



OVP Debugging with INSIGHT User Guide

Imperas Software Limited

Imperas Buildings, North Weston,
Thame, Oxfordshire, OX9 2HA, UK
docs@imperas.com



Author:	Imperas Software Limited
Version:	0.3
Filename:	OVPsim_Debugging_with_INSIGHT_User_Guide.doc
Project:	OVP Debugging with INSIGHT User Guide
Last Saved:	July 22, 2015
Keywords:	OVP insight debug

Copyright Notice

Copyright © 2015 Imperas Software Limited All rights reserved. This software and documentation contain information that is the property of Imperas Software Limited. The software and documentation are furnished under a license agreement and may be used or copied only in accordance with the terms of the license agreement. No part of the software and documentation may be reproduced, transmitted, or translated, in any form or by any means, electronic, mechanical, manual, optical, or otherwise, without prior written permission of Imperas Software Limited, or as expressly provided by the license agreement.

Right to Copy Documentation

The license agreement with Imperas permits licensee to make copies of the documentation for its internal use only. Each copy shall include all copyrights, trademarks, service marks, and proprietary rights notices, if any.

Destination Control Statement

All technical data contained in this publication is subject to the export control laws of the United States of America. Disclosure to nationals of other countries contrary to United States law is prohibited. It is the reader's responsibility to determine the applicable regulations and to comply with them.

Disclaimer

IMPERAS SOFTWARE LIMITED, AND ITS LICENSORS MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

Table of Contents

1	Preface.....	4
1.1	Notation.....	4
1.2	Related OVP Documents	4
2	Introduction.....	5
2.1	Prerequisites	5
3	Installation and Configuration of INSIGHT	6
3.1	SourceForge MinGW Project	6
3.2	Obtain and Install INSIGHT	6
3.3	Obtain and Install GDB	7
3.4	Configure the Environment.....	8
3.5	Check the Install	8
4	Debugging a Processor Model.....	10
4.1	Introduction.....	10
4.2	A Basic Platform.....	10
4.2.1	Building the Platform.....	11
4.2.2	Running the platform	12
4.3	Starting a Debug Session	12
4.4	Model DLL Load	13
4.5	An Example Debug Session.....	15
4.5.1	The Constructor Function	15
4.5.2	The Code Morph Function.....	17
4.5.3	The Decode Function.....	18
4.5.4	The Disassemble Function	19

1 Preface

This document describes how to debug an OVP simulator processor model using the INSIGHT debugger.

1.1 *Notation*

Code Code and command extracts

1.2 *Related OVP Documents*

- CpuManager and OVPsim User Guide

2 Introduction

The *OVPsim and CpuManager User Guide* describes how platforms containing any number of processor models and peripheral models can be constructed. This document describes how to debug models contained within a platform while it is simulating using the freely-available OVPsim simulation environment.

The debugging of models within the OVPsim environment is by no means limited to INSIGHT. This is one of a number of graphical user interfaces that could be used but has been chosen here for its simplicity of use and good feature set.

2.1 Prerequisites

GCC Compiler Versions

Linux32	4.5.2	i686-nptl-linux-gnu (Crosstool-ng)
Linux64	4.4.3	x86_64-unknown-linux-gnu (Crosstool-ng)
Windows32	4.4.7	mingw-w32-bin_i686-mingw
Windows64	4.4.7	mingw-w64-bin_i686-mingw

For Windows environments, Imperas recommends using MinGW (www.mingw.org) and MSYS.

There are two components used in the example given in this document. The processor model uses the opencores OR1K processor model and tool chain. The peripheral model uses a DMA Controller. Both are available to download from the www.ovpworld.org website or as part of an Imperas installation.

3 Installation and Configuration of INSIGHT

3.1 *SourceForge MinGW Project*

To use the INSIGHT graphical debug interface on Windows with OVPsim, two modules need to be downloaded from the mingw project on sourceforge, <http://sourceforge.net/projects/mingw>

Select the MinGW download pages using the link:



Download MinGW - Minimalist GNU for Windows

Project Admins: earnie, infidel, keithmarshall
Operating System: MinGW/MSYS (MS Windows), All 32-bit MS Windows (95/98/NT/2000/XP)
License: BSD License, GNU General Public License (GPL), Public Domain
Category: Build Tools, Code Generators, Compilers, Debuggers, Interpreters

