XAPP1037 (v1.0) February 28, 2008

Introduction to Software Debugging on Xilinx MicroBlaze Embedded Platforms

Author: Brian Hill

Summary

This application note discusses the use of the Xilinx Microprocessor Debugger (XMD**)** and the GNU software debugger (GDB**)** to debug software defects.

Included Systems

Included with this application note is one MicroBlaze™ ML403 reference system:

www.xilinx.com/support/documentation/application_notes/xapp1037.zip

Introduction

This application note offers an introduction to software debugging of Xilinx MicroBlaze embedded processing platforms using XMD and GDB.

XMD is used to download executables to the system, to control the running of these applications with breakpoints, and to examine and modify memory and CPU registers. XMD includes a TCL parser. TCL is a full featured industry standard scripting language. This combination of built-in command and scripting provides powerful debugging possibilities.

GDB is a full featured symbolic software debugger. It makes certain tasks which are cumbersome to accomplish with XMD more streamlined. GDB is used to debug software locally - a local process running on the same machine and operating system as GDB itself, or remotely. In this document, GDB running on the local machine, is used to connect to the **GDB stub** (also called GDB server) running within XMD. XMD automatically starts the GDB server after the connection to the target processor is made.

An implemented system is provided with sample applications which contain intentional software defects. This document will discuss the tools available to identify these defects.

Hardware and Software Requirements

The hardware and software requirements are:

- Xilinx ML403 Development Board
- Xilinx Platform USB Cable or Parallel IV Cable
- RS232 Cable
- Serial Communications Utility Program, such as HyperTerminal
- Xilinx Platform Studio 9.2.02i
- Xilinx Platform Studio SDK 9.2.02i
- Xilinx Integrated Software Environment (ISE™) 9.2.03i

MicroBlaze Reference System Specifics

The included ML403 system was created with Base System Builder. It includes a MicroBlaze processor, XPS UART Lite, XPS Interrupt Controller, XPS Timer, XPS Ethernet Lite controller, MicroBlaze Debug Module, and 8k of LMB BRAM. Please refer to Table 1 for the system address map.

Address Map

Table 1: **Reference System Address Map**

Instance	Peripheral	Base Address	High Address
dlmb_cntlr/ ilmb_cntlr	lmb_bram_if_cntlr	0x00000000	0x00001FFF
Ethernet_MAC	xps_ethernetlite	0x81000000	0x8100FFFF
xps_intc_0	xps_intc	0x81800000	0x8180FFFF
xps_timer_1	xps_timer	0x83C00000	0x83C0FFFF
RS232_Uart	xps_uartlite	0x84000000	0x8400FFFF
debug_module	mdm	0x84400000	0x8440FFFF
DDR_SDRAM	mpmc	0x8C000000	0x8FFFFFFF

Software Applications

This system contains applications that use the Xilinx standalone software library. These applications have intentional software defects which will be investigated using the procedures outlined in this application note. The source for these applications is provided within the `SDK_projects` directory.

The TestApp_Crash application can be made to fail, or *crash,* by intentionally causing the processor to execute invalid instructions. How to find the cause of the application failures, or *crashes*, will be discussed using this application as a model.

TestApp_StackOverflow is used to examine software failures (crashes), this time providing a more realistic example of likely software errors. Gathering crash debug information from deployed systems is discussed.

TestApp_malloc is used to demonstrate how memory leaks may be identified.

The TestApp_emaclite application is a simple application which uses the Light Weight IP (lwIP) library. In its present form, the application cannot be pinged successfully. This will be investigated.

Compiler Options

When debugging an application, it is best to compile images with symbolic debugging information included (**-g**), and with no optimization (**-O0**). Images which have been stripped (contain no symbols) will be difficult to use in a meaningful way with the debugger. Highly optimized images (**-O2**) will be more confusing to debug because operations may not occur in the sequence they appear in the source text or they may not occur at all. For that reason, all applications provided with this application note have been configured to compile with **-g -O0** compiler options.

Command Conventions

This application note provides instructions for entering various commands. All commands are displayed as **`bold`**. The prompt displayed with the command indicates the environment for which the command is intended as shown in Table 2.

Table 2: **Command Prompts**

Prompt	Environment
$	EDK Shell
(gdb)	GDB
XMD%	XMD

Executing the Reference System

Executing the Reference System using the Pre-Built Bitstream and the Compiled Software Applications

Execute the system using files inside the `ready_for_download/` directory in the project root directory, by following the steps below.

1. Change directories to the `ready_for_download` directory.
2. Use iMPACT to download the bitstream by using the command,

   ```
   $ impact -batch xapp1037.cmd
   ```

3. Invoke XMD and connect to the processor by using the command,

   ```
   $ xmd -opt xapp1037.opt
   ```

4. Download the executables by using the command,

   ```
   XMD% dow <executable name>.elf
   ```

Executing the Reference System from XPS

Execute the system using XPS, by following the steps below.

1. Open `system.xmp` inside XPS.
2. Select **Hardware** → **Generate Bitstream** to generate a bitstream for the system.
3. Select **Device Configuration** → **Download Bitstream** to download the bitstream to the board.
4. Select **Debug** → **Launch XMD...** to launch XMD.
5. Download the executables by using the XMD command,

   ```
   XMD% dow <executable name>.elf
   ```

Console connection

Connect a serial cable to the RS232 port on the ML403. The terminal application, such as HyperTerminal, is configured with the settings: Bits per second: **960,** Data bit:, **8**; Parity: **None**; Stop Bits: **1**; and Flow Control: **None**, as shown in Figure 1.

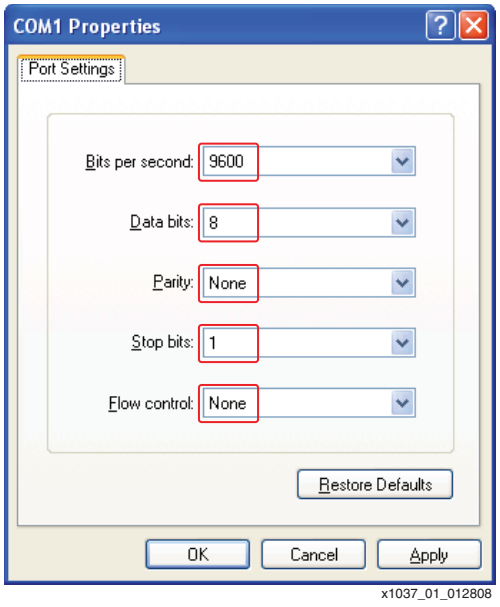

Figure 1: **ML555 PLBv46 PCI Reference System Block Diagram**

MicroBlaze Synthesis Parameters

The MicroBlaze processor is a soft core microprocessor with many IP options which can be customized. The defaults for these parameters are intended for the smallest possible FPGA resource usage. This environment is not suitable for software debugging.

By default, **there are no software exceptions** — meaning that when software errors occur, the MicroBlaze processor will ignore them. However, the software exceptions may be changed by enabling exceptions as in Figure 2.

