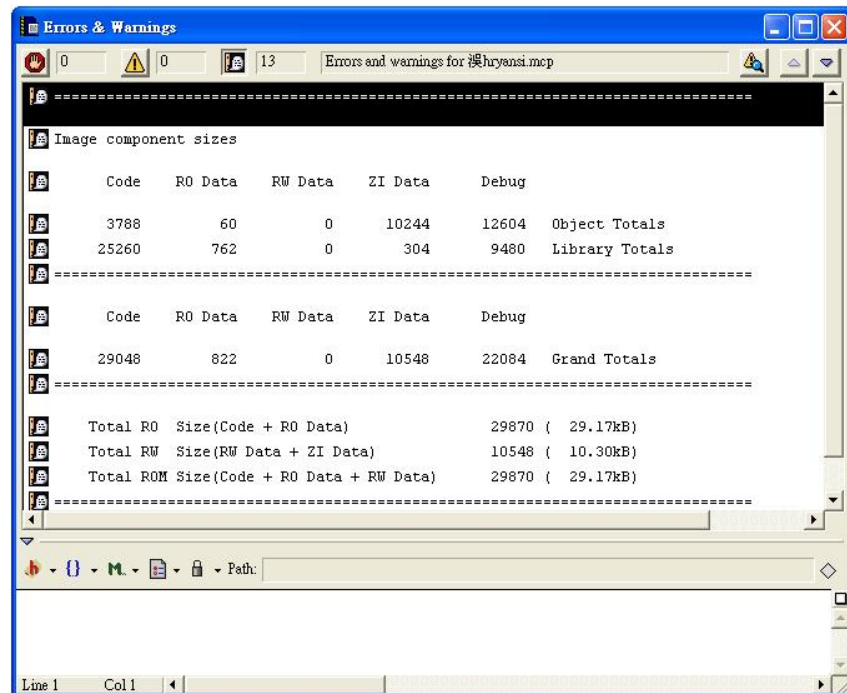


Figure 22. Show Code and Data Size by using -info totals.

2. Select **DebugRel Settings** from **Project Window**
 - 2.1 Change the ARM Linker option from **-info totals** to **-info sizes** in Equivalent Command Line, as shown in Figure 23.
 - 2.2 Make the project again and then check the **Errors and Warnings window**. A much more detailed memory requirement for each object file and library file is listed.

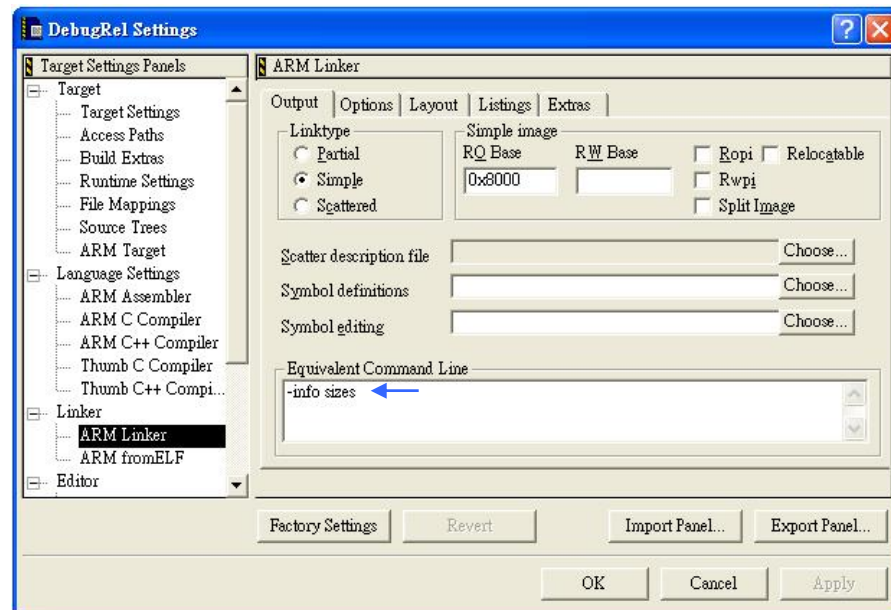


Figure 23. DebugRel Settings for ARM Link output

Profiling

1. After making the project, launch AXD Debugger. Then
 - 1.1 Select **File** → **Load Image** to load image file from C:\ARMSoC\Lab_02\dhryansi_Data\DebugRel\dhryansi.axf.
 - 1.2 Check the **Profile** checkbox, as shown in Figure 24.

