UVic

The ARMSIM User Guide

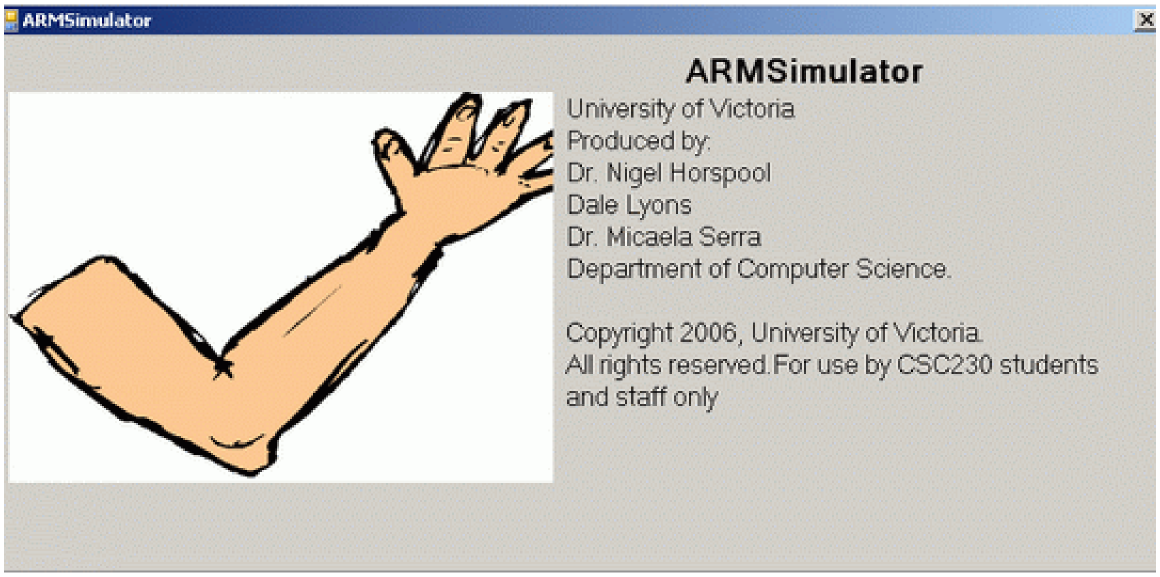
© R. N. Horspool, W. D. Lyons, M. Serra, 2007–2008
Department of Computer Science, University of Victoria

1. Overview

ARMSim is a desktop application running in a Windows environment. It allows users to simulate the execution of ARM assembly language programs on a system based on the ARM7TDMI processor. ARMSim includes both an assembler and a linker; when a file is loaded, the simulator automatically assembles and links the program. ARMSim also provides features not often found in similar applications. They enable users both to debug ARM assembly programs and to monitor the state of the system while a program executes. The monitoring information includes both cache states and clock cycles consumed.

The purpose of this user guide is to explain how to use the tools and views[1] provided by ARMSim. Therefore, the scope of the document has been limited to the features of the simulator. It does not cover ARM assembly programming or computer architecture. Users who are unfamiliar with these topics should consult other material, some of which is listed in the references.

The topics in this document have been organized to provide a step-by-step introduction to ARMSim. Sections 2 and 3 begin with the features of the simulator and explain how to customize its layout. Then, section 4 describes the most commonly used debugging features and views. Finally, sections 5 and 6 conclude by detailing additional debugging features, views and performance monitoring tools.



ARMSimulator
University of Victoria
Produced by:
Dr. Nigel Horspool
Dale Lyons
Dr. Micaela Serra
Department of Computer Science.

Copyright 2006, University of Victoria.
All rights reserved.For use by CSC230 students and staff only

1. In this document, a *view* is a window displayed by the ARMSim simulator that shows the state of some aspect of the program being run.

2. Features

The ARMSim toolbar and views give the user access to a variety of tools to debug and monitor ARM assembly language programs. The following sections describe the controls provided by the toolbar and the information displayed in the views.

2.1 Toolbar

The ARMSim toolbar provides easy access to many of the debugging features of the simulator, especially those features that allow the user to control the execution of a program. The functions of the buttons on the toolbar are summarized in Table 1.

Table 1. Toolbar Buttons.

	The **Step Into** button causes the simulator to execute the highlighted instruction and move to the next instruction in the program. If the highlighted instruction is a subroutine call (BL or BX instruction) then the next highlighted instruction will be the first instruction of the subroutine.
	The **Step Over** button causes the simulator to execute the highlighted instruction and move to the next instruction in the current subroutine. If the highlighted instruction is a subroutine call (BL or BX instruction) then the program is run until the subroutine returns. Thus, unless a breakpoint is encountered, the next highlighted instruction will be at the return point from the subroutine call.
■	The **Stop** button causes the simulator to stop the execution of the program.
▶	The **Continue** button causes the simulator to run the program until it encounters a breakpoint, an SWI 0x11 instruction (end of execution), or a run-time error.
	The **Restart** button causes the simulator to start the execution of the program from the beginning.
	The **Reload** button causes the simulator to load a new version of the program file from the hard drive and start the execution of the program from the beginning.

2.2 Views

The ARMSim views display the simulator's output and the contents of the system's storage. ARMSim provides seven views, which are summarized in Table 2.

3. Setting up the Simulator

The appearance of ARMSim, including the location, font, and colour of the views, can be customized to suit the user's preferences. When the simulator is closed, the settings are remembered for next time the user starts up ARMSim. The following sections describe how to customize ARMSim's appearance.

3.1 Docking Windows

All of the views described in section 2.2, except the **Code View**, appear in docking windows (see Figure 1). Each window can be docked along any side of the application window, or it can float above the appli-

Table 2. ARMSim Views

Code View	It displays the assembly language instructions of the program that is currently open. This view is always visible and cannot be closed.
Registers View	It displays the contents of the 16 general-purpose user registers available in the ARM processor, as well as the status of the Current Program Status Register (CPSR) and the condition code flags. The contents of the registers can be displayed in hexadecimal, unsigned decimal, or signed decimal formats. Additionally the contents of the Vector Floating Point Coprocessor (VFP) registers can be displayed. They include the overlapped Single Precision Registers (s0-s31) and the Double Precision Floating Point Registers (d0-d15).
Output View	It displays any text printed to standard output, plus any automatic success and error messages produced by the simulator.
Stack View	It displays the contents of the system stack. In this view, the top word in the stack is highlighted.
Watch View	It displays the values of variables that the user has added to the watch list, that is, the list of variables that the user wishes to monitor during the execution of a program.
Cache Views	They display the contents of the L1 cache. This cache can consist of either a unified data and instruction cache, displayed in the **Unified Cache View**, or separate data and instruction caches, displayed in the **Data Cache** and **Instruction Cache Views**, respectively, depending on the cache properties selected by the user.
Board Controls View	It displays the user interfaces of any loaded plug-ins. If no plug-ins were loaded at application start, this view is disabled.
Memory View	It displays the contents of main memory, as 8-bit, 16-bit, or 32-bit words. There can be multiple memory views, each displaying a different region of memory.

cation window. In addition, each docking window can be displayed or hidden, and each displayed window has an auto-hide option.

To move a docking window, click the title bar of the window, and drag the window to the desired location. If multiple views have been stacked within a single docking window, select the tab with the desired view name from the tabs along the bottom of the docking window, click this tab, and drag it to the desired location.

To toggle a docking window between the show and hide states, select the view name from the **View** menu. Alternatively, to hide a docking window that is currently displayed, click the **X** in the top right corner of the docking window. To toggle a docking window between the show and auto-hide modes, click the pin in the top right corner.