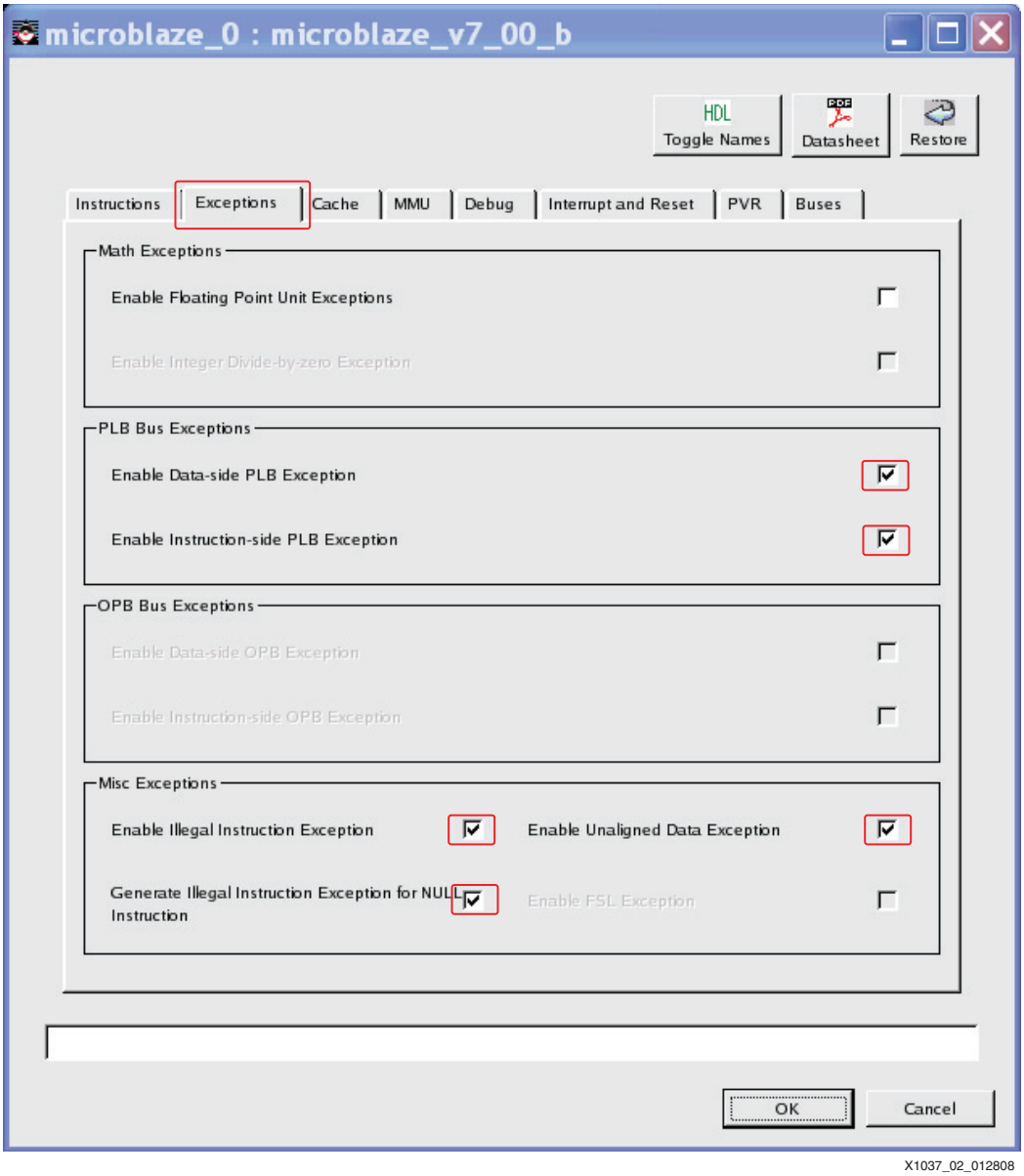

Figure 2: **MicroBlaze Exceptions IP Configuration**

The MicroBlaze processor provides information about how it was synthesized in the Processor Version Registers (**PVR**). These registers include information about which exceptions the processor can generate when errors occur. By default **none of the PVR registers exist**. This is changed by setting the Processor Version Register to FULL as shown in Figure 3.

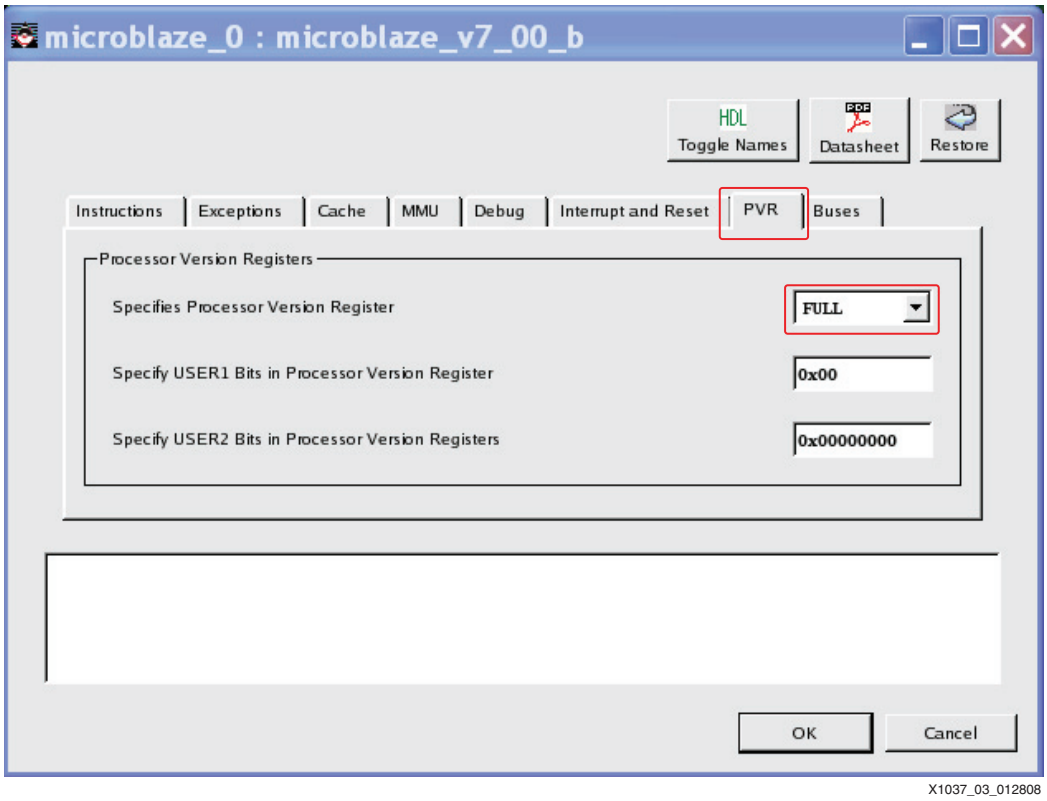

Figure 3: **MicroBlaze PVR IP Configuration**

XMD and TCL Scripting

Please note that this section of the software debugging document is meant to introduce some of the capabilities of TCL scripting within XMD. The TCL language is beyond the scope of this document.

XMD includes a TCL parser. TCL is a full featured industry standard scripting language. This combination provides powerful debugging possibilities. All the functionality of XMD (read/write registers, memory, and memory mapped devices, breakpoints, etc...) is available to user-supplied scripts which can enhance the base functionality of XMD. Any valid TCL command can be entered interactively at the XMD prompt:

```
XMD% expr 8 + 1
9
XMD% puts "hello world"
hello world
```

By writing TCL procedures, it is possible to extend XMD.

```
XMD% proc myprocedure {mynumber1 mynumber2} {
> set retval [expr $mynumber1 + $mynumber2]
> return $retval
> }
XMD%
XMD% myprocedure 8 1
9
```

The script files can be loaded into XMD (rather than typing them in, as above) with the **source** command:

```
XMD% source myscriptfile.tcl
XMD% myprocedure 8 1
9
```

The real power becomes evident when scripting is combined with XMDs abilities to access CPU registers and memory. XMD can access CPU registers interactively, as shown below:

```
XMD% rrd msr
   msr: 00000400
```

This reads the MicroBlaze **MSR** register. The small script shown below is an example of how to use this ability to access registers or memory to display information:

```
# Read Microblaze MSR Register and examine the EE bit.
# Print in plain English if Exceptions are presently
# enabled.
proc microblaze_exceptionenable_print {} {
    # Read the MSR register.  Trim off extra text, keeping only the number.
    # "   msr:  12345678 " becomes "12345678"
    set regval [string trimleft [rrd msr] "msr: "]

    # make the number read above appear like conventional hexadecimal
    # "12345678" becomes "0x12345678"
    set regval [format "0x%08x" 0x0$regval ]

    puts -nonewline "Microblaze Exceptions:"
    # Test the 'EE' bit
    if {$regval & 0x00000100} {
        puts "ENABLED"
    } else {
        puts "DISABLED"
    }
}
```

This example script is provided in the `xmd_tcl_scripts` directory as `mb_exenable_print.tcl`. If presently in the `ready_for_download` directory, it would be loaded like this:

```
XMD% source ../../xmd_tcl_scripts/mb_exenable_print.tcl
```

When the procedure is executed human-readable state information is displayed:

```
XMD% microblaze_exceptionenable_print
Microblaze Exceptions: DISABLED
XMD%
```

When XMD starts, it will automatically execute any TCL commands in a file called `.xmdrc` (if it exists). This file should be placed in the users' home directory. Commands can be placed here to source all of your debugging scripts when XMD is started.

This application note includes several TCL scripts found in the `xmd_tcl_scripts` directory for use with XMD as debugging aids for Xilinx embedded systems. These scripts display CPU and peripheral register values, decoding many register fields. To utilize these scripts, copy ***dotxmdrc*** from the `xmd_tcl_scripts` directory to **`$HOME/.xmdrc`**.

An example, as entered from the EDK Shell within the xmd_tcl_scripts directory is shown:

```
$ cp dotxmdrc $HOME/.xmdrc
```

The users `.xmdrc` file should be edited to reflect the directory where these TCL scripts have been placed.

Now, when XMD is started these scripts will alert the user to new commands available:

```
$ xmd
...
Loading custom commands:
ppc405_print
```

```
mb_print
emaclite_print
lltemac_print
lltemac_read_phy
lldma_mm_print
uartlite_print
uartns550_print
xps_intc_print
XMD%
```

TestApp_Crash

From the ready_for_download area, use XMD to download and run TestApp_Crash (assumes bitstream is already downloaded):

```
$ xmd -opt xapp1037.opt
XMD% dow TestApp_Crash.elf
XMD% run
```

If working properly, (in the present form it is not), the expected output would be:

```
-- Entering main() --
-- Exiting main() --
```

Identifying the Problem

When run in it's present form, only the first line is printed correctly. The application has "crashed", resulting in only half of the expected output to appear. The cause of this error in such a tiny application as TestApp_Crash is easy to find by examining the source code, but if this were a much larger application the task would be much more difficult. If the application were a Unix process, the user would expect the operating system to indicate that the process had been terminated, and some indication of why. The Xilinx standalone library is a very minimal environment. Unless the application explicitly sets up exception handling for software errors there will be no indication of what has happened.