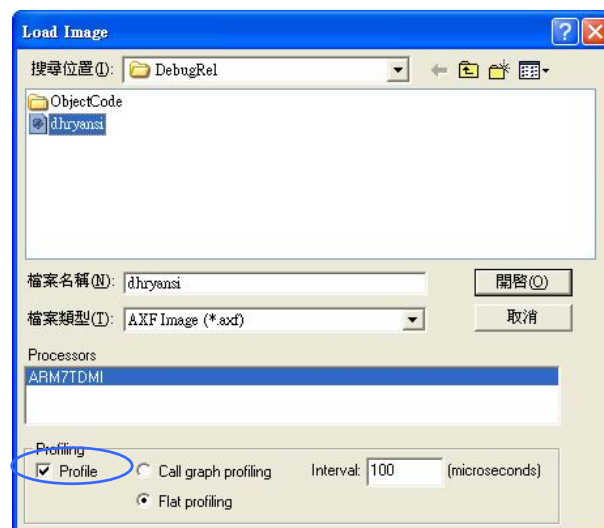


Figure 24. Load image with Profiling functionality.

- 1.3 Select **Options** → **Profiling** → **Toggle Profiling** if necessary to ensure that Toggle Profiling is checked in the Profiling submenu of the Options menu.
 - 1.3.1 Select call graph profiling or flat profiling as your need.

- 1.4 Select **Options → Profiling → Clear Collected** to clear previous profiling data if necessary.
- 1.5 Execute your program (Hit the **Go** button). Find your source file if asked, and then Hit **Go** button again.
- 1.6 Type **8000** in the **Console Window** when be asked.
- 1.6 When the program terminates, select **Options → Profiling → Write to File**.
- 1.7 A **Save** dialog appears. Enter a file name (e.g., **dhryansi**) and specify a directory if necessary. Click the **Save** button.
- 1.8 Launch DOS command line window and change to the directory you specified in last directory. Type **armprof dhryansi.prf** under the DOS command line to view the profiling information. Save the information. It will be used later.

-----Note-----

- You cannot display profiling information in AXD. Use the Profiling functions on the Options menu to capture profiling information, then use the **armprof** command-line tool.
 - If you want to save the profiling information. Type **armprof filename.prf > filename.log** in command line. Profiling information will save into the filename.log
-

To collect information on a specific part of the execution:

1. Load (or reload) the program with profiling enabled.
2. Open the Source Window (Ctrl+S, and then choose dhry_1.c). Set a breakpoint at the beginning of the region of interest (e.g., start of for loop at line number **155**), and another at the end (e.g., the end of for loop at line number **199**, then a breakpoint icon will be shown at line number 213).
3. Execute the program and type **8000** in the **Console Window** when be asked in **Console**.
4. Execute the program as far as the beginning of the region of interest. Clear any profiling information already collected by selecting **Options → Profiling → Clear Collected**, and ensure that **Toggle Profiling** is checked.
5. Execute the program as far as the breakpoint at the end of the region of interest.
6. Select **Options → Profiling → Write to File** and specify the name of a file in which to save the profiling information.
7. Compare this result with the one you saved before.

Performance benchmarking

This section is based on ARM Application Note 93: Benchmarking with ARMulator, March 2002.

1. Cycle counting example: Dhrystone using the ARM7TDMI

1. If necessary, select **File → Load Image** to load image file from **C:\ARMSoC\Lab_02\dhryansi_Data\DebugRel\dhryansi.axf**.
2. Within AXD **select Options → Configure Target...**