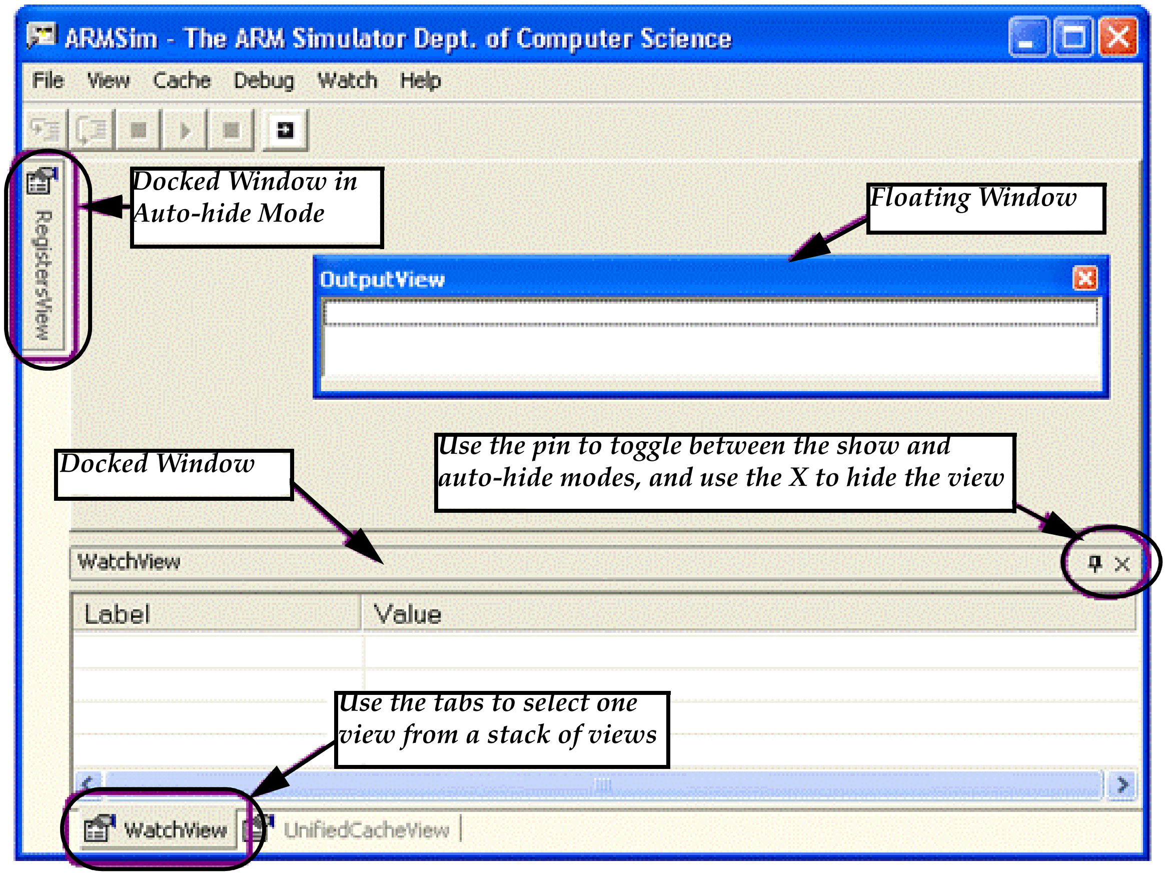

ARMSim - The ARM Simulator Dept. of Computer Science
File View Cache Debug Watch Help
RegistersView
Docked Window in Auto-hide Mode
Floating Window
OutputView
Docked Window
Use the pin to toggle between the show and auto-hide modes, and use the X to hide the view
WatchView
Label
Value
Use the tabs to select one view from a stack of views
WatchView
UnifiedCacheView

Figure 1. Docking Windows

3.2 Fonts

To change the font, size, style, or colour of the text in a view, move the cursor into the view, click the right mouse button, and select **Font** from the context menu. Then, make changes in the **Font** dialog box, and click **OK**. To restore the original font settings, move the cursor into the view, click the right mouse button, and select **Restore Defaults** from the context menu. Note that **Restore Defaults** will also restore the default background and highlight colours.

3.3 Colours

To change the background (highlight) colour in a view, move the cursor into the view, click the right mouse button, and select **Background Colour (Highlight Colour)** from the context menu. Then, make the changes in the **Color** dialog box, and click **OK**. To restore the original background and highlight colours, move the cursor into the view, click the right mouse button, and select **Restore Defaults** from the context menu. Note that **Restore Defaults** will also restore the default font settings.

The use of the highlight colour depends on context. For example, in the **Code** and **Stack Views**, it is used as a background colour on the highlighted line, but in the **Register** and **Cache Views**, it is used as a text colour for storage locations that have been written to.

4. Getting Started

Using ARMSim to simulate the execution of a program on an ARM processor involves two activities—actually running the program and observing the output. Sections 4.1 to 4.4 provide information on running programs with the simulator, while sections 4.5 and 4.6 describe two of the views available in the simulator.

4.1 Creating a File

ARMSim accepts both ARM assembly source files that use the Gnu Assembler (*gas*) syntax and ARM object files generated by the Gnu tools provided with Cygwin or CodeSourcery. ARM assembly source files can be created using any text editor (e.g. TextPad) and must be saved with a .s filename extension. ARM object files can be generated from ARM assembly files or C source files and must be compiled according to the instructions in the accompanying document on “C and ARM”. For details on ARM assembly programming consult the references.

4.2 Opening and Loading a File

To open a file, select **File > Load**. Then navigate to the folder in which the file is stored and double-click the file to be opened. When a file is opened, it is automatically assembled (if it is a source file) and linked. If the assembly and linking processes are successful, the contents of the file appear in the **Code View** with the first instruction in the _start (or main) subroutine highlighted. If the contents of the file appear in the **Code View**, but the first instruction is *not* highlighted, one must check the **Output View** for compiler errors (see section 6.3).

Notes:

- The file to be opened must be a source (.s) file or an object (.o) file.
- If the file to be opened does not appear in the directory listing in the dialog box, check to make sure that the appropriate file type has been selected.
- The source code cannot be edited in the **Code View** window, but must be changed in the original text editor and then reloaded.

4.3 Running a Program

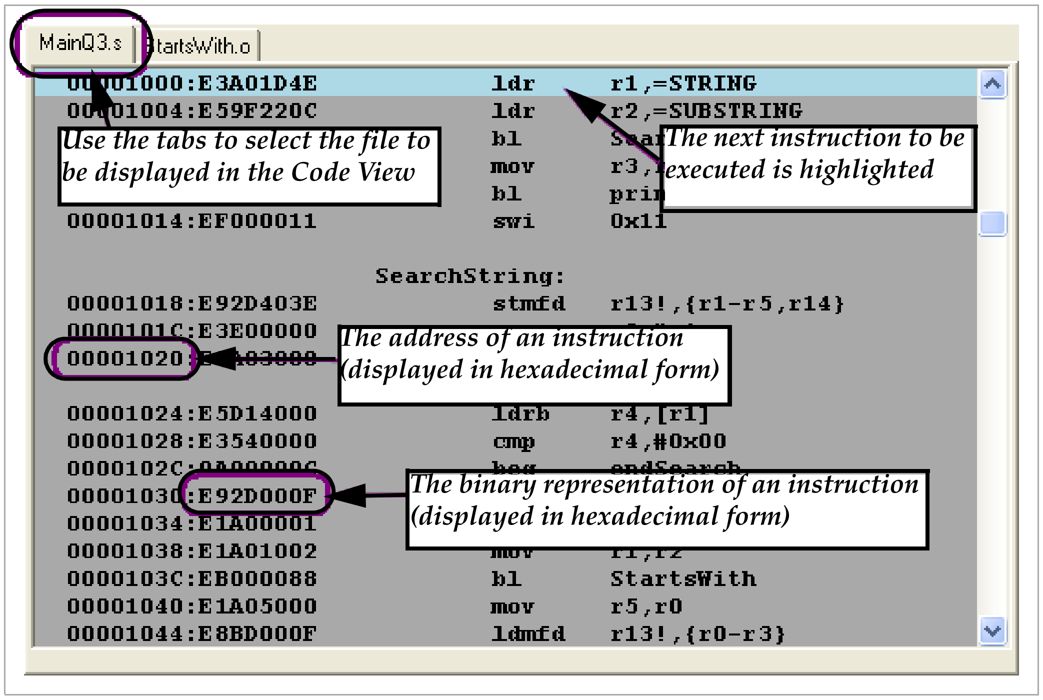
To run the program displayed in the **Code View**, select **Debug > Run**, or click the **Continue** button on the toolbar (see Table 1). The program runs until the simulator encounters a breakpoint (see section 5.5 for an explanation of breakpoints) or an SWI 0x11 instruction (to exit the execution), or a fatal error.

4.4 Stopping a Program

To stop a program that is currently running, select **Debug > Stop**, or click the **Stop** button on the toolbar (see Table 1). When the program has stopped, any storage locations in the **Register**, **Cache**, and **Memory Views** that have been written to since the program started running are highlighted.