A "crash" can be one of several events: The processor executed an invalid instruction, an attempt to execute privileged code from user mode, or any other access violation (violating memory protections set in TLB entries). When any such event occurs, the processor will generate an exception, often referred to as an interrupt. Exceptions cause the processor to execute code at the appropriate exception vector for the type of exception encountered. Software errors generate a Hardware Exception on the MicroBlaze, which is vector 0x20.

This means that the processor will execute instructions at offset 0x20 after the start of the exception vector table, 0x00000000.

To determine the cause of this crash, set a hardware breakpoint at the Hardware Exception vector, 0x20. Exceptions must be enabled. This is controlled by bit 23 in the MSR register. In the below example, it is observed that the processor will execute instructions at 0x20 when a Hardware Exception occurs.

From the ready_for_download area, use XMD to download and run TestApp_Crash (assumes bitstream is already downloaded):

```
$ xmd -opt xapp1037.opt
XMD% dow TestApp_Crash.elf
```

Enable exceptions:

```
XMD% rwr msr 0x100
```

Set a hardware breakpoint at the Hardware Exception vector:

```
XMD% bps 0x20 hw
Setting breakpoint at 0x00000020
XMD% run
Info:Processor started. Type "stop" to stop processor
```

The mb.tcl script, provided in the xmd_tcl_scripts directory, can provide some information about the cause of the crash.

Execute the "mb_print" procedure after the application hits the breakpoint as shown below:

```
XMD% mb_print
MSR:   0x00000600 EIP PVR
ESR:   0x00000002 EXCEPTION-ILLEGAL_OPCODE
PVR0:  0x94000700 PVR_FULL MUL EXC VER_7.00.b
PVR2:  0x578311f0 DLMB ILMB IRQPOS DPLB IPLB INTERCON MSR PCMP MUL IPLBEXC
DPLBEXC OP0EXC UNEXC OPEXC
PVR4:  0x034d0000 NO_ICU ICW
PVR5:  0x034d0000 NO_DCU DCW
PVR10: 0x07000000 ARCH_Virtex4
PVR11: 0x0ae00000 MMU_NONE TLBACC_MINIMAL
```

This script prints the MSR, ESR, and all available PVR MicroBlaze registers, decoding many of the bits. This is applicable to TestApp_Crash because the Exception Syndrome Register (ESR) contains the specific reason an exception has occurred (there are many possibilities). In this case, an Illegal Opcode is indicated, as expected.

This script also decodes PVR2, which indicates which Hardware Exceptions the MicroBlaze was configured to generate when it was synthesized.

Explanation of the Problem

TestApp_Crash executes an invalid MicroBlaze instruction. It accomplishes this by filling an array with arbitrary data, and then executing this arbitrary data as code.

Excerpt from SDK_projects/TestApp_Crash/src/TestApp_Crash.c:

```
unsigned crash_instructions[10];

/*
 * crash_function:
 * This function will generate an Illegal Instruction Hardware Exception
 * (Microblaze vector 0x20) by filling the array crash_instructions[] with
 * arbitrary data, then branching to this array address causing the
 * processor to attempt to execute this data as code.
 */
void
crash_function (void) {
   void (*crash_instructions_p) (void);

   crash_instructions[0] = 0;
   crash_instructions[1] = 1;
   crash_instructions[2] = 2;
   crash_instructions[3] = 3;

   crash_instructions_p = (void*)crash_instructions;
   (*crash_instructions_p)();
}
```

When a Hardware Exception occurs, the processor sets the effective address of the instruction that caused the exception in R17:

```
XMD% rrd r17
   r17: 8c000e0c
```

This is the address of the instruction (0x8C000E0C) which, when executed, generated the exception. To display the addresses of all symbols in the executable:

```
$ mb-nm --numeric-sort TestApp_Crash.elf
```

It can now be seen that the illegal instruction is at an address within the array crash_instructions, where crash_function() branched to.

```
...
8c000dec b object.2269
8c000e04 b XAssertCallbackRoutine
8c000e08 B crash_instructions
8c000e30 B XAssertStatus
8c000e34 B __bss_end
...
```

Other useful information will be found in the contents of the Link Register. This contains the return address for the last function call (the last branch-with-link type instruction). The MicroBlaze Application Binary Interface (ABI) uses register R15 as the Link Register by convention.

```
XMD% rrd r15
     r15: 8c0001e4
```

The symbol listing shows that the last function call was made from somewhere within crash_function() -- the return address of 0x8C0001E4 is greater than the start of crash_function() at 0x8C000198 and less than the end crash_function() at 0x8C000204:

```
...
8c0000fc T _crtinit
8c000198 T crash_function
8c000204 T main
8c000258 T print
...
```

It could be helpful to see what function was called from within crash_function(). To do this, disassemble the executable and examine the listing:

```
$ mb-objdump -S TestApp_Crash.elf > TestApp_Crash.dis
```

Note: For this command to include intermingled C source, as shown here, it must be run on the executable in the directory where it was compiled, SDK_projects/TestApp_Crash/Debug/. A previously generated file has been provided in ready_for_download/TestApp_Crash.dis.

```
void
crash_function (void) {
8c000198:       3021ffdc        addik   r1, r1, -36
8c00019c:       f9e10000        swi     r15, r1, 0
8c0001a0:       fa610020        swi     r19, r1, 32
8c0001a4:       12610000        addk    r19, r1, r0
   void (*crash_instructions_p) (void);

   crash_instructions[0] = 0;
8c0001a8:       b0008c00        imm     -29696
8c0001ac:       f8000e08        swi     r0, r0, 3592
   crash_instructions[1] = 1;
8c0001b0:       30600001        addik   r3, r0, 1
8c0001b4:       b0008c00        imm     -29696
8c0001b8:       f8600e0c        swi     r3, r0, 3596
   crash_instructions[2] = 2;
8c0001bc:       30600002        addik   r3, r0, 2
8c0001c0:       b0008c00        imm     -29696
8c0001c4:       f8600e10        swi     r3, r0, 3600
   crash_instructions[3] = 3;
```

```
8c0001c8:       30600003        addik   r3, r0, 3
8c0001cc:       b0008c00        imm     -29696
8c0001d0:       f8600e14        swi     r3, r0, 3604

   crash_instructions_p = (void*)crash_instructions;
8c0001d4:       b0008c00        imm     -29696
8c0001d8:       30600e08        addik   r3, r0, 3592
8c0001dc:       f873001c        swi     r3, r19, 28
   (*crash_instructions_p)();
8c0001e0:       e873001c        lwi     r3, r19, 28
8c0001e4:       99fc1800        brald   r15, r3
...
```

The last function call was a function pointer crash_instructions_p in crash_function(). The code pointed to by the crash_instructions_p was not code at all, but invalid data. The instruction of the return address (the contents of the Link Register, R15) `0x8C0001E4` is "brald <link register> <register with absolute address to branch to>".

How to Solve the Problem

TestApp_Crash is not a realistic example of any common programming error. No application design would ever specifically include code to execute arbitrary data. This application does provide a useful framework to locate software errors with the debugger, and an introduction to other useful software tools. This framework is a foundation for the other applications in this application note. The “problem” in TestApp_Crash is easily solved by simply removing the code inside crash_function() which branched to the array crash_instructions.

Introduction to GDB

Identifying the Problem

Using the TestApp_Crash application again, the same problem debugged previously with XMD is re-examined, this time using GDB. As previously observed, this application will “crash” because it executes an illegal instruction. The same information previously gathered with XMD is collected again with GDB, with some of the features present only in GDB introduced.

Explanation of the Problem

Before beginning with GDB the bitstream should already have been downloaded.

Start (or re-start if already running) XMD from the `ready_for_download` area:

```
$ xmd -opt xapp1037.opt
```

Start GDB from an EDK shell in the `ready_for_download` directory. GDB is used in textual mode as indicated by the -nw switch.

