

## Introduction

The MIPS Architecture has historically provided a set of primitives for debugging software and systems that is consistent with the RISC philosophy of an integrated hardware/software architecture, providing functionality at a minimum cost in silicon. As SoC's grew increasingly sophisticated, it became clear that additional On-Chip Debug (OCD) functionality beyond EJTAG was needed. Hard to track code errors such as null pointers or writes to non-existent memory become more prevalent as applications and SoCs become more complex. The advent of more complex applications also highlighted the need for better performance analysis tools. To address these new debugging/profiling needs, MIPS Technologies defined MIPS Trace.

MIPS Trace extends the EJTAG debug environment, providing a hardware mechanism for tracking the history of execution through a program. This document describes the MIPS Trace functionality as implemented in the MIPS Technologies 4KE family of cores (4KE, 4KSD, and M4K). Concrete examples illustrating the functionality and utility of PDtrace in a real development environment are provided in this paper.

The development environment used for the examples consists of the SDE MIPS toolkit, GDB/Insight GUI environment, a Malta motherboard combined with a PDtrace enabled MIPS 4KEc Core-FPGA, and an FS2 ISA MIPS/T EJTAG probe. Software support for the probe and for PDtrace operation is provided by FS2 through their Command Console software. The Console code has been integrated with the MIPS Software Toolkit and the GDB/Insight GUI environment.

This document assumes that the development environment has been installed, configured, and is working properly. Details about installation and set-up of the MIPS Software Toolkit, GDB/Insight, the Malta 4KEc and the FS2 probe hardware and software are available from MIPS and FS2 respectively and are not covered here.

## MIPS Basic Debug Architecture Overview

The basic MIPS Debug Architecture set of primitives include:

- A breakpoint instruction, BREAK, whose execution causes a specific exception
- A set of trap instructions, whose execution causes a specific exception when certain register value criteria are satisfied
- A pair of optional Watch registers that can be programmed to cause a specific exception on a load, store, or instruction fetch access to a specific 64-bit doubleword in virtual memory
- An optional TLB-based MMU that can be programmed to trap on any store to a page of memory

All these mechanisms assume software support in the form of at least a software monitor that can manage these primitives and communicate with a user. While this debug model works well in a processor board based system, it has a number of shortcomings when applied to System-on-Chip (SoC) development.

To address these shortcomings, Enhanced JTAG (EJTAG) was specified by the MIPS community as the debug methodology to be used in SoC development. A processor or SoC implementing EJTAG can be tied into a JTAG Tap Controller and debugged using an external EJTAG probe connected to the system's JTAG TAP interface.

In addition to providing a standard debug I/O interface, EJTAG provides the following new capabilities for software and system debug:

- Off-board EJTAG memory - A MIPS processor in Debug Mode sees EJTAG memory mapped as physical memory, but references to this memory are converted into transactions on the TAP interface. Both instructions and data can be accessed in EJTAG memory, which allows debugging of systems without requiring the presence of a ROM monitor or debugger scratch pad RAM. It also provides a communication channel between debug software executing on the processor and an external debugging agent.
- Hardware breakpoints - Two types of hardware breakpoints have been added. They can be configured to cause a debug exception (1) on an instruction fetch from a specific virtual address and (2) on a memory reference from a specific virtual address that can be additionally qualified by a data value.
- Single-Step execution - Code can be single stepped without it being resident in RAM.
- System access via the EJTAG TAP - An external debugging agent can obtain information about the configuration and state of the processor under test and can force processor entry into Debug Mode, allowing further system access via EJTAG memory.
- Debug Breakpoint Instruction - the new breakpoint instruction SDBBP places the processor in Debug Mode and can fetch its associated handler code from EJTAG memory.

## **MIPS Trace overview**

MIPS Trace is an extension to the MIPS debug architecture. Figure 1 shows an implementation of MIPS Trace. The implementation consists of the EJTAG TAP controller module, a PDtrace module, a Trace Control Block (TCB) module and the trace data path from the PDtrace interface to the probe interface.

FIGURE 1 -

The PDtrace module is an architecturally defined extension to a MIPS synthesizable core. The module's function is to track instructions that actually complete and to send extract trace information from the processor pipeline and to stream some subset of information through the core's PDtrace interface. Stalls, Branch Taken target instructions, and Load or Store Instructions along with their associated addresses and data values are flagged and can be identified in the trace stream. The PDtrace module is described in the *PDtrace™ Interface Specification*, MIPS document number MD00136.

The cycle-by-cycle data streamed out of the core through the PDtrace interface is captured and formatted (compressed) by the Trace Control Block (TCB), a pipeline-independent module separate from the core. The Trace Control Block is described in the *EJTAG Trace Control Block Specification*, MIPS document number MD00148.

The compressed trace data can either be stored in on-chip trace memory (if any) or streamed off the chip through the EJTAG TCB's Probe Interface Block (PIB) into memory on a trace capable probe. Small amounts of data can be stored in the on-chip memory and accessed through the EJTAG TAP interface. Large amounts of data should be transmitted through the PIB's parallel interface to the attached probe.

### ***Enabling and Configuring MIPS Trace***

In the MIPS architecture, configuration of cores is done through registers in Coprocessor 0 (CP0). In PDtrace enabled cores, the core powers up with PDTrace configuration defaulted to the EJTAG Trace Control Block Registers.

MIPS Trace is normally enabled and configured through the EJTAG TAP interface. An EJTAG probe sends commands through the EJTAG TAP interface to program the Trace Control Block registers. These TCB registers configure both the core's PDtrace features and the Trace Control Block and the Probe Interface Block. In some unusual debug situations it is useful to control tracing from software running on the core rather than through a relatively slow EJTAG control channel. In those circumstances, debug software would transfer Pdtrace configuration control to CP0 Trace Control registers (CP0 Register 23, selects 1, 2, 3, 4). Application or debug software would then configure or re-configure the core's PDtrace features to collect whatever data is required.

### **MIPS Trace Tools Overview**

Support for MIPS Trace is provided through a combination of products from MIPS Technologies and First Silicon Solutions (FS2). MIPS provides the SDE tool chain. FS2 provides the ISA MIPS/T EJTAG trace enabled probe. The tool chain communicates with the probe through a software API called the *Microprocessor Debug Interface (MDI)*.