- 2.1 Select **ARMUL** as the target and click on the **Configure** button.
- 2.2 Select the **ARM7TDMI** as the processor variant, select the **debug endian** as **start target endian**, and ensure that the check box for **Floating Point Emulation** is cleared, then click **OK**.
- 2.3 Choose **OK** in the configuration dialog.
- 2.4 Click **Yes** when asked to reload the last image
3. Select **Processor Views → Low Level Symbols** and locate **Proc_6** in the Low Level Symbols window.
 - 3.1 Right-click on it and select **Locate Disassembly**.
 - 3.2 Place a breakpoint on this line (**Proc_6**) in the **Disassembly window**.
4. Click on the **Go** button (or press F5) to begin execution, the program will run to **main**.
 - 4.1 Click on **Go** again, the program will run, when prompted, request at least **two** runs through Dhrystone. The program will then run to the breakpoint at **Proc_6** and stop.
5. Select **System Views → Debugger Internals** and click on the **Statistics tab** in the Debugger Internals window.
 - 5.1 Right-click in the Statistics pane and select **Add New Reference Point**.
 - 5.2 Enter a suitable name (e.g., **Proc_6_cycle**) when prompted and click on **OK**.
6. Click on the **Go** button.
 - 6.1 When the breakpoint at **Proc_6** is reached again, the contents of the reference point are updated to reflect the number of instructions and cycles consumed for one iteration of the loop.
 - 6.2 The result shown in Console window also reveals some information about running the benchmark program

II. Estimate the execution time

1. Clear all breakpoints. (**Alt+K → Delete All**)
2. **Select Options → Configure Target...** then click on the **Configure** button.
3. In ARMulator Configuration window, mark the clock as **Emulated** and set **Speed** as **10MHz**
4. Reload the executable image
5. Click on the **Go** button.
6. When prompted, request **30000** (don't use 30,000) runs through Dhrystone.
7. Check the result on **Console window**.

The information reveals the **Microseconds for one run through Dhrystone** (the smaller the better) and **Dhrystones per Second** (the larger the better). Record these values.

8. Check internal variable **\$sys_clock**, which records the number of **centiseconds** since the simulation started.
 - 8.1 To display this value, select **System Views → Debugger Internals → Internal Variables**)
 - 8.2 You may change the format of **\$sys_clock** to decimal. Record this value, said sys_time.
 9. Check the total cycle count **.Total** shown in the **Statistics tab** .
 - 9.1 The **execution time = Cycle count / Cycle Frequency**. As we set the bus frequency to 10MHz, we can calculate the total execution time = (total cycle count/ (10×10⁶)) in **second**. Then (**sys_time/100**) should approximate to **cyc_time**.
 10. Reload the executable image, repeat the steps **5~7** except that request **40000** runs through Dhrystone. The results shown on **Console window** should be the same that in step 6.
 11. Reload the executable image, repeat the steps **2~7** except that set the **Emulated Speed** as **20MHz**.
 - 11.1 What message is shown on **Console window**?
 12. Repeat the actions at step 11 except that request **60000** runs through Dhrystone.
 - 12.1 What is the difference between this result and the result at step 7?
- Note-----
- If the system clock is set to Real-time, then **\$sys_clock** will return actual time using the host computer's real-time clock rather than simulated execution time. This will benchmark the performance of the host computer!
 - Note that entering a speed without specifying units assumes for example 50 assumes 50Hz rather than 50MHz. Speeds given in kHz and GHz are also acceptable.
-

III. Performance estimation using different Memory models

The default setting for the ARMulator is to model a system with 4GB of zero wait state 32bit memory. However, real systems are unlikely to have such an ideal memory system! Hence an alternative memory model called **mapfile** can be used. The mapfile memory model reads a memory description file called a map file which describes the type and speed of memory in a simulated system.

-----Note-----

- ARMulator accepts a map file of any name. The file must have the extension **.map** or **.txt** for the browse facility to recognize it; however, any extension may be used if you are entering the path and filename explicitly in the map file text entry field.
-

To calculate the number of wait states for each possible type of memory access, the ARMulator uses the *values supplied in the map file* and the *clock frequency specified to the model*.

For cached cores, the clock frequency specified is the *core clock frequency*. The *bus clock frequency = core clock frequency / MCCFG*. The derived bus clock frequency is used to calculate *wait states* in cached cores.

-----Note-----

- ARMulator constant - *MCCFG* is specified in *install_directory\Bin*.ami*. If there is no other processor specified, *Default.ami* will be used. The default setting in *default.ami* is *MCCFG=3*. See *ARM® Developer Suite Debug Target Guide* for more information.

