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:	January 13, 2020
Keywords:	OVP insight debug

Copyright Notice

Right to Copy Documentation

Destination Control Statement

Disclaimer

Table of Contents

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.

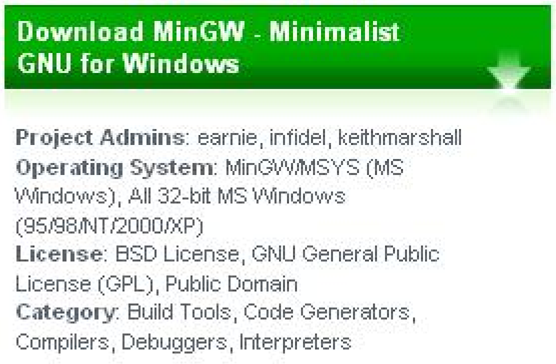
There are two components used in the example given in this document. The processor model uses the opencores OR1K processor model and tool chain. The peripheral model uses a DMA Controller. Both are available to download from the www.ovpworld.org website or as part of an Imperas installation.

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:

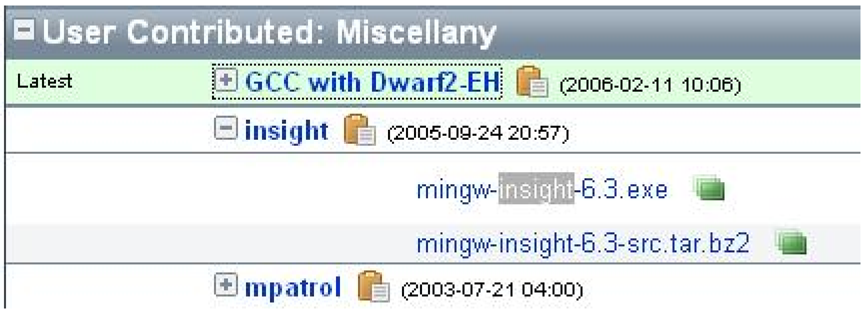


Download MinGW - Minimalist GNU for Windows

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



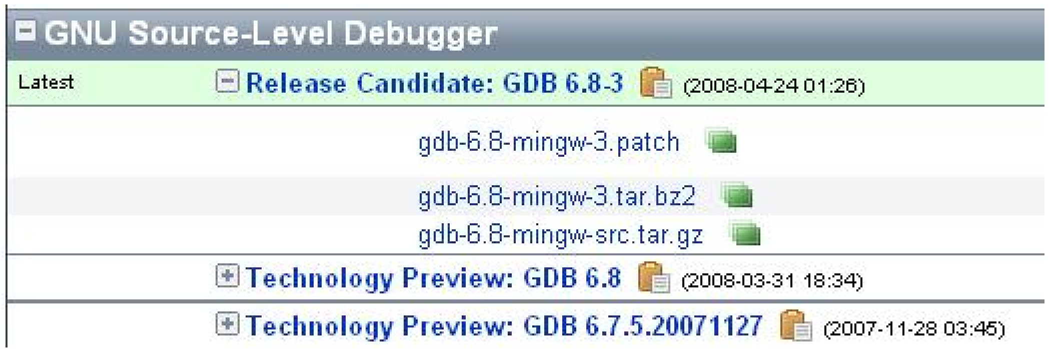
User Contributed: Miscellany

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.

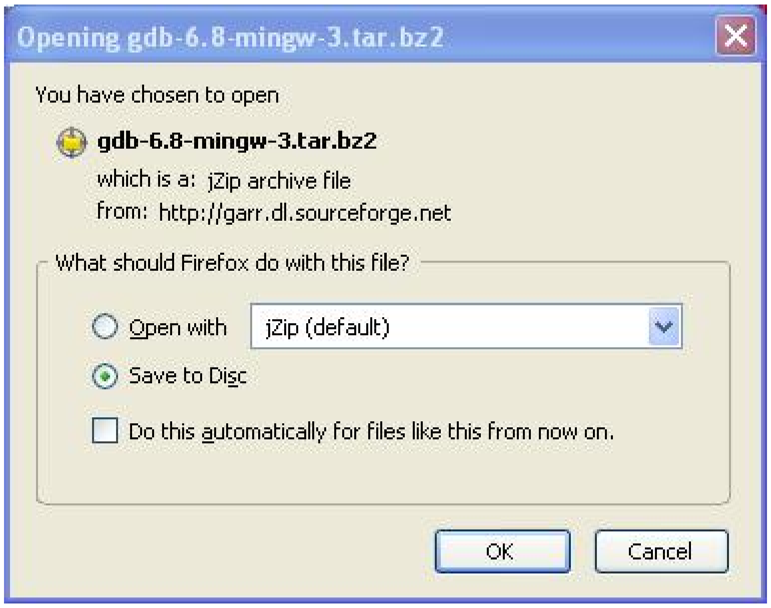
3.3 Obtain and Install GDB

Download the bzip tar file



GNU Source-Level Debugger

Latest	Release Candidate: GDB 6.8-3	(2008-04-24 01:26)
	gdb-6.8-mingw-3.patch	
	gdb-6.8-mingw-3.tar.bz2	
	gdb-6.8-mingw-src.tar.gz	
	Technology Preview: GDB 6.8	(2008-03-31 18:34)
	Technology Preview: GDB 6.7.5.20071127	(2007-11-28 03:45)