The *MDI* interface is a software library that allows debuggers to connect to MIPS targets in a device independent manner. SDE version 5.03 or later supports *gdb* connections to either MIPSsim (a software target) or to *MDI* compliant EJTAG probes (a hardware target). SDE 5.03 has been tested with the FS2 ISA-MIPS EJTAG probes.

The ISA MIPS/T probe is an ISA-MIPS EJTAG probe that includes support for MIPS Trace. The Trace enabled probes are supported by FS2 with their Command Console software that adds Tcl/Tk based MIPS Trace control and display to the *gdb/Insight* debug environment. The three Tcl/tk based programs are the following:

1. Trace Mode Window - Configures PDtrace, defining the trace information to be collected. It also sets up the Trace Control Block and the Probe Interface Block
2. HW Triggers Window - Configures the trigger features, defining when and under what condition trace data will be collected
3. HW Trace Window - provides various views of the trace results, including source line insertion and variable name lookup.

## 4KEc MIPS Trace Implementation

The 4KE synthesizable core family (4KE, 4KSD, and M4K cores) includes trace capability. Both the PDtrace module and the TCB module are included as part of the MIPS core deliverables. The modules are optional blocks that can be selected at synthesis time. Trace enabled FPGAs can be built using the scripts provided with the 4KE core deliverables.

The standard FPGAs provided as part of Malta and SEAD2 4KE family Development Systems include MIPS Trace. The FPGAs are built with both an on-chip trace buffer and a Probe Interface Block. Configuring the FPGA in this way allows the user to benefit from PDtrace even if an ISA-MIPS/T probe is not immediately available. Performance and versatility are significantly enhanced when using a trace enabled probe, but any EJTAG probe compliant with the 2.60 version of the MIPS EJTAG Specification [1] can access the on-chip memory through the EJTAG TAP.

The 4KEc FPGAs are built with the following configuration options selected:

- 16Kb I-cache, organized 4Kb by 4 way set associativity
- 16K D-cache, organized 4Kb by 4 way set associativity
- No SPRAM
- 1 register file
- No COP2 Interface
- MIPS16e optimized for size (1 decoder)
- EJTAG with a single TAP connection
- 4 Instruction and 2 Data breakpoints
- 2Kb on-chip trace buffer, organized 256 addresses by 64-bits

## Debugging with MIPS Trace

### ***Building and starting an Application for MIPS Trace analysis***

The SDE Toolkit includes the example, `fs2_ex.c`, is found in the directory `../sde/examples/fs2_ex`. The example can be built by opening a bash shell, `cd`'ing to the `../examples/fs2_ex` directory and typing `sde-make fs2`. The switch `fs2` directs the makefile to create the symbol file that will be used by FS2's Command Console software.

Typing `sde-gdb fs2_ex` will bring up the FS2 Console window and the Insight window with the program source loaded in the Insight window. When Insight has started and the `fs2_exram` program has loaded, the Insight Source Window will look like the following:

```

18 /* globals */
19 /*****/
20 unsigned char global_x;
21 unsigned short global_y;
22 unsigned long global_z;
23
24 /*****/
25 /* main program */
26 /*****/
27
28 //-----
29 // main has 3 local variables a, b, and c
30 // used as return values from func<x>
31 //-----
32 int main (void) {
33     volatile unsigned char a, b, c;
34     static unsigned char d;
35     unsigned long loopx = 0;
36
37     // set globals global_x, global_y, and global_z
38     global_x = 0x11;
39     global_y = 0x22;
40     global_z = 0x33;
41
42     // this loop increments local variable loopx
43
44     while ( 1 ) {
45         a = func1(loopx);
46         b = func2(loopx);
47         c = func3(loopx);
48         d = a + b + c;
49         loopx++;
50     }
51     return 0;
52 } // end main()

```

Program stopped at line 35

fs2\_ex.c main SOURCE

The FS2 console window looks like the following:



To bring up the associated Trace Mode, HW Trace or HW Triggers windows, click on the appropriate menu item under the Console Window menu: The operation, menus, and setup fields of each window are described in detail in the FS2 on-line help for that window.

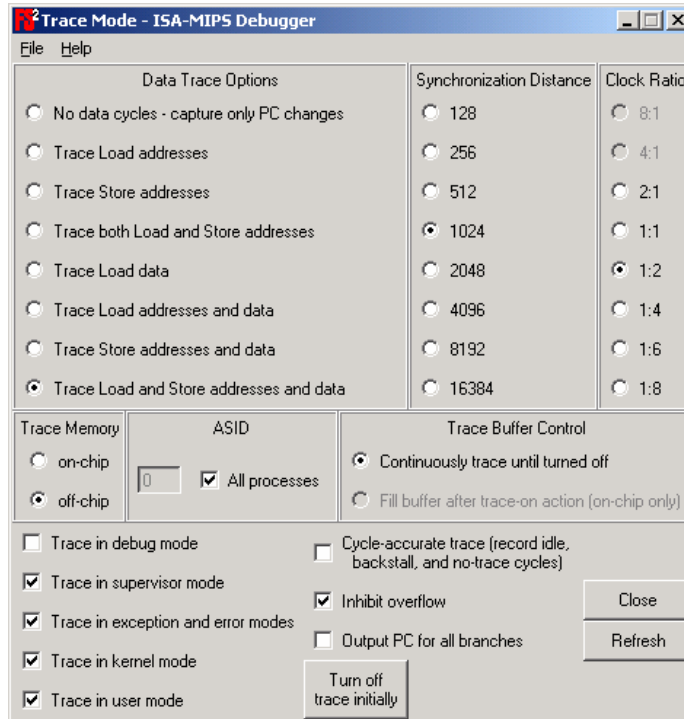
### ***Configuring PDtrace Operation***

By default, MIPS Trace is set up to capture only Program Counter changes; trace in user, kernel, exception and error modes; send data to off-chip memory; continuously trace until turned off with overflow inhibited; and using a synchronization distance of 1024 frames. These default settings can all be modified through the Trace Mode window. This window is accessed through the Window pull down of the Console Window, shown in Figure 3 above. The Trace Mode Window is shown in Figure 4 below.