4.5 Code View

The **Code View** displays the assembly language instructions of the program that is currently active. Next to each instruction, the simulator shows the memory address of the instruction and the binary representation of the instruction, separated by a colon and displayed in hexadecimal format (see Figure 2).



```
00001000:E3A01D4E          ldr      r1,=STRING
00001004:E59F220C          ldr      r2,=SUBSTRING
                           bl       Sear
                           mov      r3,
                           bl       prin
00001014:EF000011          swi      0x11

                SearchString:
00001018:E92D403E          stmfd    r13!,{r1-r5,r14}
0000101C:E3E00000
00001020:

00001024:E5D14000          ldrb     r4,[r1]
00001028:E3540000          cmp      r4,#0x00
0000102C:0A00000C          beq      endSearch
00001030:E92D000F
00001034:E1A00001
00001038:E1A01002          mov      r1,r2
0000103C:EB000088          bl       StartsWith
00001040:E1A05000          mov      r5,r0
00001044:E8BD000F          ldmfd    r13!,{r0-r3}
```

Use the tabs to select the file to be displayed in the Code View

The next instruction to be executed is highlighted

The address of an instruction (displayed in hexadecimal form)

The binary representation of an instruction (displayed in hexadecimal form)

Figure 2. Code View

When a file is opened and successfully assembled and linked, its contents are displayed in the **Code View**, as described above, and the first instruction to be executed is highlighted. When multiple files are opened (see section 5.4), the file in which execution must start is displayed in the **Code View** with the first instruction highlighted. The other files can be viewed by clicking on the tabs at the top of the **Code View**.

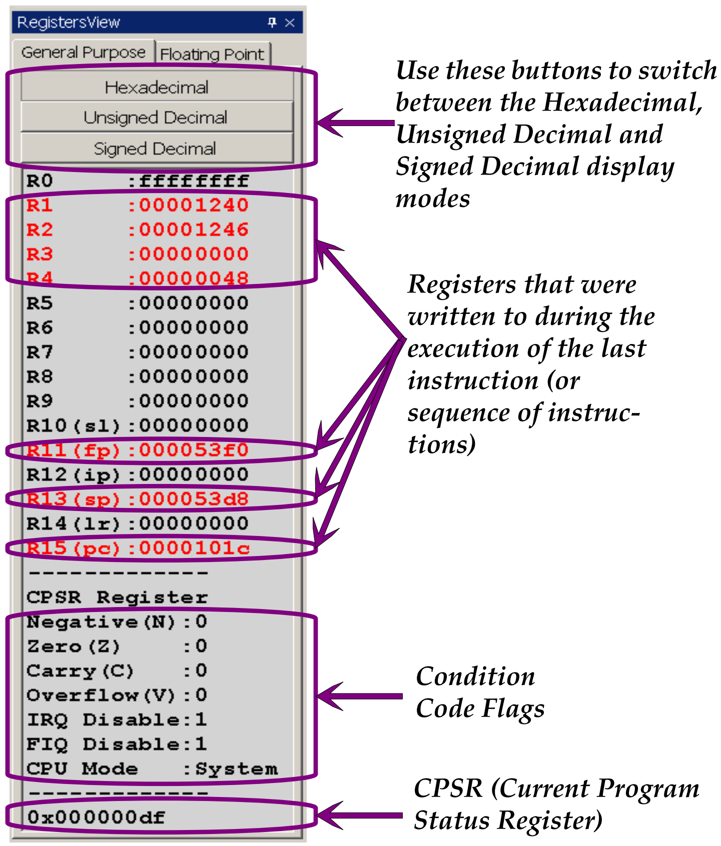
4.6 Registers View

The **Registers View** displays the contents of the 16 general-purpose user registers available in the ARM processor, as well as the status of the Current Program Status Register (CPSR) and the condition code flags (the leftmost 4 bits of the CPSR, as displayed below the condition code flags in the simulator). Additionally, the Vector Floating Point (VFP) registers are available for display in the tab labelled "Floating Point". These registers represent the 32 Single Precision registers or the 16 Double Precision Registers of the VFP. Note that these two sets of registers are overlapped.

The General Purpose Registers are selected by clicking on the "General Purpose Registers" tab in the Registers View. The contents of the general purpose registers can be displayed in hexadecimal, signed decimal, or unsigned decimal formats. Use the **Hexadecimal**, **Signed Decimal**, and **Unsigned Decimal** buttons at the top of the **Registers View** to switch between display formats (see Figure 3).

When an instruction is executed using one of the step commands (see section 5.1) or when a sequence of instructions is executed using the **Debug > Run** option or the **Continue** button (see section 4.3), any registers and condition code flags that were written to during the execution of the instruction(s) are highlighted after the execution of the instruction(s) has finished.

The Floating Point Registers are selected by clicking on the "Floating Point" tab in the Registers View. The Floating Point Registers can be viewed as Single Precision or Double Precision registers. Use


RegistersView
General Purpose
Floating Point
Hexadecimal
Unsigned Decimal
Signed Decimal
Use these buttons to switch between the Hexadecimal, Unsigned Decimal and Signed Decimal display modes
Registers that were written to during the execution of the last instruction (or sequence of instructions)
Condition Code Flags
CPSR (Current Program Status Register)

```
R0      :ffffffff
R1      :00001240
R2      :00001246
R3      :00000000
R4      :00000048
R5      :00000000
R6      :00000000
R7      :00000000
R8      :00000000
R9      :00000000
R10(sl):00000000
R11(fp):000053f0
R12(ip):00000000
R13(sp):000053d8
R14(lr):00000000
R15(pc):0000101c
-------------
CPSR Register
Negative(N):0
Zero(Z)    :0
Carry(C)   :0
Overflow(V):0
IRQ Disable:1
FIQ Disable:1
CPU Mode   :System
-------------
0x000000df
```

Registers R10-R15 are also labelled:

R10	sl	stack limit
R11	fp	frame pointer
R12	ip	intra-procedure-call scratch register
R13	sp	stack pointer
R14	lr	link register
R15	pc	program counter

Figure 3. General Purpose Registers View.

the Single Precision or Double Precision tabs at the top of the Registers View to switch between the display types (see Figure 4).

5. Debugging a Program

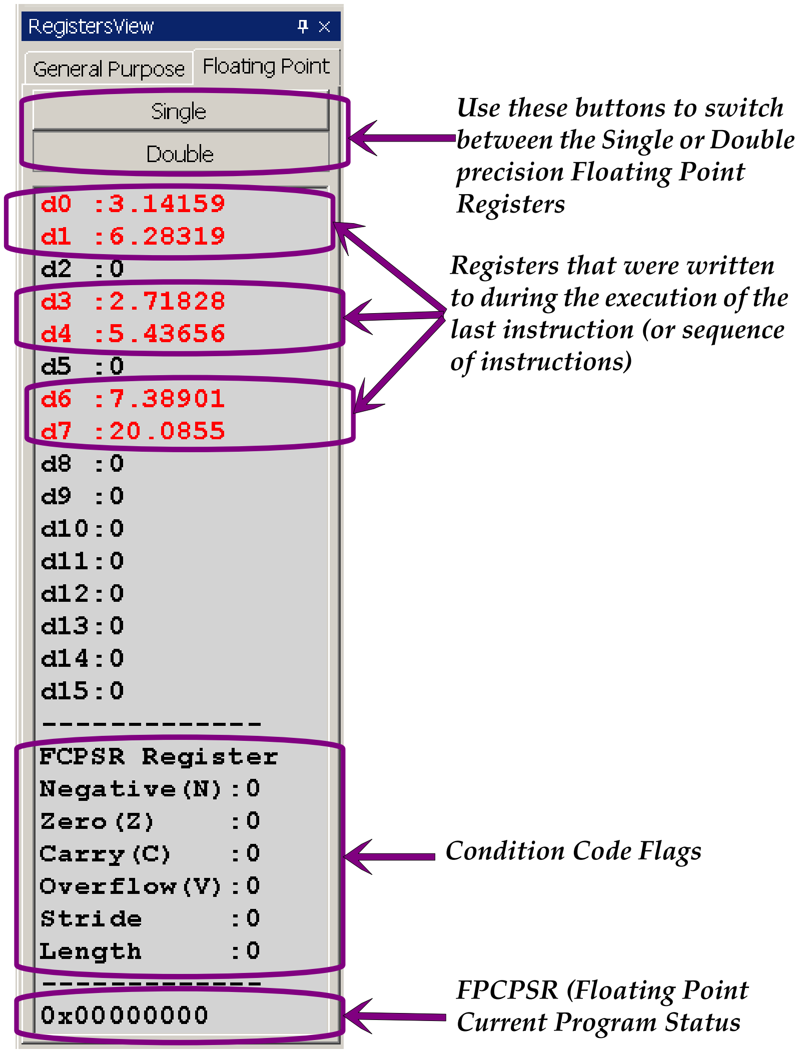
ARMSim provides a number of features that enable users to debug ARM assembly programs, including execution controls to step through and restart programs, **Reload** and **Open Multiple** commands, and breakpoints. Sections 5.1 and 5.2 describe the execution controls. Sections 5.3 and 5.4 describe the **Reload** and **Open Multiple** commands, respectively, and section 5.5 explains how to manage breakpoints.