Note: Most of the advanced features of GDB are only available through the GDB command prompt. The GDB GUI provides no graphical access to these features. For this reason, GDB is used entirely in textual mode throughout this application note.

```
$ mb-gdb -nw TestApp_Crash.elf
```

Next, have GDB connect to the "target" - in this case the GDB server within XMD. Since this is a network connection, GDB and XMD can be running on different machines:

```
(gdb) target remote (ip address of machine running XMD):1234
Remote debugging using (remote machine ip address):1234
0x00000000 in _start ()
```

If GDB and XMD are run on the same machine, “localhost” may be used to specify the machine GDB should connect to:

```
(gdb) target remote localhost:1234
```

Now, tell GDB to download the application into memory:

```
(gdb) load
```

As was done with XMD, a hardware breakpoint is set at the Hardware Exception vector:

Note: Exception handling has not been initialized by software. Due to this, some output in the remainder of this section may vary from one instance to another, and is unlikely to exactly match this text.

```
(gdb) hbreak *0x20
Hardware assisted breakpoint 1 at 0x20
```

Note: By default, MicroBlaze only supports one hardware breakpoint. If XMD has not been restarted since the previous exercise, or the hardware breakpoint set in that exercise has not been removed, GDB will fail to utilize the hardware breakpoint set above, and will instead produce an error when the program is run.

Set the EE bit in the MSR register to enable Exceptions:

```
(gdb) set $rmsr = 0x100
```

Note: GDB can have difficulty writing to CPU registers. This operation is most likely to be successful if the processor has already run an application prior to starting this GDB debugging session. If this bug is encountered, run any of the provided applications to completion and then attempt this section.

Now, start the application and watch it crash:

```
(gdb) continue
Continuing.
Breakpoint 1, 0x20 in _vector_hw_exception ()
```

Execution has stopped at the Hardware Exception vector. GDB can display the nesting of function calls (the callstack) which have occurred up to the time of the exception. Have GDB display the callstack with a backtrace command (bt):

```
(gdb) bt
#0  0x00000020 in _vector_hw_exception ()
#1  0x8c0001ec in crash_function () at ../src/TestApp_Crash.c:69
#2  0x8c00022c in main () at ../src/TestApp_Crash.c:78
```

It is seen that from somewhere in crash_function() or a function called within crash_function() has caused the application to crash. The numbers (#0, #1) indicate stack frames, one for each function in the callstack. The stack and stack frames are discussed in detail in the next section of this application note. For now, it is not necessary to understand how GDB accomplishes this.

Examine stack frame 1 (examine what happened in the function crash_function()):

```
(gdb) frame 1
#1  0x8c0001ec in crash_function () at ../src/TestApp_Crash.c:69
69 (*crash_instructions_p)();
```

This indicates that the last thing which happened within the function crash_function() was a call to the address stored in function pointer crash_instructions_p(). GDB can be told to display the pertinent lines of source code from within crash_function() with the **list** command:

```
(gdb) list
64          crash_instructions[1] = 1;
65          crash_instructions[2] = 2;
66          crash_instructions[3] = 3;
67
68          crash_instructions_p = (void*)crash_instructions;
69          (*crash_instructions_p)();
70      }
71
72      int
73      main (void) {
```

And, in TestApp_Crash.c at line 69 the offending call to invalid instructions is seen.

As was done with XMD the exception address the CPU placed in R17 is examined:

```
(gdb) info registers r17
r17            0x8c000e0c       -1946153460
```

GDB will look up an address in the symbol table:

```
(gdb) info symbol 0x8c000e0c
crash_instructions + 4 in section .bss
```

The backtrace provided by GDB is very useful, but GDB can examine the Link Register directly as was done with XMD:

```
(gdb) info register r15
r15            0x8c0001e4       -1946156572
(gdb) info symbol 0x8c0001e4
crash_function + 76 in section .text
```

GDB will also disassemble instructions in memory, so it can be seen exactly what is at the address stored in the Link Register:

```
(gdb) disassemble crash_function
Dump of assembler code for function crash_function:
0x8c000198 <crash_function+0>:  addik   r1, r1, -36
0x8c00019c <crash_function+4>:  swi     r15, r1, 0
0x8c0001a0 <crash_function+8>:  swi     r19, r1, 32
0x8c0001a4 <crash_function+12>: addk    r19, r1, r0
0x8c0001a8 <crash_function+16>: imm     -29696
0x8c0001ac <crash_function+20>: swi     r0, r0, 3592
0x8c0001b0 <crash_function+24>: addik   r3, r0, 1       // 0x1 <_start+1>
0x8c0001b4 <crash_function+28>: imm     -29696
0x8c0001b8 <crash_function+32>: swi     r3, r0, 3596
0x8c0001bc <crash_function+36>: addik   r3, r0, 2       // 0x2 <_start+2>
0x8c0001c0 <crash_function+40>: imm     -29696
0x8c0001c4 <crash_function+44>: swi     r3, r0, 3600
0x8c0001c8 <crash_function+48>: addik   r3, r0, 3       // 0x3 <_start+3>
0x8c0001cc <crash_function+52>: imm     -29696
0x8c0001d0 <crash_function+56>: swi     r3, r0, 3604
0x8c0001d4 <crash_function+60>: imm     -29696
0x8c0001d8 <crash_function+64>: addik   r3, r0, 3592
0x8c0001dc <crash_function+68>: swi     r3, r19, 28
0x8c0001e0 <crash_function+72>: lwi     r3, r19, 28
0x8c0001e4 <crash_function+76>: brald   r15, r3
0x8c0001e8 <crash_function+80>: or      r0, r0, r0
```

And again, it is observed that the instruction at `0x8C0001E4` (crash_function + 76) is a call to the invalid instructions.

How to Solve the Problem

TestApp_Crash has once again served as a useful model to demonstrate how to find software errors, this time using GDB. As before, the offending code in crash_function has been identified. To prevent TestApp_Crash from causing an exception, remove the use of the function pointer to the array crash_instructions.

Debugging Stack Errors

CPU registers are finite in number. The data which one individual function must work with can easily exceed this set. This limitation is resolved with a construct known as the stack. The stack is an area of memory used to hold temporary data - variables local to a function and saved register values. MicroBlaze, unlike some other CPU architectures (such as Intel) does not architecturally require the use of a stack, nor is explicit hardware support provided for a stack. There is no hardware-defined stack pointer, and when function calls are made or interrupts occur, MicroBlaze does not automatically store any data on the stack (as occurs with Intel x86

and its successors). The MicroBlaze Application Binary Interface specifies that R1 is used as the stack pointer by convention**.** The stack grows with each function call in a downward direction -- from higher memory addresses towards lower memory addresses. Each function has its own **stack frame** pointed to by the stack pointer in that function. A function, when called, will create a new stack frame for itself by decrementing the stack pointer (r1) by the appropriate amount to create a scratchpad for itself. Before returning, it will restore the stack pointer to that of the calling function. Stack usage will be discussed in additional detail later in this section.

Identifying the Problem

TestApp_Crash has been a useful introduction to the debugger, but it does not represent any likely errors. It does not demonstrate how a real-world application might suddenly execute unexpected code. One common reason is stack corruption. When function calls are made, register values from the calling function are saved on the stack so that they can be restored when the called function returns. Among items saved are the contents of the Link Register (R15). In the C programming language, local variables are also placed on the stack. This provides the opportunity for bugs, which shall be examined.

Explanation of the Problem

Lets examine the application TestApp_StackOverflow. The skeleton of the application is something like this:

Note: this is ***not*** the actual program text!

```
void
myfunction3 (void) {
    unsigned mylocalarray[THE_SIZE];

    printf("-- Entering myfunction3() --\r\n");
    mylocalarray[THE_INDEX] = (something);
    printf("-- Exiting myfunction3() --\r\n");
}

void
myfunction2 (void) {
    printf("-- Entering myfunction2() --\r\n");
    myfunction3();
    printf("-- Exiting myfunction2() --\r\n");
}

void
myfunction1 (void) {
    printf("-- Entering myfunction1() --\r\n");
    myfunction2();
    printf("-- Exiting myfunction1() --\r\n");
}

int main (void) {
   print("-- Entering main() --\r\n");
   myfunction1();
   print("-- Exiting main() --\r\n");
   return 0;
}
```

At first glance, the expected output would be:

```
-- Entering main() --
-- Entering myfunction1() --
-- Entering myfunction2() --
-- Entering myfunction3() --
```

```
-- Exiting myfunction3() --
-- Exiting myfunction2() --
-- Exiting myfunction1() --
-- Exiting main() --
```

But this is not the behavior observed, it is instead:

```
-- Entering main() --
-- Entering myfunction1() --
-- Entering myfunction2() --
-- Entering myfunction3() --
-- Exiting myfunction3() --
-- Exiting myfunction2()
-- Entering crash_function() --
```

Nowhere is crash_function() explicitly called or branched to. To investigate what has happened, the program is run with the debugger:

```
$ mb-gdb -nw TestApp_StackOverflow.elf
```

Connect to the XMD GDB server:

```
(gdb) target remote <ip address of machine running XMD>:1234
Remote debugging using <ip address>:1234
0x00000000 in _start ()
```

Have GDB download the executable to memory

```
(gdb) load
```

Set a hardware breakpoint at the Hardware Exception vector:

```
(gdb) hbreak *0x20
Hardware assisted breakpoint 1 at 0x20
```

Enable MicroBlaze Exceptions

```
(gdb) set $rmsr = 0x100
```

Continue program execution from the present PC

```
(gdb) c
Continuing.
Breakpoint 1, 0x00000020 in _vector_hw_exception ()
```

Note: The backtrace produced below may not appear exactly like that seen by the user. When an application “crashes”, subsequent behavior varies based upon uninitialized data (memory, CPU registers). The same application, run on the same hardware, will not necessarily always produce the same results. This is one of many important debugging lessons.