The Trace Mode window is divided into a number of sections. Each section manages a specific feature of PDtrace. The following sections describe the features of trace controlled through the Trace Mode window.

### **Clock Ratio**

The core clock frequency and the frequency of the clock controlling the transmission of data to a trace-enabled probe do not have to be the same. Usually, the off-chip clock frequency is some sub-multiple of the core clock frequency. The range of permissible values are defined when the chip is implemented. The default setting for the 4Ke family is a 1:2 ratio. A 1:2 ratio means that the Probe Interface Block frequency is  $\frac{1}{2}$  the core clock frequency. In the 4Ke, the PIB interface works in double data rate mode (DDR), meaning that at the default setting one data packet is transmitted on each edge of the off-chip clock. Care must be taken to not exceed the 100MHz maximum clock rate of the FS2 probe.



## Synchronization Distance

Complete Program Counter values are periodically transmitted, providing trace reconstruction software with the ability to check its synchronization with the arriving compressed trace data. The Synchronization Distance sets the number of core clock cycles that separate complete Program Counter value updates. The values shown in the figure are for off-chip tracing. Data being collected in on-chip memory will have synchronization updates occurring at 32 times the off-chip rate.

## Trace Memory

Trace information is captured either in on-chip memory or off-chip trace memory. Trace information stored in on-chip memory is accessible through the EJTAG port and can be read out by any EJTAG 2.6 or higher rev probe, FPGAs provided by MIPS are normally built with a 2Kb on-chip trace buffer, which is organized 256 addresses by 64-bits. Selecting off-chip memory causes trace information to be streamed through the Probe Interface Block. The FS2 ISA-MIPS/T probe will store this streamed data in on-probe memory and make it available for host processing.

## Trace Buffer Control

Trace buffer control only controls the flow of data into the on-chip trace memory. Trace data sent to off-chip (probe) memory flows continuously to the probe, stopping only when the core stops sending trace information.

When trace data is being sent to on-chip memory, trace data collection can be controlled in any of three ways:

1. Operate in Trace-To mode. This is the default operating mode.
2. Operate in Trace-From mode.
3. Operate under the control of a Trigger unit. Triggers will be discussed later.

In the Trace-To mode, trace data is continually written into the on-chip buffer, wrapping around and overwriting the oldest data. Data collection stops only when the processor reaches an end-of-trace condition. End-of-trace occurs when the processor exits the processor mode or ASID value in which tracing was requested or when an EJTAG hardware breakpoint trigger turns tracing off.

In the Trace-From mode, trace data collection starts when the processor enters a start-of-trace condition. A start-of-trace condition occurs when the processor enters a processor mode or ASID value for which tracing has been enabled or when an EJTAG hardware breakpoint trigger turns tracing on. Trace data collection will continue until the on-chip buffer is full. The trace will be a log of processor activity after the trigger. Since tracing starts when some condition is met, trace must initially be turned off. This is done by pushing the 'Turn off trace' button.

## Inhibit Overflow

The PDtrace block in the core contains a fifo to buffer the data flow between the core and TCB. However, it is still possible for data to be output from the core faster than the TCB can process it. Inhibit overflow causes the processor to stall if trace data output from the core arrives faster than the TCB can collect it.

If an overflow occurs, the fifo is flushed and tracing is restarted. Only the data in the fifo at the time of the overflow is lost. A marker is placed at the beginning of the new trace record, indicating where the break in data occurred.

## Filtering Modes - ASID and Trace in

By default, MIPS Trace collects all program code information. It is of obvious value to limit the quantity of information being collected. PDtrace uses the concept of *MatchEnable* expressions to enable selective filtering of trace data. *MatchEnable* expressions allow tracing over specific program areas or in specific processor operating modes.

In cores that implement a standard TLB-based MMU, the address space identifier (ASID) can be used to identify program code that will be traced. ASIDs are software constructs normally generated and used by MMU-based operating systems such as Linux. An ASID is an integer number generated by the operating system to identify a particular address space configuration. The ASID is written by the operating system into the **EntryHi** CP0 register and can be used by MIPS Trace to enable or disable tracing.

Both the collect-all and the ASID trace collection methods can be further qualified. Program code can be traced in one or more of the specific privilege modes allowed by a core implementation. The MIPS 4Ke family implements User, Kernel, Exception, and Debug Privilege Modes. These privilege modes reflect a hardware state of the core and can be very useful in limiting the amount of trace data collected.

### **Cycle Accurate trace**

PDtrace data transmitted from the core to the Trace Control Block is cycle-by-cycle information, including stalls. That is, PD trace data is inherently cycle accurate. Because including stall cycles in the trace increases the size of the trace, sometimes significantly, and because stall information is generally not useful in software debugging, the TCB normally does not collect the presented stall information. Setting the Cycle Accurate Trace radio button causes stall information to be included in the stored trace. Cycle accurate information is useful when trying to determine the exact operation of the core, cache and memory subsystem.

### **Output PC for all branches**

PDtrace normally only transmits Program Counter (PC) information for a branch when the branch target can't be predicted from the static program image (statically predictable), leaving address reconstruction to the trace reconstruction software. Target addresses are statically predictable for branch and all jump-immediate instructions. For statically predictable branches, a branch-taken-flag is inserted in the trace to indicate when the branch is taken.

All jump-register instructions and ERET/DERET instructions have unpredictable target addresses. PC changes due to exceptions (interrupt, reset, etc.) are also treated as having unpredictable target addresses. In the unpredictable cases, the complete PC is transmitted unless it is dynamically determined that transmitting the delta value will reduce the number of bits necessary to indicate the new PC.

Setting the Output PC (Program Counter) check box for all branches cause complete addresses for all branches taken to be stored. This is not normally necessary, especially since PDtrace transmits a full PC periodically as a synchronization safety check.