3.2 *Obtain and Install INSIGHT*

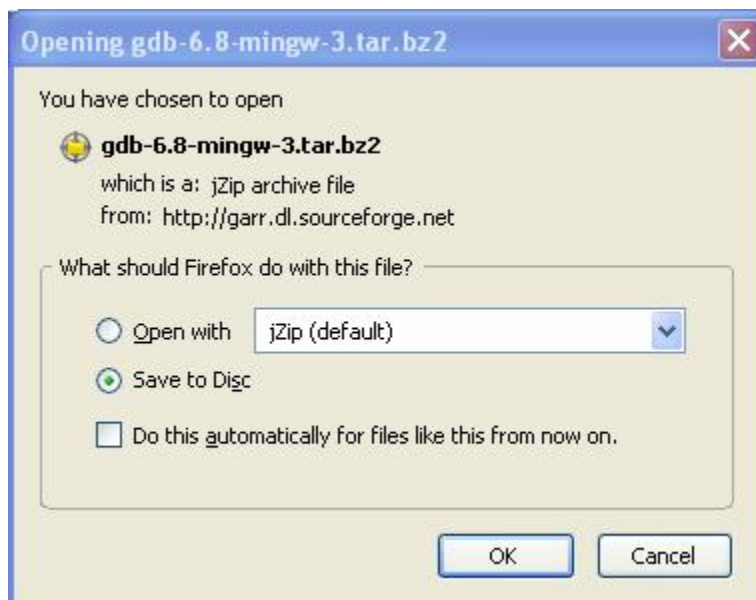
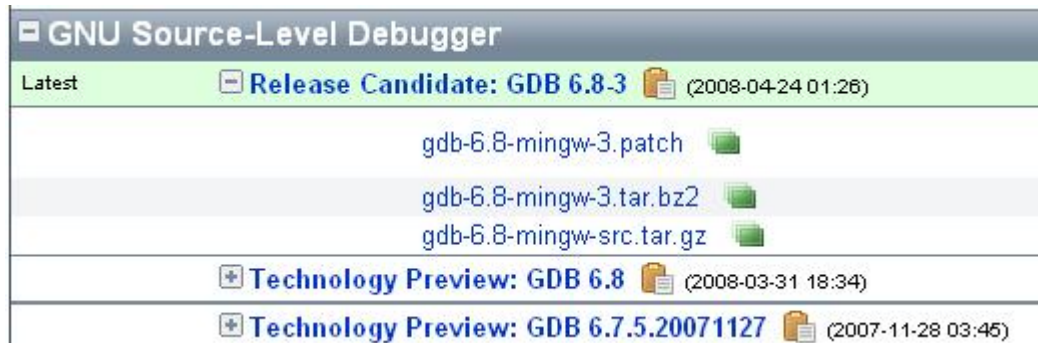
Download the win32 installer

User Contributed: Miscellany		
Latest	+ GCC with Dwarf2-EH	(2006-02-11 10:06)
	insight	(2005-09-24 20:57)
	mingw-insight-6.3.exe	
	mingw-insight-6.3-src.tar.bz2	
	+ mpatrol	(2003-07-21 04:00)

Execute the installer and select the base of the MSYS install for the INSIGHT install location, for example C:/msys/1.0. This will add files into the bin and lib directories.

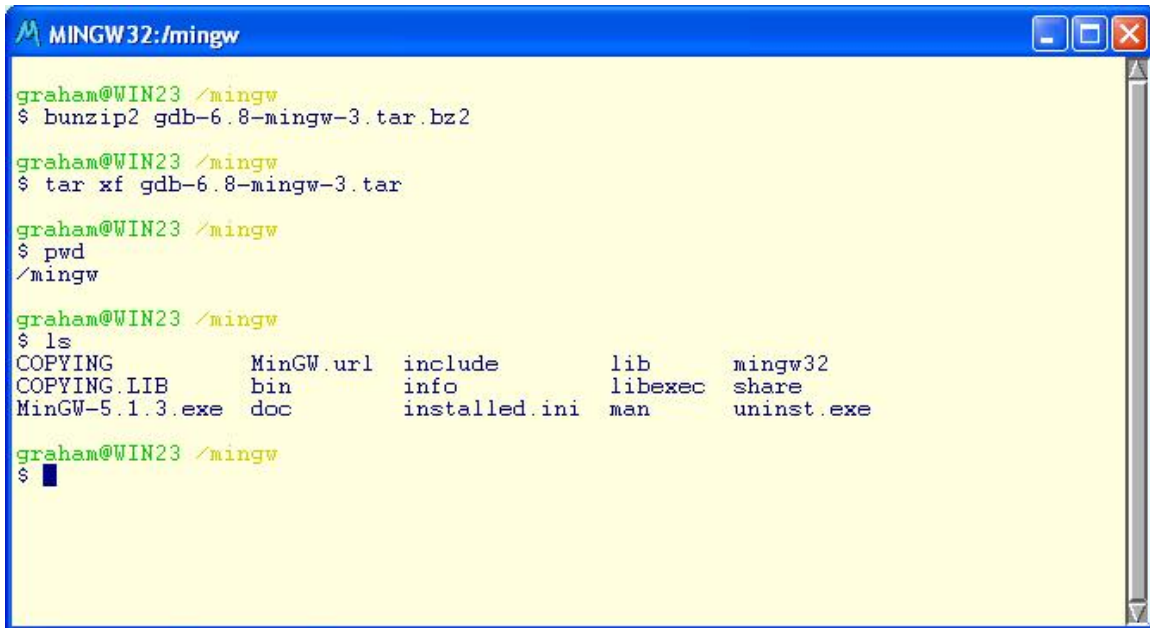
3.3 Obtain and Install GDB

Download the bzip tar file



Uncompress the gdb tar file in the MSYS shell using bunzip2 or another equivalent program.