Display the callstack backtrace:

```
(gdb) bt
#0  0x00000020 in _vector_hw_exception ()
#1  0x8c0001f4 in crash_function () at ../src/TestApp_StackOverflow.c:78
#2  0x8c000348 in main () at ../src/TestApp_StackOverflow.c:140
```

From looking at the source, it is seen that main() calls myfunction1() which called myfunction2() which calls myfunction3(). There is no call in main() to crash_function() as the above backtrace seems to indicate. Plus, the program output indicates that myfunction1() myfunction2() and myfunction3() have actually been executed.

Examine the main() stack frame more closely. The frame number is displayed in the above backtrace output.

```
(gdb) info frame 2
Stack frame at 0x8c009034:
```

```
 rpc = 0x8c000348 in main (../src/TestApp_StackOverflow.c:140); saved rpc
0x8c000348
 caller of frame at 0x8c009038
 source language c.
 Arglist at 0x8c009034, args:
 Locals at 0x8c009034, Previous frame's sp is 0x8c009034
 Saved registers:
  r15 at 0x8c009014, r19 at 0x8c009030, rpc at 0x8c009014
```

This shows that main() left it's frame at pc `0x8C000348`. Disassemble the function to see what instruction is at `0x8C000348`:

```
(gdb) disassemble main
Dump of assembler code for function main:
0x8c000320 <main+0>:    addik   r1, r1, -32
0x8c000324 <main+4>:    swi     r15, r1, 0
0x8c000328 <main+8>:    swi     r19, r1, 28
0x8c00032c <main+12>:   addk    r19, r1, r0
0x8c000330 <main+16>:   imm     -29696
0x8c000334 <main+20>:   addik   r5, r0, 22360
0x8c000338 <main+24>:   brlid   r15, 1732       // 0x8c0009fc <print>
0x8c00033c <main+28>:   or      r0, r0, r0
0x8c000340 <main+32>:   brlid   r15, -112       // 0x8c0002d0 <myfunction1>
0x8c000344 <main+36>:   or      r0, r0, r0
0x8c000348 <main+40>:   imm     -29696
0x8c00034c <main+44>:   addik   r5, r0, 22384
0x8c000350 <main+48>:   brlid   r15, 1708       // 0x8c0009fc <print>
0x8c000354 <main+52>:   or      r0, r0, r0
0x8c000358 <main+56>:   addk    r3, r0, r0
0x8c00035c <main+60>:   lwi     r15, r1, 0
0x8c000360 <main+64>:   addk    r1, r19, r0
0x8c000364 <main+68>:   lwi     r19, r1, 28
0x8c000368 <main+72>:   addik   r1, r1, 32
0x8c00036c <main+76>:   rtsd    r15, 8
0x8c000370 <main+80>:   or      r0, r0, r0
```

So, the last thing main() did was call myfunction1() (which is clearly not a call to crash_function()).

This calls for some logical thinking. Clearly, myfunction2() calls myfunction3(). The first thing any function will do is set up a stack frame. It saves the link register (R15) it will use to return to the caller. This is at a known, fixed offset (stackpointer + 0 bytes). The link register is saved because function calls may occur in myfunction3 (and, in fact, do occur), which would overwrite the current link register contents. If the value of R15 saved on the stack were to be modified before the function returned, the function would then return to the incorrect address. Examine if this is what is happening with TestApp_StackOverflow.

A model of the stack in this application is shown in Figure 4.

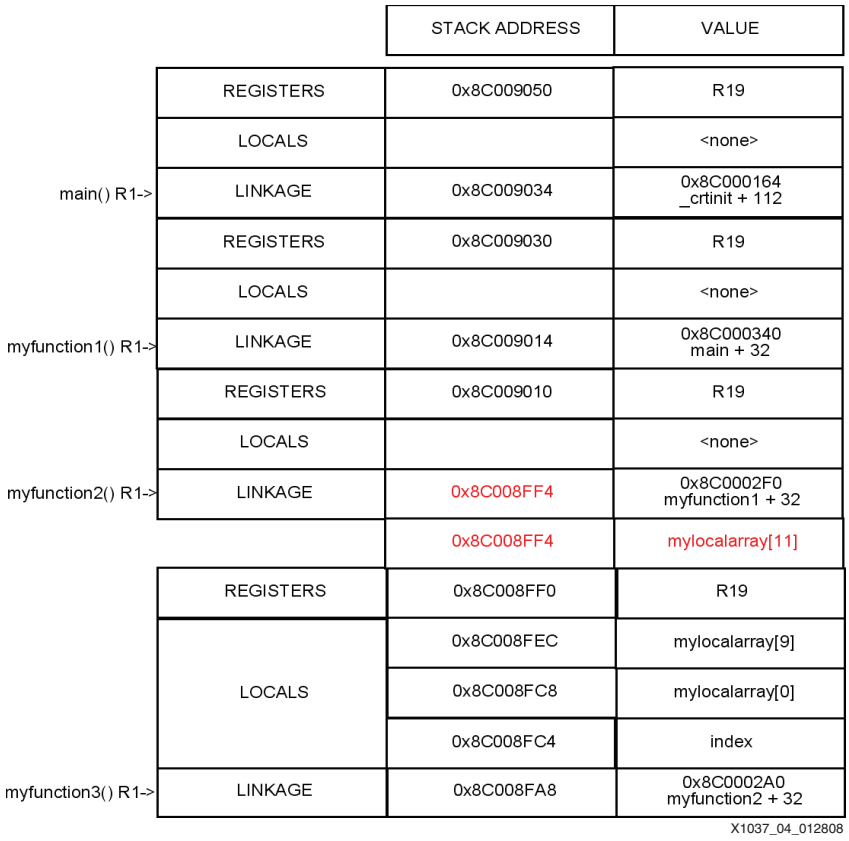

Figure 4: **TestApp_StackOverflow Stack**

The disassembly of myfunction3() is examined below to get a better understanding of the stack manipulation that occurs in this function. Additional commentary has been added to the listing to explain what is occurring.

```
$ mb-objdump -S TestApp_StackOverflow.elf
void
myfunction3 (void) {
# Assume that at call time, R1(the stack pointer) = 0x8C008FF4
# sp = sp -76 bytes. sp = 0x8C008FA8
8c00020c:       3021ffb4        addik   r1, r1, -76
# store linkage (the return address from this function) at sp + 0
8c000210:       f9e10000        swi     r15, r1, 0
# store R19 at sp + 72
8c000214:       fa610048        swi     r19, r1, 72
8c000218:       12610000        addk    r19, r1, r0
    unsigned mylocalarray[10];
    int index;

#define SAVEDR19     0
#define SAVEDR15     1

    printf("-- Entering myfunction3() --\r\n");
8c00021c:       b0008c00        imm     -29696
8c000220:       30a05698        addik   r5, r0, 22168
8c000224:       b9f41618        brlid   r15, 5656       // 8c00183c <puts>
8c000228:       80000000        or      r0, r0, r0
# index = 10 + 1 = 11
# index is beyond the end of the array. Using this index will corrupt
```

```
# the stack.
    index = (sizeof(mylocalarray)/sizeof(unsigned)) + SAVEDR15;
8c00022c:       3060000b        addik   r3, r0, 11
8c000230:       f873001c        swi     r3, r19, 28
    /* the value of index is out of bounds - this will write past the end
     * of the array.
     */
# the address of mylocalarray[11] = 0x8C008FF4, or sp + 76. This is in the
# callers' stack frame (myfunction2), where it has stored its return
# address to myfunction1. Now, when myfunction2 returns, it will not return
# to myfunction1 but to crash_function().
    mylocalarray[index] = (unsigned) crash_function;
8c000234:       e893001c        lwi     r4, r19, 28
8c000238:       b0008c00        imm     -29696
8c00023c:       30600190        addik   r3, r0, 400
8c000240:       10a30000        addk    r5, r3, r0
8c000244:       60840004        muli    r4, r4, 4
8c000248:       3073001c        addik   r3, r19, 28
8c00024c:       10641800        addk    r3, r4, r3
8c000250:       30630004        addik   r3, r3, 4
8c000254:       f8a30000        swi     r5, r3, 0

    printf("-- Exiting myfunction3() --\r\n");
8c000258:       b0008c00        imm     -29696
8c00025c:       30a056b8        addik   r5, r0, 22200
8c000260:       b9f415dc        brlid   r15, 5596       // 8c00183c <puts>
8c000264:       80000000        or      r0, r0, r0
}
8c000268:       e9e10000        lwi     r15, r1, 0
8c00026c:       10330000        addk    r1, r19, r0
8c000270:       ea610048        lwi     r19, r1, 72
8c000274:       3021004c        addik   r1, r1, 76
8c000278:       b60f0008        rtsd    r15, 8
8c00027c:       80000000        or      r0, r0, r0
```

There is a local variable called "mylocalarray". Because it is a local variable, this is instantiated on the stack. This array is only written to once:

mylocalarray[index] = (unsigned) crash_function;

This appears to be a perfectly correct line of C, yet it causes myfunction2() to return to the incorrect address. This is because *index* is outside the bounds of the array. *mylocalarray* only has 10 elements, 0-9. In this case, index = 11. This code will write to a location on the stack that it should not - in this case, the location where myfunction2() has saved the contents of its Link Register. This is commonly called "Smashing the stack" .