Opening gdb-6.8-mingw-3.tar.bz2

You have chosen to open

gdb-6.8-mingw-3.tar.bz2

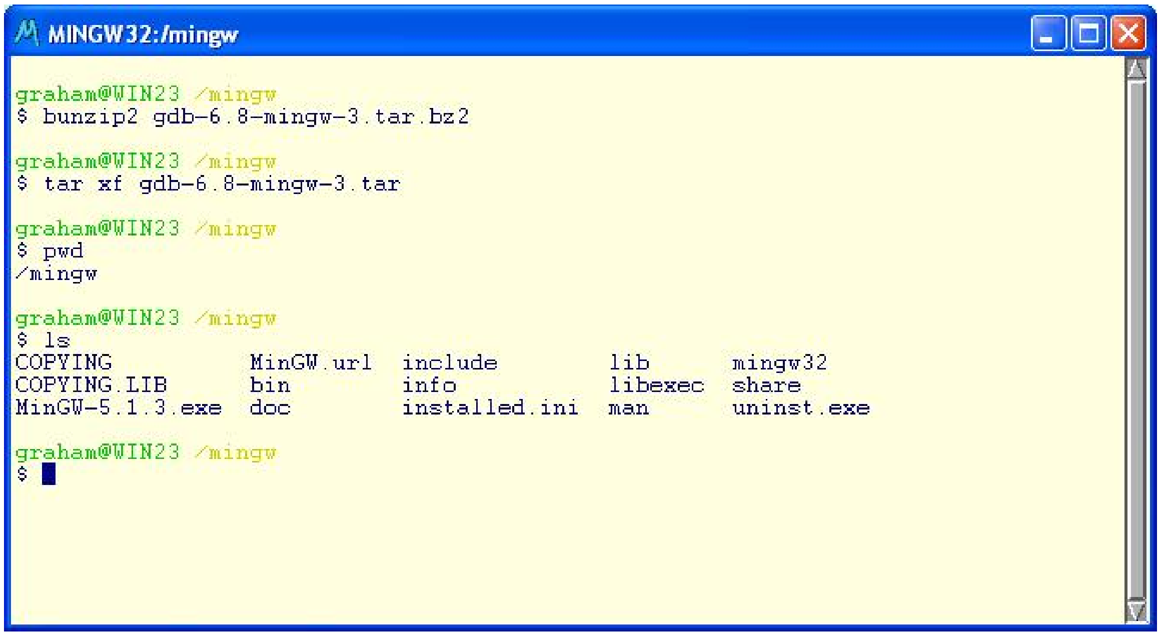
which is a: jZip archive file
from: http://garr.dl.sourceforge.net

What should Firefox do with this file?

( ) Open with jZip (default)
[✓ Save to Disc]
[ ] Do this automatically for files like this from now on.

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

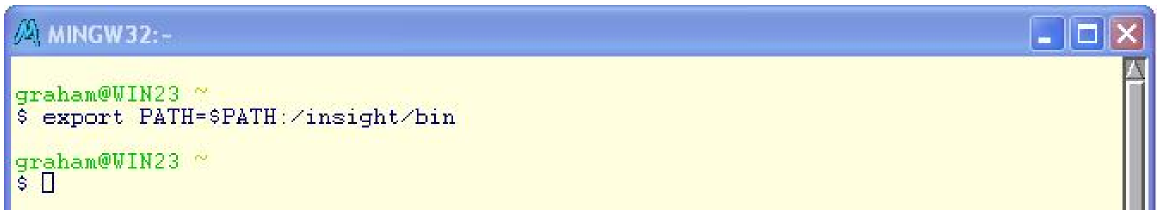
The tar file should be extracted into the MINGW install directory, for example C:/msys/1.0/mingw. This can all be done in an MSYS shell



```
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.

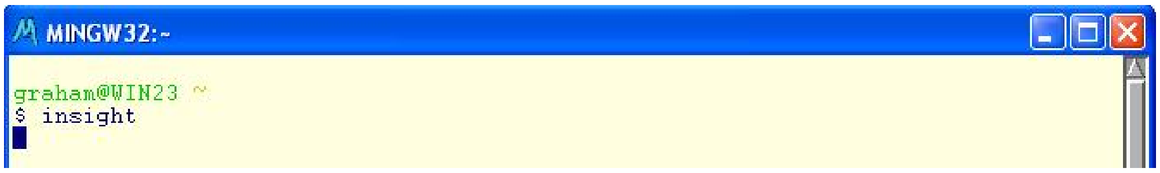


```
graham@WIN23 ~
$ export PATH=$PATH:/insight/bin

graham@WIN23 ~
$ 
```

3.5 Check the Install

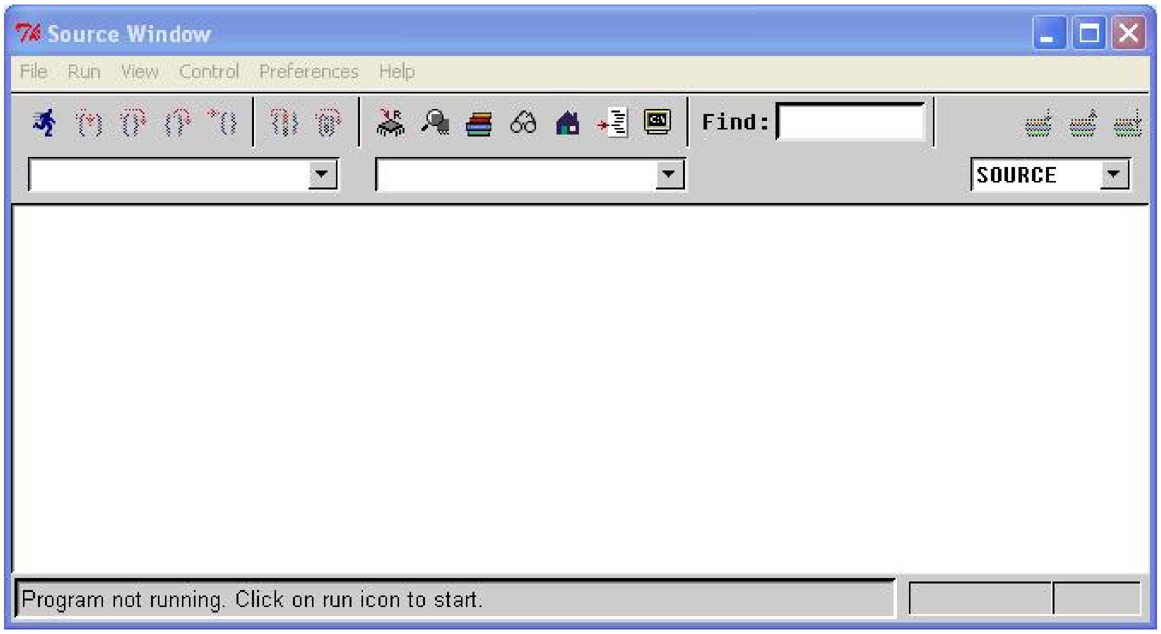
You now have a graphical debugger that can be used to debug the processor and peripheral models in an OVP simulation.

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