The tar file should be extracted into the MINGW install directory, for example C:/msys/1.0/mingw. This can all be done in an MSYS shell



```
MINGW32:/mingw
graham@WIN23 /mingw
$ bunzip2 gdb-6.8-mingw-3.tar.bz2

graham@WIN23 /mingw
$ tar xf gdb-6.8-mingw-3.tar

graham@WIN23 /mingw
$ pwd
/mingw

graham@WIN23 /mingw
$ ls
COPYING      MinGW.url    include      lib           mingw32
COPYING.LIB  bin          info         libexec       share
MinGW-5.1.3.exe doc          installed.ini man           uninst.exe

graham@WIN23 /mingw
$
```

3.4 *Configure the Environment*

The environment requires that the insight/bin directory is added to the PATH so that the executable is found. All other libraries and binaries should be found using the standard MSYS and MINGW environments.



```
MINGW32:-
graham@WIN23 ~
$ export PATH=$PATH:/insight/bin

graham@WIN23 ~
$
```

3.5 *Check the Install*

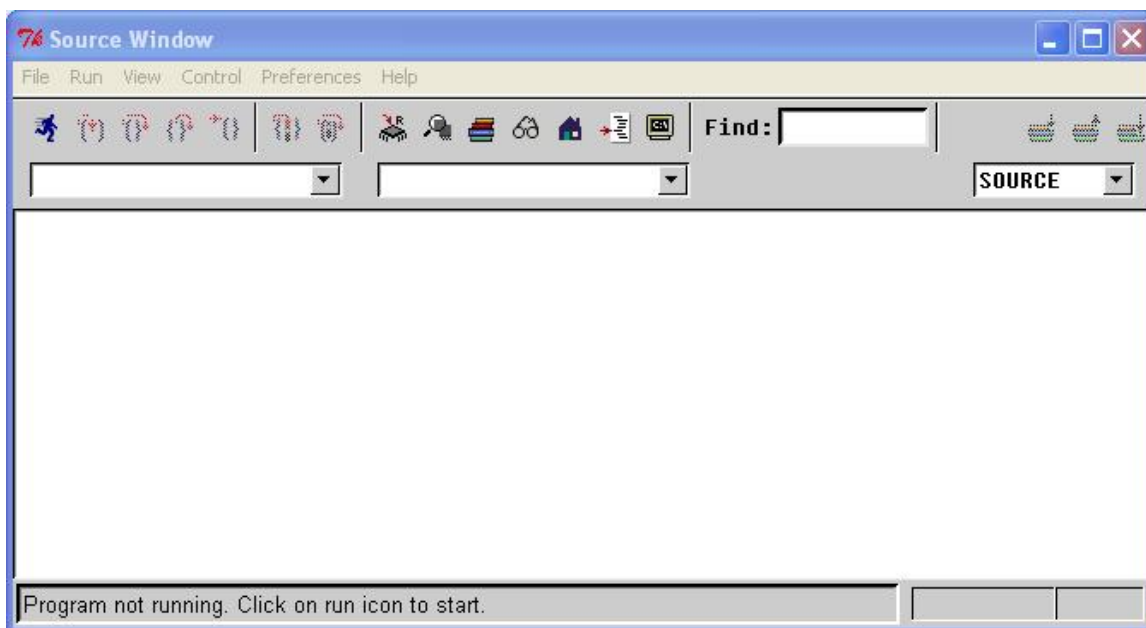
You now have a graphical debugger that can be used to debug the processor and peripheral models in an OVP simulation.

The Insight GUI may be started from an MSYS shell by typing 'insight'.



```
MINGW32:-
graham@WIN23 ~
$ insight
```

The insight graphical interface should start



4 Debugging a Processor Model

4.1 Introduction

A processor model is a set of functions that define the behavior of individual instructions using the VMI API. This generates a code morpher i.e. morphs 'your' processor instructions into native x86 processor instructions. The morph code program is run just-in-time to satisfy the requirements of the simulator.

This document deals with debugging the flow through the morph program.

4.2 A Basic Platform

A suitable single-processor platform example is available in the directory:

```
$IMPERAS_HOME/Examples/Platforms/simple
```

This uses the freely-available OR1K processor (see <http://www.opencores.org/projects.cgi/web/or1k/architecture>).

The test platform source is in file platform/platform.c:

```
// enable tracing etc. on processor model
#define MODEL_ATTRS (ICM_ATTR_DEFAULT)

//
// Create platform
//
void createPlatform() {

    // Initialize ICM
    icmInitAttrs icmAttrs = ICM_VERBOSE | ICM_STOP_ON_CTRL_C;
    icmInitPlatform(ICM_VERSION, icmAttrs, 0, 0, "platform");

    // select library components
    const char *vlnvRoot = NULL; //When NULL use default library
    const char *model = icmGetVlnvString(
        vlnvRoot, "ovpworld.org", "processor", "or1k", "1.0", "model"
    );
    const char *semihosting = icmGetVlnvString(
        vlnvRoot, "ovpworld.org", "modelSupport", "imperasExit", "1.0", "model"
    );

    // create a processor
    icmNewProcessor(
        "cpu1",           // CPU name
        "or1k",           // CPU type
        0,                // CPU cpuId
        0,                // CPU model flags
        32,               // address bits
        model,            // model file
        "modelAttrs",     // morpher attributes
        MODEL_ATTRS,     // simulation attributes. enable tracing etc
    );
}
```

```
        0,                // user-defined attributes
        semihosting,      // semi-hosting file
        "modelAttrs"      // semi-hosting attributes
    );

    // No memory or bus connections are created in this platform
    // The simulator assumes and creates a memory connection to the
    // full memory space this processor can access.
}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//                                     M A I N                                     //
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

static Bool cmdParser(int argc, const char *argv[]);

int main(int argc, const char *argv[])
{
    // Check arguments and ensure application to load specified
    if(!cmdParser(argc, argv)) {
        icmMessage("E", "platform", "Command Line parser error");
        icmExitSimulation(1);
    }

    // the constructor
    createPlatform();

    icmSimulationStarting();

    // run simulation
    icmSimulatePlatform();

    // terminate simulation and free simulation data structures
    icmTerminate();

    icmExitSimulation(0);
}
```

For a full explanation of OVPsim platform construction please see the *CpuManager and OVPsim User Guide*.