## Data Trace Collection Options

The Data Trace Radio buttons are used to configure PDtrace's trace collection modes. The choices range from a simple tracking of the Program Counter addresses (PC) when unpredictable program flow occurs to tracking all operand activity. Choosing only the information needed will minimize the bandwidth needed to move data out of the core and into the Trace Control Block (TCB)

## Debugging an application

We had previously started Insight and loaded the demonstration program fs2\_ex into our target. Setting a breakpoint on line 50 and clicking on Continue causes program execution and trace collection. The Source Window is updated as follows:

```

35  unsigned long loopx = 0;
36
37  // set globals global_x, global_y, and global_z
38      global_x = 0x11;
39      global_y = 0x22;
40      global_z = 0x33;
41
42  // the while loop increments local variable loopx
43
44      while ( 1 ) {
45          a = func1(loopx);
46          b = func2(loopx);
47          c = func3(loopx);
48          d = a + b + c;
49          loopx++;
50  }
51  return 0;
52 } // end main()

```

Program stopped at line 50

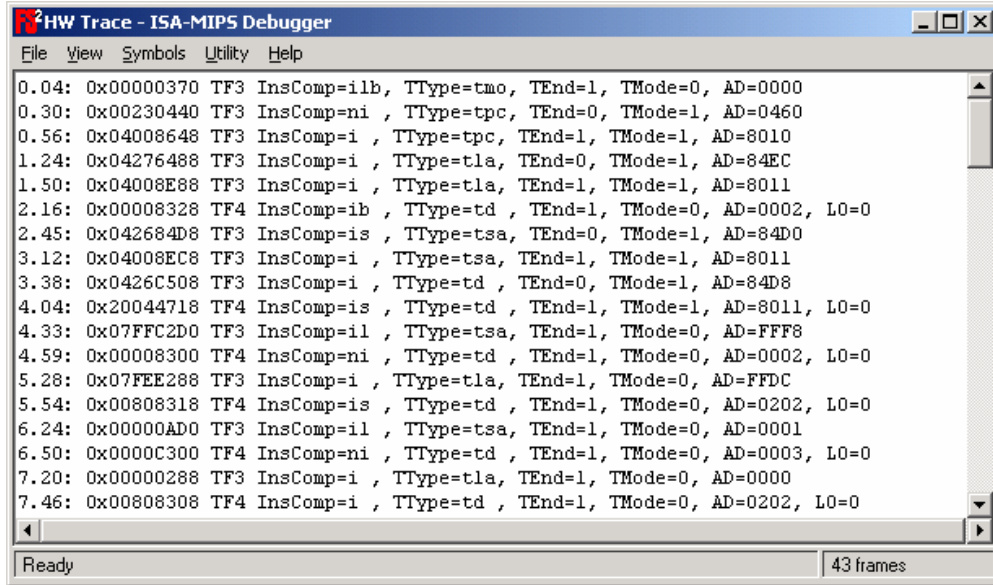
fs2\_ex.c | main | SOURCE

## Views

Trace output is displayed in the HW Trace window. To open the HW Trace window, go to the top of the FS2 Console and select Windows -> HW Trace. Several Views of the collected trace information are possible. For debugging purposes, the most useful are intermixed source and disassembly lines. As part of this tutorial, we'll briefly show the other possible views.

### Trace Message view

Select View -> Trace Message to display trace frame data in a parsed format annotated with the labels defined in the TCB specification.




### TRACE Message View

The individual trace lines are interpreted as follows;

The first entry, two numbers separated by a dot, is the Trace Frame. A Trace Frame is one 64-bit word in trace memory. The number before the dot is the frame number. Each frame typically contains several compressed trace messages. The entry after the decimal point indicates the bit position in the frame where the message begins. For example, the number 2.16 means Trace Frame 2, bit position 16.

## MDI view

Select View -> MDI to display trace frame data as MDI messages passed to the debugger. This display shows Program Counter values where the executed branches took place and Loads and Store information. It does not include disassembly between these branches or source line insertions.



```

HW Trace - ISA-MIPS Debugger
File View Symbols Utility Help
0.04: mode kernel (exl=0, erl=0), isa=MIPS32, asid=0x00
0.04: pc 0x00:0x80100460
0.04: ld8 0x00:0x801184EC 0x02
0.56: pc 0x00:0x80100464
1.24: pc 0x00:0x80100468
1.50: pc 0x00:0x8010046C
2.16: pc 0x00:0x8010033C
2.45: pc 0x00:0x80100340
2.45: st32 0x00:0x801184D0 0x801184D8
3.12: pc 0x00:0x80100344
3.38: pc 0x00:0x80100348
4.04: pc 0x00:0x8010034C
4.04: st8 0x00:0x801184C8 0x02
4.33: pc 0x00:0x80100350
4.33: ld8 0x00:0x801184C8 0x02
5.28: pc 0x00:0x80100354
5.54: pc 0x00:0x80100358
5.54: st8 0x00:0x801184C9 0x03
6.24: pc 0x00:0x8010035C
6.24: ld8 0x00:0x801184C8 0x02
7.20: pc 0x00:0x80100360
7.46: pc 0x00:0x80100364
8.16: pc 0x00:0x80100368
8.16: ld8 0x00:0x801184C8 0x02
9.12: pc 0x00:0x8010036C
9.16: pc 0x00:0x80100370
9.20: pc 0x00:0x80100374
9.20: st8 0x00:0x80107C0C <global_x> 0x12
9.46: pc 0x00:0x80100378
10.12: pc 0x00:0x8010037C
10.41: pc 0x00:0x80100390
10.45: pc 0x00:0x80100394
Ready 43 frames

```

### MDI View

## Interleaved Source and Disassembly View

In order to see the disassembled instructions and source line insertions, it is necessary to load symbols. Symbols can be loaded using the Symbols -> Load Symbol File menu. A standard file load dialog will open up with a "File of type:" field of "objdump files (\*.fs2)". Navigate to the "../examples/fs2\_ex" directory and double click on "fs2\_exram.fs2". A pop-up message reports the completion of symbol loading with the number of symbols loaded.

If you recompile your program (while Insight is still running) and want to reload symbols, use the "Symbols->Reload Symbol File" menu item. Alternatively, if you close the trace window then open it again, the last symbol file that was loaded will be automatically reloaded.

When symbols are loaded and enabled (Symbols > Show Symbols is checked) the filename, line number, and source line are inserted in a trace line above the code whose address exactly matches the line number address.

