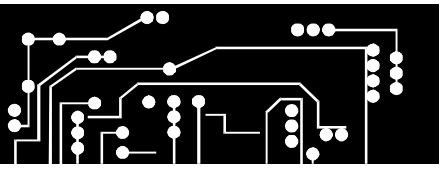
PowerPC Embedded Processors
Application Note



Real-Time Instruction Trace in the PowerPC 400 Family of Embedded Controllers

IBM Microelectronics
Dept PDXA/Bldg 060
3039 Cornwallis Road
Research Triangle Park, NC 27709
Version: 1.0

April 10, 1998

HISTORY

Software development and debug time have become the critical factors in the embedded system development cycle. But the embedded controllers of today are far different animals from those of the past. Previously, controllers ran at speeds less than 10 MHz, and contained no caches. These controllers were built by only a few companies, and enjoyed a long product life. With a long product life, manufacturers could justify the cost of producing a special "bond-out" version of a chip. This bond-out chip contained additional signals not found in the production chip, and was attached to emulators and logic analyzers to perform debug and real-time trace. But because of their high pin-out and cost of manufacturing, it was cost prohibitive to ship them in a production product, and instead they were only used for initial system development and test.

THE PROBLEM

Today's environment has changed dramatically. Now, in the 32-bit embedded RISC[1] market, controllers are running in excess of 100 MHz, and contain large caches -- 16 KB and higher. There are a large number of suppliers making many different controllers with short product lives. Some manufacturers offer system-on-a-chip (also known as Core+ASIC[2]) embedded solutions, in which customers can choose the peripherals and packaging that surround the embedded RISC processor core. Meanwhile, the size and complexity of the code executed by these controllers have increased. All of these factors have dictated that better forms of debug and emulator-like functions exist on-chip. Creating an emulator to operate at these high speeds is much more difficult. Furthermore, emulators have no access inside the chip, so it is impossible to know the code flow of a controller with an instruction cache just by analyzing the controller's external bus. Trying to produce emulator pods to hook up to all the available packages for Core+ASIC chips is unrealistic. This is especially true since a particular Core+ASIC's development and product life may be shorter than the time it takes to build a new emulator. No company is going to design a new emulator of which they will only sell one or two units to a single company over the controller's entire lifetime. As the cost of developing a Core+ASIC embedded controller decreases, there will be more of these custom controllers with real-time debugging needs and no emulator support.

Yet the need to shorten cycle time, especially software development and debug time, is enormous. A lot of work has been successfully applied to this area, through the use of Background Debug Mode (BDM) and enhanced JTAG[3] ports on existing RISC controllers. This allows developers to do hardware level debug, such as running, stopping, and stepping the processor, and interrogating processor resources -- reading and writing processor registers and memory. In some cases, these ports may also provide a window into things like the processor's caches and translation lookaside buffers (TLBs), or other special latches not normally accessible through software. This processor control and resource manipulation is accomplished through a serial interface that uses a small number of pins, with no debugger code running on the target.
But there is still the problem of real-time trace. The BDM-style debuggers mentioned typically stop processor execution to perform their debug. Although this method does not alter code flow, it dramatically changes the timing of running code, and may mask problems or make difficult timing related problems impossible to catch. Encountering a bug might require running an exact set of instruction sequences, possibly hundreds of thousands of instructions, without altering the normal code path. For these cases, it is imperative to run at full speed in a non-invasive manner.

Solution Requirements

To summarize, for the 32-bit RISC high performance embedded controller market, there exists a need for an instruction trace which has the following characteristics:

1. It must run at the full speed of the part
2. It must be non-invasive
3. It must exist on production level parts
4. It must work with the caches enabled
5. It must have a small I/O count to be cost effective