4.2.1 Building the Platform

The OVPsim examples are written to work with GCC and MAKE which are typically available on Linux and can be installed on Windows as part of MinGW and MSYS (see section 2.1.) The example commands below assume you are using a shell on Linux or MSYS.

Take a copy of one of the processor model example files, here we are using the version showing the addition of simple behavior:

```
cp -r $IMPERAS_HOME/Examples/Models/Processor/4.orlkBehaviourSimple .
```

The test platform, application and processor model can be compiled to produce an executable, *platform.\${IMPERAS_ARCH}.exe*, an elf file, *application.OR1K.elf*, and a DLL, *model.dll*, by using `make` in the example directory:

```
cd 4.orlkBehaviourSimple
```

```
make
```

NOTE: This will, by default, build a non-optimized version of the processor model for debugging.

4.2.2 Running the platform

Start the OVP simulator with the example platform by running the native platform executable built earlier. This simple platform takes a single argument which is OR1K application to run on the simulated processor.

```
platform/platform.${IMPERAS_ARCH}.exe \  
--program application/application.OR1K.elf
```

```
OVPsim v20080625.0 Open Virtual Platform simulator from www.OVPworld.org.  
Copyright (C) 2005-2008 Imperas Ltd.  Contains Imperas Proprietary Information.  
Licensed Software, All Rights Reserved.  
Visit www.imperas.com for multicore debug, verification and analysis solutions.  
OVPsim started: Wed Jul 23 11:50:31 2008
```

```
Info 'cpul' REGISTERS  
CPU cpul (instruction 1):  
    0: 00000000 00000000 deadbeef deadbeef  
   16: deadbeef deadbeef deadbeef deadbeef  
   32: deadbeef deadbeef deadbeef deadbeef  
  
... etc ...  
  
   80: 00000000 00000000 00000000 00000000  
   96: 00000000 00000000 00000000 00000000  
  112: 00000000 00000000 00000000 00000000  
Info 'cpul', 0x00000000000000178: ??? instruction:0x1820ffff  
CPU 'cpul' 0x000000178:0x1820ffff *** undecoded instruction: exiting ***
```

```
OVPsim finished: Wed Jul 23 11:50:31 2008  
Visit www.imperas.com for multicore debug, verification and analysis solutions.  
OVPsim v20080625.0 Open Virtual Platform simulator from www.OVPworld.org.
```

4.3 Starting a Debug Session

Ensure that the processor model has been built with debug enabled, i.e. using `-g -gdwarf2`

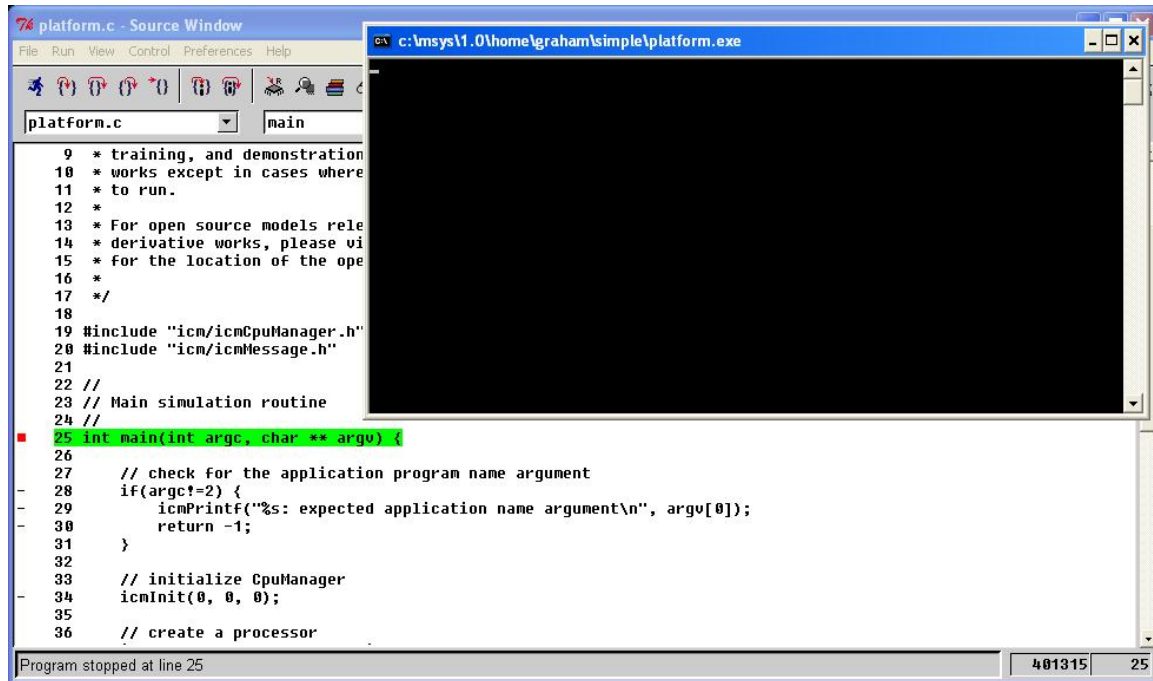
Invoke insight on the platform executable.

```
insight --args platform/platform.${IMPERAS_ARCH}.exe \  
--program application/application.OR1K.elf
```

By using the `--args` argument we pass the command line parameters from this command line directly through to the execution on GDB.