In the following steps, we will use *armsd.map* located at *C:\Program Files\ARM\ADSV1_2\Examples\dhry* as the map file. This map file describes a system

00000000 80000000 RAM 4 RW 135/85 135/85

- Memory section: start at address 0x0, length 0x80000000 bytes
 - labeled as RAM, has a 4-byte(32-bit) bus
 - read and write access
 - read access times of 135ns nonsequential and 85ns sequential
 - write access times of 135ns nonsequential and 85ns sequential
1. Clear all breakpoints
 2. **Select Options → Configure Target...** then click on the **Configure** button.
 - 2.1 Mark the clock as **Emulated** and set **Speed** as **10MHz**
 - 2.2 Specify the **Map File** through browsing to *C:\Program Files\ARM\ADSV1_2\Examples\dhry\armsd.map*
 4. Press OK's and then Reload the executable image.
 5. Click on the **Go** button.
 6. When prompted, request **30,000** runs through Dhrystone.
 7. Check the result on **Console window**.

The information reveals the *Microseconds for one run through Dhrystone* (the smaller the better) and *Dhrystones per Second* (the larger the better). Record these values and compare with those values you recorded at step 7 in Section II. Estimate the execution time.
 8. Check internal variable **\$sys_clock** and compare with that you got at step 8 in Section II. Estimate the execution time. Remember the data format should be the same, i.e., decimal. The performance should be worse.
 9. Read the memory statistics

- 9.1 Open **Command Line Interface Window** (ALT+L)
- 9.2 Enter command **di** (short form of **dbginternal**), and press any key (e.g., enter) until **\$memstates** are displayed. In this case, only single memory is used and therefore **\$memstates[0]** is displayed. You can also read memory statistics in **Debugger Internals**.

IV. Benchmarking cached cores

1. Edit **default.ami** located at **C:\Program Files\ARM\ADSV1_2\Bin**
 - 1.1 If **MCCFG**≠3, set **MCCFG=3**, quit AXD and launch it again.
2. **Select Options** → **Configure Target...** then click on the **Configure** button.
 - 2.1 Select Processor variant as **ARM940T**
 - 2.2 Mark the clock as **Emulated** and set **Speed** as **10MHz**
 - 2.3 Specify the memory file through browsing to **C:\Program Files\ARM\ADSV1_2\Examples\dhry\armsd.map**
3. After the step 2, check the message shown in ARMulator startup banner: **System Output Monitor - RDI Log**. An example result is displayed in Figure 25.

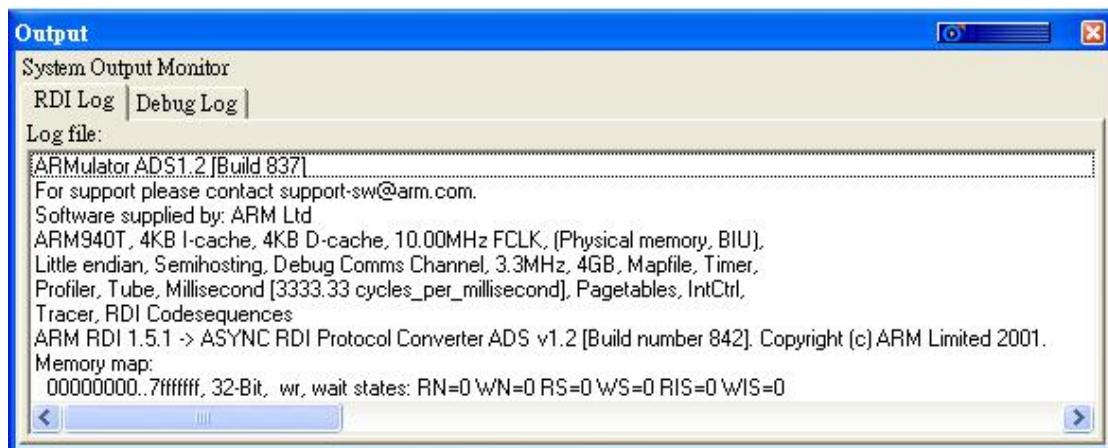
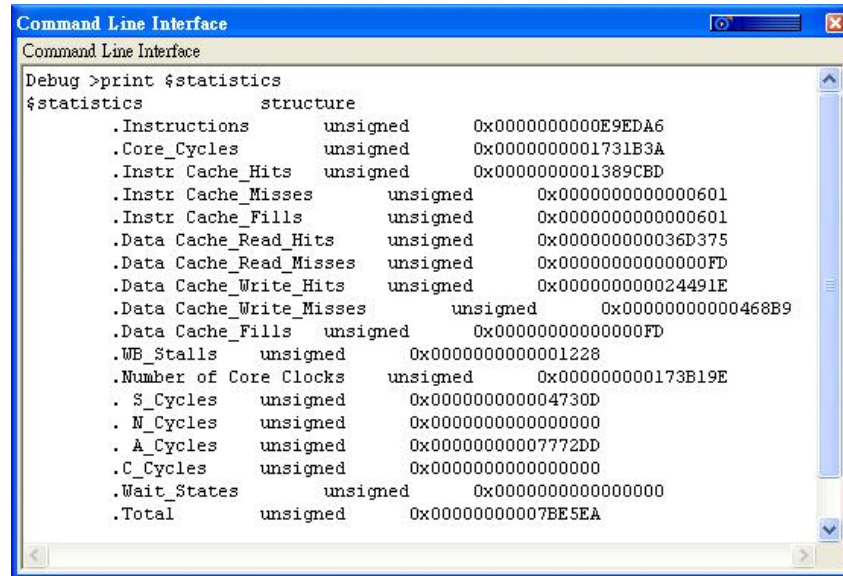