```
graham@WIN23 ~
$ insight
```

The insight graphical interface should start



Program not running. Click on run icon to 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/PlatformsICM/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_CTRLC;
    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.or1kBehaviourSimple .
```

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.or1kBehaviourSimple
    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 'cpu1' REGISTERS
CPU cpu1 (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 'cpu1', 0x0000000000000178: ??? instruction:0x1820ffff
CPU 'cpu1' 0x00000178:0x1820ffff *** 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

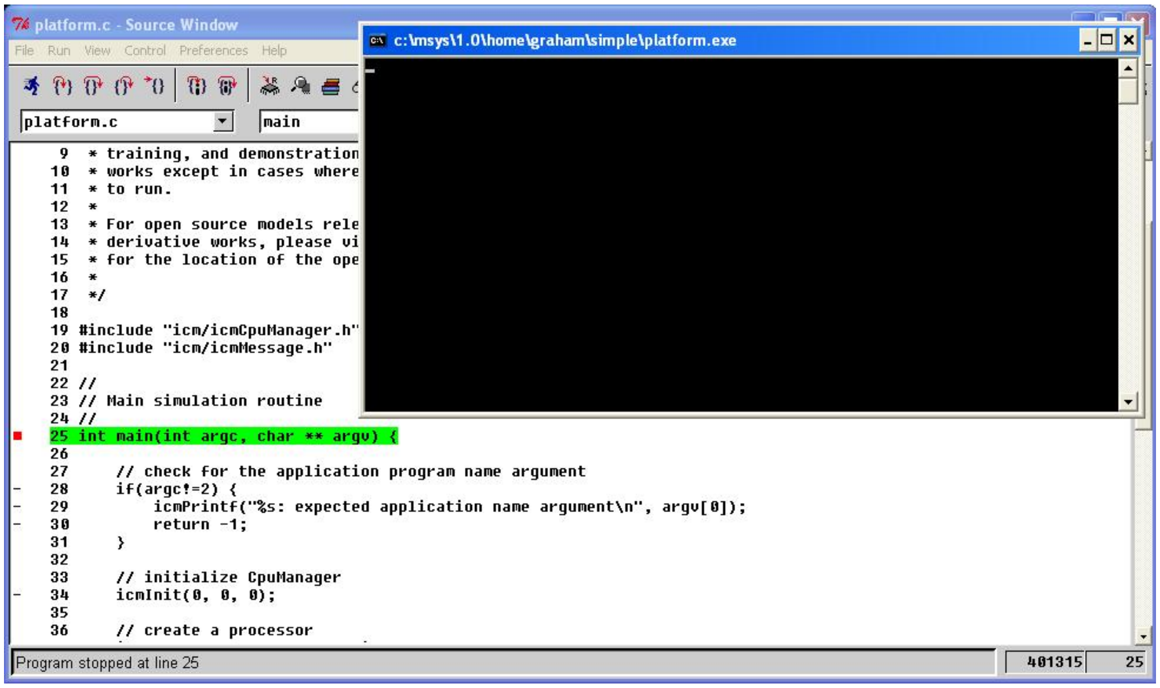
Invoke insight on the platform executable.

```
 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.


platform.c - Source Window
File Run View Control Preferences Help
platform.c
main
c:\msys\1.0\home\graham\simple\platform.exe

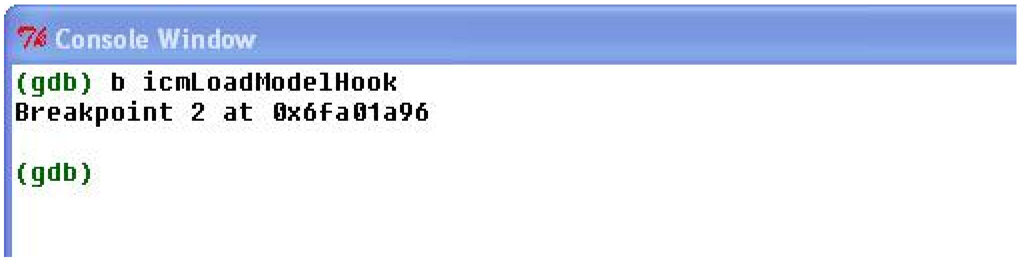
```
 * training, and demonstration
 * works except in cases where
 * to run.
 *
 * For open source models rele
 * derivative works, please vi
 * for the location of the ope
 *
 */

#include "icm/icmCpuManager.h"
#include "icm/icmMessage.h"