The screen shown below turns on instructions, load/stores, and full path and filenames. This is useful for knowing where the actual source file is located, or to distinguish which source file executed if there were duplicate filenames in different directories.

```

C:\Cygwin\usr\local\sde5.03.03\sde\examples\fs2_ex\fs2_ex.c:45
    a = func1(loopx);
0.04: 00:80100460 0x93C20014 lbu    $v0,0x0014($fp)
0.04: ld8 0x00:0x801184EC 0x02
0.56: 00:80100464 0x00402025 or     $a0,$v0,$0
1.24: 00:80100468 0x0C0400CF jal   0x8010033C
1.50: 00:8010046C 0x00000000 nop

func1:
2.16: 00:8010033C 0x27BDDFF0 addiu  $sp,$sp,0xFFFF
2.45: 00:80100340 0xAFEE0008 sw    $fp,0x0008($sp)
2.45: st32 0x00:0x801184D0 0x801184D8
3.12: 00:80100344 0x03A0F025 or     $fp,$sp,$0
3.38: 00:80100348 0x00801025 or     $v0,$a0,$0
4.04: 00:8010034C 0xA3C20000 sb    $v0,0x0000($fp)
4.04: st8 0x00:0x801184C8 0x02
C:\Cygwin\usr\local\sde5.03.03\sde\examples\fs2_ex\fs2func1.c:10
    local_l = num + 0x01;
4.33: 00:80100350 0x93C20000 lbu    $v0,0x0000($fp)
4.33: ld8 0x00:0x801184C8 0x02
5.28: 00:80100354 0x24420001 addiu  $v0,$v0,0x0001
5.54: 00:80100358 0xA3C20001 sb    $v0,0x0001($fp)
5.54: st8 0x00:0x801184C9 0x03
C:\Cygwin\usr\local\sde5.03.03\sde\examples\fs2_ex\fs2func1.c:11
    if (num < 0x80) {
6.24: 00:8010035C 0x83C20000 lb     $v0,0x0000($fp)
6.24: ld8 0x00:0x801184C8 0x02
7.20: 00:80100360 0x04400007 bltz   $v0,0x80100380
7.46: 00:80100364 0x00000000 nop
C:\Cygwin\usr\local\sde5.03.03\sde\examples\fs2_ex\fs2func1.c:12
    global_x = num + 0x10;
8.16: 00:80100368 0x93C20000 lbu    $v0,0x0000($fp)
8.16: ld8 0x00:0x801184C8 0x02

```

### Interleaved Source and Disassembly View

Other, less complete, views are available through the View pull-down menu:

- The full path and pathnames display can be turned off with View-> Dasm Option->Short File Paths menu entry.
- The display of Instruction disassembly is controlled through the View -> Dasm Option -> No Instructions menu entry.
- In the figure above, the display of Load and Store instructions is turned on and they are displayed in color (Store in red, Loads in green), distinguishing data cycles from disassembled code. The display of Loads and Stores is controlled through the View ->

No data cycles menu. The colors, fonts, and styles can be changed with View -> Highlighting menus.

- There are no idle cycles in the above display. The View -> No idle cycles entry hides notrace cycles trace messages which are generated if the cycle-accurate mode is enabled.

## Using Hardware Breakpoints to Control Trace

### *Hardware Triggers*

The EJTAG Debug architecture includes hardware instruction and data breakpoints as optional features. These hardware breakpoints can be used to turn tracing on and/or off. This is useful for collecting trace information over limited sections of code, such as a tracing execution in a single function.

The FPGA implementation of the 4KE includes 4 Instruction and 2 Data hardware breakpoints.

- Instruction hardware breakpoints can be configured to generate a debug exception when an instruction is executed anywhere in the virtual address space. Bit mask and Address Space Identifier (ASID) values may be used to qualify the address comparison.
- Data hardware breakpoints can be configured to generate a debug exception on a data transaction. Bit mask and ASID values may apply in the address compare and byte mask may qualify the value compare.

### *Instruction Triggered Trace Example*

In order for trace on-off to work, the initial state of trace needs to be turned off in the Trace Mode window. Turning trace off will uncheck "Trace in exceptions and error modes", "Trace in kernel mode", and "Trace in user mode".

In this example, trace is turned on at the entry to function func1() and turned off at the exit of the same function, thus tracing only the execution in this function. Pull up the HW trigger (Execution Trigger) window using the FS2 Command Console Window. The HW Trigger Window is used to turn trace on and off at specific hardware addresses.

The Trigger Window requires entering the both the start and end hex or decimal addresses of the function func1(). To determine the start and end address of func1(), bring up the function in the Source window. Then switch to Mixed mode to show both source and assembly. Select func1() from the function list menu, the second pull-down list from the left, at the bottom of the window.