Figure 25. ARMulator startup Message.

4. Load **C:\ARMSoC\Lab_02\dhryansi_Data\DebugRe\dhryansi.axf**.
5. Click on the **Go** button.
6. When prompted, request **40000** runs through Dhrystone.
7. When the program is terminated, open **Command Line Interface Window** and enter **print \$statistics**. An example result is displayed in Figure 25.



```

Command Line Interface
Command Line Interface
Debug >print $statistics
$statistics      structure
.Instructions      unsigned      0x000000000000E9EDA6
.Core_Cycles       unsigned      0x00000000001731B3A
.Instr Cache_Hits  unsigned      0x00000000001389CED
.Instr Cache_Misses unsigned      0x00000000000000601
.Instr Cache_Fills unsigned      0x00000000000000601
.Data Cache_Read_Hits unsigned    0x0000000000036D375
.Data Cache_Read_Misses unsigned    0x00000000000000FD
.Data Cache_Write_Hits unsigned    0x0000000000024491E
.Data Cache_Write_Misses unsigned    0x000000000000468B9
.Data Cache_Fills  unsigned      0x00000000000000FD
.WB_Stalls         unsigned      0x0000000000001228
.Number of Core Clocks unsigned    0x0000000000173B19E
.S_Cycles          unsigned      0x000000000004730D
.N_Cycles          unsigned      0x0000000000000000
.A_Cycles          unsigned      0x000000000007772DD
.C_Cycles          unsigned      0x0000000000000000
.Wait_States       unsigned      0x0000000000000000
.Total            unsigned      0x000000000007BE5EA
  
```

Figure 26. Brief statistics.

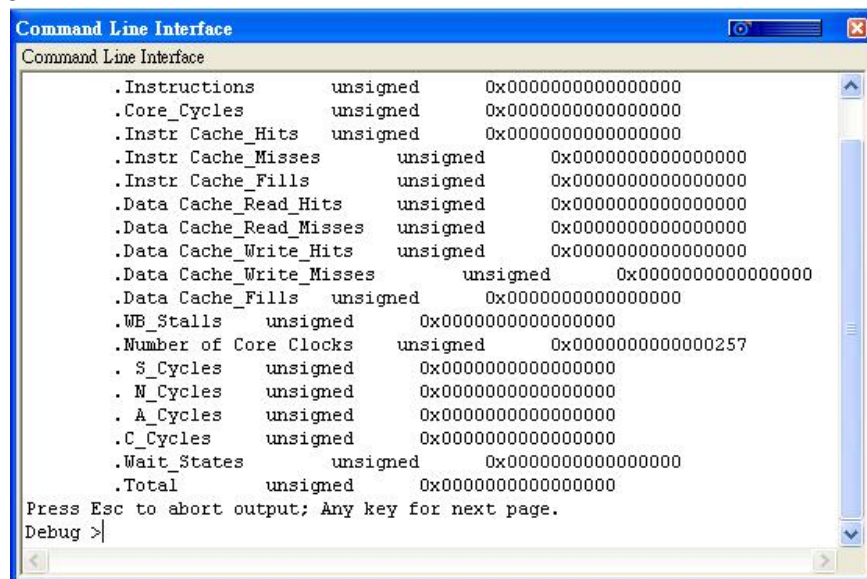
8. Edit *default.am*

- 8.1 For ADS 1.2, set Counters=True after the line setting MCCFG=3;
for ADS 1.1, add Counters=True after the line setting MCCFG.