How to Solve the Problem

This application contains a common error -- accessing an array outside of its bounds. The problem can be resolved by increasing the size of the array to at least 12 elements. Ordinarily, it is desirable for software to verify that any value to be used as an index to an array will not be past the end of the array.

Another common cause of stack problems is a stack overflow. Standalone applications are assigned a fixed amount of stack space at compile time. Too many nested function calls and/or local variables can exceed this amount. If problems which appear to be stack corruption appear, it may be a useful test to try increasing the stack space available to your application. For Xilinx standalone executables, this can be set by modifying the applications linker script.

Debugging Crashes in the Field

Identifying the Problem

When an embedded system is deployed in the field debugging application crashes takes on new challenges. It is generally not possible to run the application with the debugger in this environment. Some kind of useful error reporting from the field is necessary. Software is installed at the Hardware Exception vector which will supply debugging information that customers can forward to the appropriate technical support staff.

Explaining the Problem

In the TestApp_StackOverflow application there is provided a **minimal** example of crash debugging software. The project file **mb_crashtrace.c** contains code to set up a handler for this exception and print out useful information when a crash occurs. The necessary setup is presently commented out:

```
int
main (void) {
#ifdef SETUP_CRASHTRACE
   setup_crashtrace();
#endif
```

The **#ifdef** is removed and the application recompiled. A pre-built binary with this modification has been provided as `ready_for_download/TestApp_StackOverflow-crashtrace.elf`

Program output is now:

```
-- Entering main() --
-- Entering myfunction1() --
-- Entering myfunction2() --
-- Entering myfunction3() --
-- Exiting myfunction3() --
-- Exiting myfunction2() --
-- Entering crash_function() --

STACK MAXIMUM DEPTH: 0x20C bytes out of 0x2000 total.
============================================================
ILLEGAL_OPCODE Program Exception at: 0x8C006050
FRAME: 0x8C008FD0 RETURN 0x8C0001EC
FRAME: 0x8C00901C RETURN 0x8C000348
============================================================
```

This information should be saved to a text file for later analysis. A TCL script **stackscan** is provided with this application note which will look up addresses from the crash output in a symbol file:

Produce a symbol file for this application:

```
$ mb-nm --numeric-sort TestApp_StackOverflow-crashtrace.elf >
TestApp_StackOverflow-crashtrace.sym
```

Execute the script using XMD as the TCL parser from within the `ready_for_download` directory:

```
$ xmd -tcl ../../xmd_tcl_scripts/stackscan.tcl
TestApp_StackOverflow-crashtrace.crash TestApp_StackOverflow-
crashtrace.sym
EXCEPTION ADDR: 0x8C006050 crash_instructions+4
FRAME:          0x8C0001EC crash_function+92
FRAME:          0x8C000348 main+40
```

The script can also be executed with **tclsh** as the TCL interpreter. **TCLSH** is part of the standard TCL installation available at http://sourceforge.net/projects/tcl and is NOT part of the

Xilinx EDK install. If TCL is installed, the script can be run like any ordinary executable installed somewhere in a place specified with the users PATH environment variable:

```
$ stackscan.tcl TestApp_StackOverflow-crashtrace.crash
TestApp_StackOverflow-crashtrace.sym
```

How to Solve the Problem

The callstack output with symbols provided by stackscan can give developers a good clue of what activity was taking place when the application crashed. From this output it is seen that an illegal instruction was executed from within crash_instructions -- an array. The processor should not be attempting to execute code from an array, which contains data. It is seen that the last function called was crash_function, which seems to have been called by main(). It is known from examining the source that main() never calls crash_function(). Any irregularities in the callstack should make the developer suspect that stack corruption has occurred. The developer now knows to begin looking at code within main() and any functions that it calls, perhaps specifically looking for any references to crash_function or the variable crash_instructions.

Debugging memory allocation

Identifying the Problem

Stack errors are not the only memory pool errors that can occur. Most software of any complexity will require dynamic memory allocation where the memory allocated is more persistent than the life of a single function call (as happens with the stack). The well known libc functions **malloc**() and **free**() are used to dynamically carve up a block of dynamic memory known as the **heap**. Like all system resources, space on the heap is finite. One of the more difficult software errors to track down is a **memory leak**. A memory leak occurs when memory which was allocated for a transient purpose is never freed, and is therefore lost to the system until a reboot occurs. Eventually no memory will be left in the pool, and all calls to **malloc**() will fail.

Explanation of the Problem

The most effective method to debug any heap related issue is to replace the library versions of **malloc**() and **free**() with debug versions which have been specially instrumented. The Xilinx EDK does not provide source code for the malloc implementation used. This prevents easy modification of malloc itself, or close examination of the data structures it uses.

At first glance, it would seem that the proper course of action is to download a suitable library source (such as newlib) and build it. With the assistance of a special linker feature, however, it is possible to produce a debug solution without modifying the library version of malloc() or knowledge of any of its internals. A **wrapper** shall be created.

The linker parameter **--wrap** allows for any library function to be intercepted. For example, "--wrap malloc" would cause any call to malloc() to be intercepted with the function __wrap_malloc() which is provided in the application. The __wrap_malloc() function will perform appropriate debugging tasks, and then allocate the requested memory by calling __real_malloc(), which is the actual library malloc function.

SDK provides an option for user-specified compiler flags. The compiler, in turn, provides a method for flags to be passed to the linker with the **-Xlinker** compiler option. The proper compiler options to create a wrapper for the library function main() is:

-Xlinker --wrap -Xlinker malloc

See Figure 5 for an example SDK compiler options configuration to create a wrapper for malloc() and free(). These are used in the provided application TestApp_malloc.