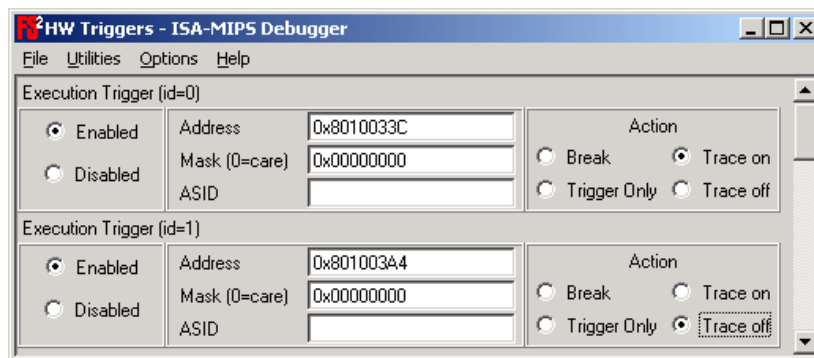
```

7 unsigned char func1 (unsigned char num) {
- 0x8010033c <func1>:          addiu   $sp,$sp,-16
- 0x80100340 <func1+4>:       sw      $s8,8($sp)
- 0x80100344 <func1+8>:       move   $s8,$sp
- 0x80100348 <func1+12>:      move   $v0,$a0
- 0x8010034c <func1+16>:      sb     $v0,0($s8)
      8                unsigned char local_1;
      9
     10                local_1 = num + 0x01;
- 0x80100350 <func1+20>:     lbu    $v0,0($s8)
- 0x80100354 <func1+24>:     addiu   $v0,$v0,1
- 0x80100358 <func1+28>:     sb     $v0,1($s8)
     11                if (num < 0x80) {
- 0x8010035c <func1+32>:     lb     $v0,0($s8)
- 0x80100360 <func1+36>:     bltz   $v0,0x80100380 <func1+68>
- 0x80100364 <func1+40>:     nop
     12                global_x = num + 0x10;
- 0x80100368 <func1+44>:     lbu    $v0,0($s8)
- 0x8010036c <func1+48>:     addiu   $v0,$v0,16
- 0x80100370 <func1+52>:     lui    $at,0x8010
- 0x80100374 <func1+56>:     sb     $v0,31756($at)
     13                } else {
- 0x80100378 <func1+60>:     b      0x80100390 <func1+84>
- 0x8010037c <func1+64>:     nop
     14                global_x = num - 0x10;
- 0x80100380 <func1+68>:     lbu    $v0,0($s8)
- 0x80100384 <func1+72>:     addiu   $v0,$v0,-16
- 0x80100388 <func1+76>:     lui    $at,0x8010
- 0x8010038c <func1+80>:     sb     $v0,31756($at)
     15                }
     16                return (global_x);
- 0x80100390 <func1+84>:     lui    $v0,0x8010
- 0x80100394 <func1+88>:     lbu    $v0,31756($v0)
- 0x80100398 <func1+92>:     move   $sp,$s8
- 0x8010039c <func1+96>:     lw     $s8,8($sp)
- 0x801003a0 <func1+100>:    addiu   $sp,$sp,16
- 0x801003a4 <func1+104>:    jr     $ra
- 0x801003a8 <func1+108>:    nop
    
```

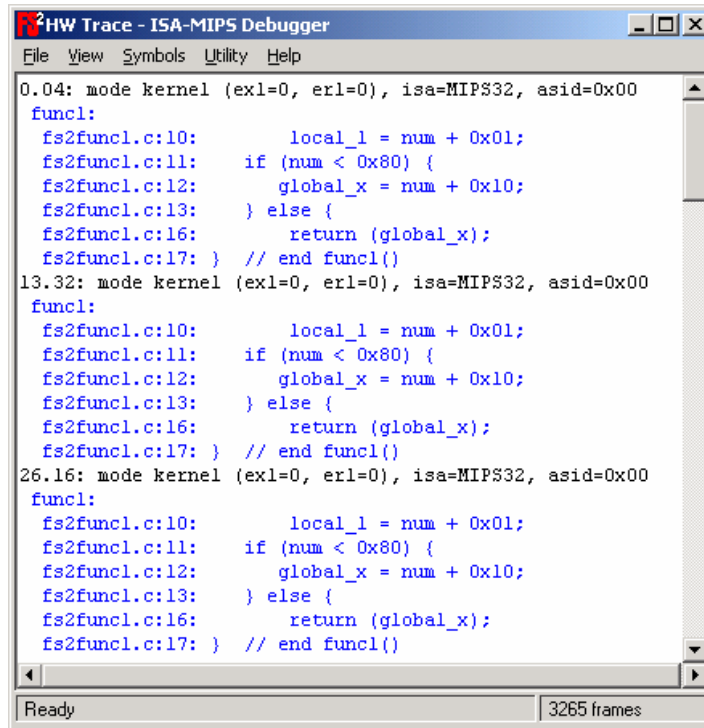
Program stopped at line 10, 0x80100350

fs2func1.c | Func1 | MIXED

As shown below, enter the function entry address (0x8010033C) in Execution Trigger (id=0), set the Mask to 0 (break only on this address, no address bits are masked), and set the Action to "Trace on". Enter the function exit address (0x801003A4) in Execution Trigger (id=1) with the Action set to "Trace off".

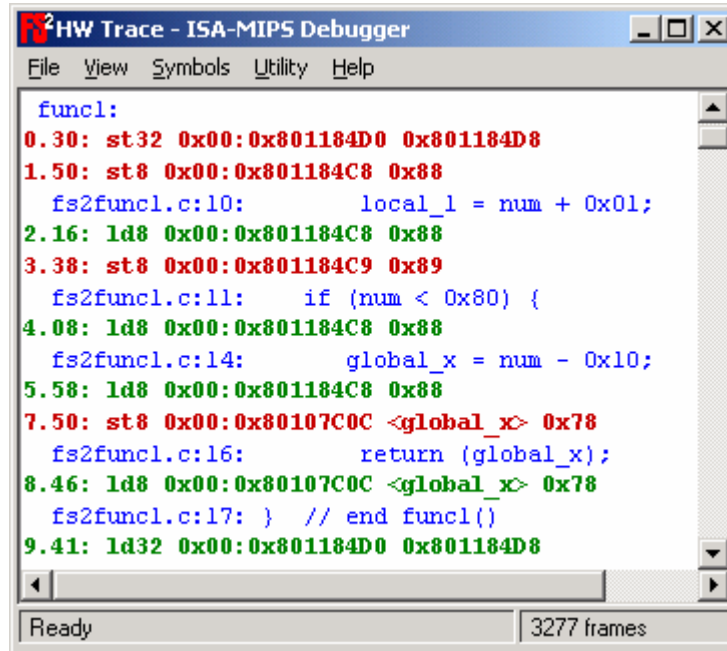


Make sure no software breakpoints are set in the program, Start the program then stop it manually with the far left “Stop” icon. A segment of the resulting trace is shown in the screen capture below. The screen shows only executions of func1() with no other code in the trace:



```
HW Trace - ISA-MIPS Debugger
File View Symbols Utility Help
0.04: mode kernel (ex1=0, er1=0), isa=MIPS32, asid=0x00
func1:
fs2func1.c:10:     local_1 = num + 0x01;
fs2func1.c:11:     if (num < 0x80) {
fs2func1.c:12:         global_x = num + 0x10;
fs2func1.c:13:     } else {
fs2func1.c:16:         return (global_x);
fs2func1.c:17: } // end func1()
13.32: mode kernel (ex1=0, er1=0), isa=MIPS32, asid=0x00
func1:
fs2func1.c:10:     local_1 = num + 0x01;
fs2func1.c:11:     if (num < 0x80) {
fs2func1.c:12:         global_x = num + 0x10;
fs2func1.c:13:     } else {
fs2func1.c:16:         return (global_x);
fs2func1.c:17: } // end func1()
26.16: mode kernel (ex1=0, er1=0), isa=MIPS32, asid=0x00
func1:
fs2func1.c:10:     local_1 = num + 0x01;
fs2func1.c:11:     if (num < 0x80) {
fs2func1.c:12:         global_x = num + 0x10;
fs2func1.c:13:     } else {
fs2func1.c:16:         return (global_x);
fs2func1.c:17: } // end func1()
Ready 3265 frames
```