Choose 8.2a or 8.2b step:

- 8.2a **Select Options**→**Configure Target...**then click on the **Configure** button. Select **OK**
- 8.2b Quit AXD and then restart it.

9. Open **Command Line Interface Window** and enter **print \$statistics**. Additional statistics for cached core is displayed, as one example displayed in Figure 27. Because we do not start the execution of the program, all values are zero.



```

Command Line Interface
Command Line Interface
.Instructions      unsigned      0x0000000000000000
.Core_Cycles       unsigned      0x0000000000000000
.Instr Cache_Hits  unsigned      0x0000000000000000
.Instr Cache_Misses unsigned      0x0000000000000000
.Instr Cache_Fills unsigned      0x0000000000000000
.Data Cache_Read_Hits unsigned    0x0000000000000000
.Data Cache_Read_Misses unsigned    0x0000000000000000
.Data Cache_Write_Hits unsigned    0x0000000000000000
.Data Cache_Write_Misses unsigned    0x0000000000000000
.Data Cache_Fills  unsigned      0x0000000000000000
.WB_Stalls         unsigned      0x0000000000000000
.Number of Core Clocks unsigned    0x00000000000000257
.S_Cycles          unsigned      0x0000000000000000
.N_Cycles          unsigned      0x0000000000000000
.A_Cycles          unsigned      0x0000000000000000
.C_Cycles          unsigned      0x0000000000000000
.Wait_States       unsigned      0x0000000000000000
.Total            unsigned      0x0000000000000000
Press Esc to abort output; Any key for next page.
Debug >|
  
```

Figure 27. Cached core additional statistics.

10. Click on the **Go** button.
 - 10.1 Set a breakpoint on the line **158** of the source file, the **Proc_5()**.
 - 10.2 Select **System Views → Debugger Internals → Statistics**.
11. Click on the **Go** button.
 - 11.1 When prompted, check the values at **Statistics tab**.
 - 11.2 Right Click on **Statistics tab** and select **Add New Reference Point**. Enter **iter_1** in the pop-up window. The new reference point will appear with zero values.
 - 11.2 Request **40000** runs through Dhrystone in the **Console Window**.
12. When the debugger halts at the breakpoint, check the values of **iter_1** and record **Total cycle** count.
 - 12.1 Add another new reference point, named as **iter_2**.
 - 12.2 Resume the program.
13. When the debugger halts at the breakpoint, check the values of **iter_2** and record **Total cycle** count.
 - 12.1 Add another new reference point, named as **iter_3**.
 - 13.2 Resume the program.
14. When the debugger halts at the breakpoint, check the values of **iter_3** and record **Total cycle** count.
 - 14.1 Clear or disable the breakpoint on the line **158** and resume the program.

The change of the total cycle of iter_1, iter_2 and iter_3 could be 10281 → 849 → 317. For the first iteration of the loop, the loop instructions and data would not be held in the cache memory, hence there are many cache misses and the total cycle is large. After several iterations, the Dhrystone loop will be held in cache memory and therefore the total cycle for each iteration is reduced.

Efficient C programming

1. Edit a copy of **loop.c** shown in Figure 28. Build it by using **ARM Executable Image Project** template and then record its memory requirement.

```
1  #include <stdio.h>
2
3  int acc(int n) {
4      int i; //loop index
5      int sum=0;
6
7      for (i=1; i<=n ;i++)
8          sum+=i;
9      return sum;
10
11 }
12
13
14 int main(void) {
15     int acc_val;
16
17     acc_val=acc(10);
18     printf("%d\n",acc_val);
19
20     return 0;
21
22 }
```