5.1 Stepping Through a Program

To step through a program one instruction at a time, use either the **Step Into** button or the **Step Over** button on the toolbar, or alternatively, select **Debug > Step Into** or **Debug > Step Over**.

After an instruction has been executed using either **Step Into** or **Step Over**, both the next instruction to be executed and any memory locations in the **Registers**, **Memory**, and **Cache Views** that were written to during the execution of the instruction are highlighted.

For most instructions, the results of both **Step Into** and **Step Over** are identical; however, when an instruction is a branch to a subroutine, **Step Into** executes the branch and moves to the first instruction of the subroutine. In contrast, the **Step Over** executes the whole subroutine and moves to the instruction after the branch in the original subroutine. Therefore, if a program consists of multiple files and there is a branch from a subroutine in one file to a subroutine in another file, executing the branch using **Step Into** also changes the file displayed in the **Code View**.


RegistersView
General Purpose
Floating Point
Single
Double
Use these buttons to switch between the Single or Double precision Floating Point Registers
Registers that were written to during the execution of the last instruction (or sequence of instructions)
Condition Code Flags
FPCPSR (Floating Point Current Program Status

```
d0  :3.14159
d1  :6.28319
d2  :0
d3  :2.71828
d4  :5.43656
d5  :0
d6  :7.38901
d7  :20.0855
d8  :0
d9  :0
d10:0
d11:0
d12:0
d13:0
d14:0
d15:0
-------------
FCPSR Register
Negative(N):0
Zero(Z)    :0
Carry(C)   :0
Overflow(V):0
Stride     :0
Length     :0
-------------
0x00000000
```

Figure 4. Floating Point Registers View.

5.2 Restarting a Program

To restart a program, click the **Restart** button on the toolbar, or select **Debug > Restart**. Restarting a program resets the registers, cache, and memory; it sets the program counter to the address of the first instruction in the program; and it highlights this instruction (the next instruction to be executed).

5.3 Reloading a Program

To reload a program, click the **Reload** button on the toolbar, or select **File > Reload**. Reloading a program loads a new copy of the file from the hard drive; it resets the registers, cache, memory, stack, and watches; it sets the program counter to the address of the first instruction in the program; and it highlights this instruction (the next instruction to be executed).

5.4 Opening Multiple Files

To open multiple files, select **File > Open Multiple**. Then, click the **Add** button in the **MultiFileOpen** dialog box; navigate to the folder, in which the files are stored; and double-click the file to be opened. Repeat the three steps in the previous sentence until all of the files to be opened have been added to the list in the dialog box. Then, click **OK** to open the files. When the files have been successfully opened, the contents of the file that contains the _start (or main) subroutine will appear in the **Code View** with the first instruction in this subroutine highlighted.

To remove a file from the list of files to be opened, select the filename in the dialog box, and click the **Remove** button. To remove all of the files from the list of files to be opened, click the **Clear** button.

Notes:

- The files to be opened must be ARM assembler source (.s) files, ARM object (.o) files, or a combination of source and object files.
- If a file does not appear in the directory listing in the dialog box, one must check that the appropriate file type has been selected.
- If the contents of the file appear in the **Code View**, but the first instruction is not highlighted, check the **Output View** for compiler errors (see section 6.3).
- When the file is opened, it is automatically assembled (if it is a source file) and linked.

5.5 Breakpoints

A breakpoint is a user-defined stopping point in a program (i.e. a point other than an SWI 0x11 instruction, at which execution of a program should terminate). When a program is being debugged, breakpoints are used to halt execution of the program at predefined points so that the contents of storage locations, such as registers and main memory, can be examined to ensure that the program is working correctly.

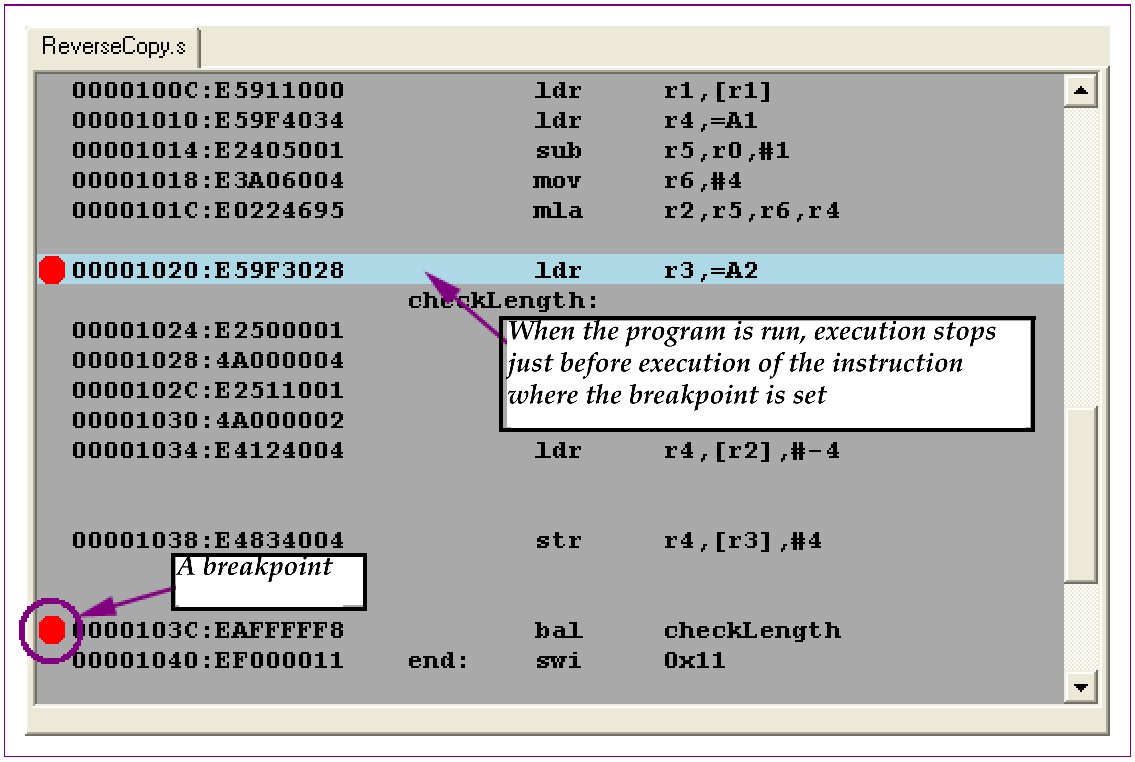
When a breakpoint is set and the program is run using either the **Debug > Run** option or the **Continue** button (see section 4.3), execution of the program stops just before execution of the instruction at which the breakpoint is set (see Figure 5).

To set a breakpoint, double-click the line of code, at which the breakpoint should be set. Alternatively, step through the code to the line, at which the breakpoint should be set, and then select **Debug > Toggle Breakpoint**. When the breakpoint is set, a large red dot appears in the **Code View** next to the address of the instruction at which the breakpoint was set.

To clear a breakpoint, double-click the line of code, at which the breakpoint is set. Alternatively, step through the code to the line, at which the breakpoint is set, and then select **Debug > Toggle Breakpoint**. To clear all of the breakpoints in a program, select **Debug > Clear All Breakpoints**.

Note:

- **Clear All Breakpoints** clears the breakpoints in *all* files that are currently open.



```
0000100C:E5911000            ldr     r1,[r1]
00001010:E59F4034            ldr     r4,=A1
00001014:E2405001            sub     r5,r0,#1
00001018:E3A06004            mov     r6,#4
0000101C:E0224695            mla     r2,r5,r6,r4

00001020:E59F3028            ldr     r3,=A2
                     checkLength:
00001024:E2500001
00001028:4A000004
0000102C:E2511001
00001030:4A000002
00001034:E4124004            ldr     r4,[r2],#-4


00001038:E4834004            str     r4,[r3],#4


0000103C:EAFFFFF8            bal     checkLength
00001040:EF000011    end:    swi     0x11
```

When the program is run, execution stops just before execution of the instruction where the breakpoint is set

A breakpoint

Figure 5. Breakpoints.

6. Additional Views

In addition to the **Code** and **Register Views** discussed in sections 4.5 and 4.6, respectively, ARMSim includes **Watch**, **Memory**, **Output**, **Stack**, and **Cache Views** that enable users to observe the data transfers within the system, as well as the output of the system. The following sections describe these additional views and explain any commands and settings associated with them.

6.1 Watch View

The **Watch View** displays the values of variables that the user has added to the watch list, which is a list of variables that the user wishes to monitor during the execution of a program.