As with all GDB a breakpoint will be created on main so we can immediately run and the simulation will stop at main. Select run from the pull down menu.

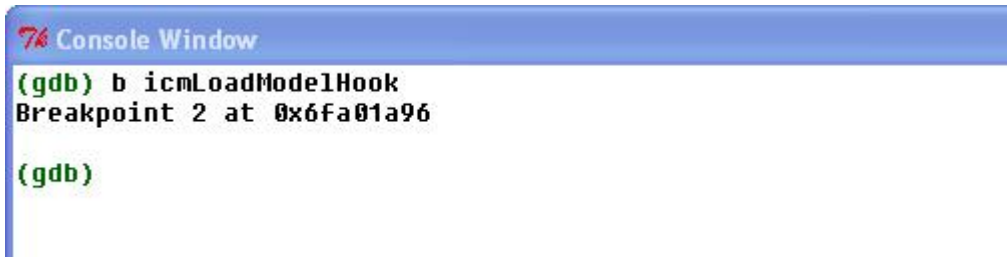
The simulation will start, a shell will be created and the simulation will stop at main in the platform.c file.



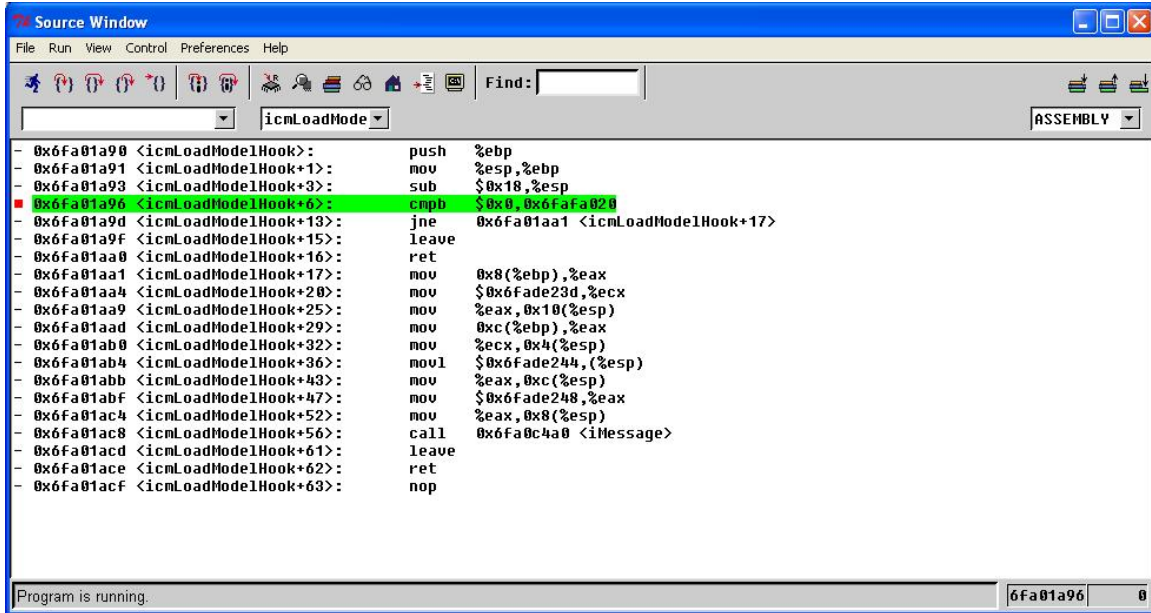
At this stage the processor model DLL has not been loaded. We can only view the source for the processor model once loaded so we must run the simulation until the point at which the DLL has been loaded. This will be shown in the next sections