Figure 28. loop.c

2. Load the executable image of **loop.c** into AXD.
 - 2.1 Open **Processor Register View** and extend the **Current Register**.
 - 2.2 **Select Options** → **Configure Target...** then click on the **Configure** button.
 - 2.3 Select Processor variant as **ARM7TDMI**
 - 2.4 Mark the **Clock** as **Real-time**
 - 2.5 Specify the **Memory Map File** as **No Map File**
3. Stepping Mode in Strong Source
 - 3.1 Right click on **Disassembly window** and set **Stepping Mode** in **Strong Source**.
 - 3.2 Set the format of **r0** as **Decimal**.
 - 3.3 **Step in** through the program, check how the argument is passed to **acc()**. (**r0** is changed to 10)
 - 3.4 During the execution of **acc()**, which registers are used?
 - 3.5 Check which register is used to pass result back to **main()**. (**sum** is passed through **r0**)
4. Click **Go** button to finish the rest of the program.
- 5 Reload the image.
 - 5.1 Set two breakpoints, one on **for (i=0; i<=n ;i++)** and the other on **return sum;**
 - 5.2 Click **Go** button. When the program halt at breakpoint on **for (i=0; i<=n; i++)**, add a new reference point, named as **loop_time**.
 - 5.3 Click **Go** button again. Record the **Total** cycle count as **loop_time** when the program halt at breakpoint on **return sum**.

- 5.4 Click **Go** button to finish the rest of the program. Record the **Total** cycle count of **&statistics**.
6. Copy **loop.c** to a new file named as **loop_opt.c**
 - 6.1 Change the statement **for (i=1; i<=n ;i++)** to **for (i=n; i!=0 ;i--)**
 - 6.2 Build it and compare its memory requirement with **loop.c**
7. Load the executable image of **loop_opt.c** into AXD.
 - 7.1 Repeat step 3 and 4. Compare the results with those of **loop.c**.
 - 7.2 Repeat step 5. Compare the result with that of **loop.c**.

2.3. Exercises

Analyze the Dhrystone benchmark according to following lab requests
(Dhrystone program: dhry_1.c, dhry_2.c, dhry.h)

1. Compile the Dhrystone benchmark program using ARM and Thumb instructions according to the setting below. Record the information of code size and performance.

Code size : ROM size

Performance: instruction count, Core_Cycles, Total_Cycles,
\$sys_clock, Dhrystones per Second

- Setting:
 - Default target : Release
 - ARMulator processor: ARM 7TDMI
 - ARMulator clock : Emulated (10MHz, 50MHz, 100MHz)
 - Memory map file : armsd.map
 - Number of run : 100000 times

Please compare the code size and performance of using ARM/Thumb instructions, and explain the result.

2. Compile the Dhrystone benchmark program to different ARM core and different emulated speed using ARM and Thumb instructions according to the setting below. Record the information of code size and performance.

Code size : ROM size

Performance: \$sys_clock, Core_Cycles, Total_Cycles,
Cache Efficiency, Instructions count, Instr Cache_Misses,
Data Cache_Read_Misses, Data Cache_Write_Misses

- Setting:
 - Default target : Release
 - ARMulator processor: ARM940T
 - ARMulator clock : Emulated (10MHz, 50MHz, 100MHz)
 - Memory map file : armsd.map
 - Number of run : 50000, 100000, 200000 times

Please compare the code size and performance of using ARM/Thumb instructions, and explain the result.

3. According to the setting below, minimize the code size and enhance the performance using skills of interworking and coding style to optimize the Dhrystone benchmark program, and record the result of code size and performance.

Code size : ROM size

Performance: \$sys_clock, Core_Cycles, Total_Cycles,
Cache Efficiency, Instructions count, Instr Cache_Misses,
Data Cache_Read_Misses, Data Cache_Write_Misses,

- Setting:
 - Default target : Release
 - ARMulator processor: ARM940T
 - ARMulator clock : Emulated (10MHz, 50MHz, 100MHz)
 - Memory map file : armsd.map
 - Number of run : 50000, 100000, 200000 times

Report the skills to optimize the code and the comparison of performance and code size between optimized code and original one. Try to explain the effect of the skills.

2.4. Reference

- DUI0151A Debugging skills: ADS Debugger Guide, “ADS Compiler, Linker, and Utilities Guide”.
- Profiling: “Application Note 93: Benchmarking with ARMulator”
- Efficient C programming: “Application Note 34: Writing Efficient C for ARM”
- http://twins.ee.nctu.edu.tw/courses/ip_core_02/index.html
- http://twins.ee.nctu.edu.tw/courses/ip_core_01/index.html