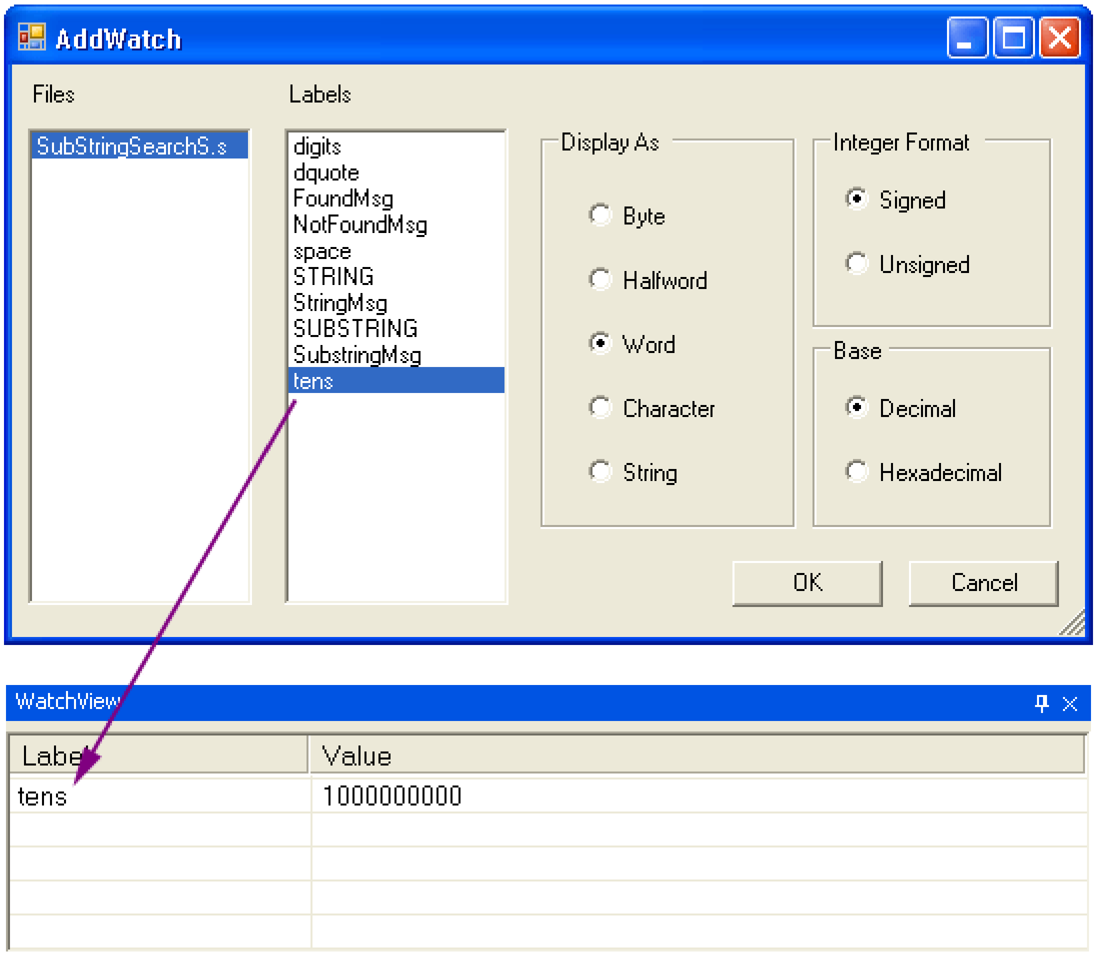
To add a variable to the watch list, select **Watch > Add Watch**. Alternatively, right-click in the **Watch View**, and select **Add Watch** from the context menu. In the **Add Watch** dialog box (see Figure 6), select the file, in which the variable appears; the label that is attached to the variable; and the display type of the variable. If applicable, specify the integer format of the variable, and select the base, in which the integer representation of the variable should be displayed. Click **OK**.

To remove a variable from the watch list, select the variable in the **Watch View**, and then select **Watch > Remove Watch**. To remove all of the variables from the watch list, select **Watch > Clear All**. Alternatively, right-click in the **Watch View**, and select **Clear All** from the context menu.

Notes:

- Although **Remove Watch** appears in the **Watch** menu, this option has not yet been implemented.

- The **Watch View** does not display arrays; however, it is possible to display the first item of an array by treating it as a scalar variable and adding it to the watch list, as described above.



Display As: ( ) Byte ( ) Halfword [✓ Word] ( ) Character ( ) String
Integer Format: [✓ Signed] ( ) Unsigned
Base: [✓ Decimal] ( ) Hexadecimal

Label	Value
tens	1000000000

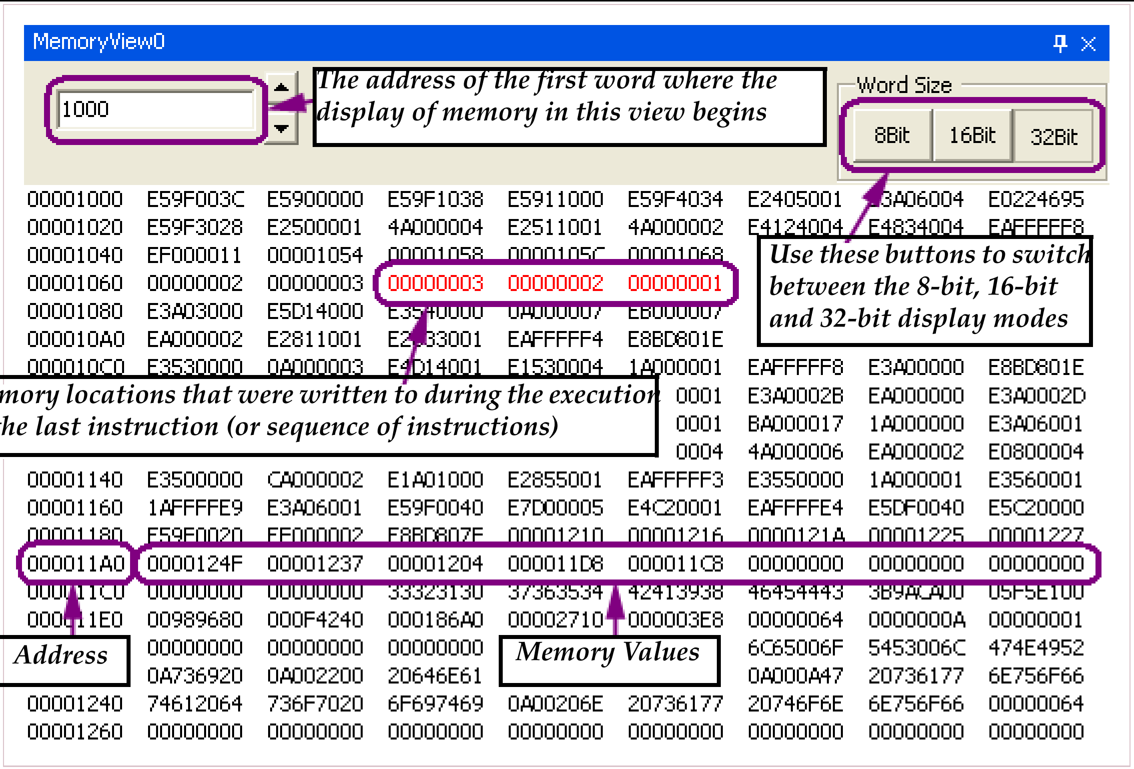
Figure 6. Adding a Watch.

6.2 Memory View

A **Memory View** displays the contents of main memory. In this view, each row contains an address followed by a series of words from memory (see Figure 7).

Since the entire main memory cannot be displayed in a single **Memory View**, each **Memory View** shows only a part of memory. The address in the top left corner of the view specifies the word, at which the part of memory displayed in the view begins, and the size of the view determines the number of words displayed.

To display a different part of memory, enter a hexadecimal address from 0 to FFFFFFFF into the text box in the top left corner of the **Memory View**. Alternatively, use the up and down arrows beside the text box to select lower and higher memory addresses, respectively. The contents of memory can be displayed as 8-bit bytes, 16-bit halfwords, or 32-bit words. Use the three buttons in the **Word Size** box in the top right corner of the **Memory View** to switch among the three display formats.