4.4 Model DLL Load

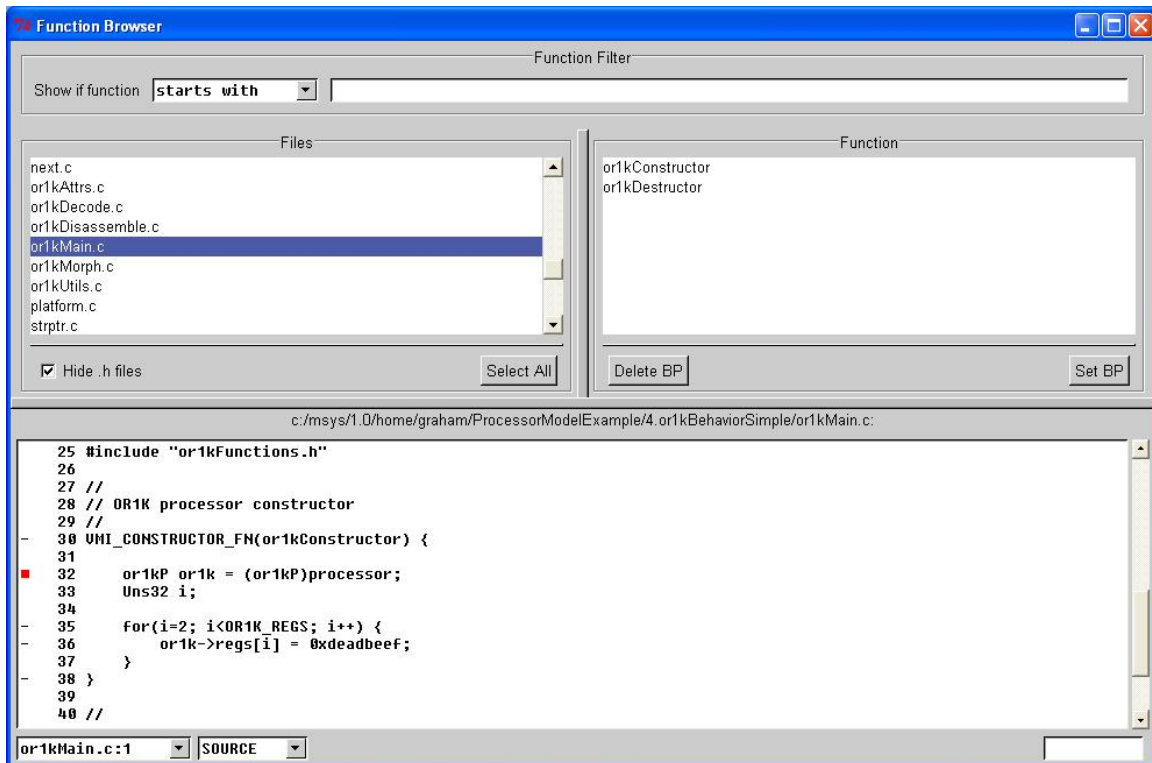
The model DLL is loaded when the `icmNewProcessor` API call is made. After this call has completed we can use the `view->Function Browser` option to select the source files for the processor model and set breakpoints. However, once this call has completed the processor model constructor will also have completed. If we wish to examine the execution of the processor model execution we need a way of stopping the simulation after the model is loaded but before anything is executed. This is done by setting a breakpoint the symbol `icmLoadModelHook`. This breakpoint will have to be set from the console, `view->Console`.



We can now continue the simulation to the breakpoint.



Once we have reached this point all the model source code is available to view in the Function Browser. We are now able to set breakpoints and debug the model.



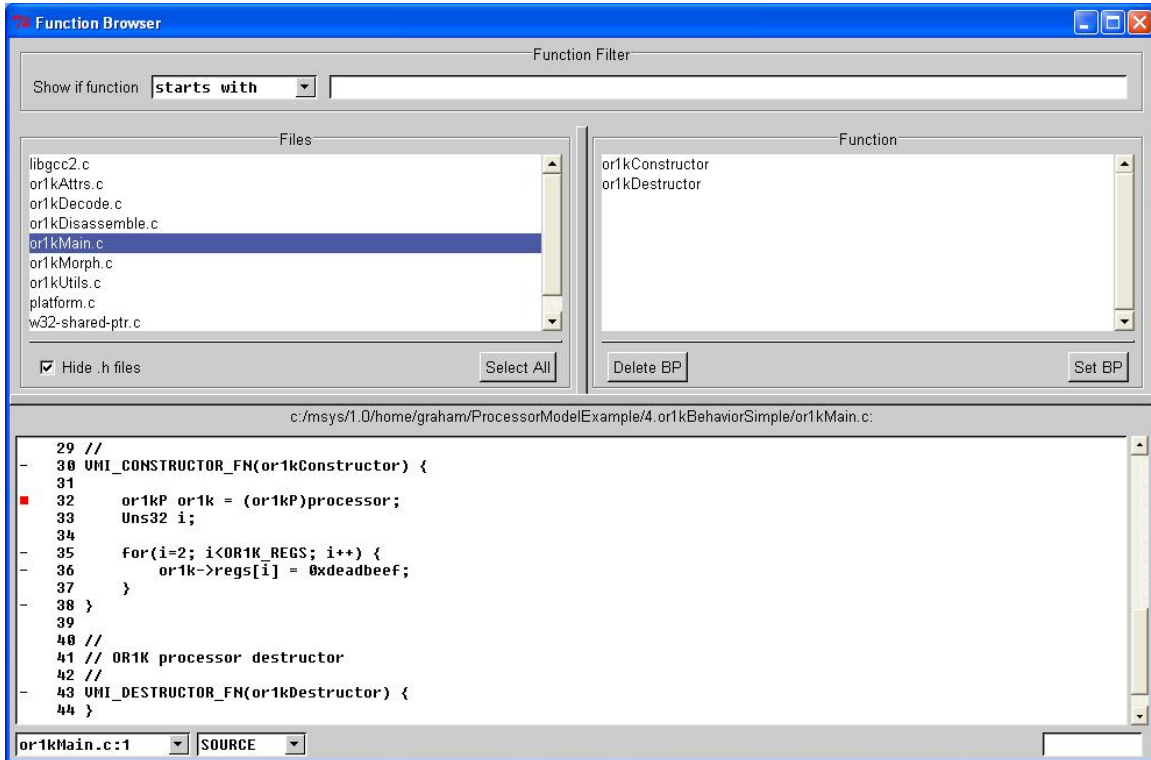
4.5 An Example Debug Session

This section takes a look at some of the aspects in the debug of the processor model.