- [1] IEEE standard specifications for the implementations of 8x8 inverse discrete cosine transform, IEEE Std 1180-1990, March 1991
- [2] Tadashi Sakamoto and Tomohiro Hase, “Software JPEG for a 32-bit MCU with dual issue,” IEEE Transactions on Consumer Electronics, Vol. 44 Issue: 4, Nov. 1998, pp. 1334 -1341.
- [3] Alan Lewis and Paul Carpenter, “Optimizing digital video codecs in ARM cores,” EE Times, Sep. 20, 2001.
- [4] <http://www.nondot.org/sabre/graphpro/line3.html#What>

2.5. Appendix

The MIPS figures which ARM (and most of the industry) quotes are "Dhrystone VAX MIPS". The idea behind this measure is to compare the performance of a machine (in our case, an ARM system) against the performance of a reference machine. The industry adopted the VAX 11/780 as the reference 1 MIP machine. The benchmark is calculated by measuring the number of Dhrystones per second for the system, and then dividing that figure by the number of Dhrystones per second achieved by the reference machine. So "80 MIPS" means "80 Dhrystone VAX MIPS", which means 80 times faster than a VAX 11/780. The reason for comparing against a reference machine is that it avoids the need to argue about differences in instruction sets. RISC processors tend to have lots of simple instructions. CISC machines like x86 and VAX tend to have more complex instructions. If you just counted the number of instructions per second of a machine directly, then machines with simple instructions would get higher instructions-per-second results, even though it would not be telling you whether it gets the job done any faster. By comparing how fast a machine gets a given piece of work done against how fast other machines get that piece of work done, the question of the different instruction sets is avoided.

There are two different versions of the Dhrystone benchmark commonly quoted:

- Dhrystone 1.1
- Dhrystone 2.1

ARM quotes Dhrystone 2.1 figures. The VAX 11/780 achieves 1757 Dhrystones per second. The maximum performance of the ARM7 family is 0.9 Dhrystone VAX MIPS per MHz. The maximum performance of the ARM9 family is 1.1 Dhrystone VAX MIPS per MHz. These figures assume ARM code running from 32-bit wide, zero wait-state memory. If there are wait-states, or (for cores with caches) the caches are disabled, then the performance figures will be lower. To estimate how many ARM instructions are executed per second then simply divide the frequency by the average CPI (Cycles Per Instruction) for the core. For example:

The average CPI for the ARM7 family is about 1.9 cycles per instruction.

The average CPI for the ARM9 family is about 1.5 cycles per instruction.

What is fixed point?

Fixed Point numbers are a simple way to store floating point information in an integer number. They are a very useful with limited computing resource. They only store a set number of decimal places, and are therefore slightly inaccurate. But they can reduce so many computing power and time. Some describe as follow.

The first example are in base 10. This is an easy base for humans to deal with, but is very cumbersome with computers. Therefore, in real applications we use base 2.

- **Example 1:**

Suppose you have the following number : 180.4527

Because you are working with performance intensive applications, you need good performance. Naturally, you try to use integer math. Unfortunately, you find that the ".4527" is an important part of the information. So you do the following:

```
IntX = 1804527;
```

You could see that the number above is that same number as the one farther above, it is just **"shifted"** four places to the left. This leaves you with a (bigger) integer number. But how do you print it out?

```
WRITELN('This number was stored in an integer: ', IntX / (104));
```

{"Shifted" four places} ←

What is great about this system is that you can add numbers very easily:

IntX := 1804527; {180.4527 }

```
IntY := 0005473;    { 0.5473 }
```

IntZ := IntX+IntY; { IntZ now = 1810000 }

$$\text{RealZ} := \text{IntZ} / (10^4); \{ \text{RealZ} = 181.0 \}$$

You can multiply two numbers with integer routines.

- **Example2:**

Suppose you have the following number: 0.125

In reality, we will be using base 2 fixed point numbers. So you can see the following:

$$\text{Int}X = 1$$

And shifted 3 place

WRITELN ('This number was stored in an integer: ', IntX / (2³));
{ "Shifted" three places } ←

You can describe numbers very easily, and without floating point:

IntX = 1; {0.125 ($1 / 2^3$), 0.5 ($1/2$), and so on}

IntY = 3; {0.375 ($3 / 2^3$), 0.1875 ($1/2^4$), and so on}

IntZ = 5; {0.15625 ($5/2^5$), 0.3125 ($1/2^4$), and so on}

This means that we will use this for divide and multiply.

this is a huge improvement over using floating point. It also lets people without a coprocessor create and use high performance routines (ex: FFT, DCT).