MemoryView0
The address of the first word where the display of memory in this view begins
Word Size
8Bit
16Bit
32Bit
Use these buttons to switch between the 8-bit, 16-bit and 32-bit display modes
Memory locations that were written to during the execution of the last instruction (or sequence of instructions)
Address
Memory Values

```
1000

00001000  E59F003C  E5900000  E59F1038  E5911000  E59F4034  E2405001  E3A06004  E0224695
00001020  E59F3028  E2500001  4A000004  E2511001  4A000002  E4124004  E4834004  EAFFFFF8
00001040  EF000011  00001054  00001058  0000105C  00001068
00001060  00000002  00000003  00000003  00000002  00000001
00001080  E3A03000  E5D14000  E3540000  0A000007  EB000007
000010A0  EA000002  E2811001  E2[illegible]33001  EAFFFFF4  E8BD801E
000010C0  E3530000  0A000003  E4D14001  E1530004  1A000001  EAFFFFF8  E3A00000  E8BD801E
                                                  0001  E3A0002B  EA000000  E3A0002D
                                                  0001  BA000017  1A000000  E3A06001
                                                  0004  4A000006  EA000002  E0800004
00001140  E3500000  CA000002  E1A01000  E2855001  EAFFFFF3  E3550000  1A000001  E3560001
00001160  1AFFFFE9  E3A06001  E59F0040  E7D00005  E4C20001  EAFFFFE4  E5DF0040  E5C20000
00001180  E59F0020  EF000002  E8BD807E  00001210  00001216  0000121A  00001225  00001227
000011A0  0000124F  00001237  00001204  000011D8  000011C8  00000000  00000000  00000000
000011C0  00000000  00000000  33323130  37363534  42413938  46454443  3B9ACA00  05F5E100
000011E0  00989680  000F4240  000186A0  00002710  000003E8  00000064  0000000A  00000001
          00000000  00000000  00000000                      6C65006F  5453006C  474E4952
          0A736920  0A002200  20646E61                      0A000A47  20736177  6E756F66
00001240  74612064  736F7020  6F697469  0A00206E  20736177  20746F6E  6E756F66  00000064
00001260  00000000  00000000  00000000  00000000  00000000  00000000  00000000  00000000
```

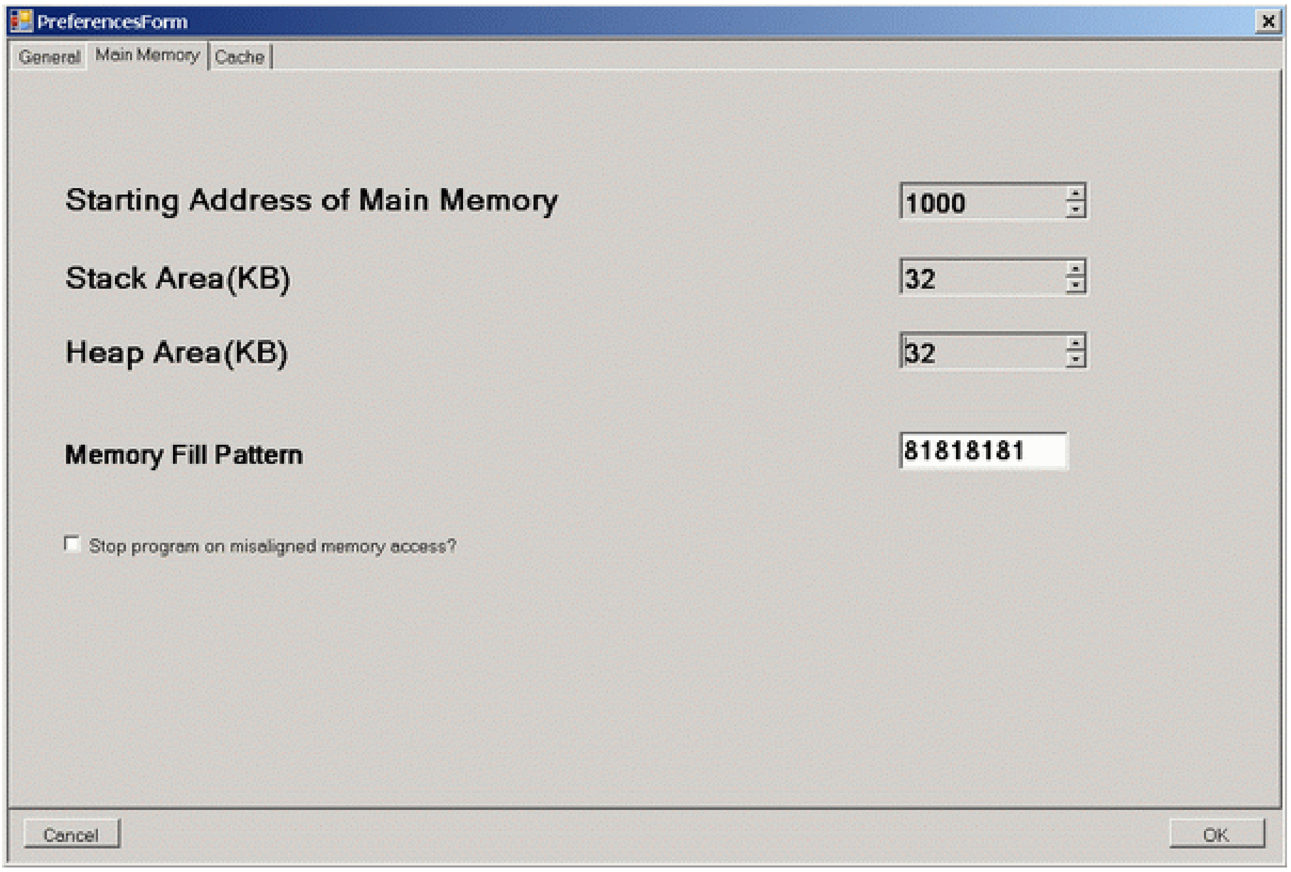
Figure 7. Memory View.

When an instruction is executed using one of the step commands (see section 5.1) or when a sequence of instructions is executed using the **Debug > Run** option or the **Continue** button (see section 4.3), any memory locations that were written to during the execution of the instruction(s) are highlighted after the execution of the instruction(s) has finished.

The properties of main memory, including its starting address, the stack area, and the heap area, can be customized to suit the user's preferences. To change these properties, select **File > Preferences**, and click the **Main Memory** tab. Type in new values for the starting address, stack area, and heap area, or use the arrow buttons beside each property to adjust the value of that property (see Figure 8). Click **OK**, and then reload the program (see section 5.3) to refresh the **Memory View(s)**.

Notes:

- If a store (STR) instruction is executed, but the value in memory does not change, check the **Cache Preferences** to make sure that the **Write Policy** is not set to **Write Back**. If it is, set it to **Write Through**. (See section 6.5 for information on setting the **Cache Preferences**.)
- The simulator can have multiple **Memory Views**, each of which displays a different region of memory. To open additional **Memory Views**, select **View > Memory**.
- When the display size is set to 8-bit, the ASCII representation of each row of bytes is displayed at the end of the row.
- When the display size is 16-bit or 32-bit, the assignment of byte addresses is little-endian.
- In the **Memory View**, all cells that are part of the memory region allocated to the program are shown in hexadecimal notation (e.g. E1A03000, 00000000); cells outside the allocated memory region are shown as question marks (e.g. ????????).



Starting Address of Main Memory: 1000

Stack Area(KB): 32

Heap Area(KB): 32

Memory Fill Pattern: 81818181

☐ Stop program on misaligned memory access?

Figure 8. Main Memory Preferences Form.