//
// Main simulation routine
//
int main(int argc, char ** argv) {

    // check for the application program name argument
    if(argc!=2) {
        icmPrintf("%s: expected application name argument\n", argv[0]);
        return -1;
    }

    // initialize CpuManager
    icmInit(0, 0, 0);

    // create a processor
```

```
Program stopped at line 25    401315    25
```

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

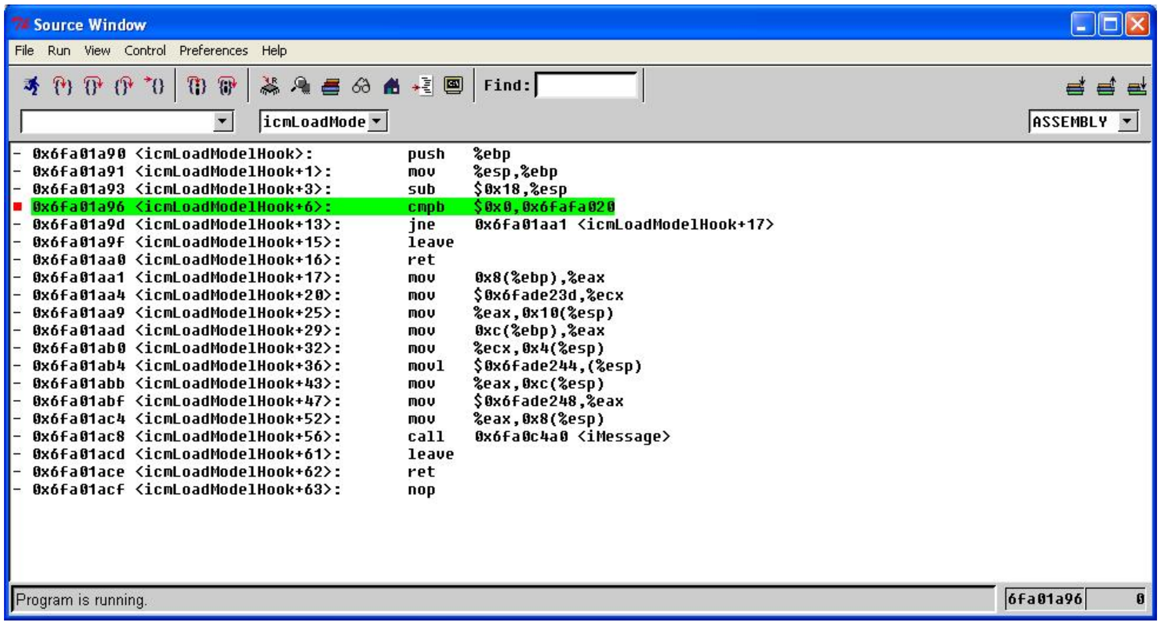
4.4 Model DLL Load

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*.



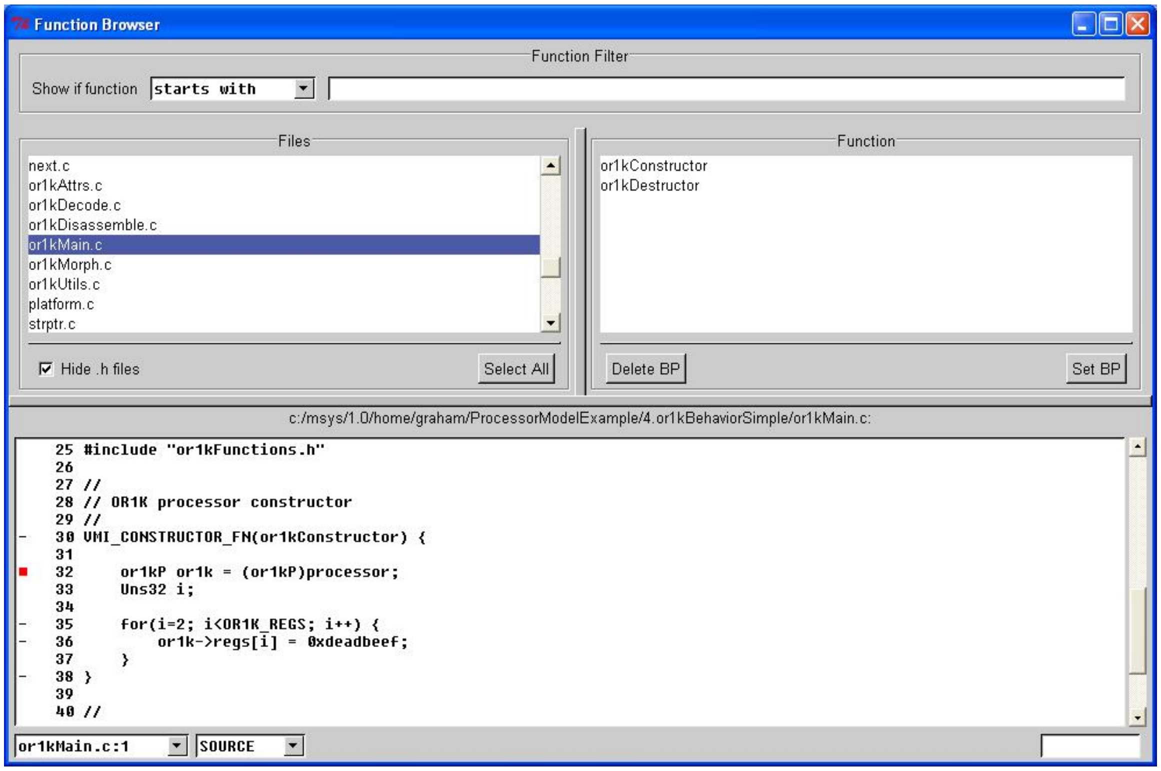
```
(gdb) b icmLoadModelHook
Breakpoint 2 at 0x6fa01a96

(gdb)
```

We can now continue the simulation to the breakpoint.