SOLUTION

Overview

One example of a solution to this problem can be found in the PowerPC* 400 family of embedded controllers from IBM. This paper will discuss the solution that is implemented in these parts, but the general concepts may be applicable to other 32-bit RISC architectures.
Several parts in the PowerPC 400 family contain a dedicated trace port used for real-time instruction tracing. Although instruction address lengths are 32-bits on these parts, the pin-out for this solution consists of only eight pins -- one for a synchronizing clock and the other seven for data. It is a real-time instruction trace, meaning that the port is used to determine the address of the current instruction at every cycle of the processor's operation. This address is also known as the instruction pointer, or IP. Since these are 32-bit RISC parts, each IP is four bytes (32 bits) wide, so an encoding and serialization is used to compress this address information, plus the current instruction execution status, down onto seven data pins.
The trace information is collected by an external tool while the processor is executing at full-speed. The data pins are broadcast at every positive transition of the core clock pin to

synchronize the information. Because the information is broadcast from inside the processor core, and not from an external processor bus, the IP information is correct even when the instruction caches are enabled. After the trace is complete, a debug tool can interpret the trace data to reconstruct the instruction code flow and then display it in a human readable form.

HOW IT WORKS

Setting up the Trigger Point

To perform an instruction trace, the first thing to do is to set a trigger point. Engineers familiar with logic analyzers and emulators are accustomed to having robust trigger capability. With these tools, users can specify a complex trigger point that must occur before a trace snapshot is taken. Unlike a simple trigger point of a particular instruction address, a complex trigger point may involve counters, logical operators, bit masking, and event sequencing. For the PowerPC 400 family controllers, there are two ways to generate a trigger point:

1. Use the hardware debug resources contained in the processor core
2. Use an external trigger source to feed into the processor core

Using Hardware Debug Resources

Most 32-bit RISC processors have registers and debug facilities that allow users to set breakpoints at a particular IP or multiple IPs, the address of one or more data addresses, when a branch or exception is taken, etc. For real-time instruction tracing, these internal processor core resources are used to determine the trace point, instead of a breakpoint.

This method has the advantage of being a precise trigger mechanism, which means that the exact point of when the trigger executes is known. In this method, all code before the trigger point is guaranteed to have been executed, and all code after the trigger point has not yet been executed.
Using only available processor resources for trigger points may sound limiting, but in reality it is not a big problem because processor cores are incorporating more and more debug resources to help the embedded developer. The IBM PowerPC 403GA, 403GB, 403GC, and 403GCX controllers all contain breakpoint/trigger points for multiple instruction address and data address values, as well as counting and sequencing mechanisms, and for when branches or exceptions occur. Future PowerPC 400 family cores will have more trigger facilities supporting logical expressions and ranges.

Using an External Trigger Source

If the internal hardware resources are not sufficient, an external trigger can be fed into the processor and is used as the trigger point. This method is an imprecise triggering mechanism, because the event has already occurred before being fed into the processor core. It is therefore likely that processor execution has continued past the trigger point (this is also known as “skid,"

since the IP skids past the desired trigger point). Fortunately, the amount of skid is usually minimal and does not hinder the usefulness of the instruction trace.

Compressing the Trace Data

The trick now is to get the necessary instruction address information onto seven data pins. Fortunately, the locality of reference associated with Von Neumann architecture machines assists in this process. The following explanation will refer to the number of finite "states" that are needed to determine the code flow, plus some dedicated pins for special address broadcasting.

Linear Code Execution

Consider the normal, linear (sequential), execution flow of a scalar 32-bit RISC architecture. Linear code flow means that after an instruction is executed at address IP, the next instruction to execute is located at IP + 4 (assuming a 32-bit instruction width). To determine instruction address flow, we need to broadcast two states during every processor core clock period -- one state to say that an instruction executed on a given clock cycle, and another state to say that no instruction executed on the given cycle. To illustrate, let us look at the following example. Consider the following trace, in which State 0 means the instruction did not finish executing on the given cycle, and State 1 indicates the instruction finished executing on that cycle:

Cycle number	State	Did instruction execute?
Cycle 1	0	instruction did not execute
Cycle 2	1	instruction did execute
Cycle 3	0	instruction did not execute
Cycle 4	0	instruction did not execute
Cycle 5	1	instruction did execute
Cycle 6	1	instruction did execute