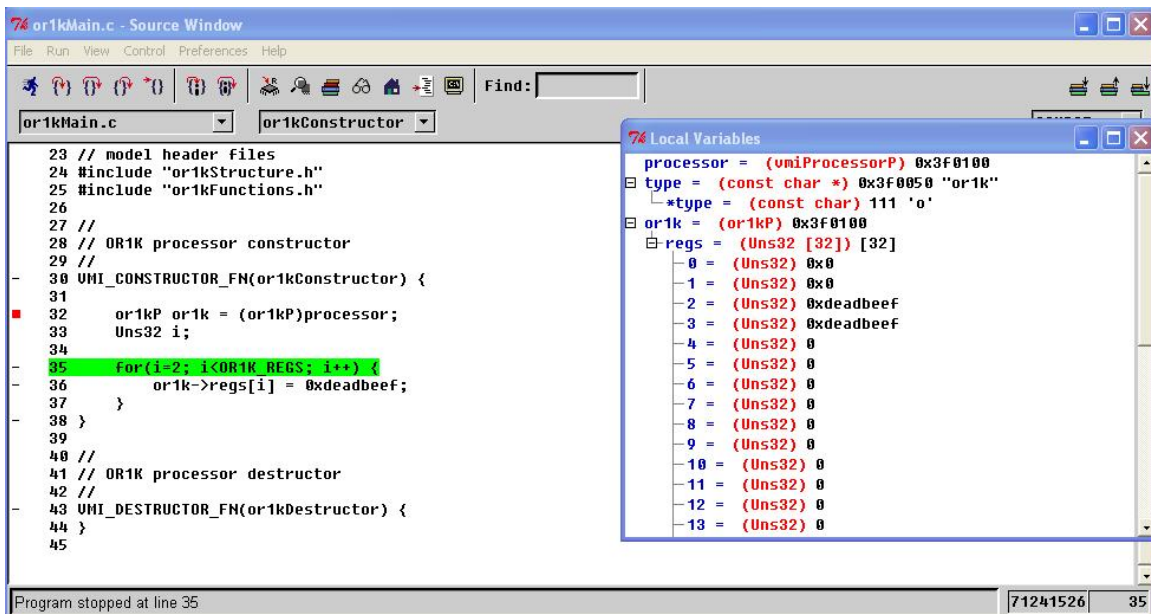
4.5.1 The Constructor Function

The very first function called by the simulator is called the constructor. This is used for carrying out initialization or other configuration of the model. These may be based on attributes passed to the model when the processor is instantiated.

The **view->Function Browser** can be used to set a breakpoint in the constructor; but only after the model is loaded.



Using the **view->Local Variables** window we can see how the processor state is effected as the constructor executes.

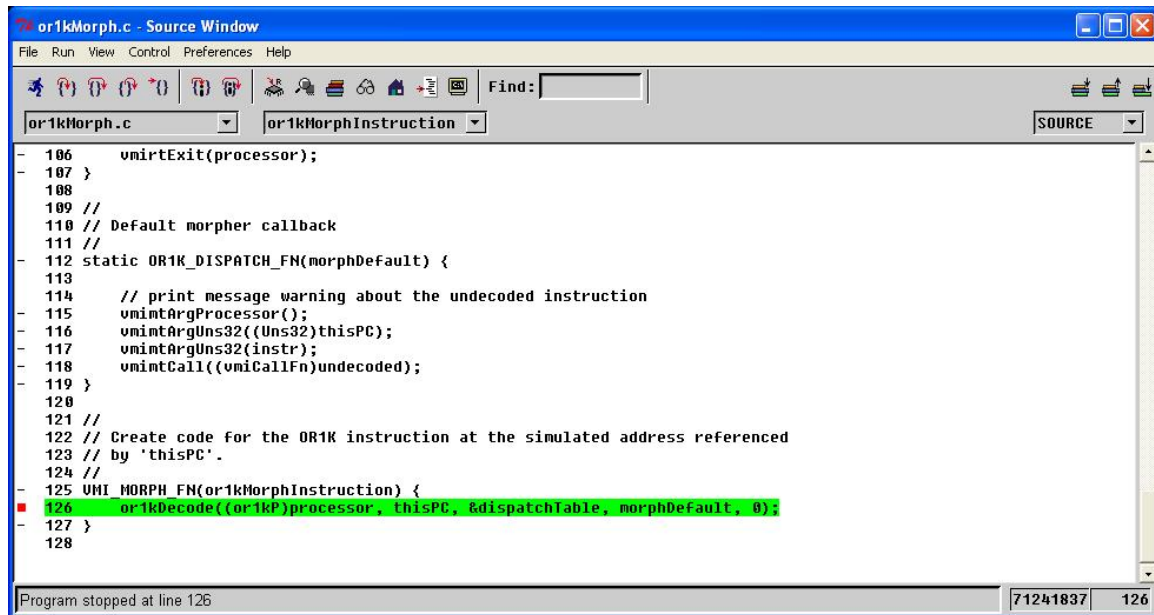


4.5.2 The Code Morph Function

The code morphing function describes on an instruction by instruction basis how to create its behavior using the VMI API.