```
- 0x6fa01a90 <icmLoadModelHook>:        push   %ebp
- 0x6fa01a91 <icmLoadModelHook+1>:      mov    %esp,%ebp
- 0x6fa01a93 <icmLoadModelHook+3>:      sub    $0x18,%esp
■ 0x6fa01a96 <icmLoadModelHook+6>:      cmpb   $0x0,0x6fafa020
- 0x6fa01a9d <icmLoadModelHook+13>:     jne    0x6fa01aa1 <icmLoadModelHook+17>
- 0x6fa01a9f <icmLoadModelHook+15>:     leave
- 0x6fa01aa0 <icmLoadModelHook+16>:     ret
- 0x6fa01aa1 <icmLoadModelHook+17>:     mov    0x8(%ebp),%eax
- 0x6fa01aa4 <icmLoadModelHook+20>:     mov    $0x6fade23d,%ecx
- 0x6fa01aa9 <icmLoadModelHook+25>:     mov    %eax,0x10(%esp)
- 0x6fa01aad <icmLoadModelHook+29>:     mov    0xc(%ebp),%eax
- 0x6fa01ab0 <icmLoadModelHook+32>:     mov    %ecx,0x4(%esp)
- 0x6fa01ab4 <icmLoadModelHook+36>:     movl   $0x6fade244,(%esp)
- 0x6fa01abb <icmLoadModelHook+43>:     mov    %eax,0xc(%esp)
- 0x6fa01abf <icmLoadModelHook+47>:     mov    $0x6fade248,%eax
- 0x6fa01ac4 <icmLoadModelHook+52>:     mov    %eax,0x8(%esp)
- 0x6fa01ac8 <icmLoadModelHook+56>:     call   0x6fa0c4a0 <iMessage>
- 0x6fa01acd <icmLoadModelHook+61>:     leave
- 0x6fa01ace <icmLoadModelHook+62>:     ret
- 0x6fa01acf <icmLoadModelHook+63>:     nop

Program is running.                                   6fa01a96    0
```

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.



next.c
or1kAttrs.c
or1kDecode.c
or1kDisassemble.c
or1kMain.c
or1kMorph.c
or1kUtils.c
platform.c
strptr.c

or1kConstructor
or1kDestructor

c:/msys/1.0/home/graham/ProcessorModelExample/4.or1kBehaviorSimple/or1kMain.c:

```
#include "or1kFunctions.h"

//
// OR1K processor constructor
//
VMI_CONSTRUCTOR_FN(or1kConstructor) {

    or1kP or1k = (or1kP)processor;
    Uns32 i;

    for(i=2; i<OR1K_REGS; i++) {
        or1k->regs[i] = 0xdeadbeef;
    }
}

//
```

or1kMain.c:1

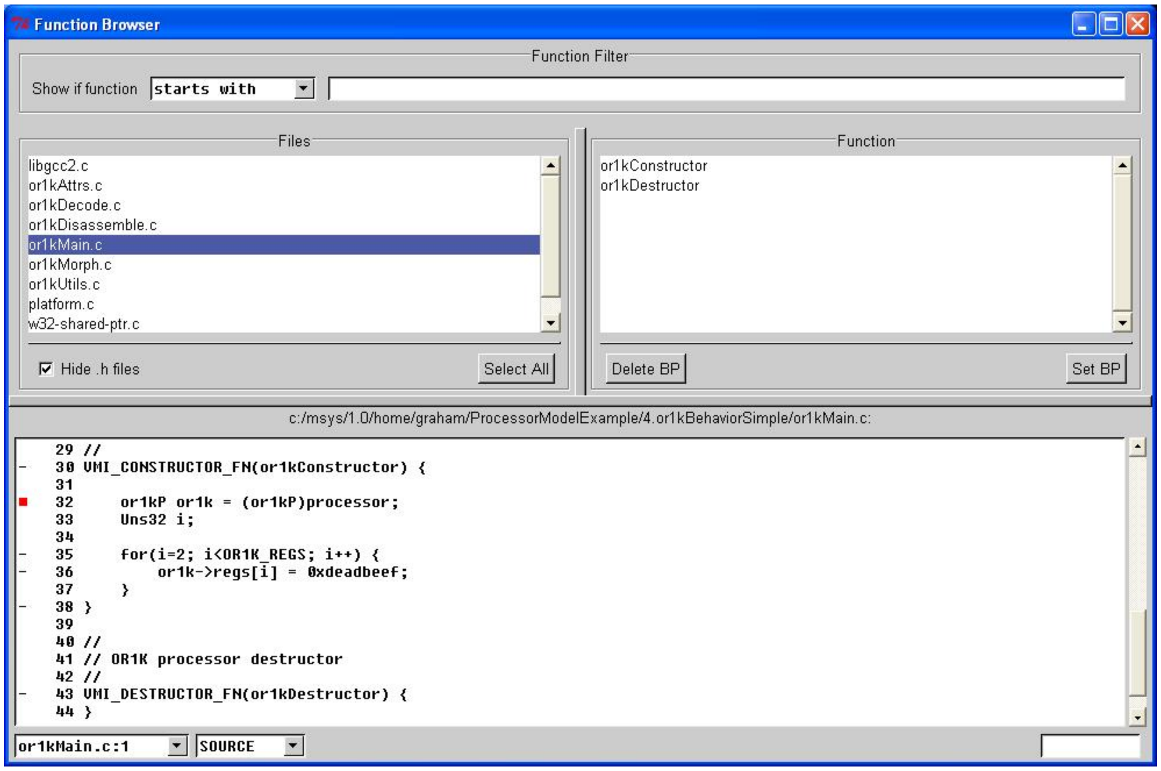
4.5 *An Example Debug Session*

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.