Other views can provide useful information. For example, turn on load/store cycles. In the figure shown below the “if” condition of source line 11 is shown to be false because the variable “num” shown in frame 4.08 has a value of 0x88. As a result the else portion of the if-else of source line 14 is executed.



```

HW Trace - ISA-MIPS Debugger
File View Symbols Utility Help

func1:
0.30: st32 0x00:0x801184D0 0x801184D8
1.50: st8 0x00:0x801184C8 0x88
   fs2func1.c:10:    local_l = num + 0x01;
2.16: ld8 0x00:0x801184C8 0x88
3.38: st8 0x00:0x801184C9 0x89
   fs2func1.c:11:    if (num < 0x80) {
4.08: ld8 0x00:0x801184C8 0x88
   fs2func1.c:14:    global_x = num - 0x10;
5.58: ld8 0x00:0x801184C8 0x88
7.50: st8 0x00:0x80107C0C <global_x> 0x78
   fs2func1.c:16:    return (global_x);
8.46: ld8 0x00:0x80107C0C <global_x> 0x78
   fs2func1.c:17: } // end func1()
9.41: ld32 0x00:0x801184D0 0x801184D8

Ready 3277 frames

```

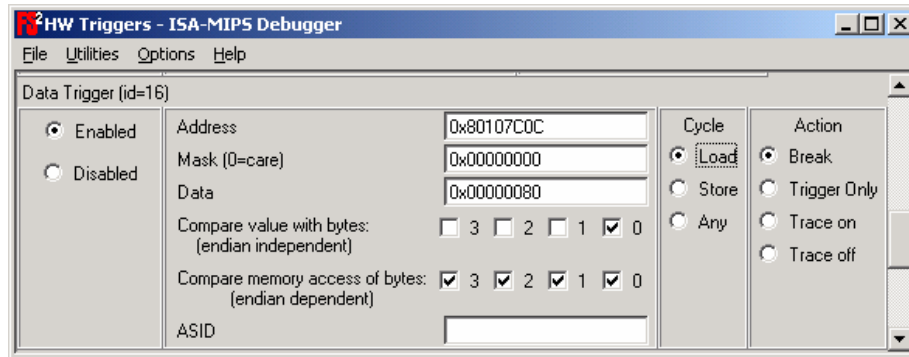
### **Data Triggered Trace Example**

Data triggers provide triggering when the value of a variable residing at a specific address matches a requested data value. The data values can be qualified by masking values found in specified byte lanes. Addresses can be qualified by a bit-wise mask of the variable’s address and by ASID comparison. Access of the variables can be qualified by masking access to specified byte-lanes, or by allowing triggering to occur on any cycle access or only on Load or Store access.

This example illustrates how to set up and trigger on a Load of the data value 0x80 into the “global\_x” variable. Pull up the Data Trigger window using the FS2 Command Console Window. The Data Trigger Window is used to turn trace on and off when specific data value conditions are satisfied.

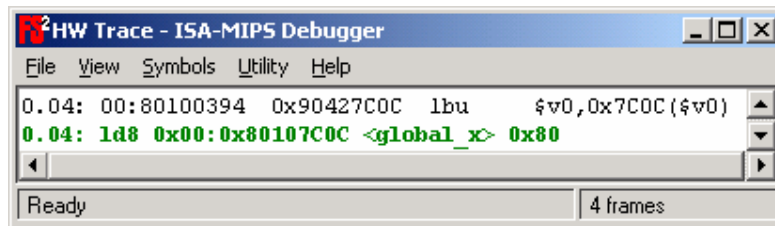
Using the Instruction breakpoint trace example, extract the address of “global\_x” which is seen to be 0x80107C0C. Enter this value in the Address field. Set the mask to 0, indicating that triggering will occur at only this address. Enter the data value 0x80 in the Data field to 0x80. Enable the 0 byte lane, indicating triggering occurs only on a match of the least significant byte.

Set Cycle type to “Load” and Action to “Break”. The Data Trigger should look like the screen below:



Start execution. The program stops when `global_x` is loaded as a return value in Source Line 16 of `fs2func1.c`.

Tracing starts when the processor is single stepped, which is required to get past this data trigger. Since the on-chip Data Trigger stopped the processor **before** it completed the Load, the return value won't have the actual load `0x80` value in it until the processor is stepped. The figure below shows the trace starting with the instruction that initiated the Load. The return value `0x80` is shown in the byte Load of variable, `global_x`.

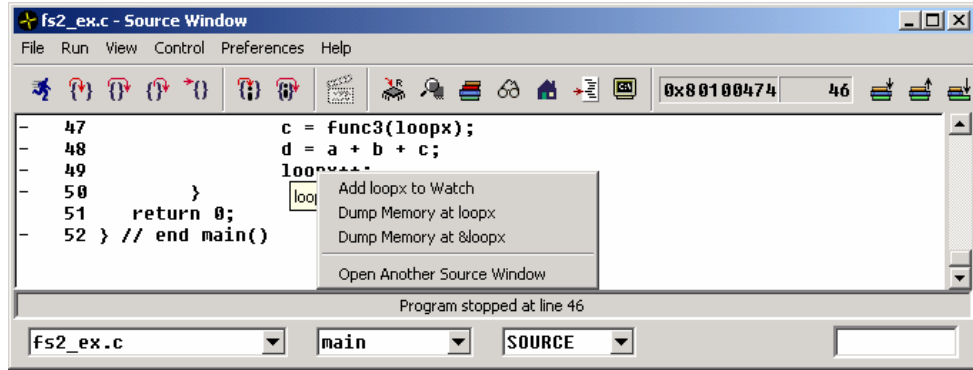


The following example sets up the trigger on `loopx` when a Store (write) of all ones (`0xFFFF`) occurs on the lower 2 bytes (16 bits) with the upper 2 bytes being a don't care. We will set a data breakpoint on `loopx`, a 32-bit variable found in `main()`. The address can be obtained by either using the Command Console:

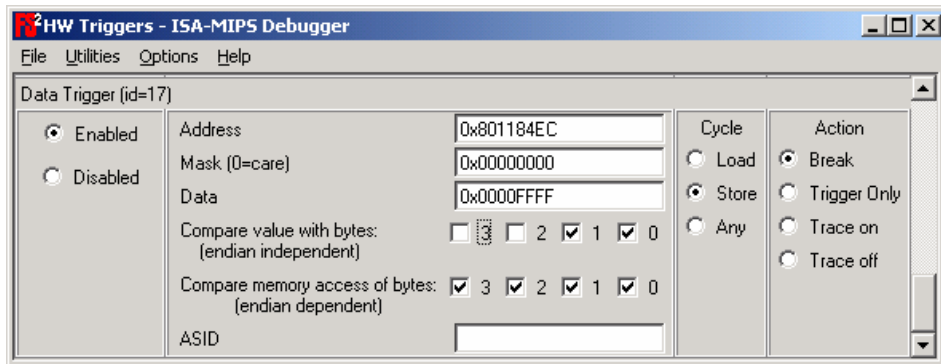
```
(gdb) print &loopx
```

```
$1 = (long unsigned int *) 0x801184ec
```

or by using Insight. As shown below, right clicking on the variable `loop_x` in the Insight window will produce a pop-up menu that allows memory to be dumped at `loopx`.

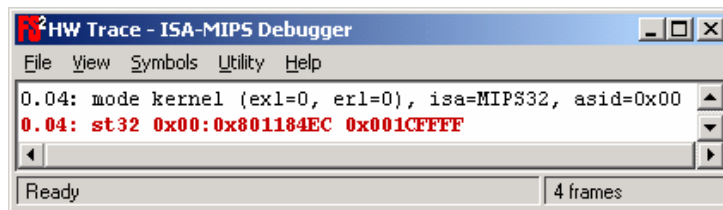


The resulting Memory window dump shows the address of loopx to be the hex value on the first line (0x801184ec). Enter the Address and Data values in the HW Trigger window. Select Store Cycle and compare with bytes 0 and 1.



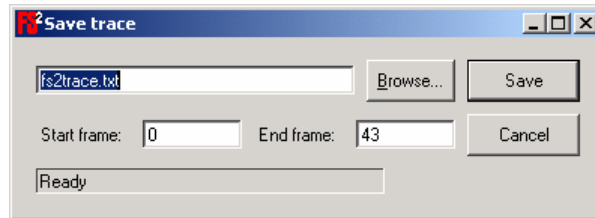
Start execution. The program will break the processor when the low order bytes of loopx are **written** with a value of 0xFFFF. Again, the processor would not have completed the write cycle until the debugger is stepped one source line.

Single step and the trace captures the write to loopx with a value of 0xFFFF on the lower two bytes:



## Saving the trace to a file

The menu entry **File > Save as** provides a dialog to save all or part of the Trace window formatted data to a file. The contents of the ASCII file will start and end at the selected frames and will reflect whatever the current display mode is. The Dialog box is shown below.



## Conclusion

MIPS Trace extends the EJTAG debug environment, providing a hardware mechanism for tracking the history of execution through a program. The combination of the SDE Toolkit, Insight debugger, and the FS2 Trace Control software provides a powerful debug addition to any programmer's toolkit.

## References

1. EJTAG Specification, MIPS Technologies, Document No. MD00047
2. PDtrace™ Interface Specification, MIPS Technologies, Document No. MD00136
3. EJTAG Trace Control Block Specification, MIPS Technologies, Document No. MD00148
4. MIPS32™ 4KE™ Processor Cores Software User's Manual, MIPS Technologies, Document No. MD00103
5. MIPS32 M4K™ Processor Core Software User's Manual, MIPS Technologies, Document No. MD00249
6. FS2-Trigger-Trace Windows for GDB/Insight for MIPS Processors User Guide, First Silicon Solutions
7. MIPS-Getting-Started, First Silicon Solutions
8. MIPS SDE 5.03 Programmer's Guide, MIPS Technologies, Document No. MD00

## Appendix: Some ideas from Bruce:

\* trace is useful to look back in the history of execution to see where the code executed at the instruction, source line, or even function level. When you break execution, you can view the call stack (e.g. with gdb/Insight) to see what the current call stack nesting is, but it doesn't give the detail of what path it took through the conditional branches. Also many calls and returns can be in a trace but a call stack only tells you the current context after a break.

With load/stores turned on, the trace can show all the memory accesses occurring in the code. With fixed address variables (static memory) you can see the reads or writes to them. Stack-based variables (passed parameters and local variables) are more difficult to figure out what the reference is to back to source. Sometimes it can be done. For pointer-based data structures (e.g. memory that was malloc'ed) you would have to relate the source code that was accessing them to the addresses to understand where it was reading or writing.

\* triggers and trace work together as debugging tools. Triggers can find actual errors in code execution and data accesses, and the trace allows you to look back to see cause-effect; i.e. what code executed that caused the failure. For example, set up a trigger on a read or write to memory address 0. Usually compilers initialize pointers to NULL (0x0) so if you trigger on a memory access to 0, this will find the C or C++ bug where the pointer wasn't initialized by user code.

You can set a trigger to catch a write to code space which should never happen. This could be a bug in, say, pointer arithmetic that causes the pointer value to be way off. Looking into the trace history will locate where the errant code is.

\* The ability to turn trace on and off allows the trace buffer to be better utilized for capturing the execution of the function or functions of interest.

\* deep traces can be post-processed to provide performance analysis by binning executed address occurrences or with timestamps, the duration in each function. The same trace can be analyzed for code coverage to help develop regression tests to cause all the code to execution (and even keep track of branches taken and not taken).

\* The MIPS trace has a mode to capture all clock information for each instruction. This can be used to uncover cache misses and determine ways of positioning code to reduce those cache misses.

--- Bruce