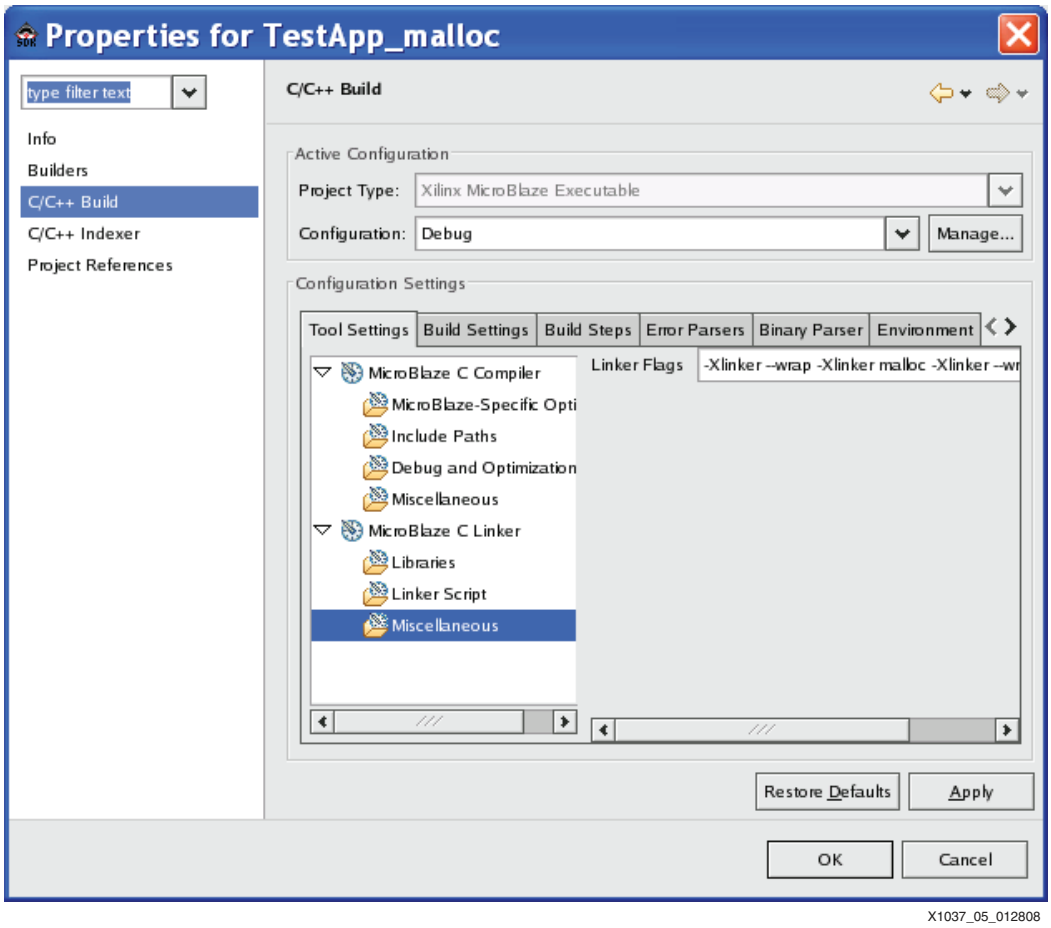

Figure 5: **Setting the linker wrapper option**

TestApp_malloc

TestApp_malloc provides code useful in locating memory leaks. This is done by building upon lessons learned with TestApp_StackOverflow. As discussed previously, it is possible to determine the precise location where a function call occurred all the way up the call tree. This is a useful tool which can be applied to determine the callers of malloc in the application. Statistics are kept on a per-caller basis. In this way it can be seen what memory has been allocated and never freed.

The caller of a function is determined, as seen previously, by examining the contents of the Link Register:

```
/*
 * Determine where malloc() was called from
 */
lr_val = mfgpr(r15);
```

This information is stored in the allocated memory itself. Additional bytes are added to the users requested memory to hold this debug info, then the user is returned a pointer past the beginning of allocated memory. See Figure 6.

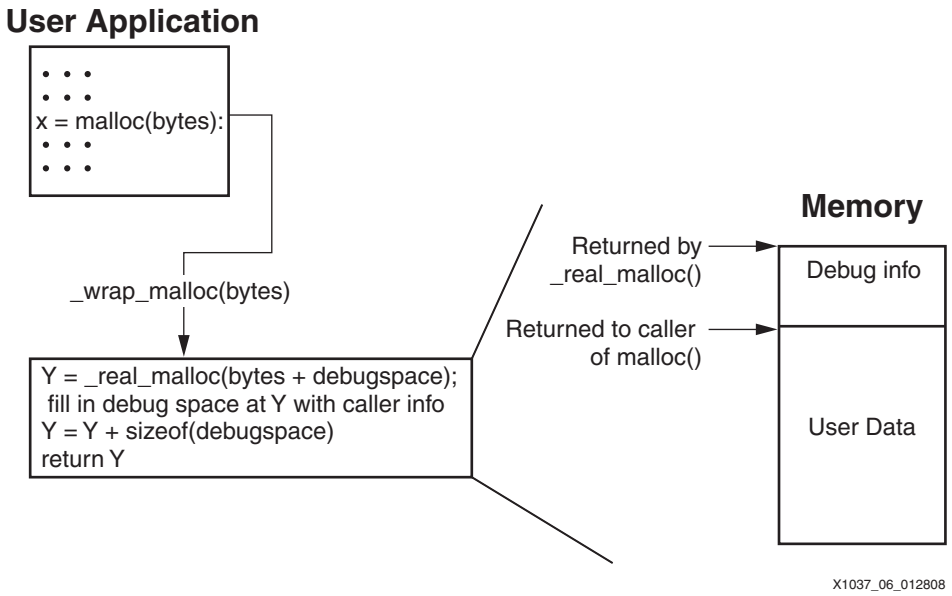

Figure 6: **Malloc Wrapper Flow**

The array *malloc_log_data[]* is used to track total bytes allocated from an individual call location. Because an array is statically allocated, the maximum number of callers to malloc() that will be tracked is determined at compile time.

```
/*
 * Data to be recorded in debug array when malloc() is called
 */
typedef struct malloc_data malloc_data_t;
struct malloc_data {
    u32    caller_return; /* Stats per caller PC */
    Xint32 bytes;         /* Bytes allocated from this caller */
    u32    total_allocs;  /* Total times malloc() called from this caller */
    u32    total_frees;   /* number of these allocs which were later freed */
};

/*
 * data from each malloc() and free() are stored here.  This is statically
 * sized since it is memory allocation itself being debugged.
 */
static malloc_data_t malloc_log_data[MAX_MALLOC_CALLER_LOG];
```

Calls to free() are intercepted in __wrap_free(). Here, the saved caller to malloc() is retrieved from the debug space. This caller information is used to find the applicable entry within *malloc_log_data[]* and decrement the amount of memory logged to the appropriate caller of malloc().

When a call to malloc fails (when malloc() returns NULL) the gathered statistics are viewed with a call to malloc_log_print(). The output of TestApp_malloc is shown below:

```
-- Entering main() --
MemWaster4 ../src/TestApp_malloc.c:144 malloc failed!
MALLOC Statistics:
Total calls to malloc: 86
Total calls to free:   27
Malloc failures:       1
```

```
MALLOC CALLER: 0x8c0001bc TOTAL BYTES:      448 ALLOCS:    40 FREES    26
MALLOC CALLER: 0x8c000280 TOTAL BYTES:     2552 ALLOCS:    44 FREES     0
MALLOC CALLER: 0x8c00036c TOTAL BYTES:      100 ALLOCS:     1 FREES     0
```

The address of each MALLOC CALLER is looked up in the symbol table for this image, as described earlier in this document with TestApp_Crash:

```
$ mb-nm --numeric-sort TestApp_malloc.elf > TestApp_malloc.sym
```

Contents of generated TestApp_malloc.sym file:

```
...
8c0001a0 T MemWaster1
8c000264 T MemWaster2
8c000304 T MemWaster3
8c0003d4 T MemWaster4
8c00045c T main
...
```

Each of the addresses provided in the MALLOC statistics is looked up in the symbol table by hand following the procedure first outlined with TestApp_Crash. It is found that:

```
MALLOC CALLER 0x8C0001BC MemWaster1+28 presently has 448 bytes allocated
MALLOC CALLER 0x8C000280 MemWaster2+28 presently has 2552 bytes allocated
MALLOC CALLER 0x8C00036C MemWaster3+104 presently has 100 bytes allocated
```

MemWaster2() is the largest consumer of dynamically allocated memory in the application at the time when malloc_log_print() was called. This information indicate a probable cause for the memory leak.

How to Solve the Problem

The best that any debug tool can do is provide data — a clue as to where the investigation should proceed. That is the case observed with TestApp_malloc. The memory usage statistics identify which places within the application allocate the most memory in a persistent manner (memory which has not been freed). Higher memory usage does not guarantee a memory leak — there may really be a large amount of data to store. Therefore, knowledge of what type of data is stored and how much of it will be necessary to debug a possible memory leak.

A Real Application

Now a more substantial application, TestApp_emaclite, is debugged. A PC is connected to the ethernet port of the board, with its adapter configured with an appropriate IP address and link speed. When successful, the PC will be able to *ping* the board.

Begin by configuring the PC with the IP address of 192.168.1.11, with a hard configured link speed of 100MB Full Duplex.

Note: The PHY on the ML403 is capable of gigabit speeds, and will autonegotiate to this speed if connected to a similarly capable device. The Emaclite is not able to operate at this link speed. For proper operation, the PC **must** be hard configured at a lower link speed.

Identifying the Problem

When the application is started, it displays the IP configuration with which it was statically built:

```
Board IP: 192.168.1.10
Netmask : 255.255.255.0
Gateway : 192.168.1.1
```

However, as in previous examples, this application is not behaving as desired. The PC can not successfully ping this host:

```
C:\> ipconfig /all
Windows IP Configuration
```

```
Ethernet adapter Local Area Connection:

        Connection-specific DNS Suffix  . :
        Physical Address. . . . . . . . . : 00-1B-2F-35-54-43
        Dhcp Enabled. . . . . . . . . . . : No
        IP Address. . . . . . . . . . . . : 192.168.1.11
        Subnet Mask . . . . . . . . . . . : 255.255.255.0
        Default Gateway . . . . . . . . . :

C:\> ping -n 1 192.168.1.10

Pinging 192.168.1.10 with 32 bytes of data:

Request timed out.

Ping statistics for 192.168.1.10:
    Packets: Sent = 1, Received = 0, Lost = 1(100% loss),
```

Explanation of the Problem

An XMD TCL script is provided which displays the present state of the xps_ethernetlite. After the PC attempts to ping the board, this script is used:

```
XMD% stop
XMD% Info:User Interrupt, Processor Stopped at 0x8c006d5c
XMD% emaclite_print 0x81000000
0x81000000 TXBUFF W0  0x00010203
0x81000004 TXBUFF W1  0x04050000 DST_MACADDR: 0x000102030405
0x81000008 TXBUFF W2  0x00000000 SRC_MACADDR: 0x000000000000
0x8100000c TXBUFF W3  0x00000000 LENGTH: 0x0000
0x810007f4 TPLR       0x00000006
0x810007f8 GIER       0x80000000
0x810007fc TSR        0x00000008 XMIT_IE

0x81001000 RXBUFF W0  0xffffffff
0x81001004 RXBUFF W1  0xffff001b DST_MACADDR: 0xffffffffffff
0x81001008 RXBUFF W2  0x2f355443 SRC_MACADDR: 0x001b2f355443
0x8100100c RXBUFF W3  0x08060001 ETHERTYPE: 0x0806 IPv4_ARP
0x810017fc RSR        0x00000009 RECV_DONE RECV_IE
```

Where `0x81000000` is the base address of the xps_ethernetlite core.