```
libgcc2.c
or1kAttrs.c
or1kDecode.c
or1kDisassemble.c
or1kMain.c
or1kMorph.c
or1kUtils.c
platform.c
w32-shared-ptr.c
```

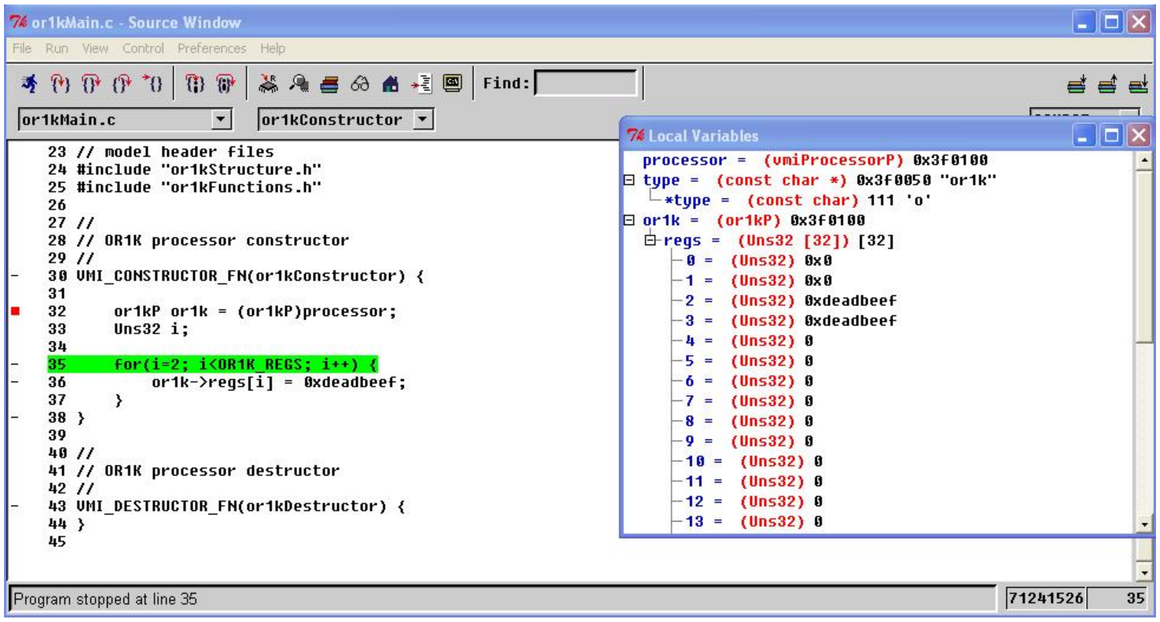
```
or1kConstructor
or1kDestructor
```

c:/msys/1.0/home/graham/ProcessorModelExample/4.or1kBehaviorSimple/or1kMain.c:

```
//
VMI_CONSTRUCTOR_FN(or1kConstructor) {

    or1kP or1k = (or1kP)processor;
    Uns32 i;

    for(i=2; i<OR1K_REGS; i++) {
        or1k->regs[i] = 0xdeadbeef;
    }
}

//
// OR1K processor destructor
//
VMI_DESTRUCTOR_FN(or1kDestructor) {
}
```

or1kMain.c:1

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



```
// model header files
#include "or1kStructure.h"
#include "or1kFunctions.h"

//
// OR1K processor constructor
//
VMI_CONSTRUCTOR_FN(or1kConstructor) {

    or1kP or1k = (or1kP)processor;
    Uns32 i;