Figure 1 - Linear Instruction Trace

Possible reasons for an instruction not executing on a cycle are: multi-cycle instructions (such as multiplies and divides), memory accesses, and pipeline stalls. As one can see from the previous table, only one data pin is needed to save the two states. Now assume, for the purposes of an example, that we already know that the beginning IP is at 0x10 when the trace is started. Because the code flow is known to be linear, the order of the addresses will always be increasing where the next IP = current IP + 4. The only remaining calculation is to determine how long each instruction took to execute. This is calculated by adding the number of non-executed cycles plus the cycle the instruction did execute. For example, at the beginning address, 0x10, the instruction took two cycles to execute, since it did not execute on cycle one but did execute on cycle two. A post-processing tool would determine that the instruction flow was the following:

Address	Cycles per instruction	Total number of cycles
0x10	2	2
0x14	3	5
0x18	1	6

Figure 2 - Linear Instruction Trace Reconstruction

Assuming that instruction memory was not modified while the processor was running and the trace was gathered, memory or a static code listing can then be read postmortem to determine the instructions at addresses 0x10, 0x14, 0x18. (It should be noted that self-modifying code is considered a practice to avoid unless absolutely necessary!)

In this six cycle example, only one bit of information per cycle needs to be saved, for a total of six bits. This is a far cry from the 32 bits per cycle needed if we were storing the IP itself on every cycle.

Another observation to note is that not only are we generating a trace that will assist in hard to catch timing related bugs, we are also getting the added benefit of a performance analysis tool. Again, post-processing tools can be used on the trace data to determine things like instructions most frequently used, instructions that took the longest time, etc.

What we have so far is a real-time instruction trace with IP information output on one external data pin, clocked at the processor core clock frequency on one clock pin. From this trace, we can determine how long each instruction takes to execute. Unfortunately, it's not a realistic picture of processor execution flow!

Handling Branches

Anyone who has debugged code before knows that normal execution flow is far from linear. Perhaps the biggest advantage of computers is the ability to analyze input choices and then *select* an output. To do this kind of decision making, the code flow must analyze input conditions and then branch to different outputs depending on the inputs. This results in non-linear code flow. Therefore, another two states must be added to determine if a branch was executed or not, so the post-mortem tool can correctly calculate the new IP from the current IP, since it may no longer be IP+4.

Here's an updated example with two new states to handle whether or not the instruction was a branch that was taken:

Cycle number	IE State	BT State	What happened?
Cycle 1	0	0	Inst. did not execute, not a taken branch
Cycle 2	1	0	Inst. did execute, not a taken branch
Cycle 3	0	0	Inst. did not execute, not a taken branch
Cycle 4	0	0	Inst. did not execute, not a taken branch
Cycle 5	1	0	Inst. did execute, not a taken branch
Cycle 6	1	0	Inst. did execute, not a taken branch
Cycle 7	1	1	Inst. did execute, taken branch
Cycle 8	1	0	Inst. did execute, not a taken branch
Cycle 9	1	1	Inst. did execute, taken branch
Cycle 10	1	0	Inst. did execute, not a taken branch

Figure 3 - Nonlinear Instruction Trace

In Figure 3, IE State refers to whether or not an instruction executed on a cycle, and BT State refers to whether or not an instruction was a branch that was taken on that cycle. From this figure, can you determine what the analyzed trace output looks like? You can't, because simply knowing if a branch was taken does not tell you what the new IP is after the result of the branch. Let us consider the most common case of PowerPC branches, in which the branch target address can be calculated from the branch instruction encoding itself. To correctly trace this type of branch, it is now required that instruction memory be unmodified . Self-modifying code will give misleading trace results. To post-process the address information, instruction memory or a code listing is read to determine the target address of the branch. Here again is the same table, but this time the branch target addresses (BTAs) are added for all branches that have been taken.

Cycle	IE State	BT State	BTA	What happened?
Cycle 1	0	0	N/A	Inst. did not execute, not a taken branch
Cycle 2	1	0	N/A	Inst. did execute, not a taken branch
Cycle 3	0	0	N/A	Inst. did not execute, not a taken branch
Cycle 4	0	0	N/A	Inst. did not execute, not a taken branch
Cycle 5	1	0	N/A	Inst. did execute, not a taken branch
Cycle 6	1	0	N/A	Inst. did execute, not a taken branch
Cycle 7	1	1	0x24	Inst. did execute, taken branch
Cycle 8	1	0	N/A	Inst. did execute, not a taken branch
Cycle 9	1	1	0x04	Inst. did execute, taken branch
Cycle 10	1	0	N/A	Inst. did execute, not a taken branch

Figure 4 - Nonlinear Instruction Trace with Branch Target Addresses

Remember, the branch target addresses (0x24 and 0x4 in this example) are NOT broadcast over the trace pins, but determined from the debug tool AFTER the trace is run by either reading instruction memory or reading a static code listing.