This output indicates that a complete packet has been received by the EMAC, as seen by the RECV_DONE bit in the Receive Status Register (RSR). It is an IP Address Resolution Protocol (ARP) packet. The source MAC address matches the PC, as seen with the preceeding "ipconfig /all" output. No reply packet has been sent.

The application is investigated with GDB:

```
$ mb-gdb -nw TestApp_emaclite.elf
(gdb) target remote localhost:1234
(gdb) load
(gdb) c
Continuing.
```

Ping the board from the PC at this time. After this fails, interrupt GDB with a Control-C. This will stop program execution. The GDB prompt is again displayed.

The lwIP library maintains packet statistics. These are now displayed:

```
(gdb) set print pretty
(gdb) p lwip_stats.icmp
$2 = {
```

```
  xmit = 0,
  rexmit = 0,
  recv = 0,
  fw = 0,
  drop = 0,
  chkerr = 0,
  lenerr = 0,
  memerr = 0,
  rterr = 0,
  proterr = 0,
  opterr = 0,
  err = 0,
  cachehit = 0
}
```

No packets have been seen by software. The output of **emaclite_print** indicated that receive interrupts are enabled, and a complete packet has been received.

The **mb_print** TCL script is used to see if MicroBlaze interrupts are enabled. Before this is done, instruction execution must be paused at a location known not to have interrupts temporarily disabled by software. The application will loop forever processing received packets:

```
/* Loop forever, processing received packets. */
while (1) {
        xemacif_input(netif);
}
```

A breakpoint is set at xemacif_input, and execution is allowed to continue until this breakpoint is reached:

```
(gdb) b xemacif_input
Breakpoint 1 at 0x8c006d60: file contrib/ports/xilinx/netif/xadapter.c,
line 195.
(gdb) c
Continuing.

Breakpoint 1, xemacif_input (netif=0x8c194810)
    at contrib/ports/xilinx/netif/xadapter.c:195
195             struct xemac_s *emac = (struct xemac_s *)netif->state;
```

Now that execution is stopped at a location where it is known that software has not temporarily disabled interrupts, the status of the MicroBlaze is examined:

```
XMD% mb_print
MSR:   0x00000402 IE PVR
PVR0:  0x94000700 PVR_FULL MUL EXC VER_7.00.b
PVR2:  0x578311f0 DLMB ILMB IRQPOS DPLB IPLB INTERCON MSR PCMP MUL IPLBEXC
DPLBEXC OP0EXC UNEXC OPEXC
PVR4:  0x034d0000 NO_ICU ICW
PVR5:  0x034d0000 NO_DCU DCW
PVR10: 0x07000000 ARCH_Virtex4
PVR11: 0x0ae00000 MMU_NONE TLBACC_MINIMAL
```

The IE bit in the MSR register indicated that MicroBlaze interrupts are presently enabled.

Ascertain that the correct code is installed at the MicroBlaze interrupt vector by disassembling these instructions:

```
(gdb) disassemble 0x10 0x18
Dump of assembler code from 0x10 to 0x18:
0x00000010 <_vector_interrupt+0>:       imm     -29695
0x00000014 <_vector_interrupt+4>:       brai    14148
End of assembler dump.
```

The branch to immediate address is decoded by hand:

```
-29695 = 0xFFFF8C01
 14148 = 0x3744
(0xFFFF8C01 << 16) | 0x3744 = branch to 0x8C013744

(gdb) info sym 0x8c013744
_interrupt_handler in section .text
```

This is correct. This vector simply branches to _interrupt_handler. The call table used by _interrupt_handler is examined:

```
(gdb) p MB_InterruptVectorTable
$3 = {{
    Handler = 0x8c014940 <XIntc_InterruptHandler>,
    CallBackRef = 0x8c0906c4
  }}
```

The xps_intc handles all external interrupts. This is correct.

The XIntc call vector table is defined as:

```
Excerpts from xintc.h and xintc_g.c
XIntc_Config XIntc_ConfigTable[]
/*
 * This typedef contains configuration information for the device.
 */
typedef struct {
        u16 DeviceId;           /**< Unique ID  of device */
        u32 BaseAddress;        /**< Register base address */
        u32 AckBeforeService;   /**< Ack location per interrupt */
        u32 Options;            /**< Device options */

        /** Static vector table of interrupt handlers */
       XIntc_VectorTableEntry HandlerTable[XPAR_INTC_MAX_NUM_INTR_INPUTS];
} XIntc_Config;
```

The individual XIntc_VectorTableEntry fields in HandlerTable are set by XIntc_Connect(). This structure is defined as:

```
/* The following data type defines each entry in an interrupt vector table.
 * The callback reference is the base address of the interrupting device
 * for the driver interface given in this file and an instance pointer for the
 * driver interface given in xintc.h file.
 */
typedef struct {
        XInterruptHandler Handler;
        void *CallBackRef;
} XIntc_VectorTableEntry;
```

Now the handlers for devices connected to the xps_intc are examined:

```
(gdb) p XIntc_ConfigTable
$4 = {{
    DeviceId = 0,
    BaseAddress = 2172649472,
    AckBeforeService = 6,
    Options = 1,
    HandlerTable = {{
        Handler = 0x8c000628 <xadapter_timer_handler>,
        CallBackRef = 0x0
```

```
        }, {
          Handler = 0x8c013c20 <XEmacLite_InterruptHandler>,
          CallBackRef = 0x8c090ca0
        }, {
          Handler = 0x8c0157e4 <StubHandler>,
          CallBackRef = 0x8c0906c4
        }}
    }}
```

The correct handler for xps_emaclite is present.

Examine the status of the interrupt controller with the provided **xps_intc_print** TCL command:

```
XMD% xps_intc_print 0x81800000
0x81800000 ISR   0x00000003
0x81800004 IPR   0x00000001
0x81800008 IER   0x00000001
0x8180000c IAR   0x00000000
0x81800010 SIE   0x00000000
0x81800014 CIE   0x00000000
0x81800018 IVR   0x00000000
0x8180001c MER   0x00000003
XMD%
```

Definitions in xparameters.h indicate which bits correspond to which device:

```
#define XPAR_XPS_TIMER_1_INTERRUPT_MASK 0X000001
#define XPAR_ETHERNET_MAC_IP2INTC_IRPT_MASK 0X000002
#define XPAR_RS232_UART_INTERRUPT_MASK 0X000004
```

It is seen that only the timer interrupt is enabled (IER), yet there are timer and ethernet mac interrupts asserted (ISR). Ethernet interrupts have not been enabled inside the xps_intc.

When platform_setup_interrupts() is examined, it is found that the code which enables Ethernet interrupts has been commented out:

```
#ifdef NOTNOW_FIXME
        /* Enable timer and EMAC interrupts in the interrupt controller */
        XIntc_mEnableIntr(XPAR_XPS_INTC_0_BASEADDR,
                          XPAR_XPS_TIMER_1_INTERRUPT_MASK |
                          XPAR_ETHERNET_MAC_IP2INTC_IRPT_MASK);
#endif
```

A pre-compiled binary with this change is provided in ready_for_download/TestApp_emaclite-pingable.elf.

How to Solve the Problem

If the code to enable all the interrupts used by the application was accidently removed or forgotten, remove the #ifdef and #endif and recompile the application the problem to resolve the problem.

Conclusion

This application note has provided several debugging tools such as XMD TCL scripts to debug Xilinx peripherals, software to handle exceptions and display useful debugging information, and software to gather memory allocation statistics. Standard tools (generally referred to as *binutils*) such as **nm** and **objdump** were introduced to the user. The Xilinx Microprocessor Debugger (XMD) and the GNU Debugger (GDB) were used to debug several software errors.

References

1. UG081 *MicroBlaze Processor Reference Guide*
2. UG111 *Embedded System Tools Reference Manual*
3. Stallman, Richard, Roland Pesch, Stan Schebs. *Debugging with GDB*. Boston: The Free Software Foundation, 2007
4. Ousterhout, John. *Tcl and the TK Toolkit*. Reading: Addison-Wesley Publishing Company, 1994.

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
2/28/08	1.0	Initial Xilinx release.

Notice of Disclaimer