    for(i=2; i<OR1K_REGS; i++) {
        or1k->regs[i] = 0xdeadbeef;
    }
}

//
// OR1K processor destructor
//
VMI_DESTRUCTOR_FN(or1kDestructor) {
}
```

```
processor =  (vmiProcessorP) 0x3f0100
⊟ type =  (const char *) 0x3f0050 "or1k"
   *type =  (const char) 111 'o'
⊟ or1k =  (or1kP) 0x3f0100
  ⊟ regs =  (Uns32 [32]) [32]
      0 =  (Uns32) 0x0
      1 =  (Uns32) 0x0
      2 =  (Uns32) 0xdeadbeef
      3 =  (Uns32) 0xdeadbeef
      4 =  (Uns32) 0
      5 =  (Uns32) 0
      6 =  (Uns32) 0
      7 =  (Uns32) 0
      8 =  (Uns32) 0
      9 =  (Uns32) 0
      10 =  (Uns32) 0
      11 =  (Uns32) 0
      12 =  (Uns32) 0
      13 =  (Uns32) 0
```

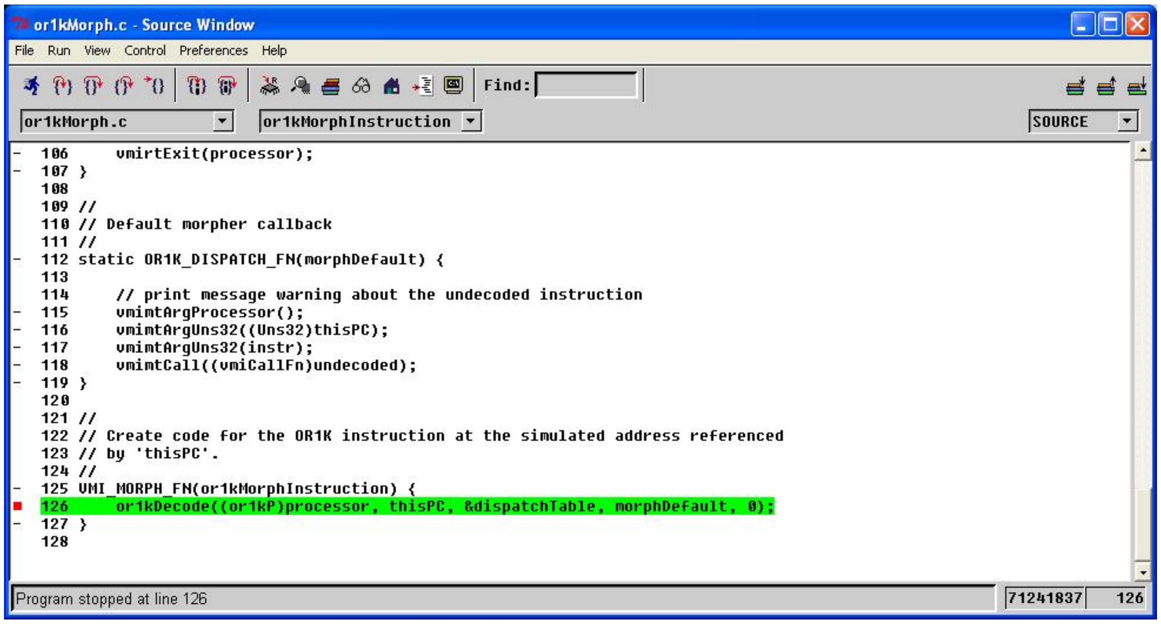
Program stopped at line 35   71241526   35

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 [1]is 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***.

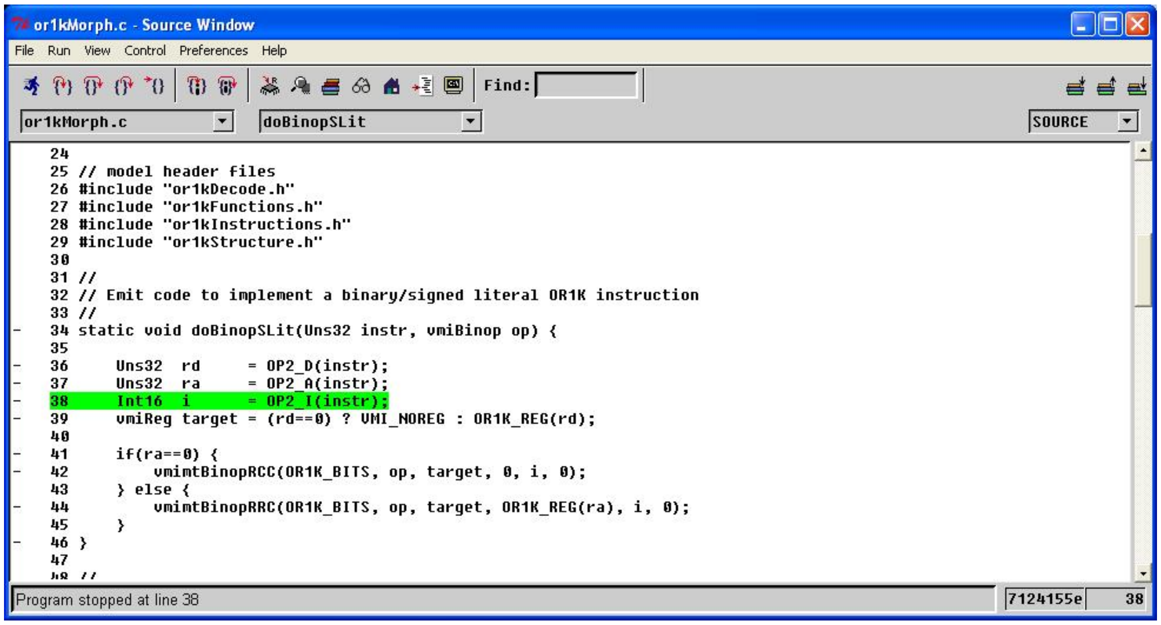

or1kMorph.c - Source Window
File Run View Control Preferences Help
Find:
or1kMorph.c
or1kMorphInstruction
SOURCE

```
106      vmirtExit(processor);
107  }
108
109  //
110  // Default morpher callback
111  //
112  static OR1K_DISPATCH_FN(morphDefault) {
113
114      // print message warning about the undecoded instruction
115      vmimtArgProcessor();
116      vmimtArgUns32((Uns32)thisPC);
117      vmimtArgUns32(instr);
118      vmimtCall((vmiCallFn)undecoded);
119  }
120
121  //
122  // Create code for the OR1K instruction at the simulated address referenced
123  // by 'thisPC'.
124  //
125  VMI_MORPH_FN(or1kMorphInstruction) {
126      or1kDecode((or1kP)processor, thisPC, &dispatchTable, morphDefault, 0);
127  }
128
```