6.3 Output View

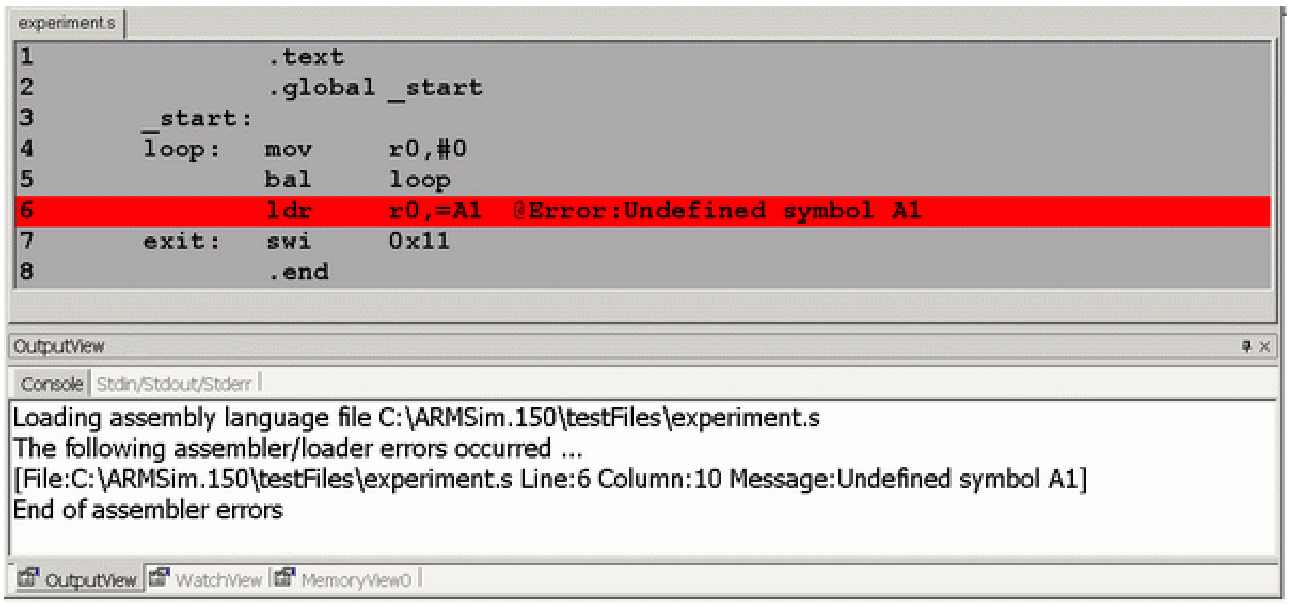
The Output View contains a row of two tabs labelled “Console” and “Stdin/Stdout/Stderr”. Selecting the tab labelled “Console” brings a window to the front where the simulator outputs success and error messages. After the simulator has loaded the program, any assembler or linker errors are displayed here (see Figure 9 for an example). To find the source of an error message displayed in the **Output View** (see Figure 9), double-click the message, and scroll up one line in the **Code View**. Additional information will be displayed here such as instruction counts and runtimes.

Selecting the tab labelled “Stdin/Stdout/Stderr” brings a window to the front where output from the user program is displayed as a result of using software interrupts (SWI instructions) to perform I/O. Output directed to either the standard output or standard error (Stdin/Stdout) are displayed in this tabbed window. Any request to read from the standard input device (Stdin) causes the program to freeze until the input is provided on the keyboard; that input is echoed in this tabbed window as well.

To copy text from the one of the **Output View** tabbed windows, right-click in the view, and select **Copy to Clipboard** from the context menu. To clear the contents of the **Output View** tabbed window, right-click in the tab, and select **Clear** from the context menu.

6.4 Stack View

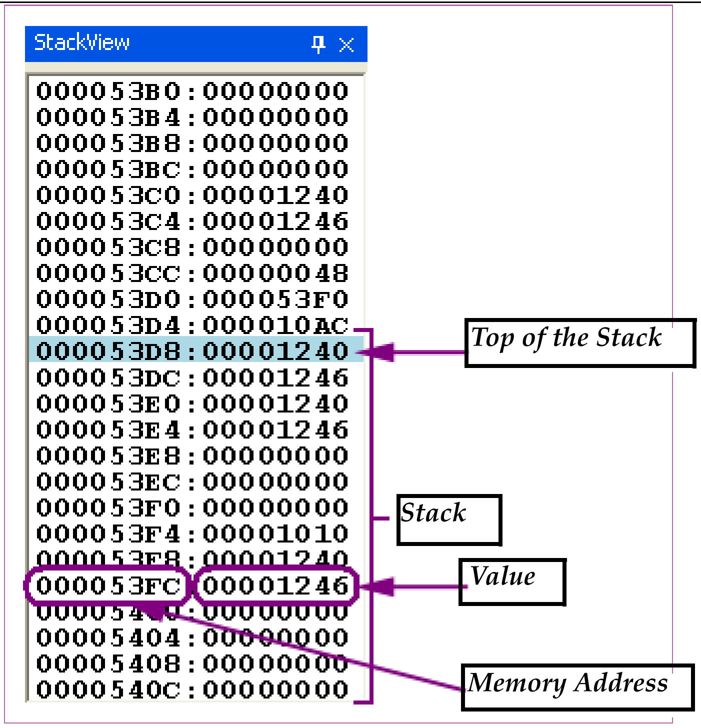
The **Stack View** displays the contents of the system stack. In this view, the memory address of a value and its binary representation are displayed on a single line, separated by a colon and displayed in hexadecimal format. Furthermore, the top word in the stack is highlighted (see Figure 10). Note that the system stack is a full descending stack.



```
1               .text
2               .global _start
3       _start:
4       loop:   mov     r0,#0
5               bal     loop
6               ldr     r0,=A1  @Error:Undefined symbol A1
7       exit:   swi     0x11
8               .end
```

```
Loading assembly language file C:\ARMSim.150\testFiles\experiment.s
The following assembler/loader errors occurred ...
[File:C:\ARMSim.150\testFiles\experiment.s Line:6 Column:10 Message:Undefined symbol A1]
End of assembler errors
```

Figure 9. Error Message.



```
000053B0:00000000
000053B4:00000000
000053B8:00000000
000053BC:00000000
000053C0:00001240
000053C4:00001246
000053C8:00000000
000053CC:00000048
000053D0:000053F0
000053D4:000010AC
000053D8:00001240
000053DC:00001246
000053E0:00001240
000053E4:00001246
000053E8:00000000
000053EC:00000000
000053F0:00000000
000053F4:00001010
000053F8:00001240
000053FC:00001246
00005400:00000000
00005404:00000000
00005408:00000000
0000540C:00000000
```

Figure 10. Stack View.

6.5 Cache Views

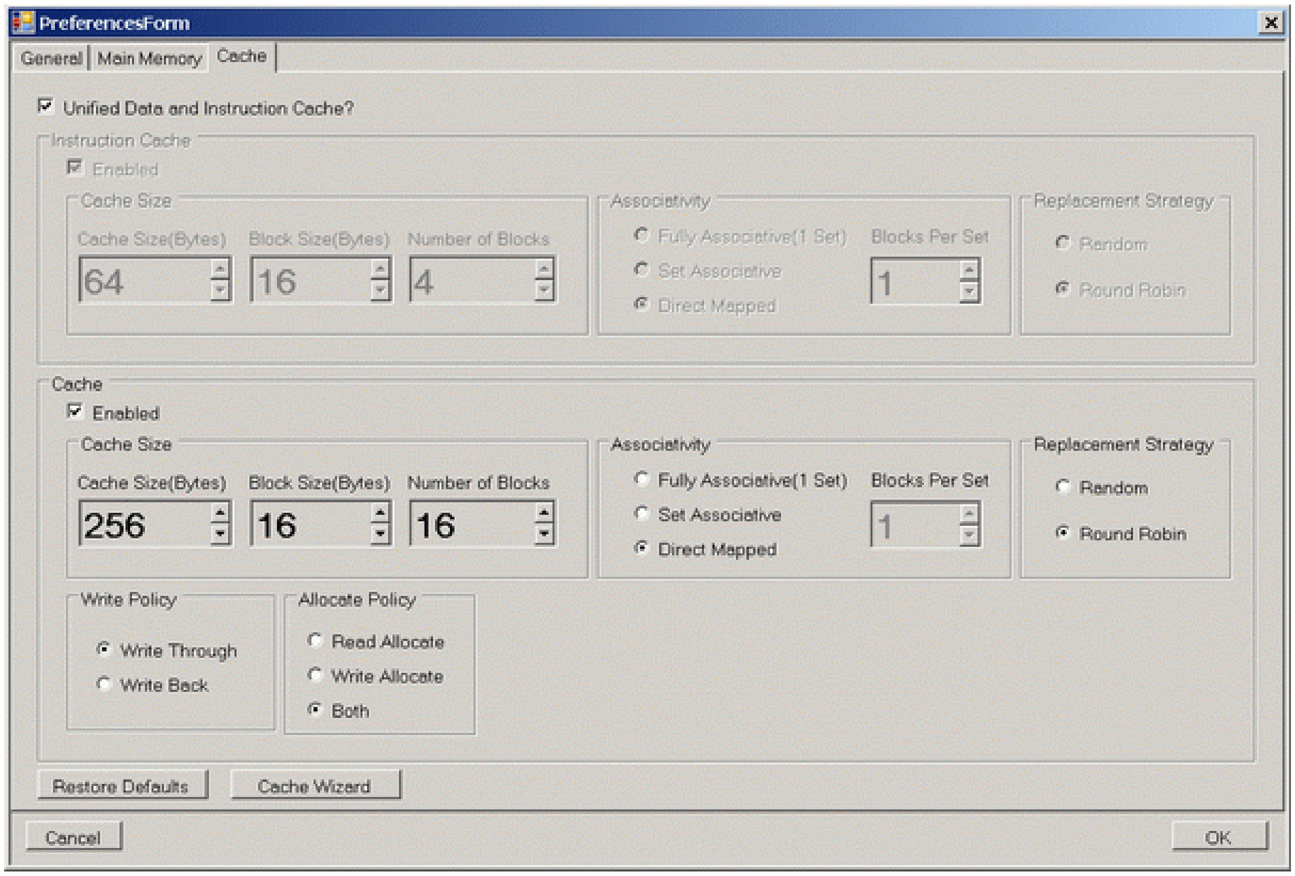
The **Cache Views** display the contents of the L1 cache. The cache can have different organizations. The one used by ARMSim can be selected by the user before an ARM program is executed. The cache can consist of either a unified data and instruction cache, displayed in the **Unified Cache View**, or separate data and instruction caches, displayed in the **Data** and **Instruction Cache Views**, respectively, depending on the cache properties selected by the user.