The morph function is called for each address the simulator reaches to create the code to be executed. A complete code block ¹is created.

A breakpoint may be set on the code morphing function. This uses the common instruction decoder and passes a table of morph functions in the *dispatchTable*.



By stepping into the decode table we can follow the flow through into the leaf function that provide the behavior for the instruction.

¹ A code block is a set of instructions that reside in contiguous memory locations. The code block may be terminated by API calls, by instructions causing a change in the flow (jumps, branches, calls etc), by memory paging etc

```

24
25 // model header files
26 #include "or1kDecode.h"
27 #include "or1kFunctions.h"
28 #include "or1kInstructions.h"
29 #include "or1kStructure.h"
30
31 //
32 // Emit code to implement a binary/signed literal OR1K instruction
33 //
34 static void doBinopSLit(Uns32 instr, vmiBinop op) {
35
36     Uns32 rd    = OP2_D(instr);
37     Uns32 ra    = OP2_A(instr);
38     Int16 i     = OP2_I(instr);
39     vmiReg target = (rd==0) ? UMI_NOREG : OR1K_REG(rd);
40
41     if(ra==0) {
42         vmiBinopRCC(OR1K_BITS, op, target, 0, i, 0);
43     } else {
44         vmiBinopRRC(OR1K_BITS, op, target, OR1K_REG(ra), i, 0);
45     }
46 }
47
48 //

```

Program stopped at line 38

7124155e 38

Integral to the morph function is the decode function.

4.5.3 The Decode Function

The decode function parses the instructions loaded from memory and then on finding a match calls into one of the entries in the table that it is passed.

```

64     or1kDispatchTableCP table,
65     or1kDispatchFn      defaultCB,
66     void                 *userData
67 ) {
68     // get the instruction at the passed address - always 4 bytes on OR1K
69     vmiProcessorP processor = (vmiProcessorP)or1k;
70     Uns32 instruction = vmiCxtFetch4Byte(processor, thisPC);
71     or1kInstructionType type;
72
73     // search for first matching entry in decode table and call the
74     // corresponding dispatch function
75     for(type=(or1kInstructionType)0; type<OR1K_IT_LAST; type++) {
76         if(decodeTable[type](instruction)) {
77             ((*table)[type])(or1k, thisPC, instruction, userData);
78             return True;
79         }
80     }
81
82     // call the default dispatch function if no match
83     defaultCB(or1k, thisPC, instruction, userData);
84     return False;
85 }
86

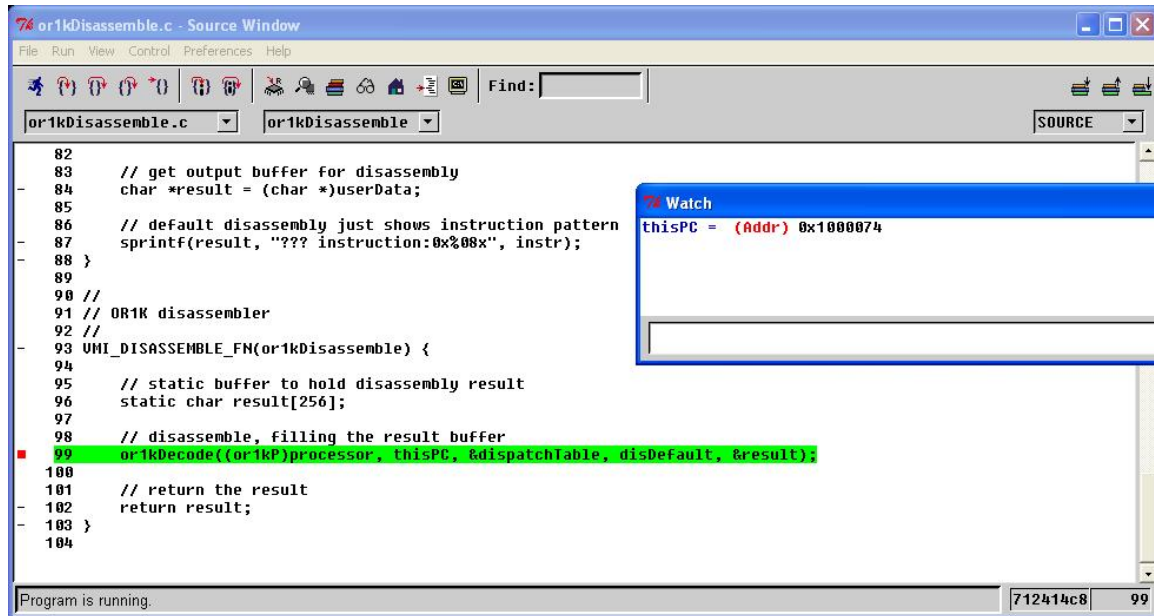
```

Program stopped at line 76

7124129b 76

4.5.4 The Disassemble Function

Finally the disassembler also calls through the decode function to a dispatch table that will display the disassembler output of the instruction at the passed address.



###