Now the post-processing for this trace can be shown. Keep in mind that the instructions that were executed at cycles 2, 5, 6, 8, and 10 MAY be conditional branch instructions, but if they were, the conditions to take the branch were not met.

Address	Cycles per instruction	Total number of cycles
0x10	2	2
0x14	3	5
0x18	1	6
0x1C	1	7
0x24	1	8
0x28	1	9
0x04	1	10

Figure 5 - Nonlinear Instruction Trace Reconstruction

Notice that the code flow is no longer linear; the instruction at address 0x20 has not been executed. Also note that the last instruction is at address 0x04, a lower address than the start of the trace. To total up the pin count, we are now using one pin for clocking, and two others to handle the four finite states (one of which will never occur -- executing a branch when an instruction has not been executed), for a total of three pins.

Branches are one example of non-linear code execution. But so far we have only handled one type of branch, one whose target address can be calculated simply by knowing the branch instruction. In the PowerPC architecture, there are other kinds of branch instructions in which the branch target address can only be determined by a value in a designated register. This is a special case that will be handled along with another special case of non-linear code flow -- interrupts.

Interrupts

Interrupts present a unique problem in that when an interrupt is taken, the next instruction address may jump to any one of a number of possible locations depending on the type of interrupt. These address locations are known as exception vectors. If pin bandwidth is used to create a state for every possible interrupt, the cost benefit of low pin count will be lost. Therefore, instead of tracing the type of interrupt, the address of the exception vector is broadcast. For a 32-bit RISC machine, instruction addresses are 32 bits in length. But the two least significant bits are not needed since they must always be zero, as instruction lengths are four bytes (32 bits). To broadcast the important 30 bits of address, only four pins are used -- one pin to indicate if an address is being broadcast, and three pins to broadcast the address in a serial fashion over 10 cycles.

For an example, let us assume the processor takes a PowerPC program exception when trying to execute an illegal instruction. In the PowerPC architecture, when a program exception occurs, the IP jumps to an address with an exception vector offset of 0x0700. To illustrate, we will assume the IP after the interrupt is 0x12340700. To broadcast this address, the two least significant bits are ignored, since they are always zero, and the resultant octal number is 0443200700. Assuming we broadcast the least significant bits first, an address broadcast portion of a trace will look like the following (starting at cycle n, with A0 as the most significant bit and A2 as the least significant bit):

Cycle	Addr BC	A0	A1	A2
Cycle n-1	0	N/A	N/A	N/A
Cycle n	1	0	0	0
Cycle n+1	1	0	0	0
Cycle n+2	1	1	1	1
Cycle n+3	1	0	0	0
Cycle n+4	1	0	0	0
Cycle n+5	1	0	1	0
Cycle n+6	1	0	1	1
Cycle n+7	1	1	0	0
Cycle n+8	1	1	0	0
Cycle n+9	1	0	0	0

Figure 6 - Trace of Address Broadcasting Pins

Although not shown in Figure 6, the IE State and BT state information would continue to be output in parallel with any address broadcast. The objection to be raised at this point is how ten cycles can be dedicated to broadcasting the address information without ever slowing down the processor. In situations that require address broadcasting less than ten cycles apart, how can the trace (and the controller) continue to run at full speed? The answer is that there is enough on-chip buffering of address broadcast information to be confident that for any realistic PowerPC code sequence, processor execution will not be halted.

This address broadcast mechanism is also used to handle the special branch instructions not previously considered. For example, in the PowerPC architecture, the “blr" instruction’s branch target address is the value contained in the link register.

Tallying up the pin count, the total is now up to six pins dedicated to data tracing and one clock pin. What about the last data pin?