```
Program stopped at line 126                    71241837   126
```

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


or1kMorph.c - Source Window
File Run View Control Preferences Help
Find:
or1kMorph.c
doBinopSLit
SOURCE

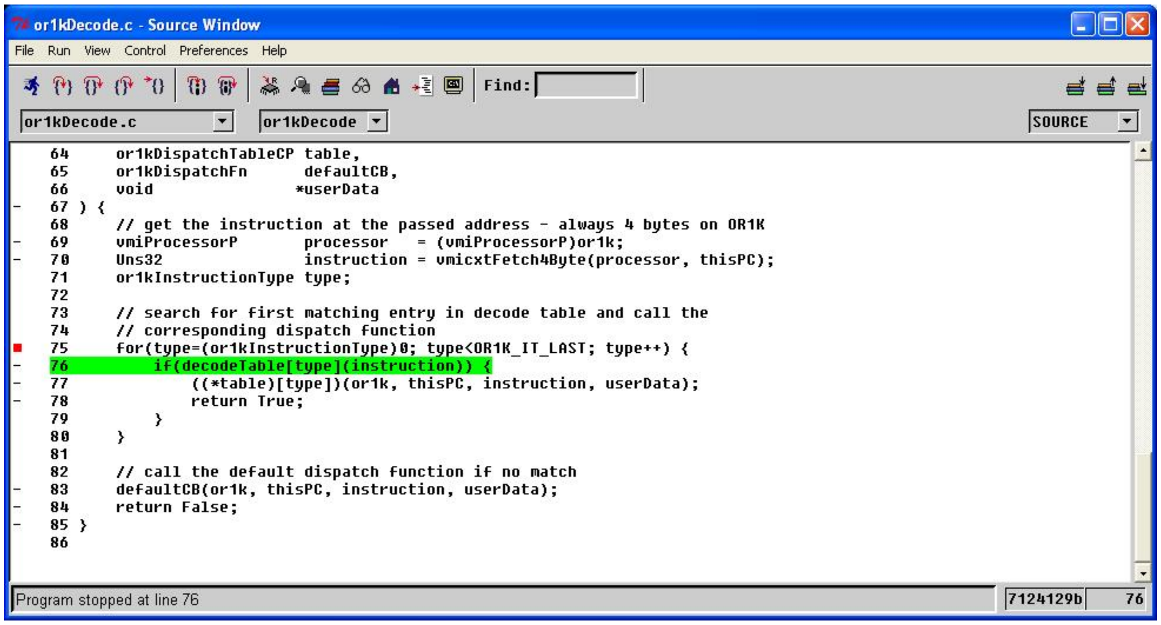
```
24
25 // model header files
26 #include "or1kDecode.h"
27 #include "or1kFunctions.h"
28 #include "or1kInstructions.h"
29 #include "or1kStructure.h"
30
31 //
32 // Emit code to implement a binary/signed literal OR1K instruction
33 //
34 static void doBinopSLit(Uns32 instr, vmiBinop op) {
35
36   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

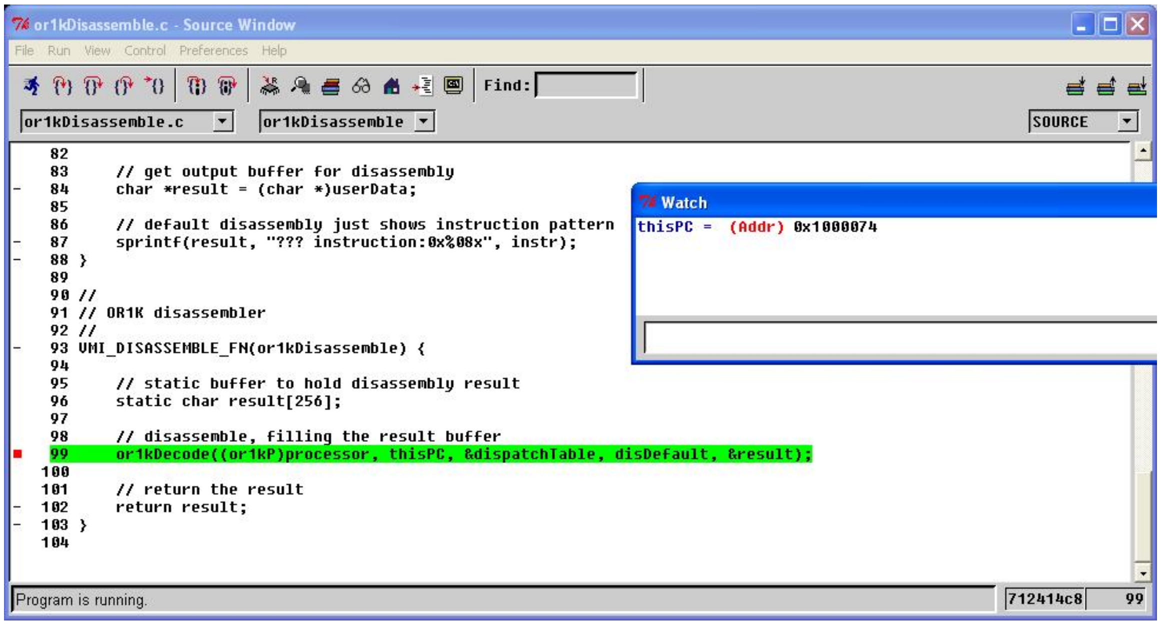

or1kDecode.c - Source Window
File Run View Control Preferences Help
Find:
or1kDecode.c
or1kDecode
SOURCE

```
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.


or1kDisassemble.c - Source Window
File Run View Control Preferences Help
Find:
or1kDisassemble.c
or1kDisassemble
SOURCE
Watch

```
    // get output buffer for disassembly
    char *result = (char *)userData;

    // default disassembly just shows instruction pattern
    sprintf(result, "??? instruction:0x%08x", instr);
}

//
// OR1K disassembler
//
VMI_DISASSEMBLE_FN(or1kDisassemble) {

    // static buffer to hold disassembly result
    static char result[256];

    // disassemble, filling the result buffer
    or1kDecode((or1kP)processor, thisPC, &dispatchTable, disDefault, &result);

    // return the result
    return result;
}
```

```
thisPC =  (Addr) 0x1000074
```

```
Program is running.    712414c8    99
```

###