To set the cache properties, select **File > Preferences**, and click the **Cache** tab. Then, use either the **Cache Preferences Form** (see Figure 11) or the **Cache Wizard** to change the current cache settings, and click **OK**. To restore the default cache properties, select the **Restore Defaults** button on the **Cache Preferences Form**.

When using the **Cache Preferences Form** to set the cache properties, begin by selecting the type of cache. Table 3 lists the available cache configurations. Then, set the size of the cache(s). Once the **Cache Size** has been set, selecting a value for either the **Block Size** or the **Number of Blocks** causes the remaining settings in the **Cache Size** box to assume the appropriate values, so that the three properties satisfy the following equation:

Cache Size (bytes) = Block Size (bytes) × Number of Blocks

Next, select the **Associativity** of the cache(s). If **Set Associative** is selected, set the **Blocks per Set**, and select a **Replacement Strategy**. Finally, select the **Write** and **Allocate Policies** for the **Cache** or **Data Cache**.



[✓] Unified Data and Instruction Cache?

Instruction Cache
[✓] Enabled
Cache Size: Cache Size(Bytes) 64; Block Size(Bytes) 16; Number of Blocks 4
Associativity: Fully Associative(1 Set) / Set Associative / [✓ Direct Mapped]; Blocks Per Set 1
Replacement Strategy: Random / [✓ Round Robin]

Cache
[✓] Enabled
Cache Size: Cache Size(Bytes) 256; Block Size(Bytes) 16; Number of Blocks 16
Associativity: Fully Associative(1 Set) / Set Associative / [✓ Direct Mapped]; Blocks Per Set 1
Replacement Strategy: Random / [✓ Round Robin]
Write Policy: [✓ Write Through] / Write Back
Allocate Policy: Read Allocate / Write Allocate / [✓ Both]

Figure 11. Cache Preferences Form.

Table 3. Cache Configurations.

Configuration	*Settings*
Unified Data and Instruction Cache	Enable the **Unified Data and Instruction Cache**.
Separate Data and Instruction Caches	Disable the **Unified Data and Instruction Cache**, and enable the **Data Cache** and the **Instruction Cache**.
Data Cache Only	Disable the **Unified Data and Instruction Cache** and the **Instruction Cache**, and enable the **Data Cache**.
Instruction Cache Only	Disable the **Unified Data and Instruction Cache** and the **Data Cache**, and enable the **Instruction Cache**.
No Cache	Disable the **Unified Data and Instruction Cache**, the **Data Cache**, and the **Instruction Cache**.

In the **Cache Views**, the boundaries of sets are marked by the blue square brackets along the left-hand side of the view (see Figure 12). Each row consists of a memory address, followed by a cache block that shows the contents of the block at this address in memory.

When an instruction is executed using one of the step commands (see section 5.1) or when a sequence of instructions is executed using the **Debug > Run** option or the **Continue** button (see section 4.3), any cache blocks that were written to during the execution of the instruction(s) are highlighted after the execution of the instruction(s) has finished.

When the **Write Policy** is set to **Write Back**, a dirty block is marked by a red dot to the left of the row.

To clear all of the cache blocks, select **Cache > Reset**. Resetting the cache purges all of the dirty blocks, invalidates all of the cache blocks, and sets all of the cache statistics to zero. To purge all of the dirty blocks in the cache, select **Cache > Purge**. This command has no effect unless the **Write Policy** is set to **Write Back**.

To view the cache statistics, including the hit and miss rates, select **Cache > Statistics**. To clear all the cache statistics, click the **Reset** button on the **Cache Statistics** display.

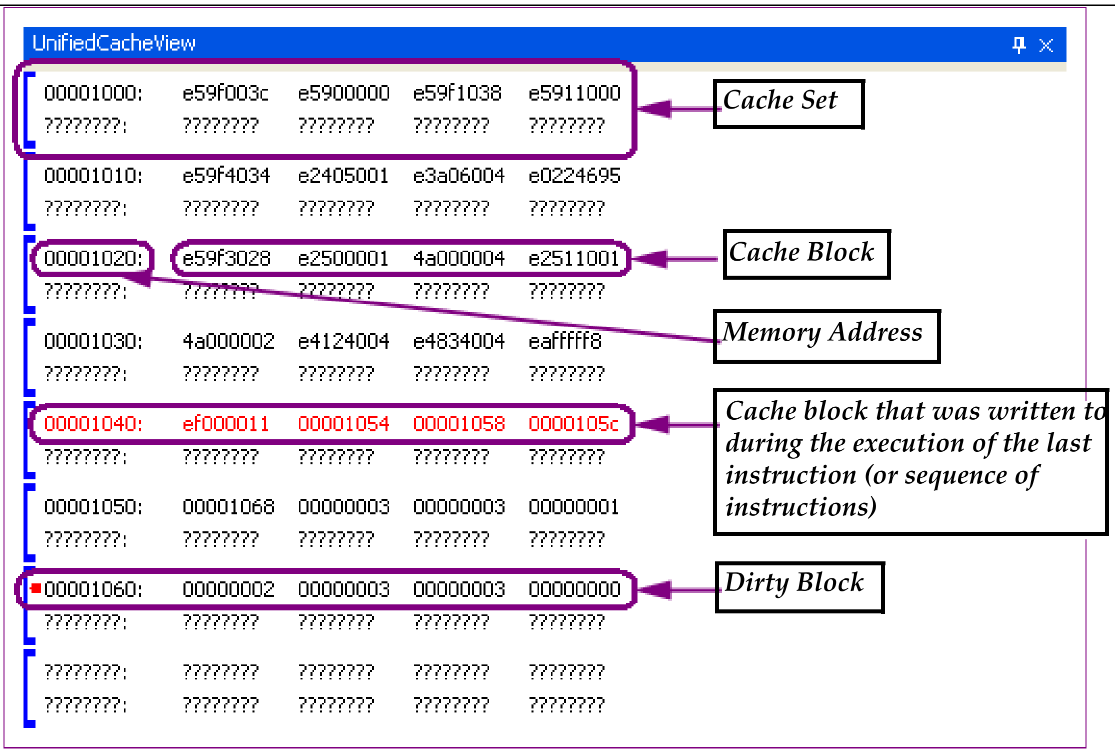
Note:

- The **Instruction Cache** is sometimes referred to as the **Code Cache**.

7. Some ARMSim Limitations

The ARMSim is an aid for learning the operation of the ARM architecture. It does not implement every feature that can be found on the ARM. Some of the more important limitations are listed below.

- The ARM architecture supports both little-endian and big-endian access to memory. The ARM-Sim supports only the little-endian format (the same as the Intel architecture which hosts the ARMSim).
- The ARM architecture has a special mode of execution called 'Thumb mode' which is intended for embedded system applications where memory is a scarce resource. Each thumb instruction occupies only 2 bytes. Thumb mode is not currently supported by ARMSim.


UnifiedCacheView
Cache Set
Cache Block
Memory Address
Cache block that was written to during the execution of the last instruction (or sequence of instructions)
Dirty Block

```
00001000:   e59f003c   e5900000   e59f1038   e5911000
????????:   ????????   ????????   ????????   ????????

00001010:   e59f4034   e2405001   e3a06004   e0224695
????????:   ????????   ????????   ????????   ????????

00001020:   e59f3028   e2500001   4a000004   e2511001
????????:   ????????   ????????   ????????   ????????

00001030:   4a000002   e4124004   e4834004   eafffff8
????????:   ????????   ????????   ????????   ????????

00001040:   ef000011   00001054   00001058   0000105c
????????:   ????????   ????????   ????????   ????????

00001050:   00001068   00000003   00000003   00000001
????????:   ????????   ????????   ????????   ????????

00001060:   00000002   00000003   00000003   00000000
????????:   ????????   ????????   ????????   ????????

????????:   ????????   ????????   ????????   ????????
????????:   ????????   ????????   ????????   ????????
```

Figure 12. Cache View.