Other States in the State Machine

For the purposes of this paper, significant detail has been omitted. For example, the 403Gx processors have a parallel branch execution unit, which allows some correctly predicted branches to be folded onto the previous cycle of execution. Also, a state is needed to indicate the trace’s starting cycle (i.e., trigger point) from within the free running trace output. The last

data pin allows for the additional states not discussed here. For qualified tool vendors who wish to support the real-time trace feature of the PowerPC 400 family controllers, these details are provided along with reconstruction code to aid in tool development.

TRACE RECONSTRUCTION EXAMPLE

RISCWatch* is an example of a debugger that provides a trace reconstruction tool called RISCTrace* for the IBM PowerPC family of embedded controllers. The following is an example of a real-time trace reconstruction from RISCWatch:

```
 # RISCTrace : Trace Output File
# DATE      : Wed Jun 04 12:25:49 1997
# TRACE TRIGGER SETTINGS   : IAC1 occurring 1 times
> TRACE TRIGGER EVENT CYCLE: 00000

#       Total Cycle/          (optional )
# Line  Cycle Instr Address   (+F_Offset) Disassembly
# ----- ----- ----- --------------------- -----------
$ FUNCTION: main  START_ADDR: 0x0000A078  FILE: demo1.c  PROGRAM: .\demo
  00001 00000       0x0000A090(+0x000018) stw       R3,0x00000038(R1)
  00002 00000                             # ** STATUS: Trigger event **
  00003 00001 00001 0x0000A094(+0x00001C) addi      R3,0,0x006F
  00004 00002 00008 0x0000A098(+0x000020) stw       R3,0x0000003C(R1)
  00005 00010 00001 0x0000A09C(+0x000024) addi      R3,0,0x0002
  00006 00011 00001 0x0000A0A0(+0x000028) stw       R3,0x00000040(R1)
                  (trace information removed for brevity)

$ FUNCTION: routine5  START_ADDR: 0x0000A1D8  FILE: demo3.c  PROGRAM: .\demo
  00058 00096 00001 0x0000A1D8(+0x000000) stwu      R1,0xFFFFFFC0(R1)
  00059 00097 00009 0x0000A1DC(+0x000004) stw       R3,0x00000058(R1)
  00060 00106 00001 0x0000A1E0(+0x000008) lwz       R4,0x00000008(R2)
  00061 00107 00001 0x0000A1E4(+0x00000C) addi      R3,0,0x0005
  00062 00108 00001 0x0000A1E8(+0x000010) stw       R3,0x00000000(R4)
  00063 00109 00001 0x0000A1EC(+0x000014) addic     R1,R1,0x0040
  00064 00109 00000 0x0000A1F0(+0x000018) blr       #  ** NOTE:  Folded
                                                       Instruction

$ FUNCTION: main  START_ADDR: 0x0000A078  FILE: demo1.c  PROGRAM: .\demo
  00065 00110 00001 0x0000A0F0(+0x000078) cror      31,31,31
```

Figure 7 - RISCTrace Output Example

We see in this actual tool output the things that were shown in the example traces -- cycles per instruction and total cycle counts. In addition, the instructions that were executed are shown, and any branches that have been folded are indicated. When the debugger has source debug information, function names and offsets are provided to relate back to the C source code.

CONCLUSION

The explosion of 32-bit RISC processors in embedded systems has driven the requirement for smarter, lower cost debug tools, including real-time instruction trace. Customers need to debug and trace production-level parts, not just expensive pinned-out cores that will not be in the final

system design. The IBM PowerPC family of embedded processors have successfully implemented a non-invasive, full-speed instruction trace using only eight pins. These pins output minimal trace information which is fed to an external debug tool to reconstruct the code flow postmortem. When this instruction trace feature is used in conjunction with a JTAG debug tool, software developers can more easily find system hardware and software errors. The result is a shorter cycle needed to bring a product to market.

© International Business Machines Corporation, 1998

All Rights Reserved

* Indicates a trademark or registered trademark of the International Business Machines Corporation.

** All other products and company names are trademarks or registered trademarks of their respective holders.

IBM and IBM logo are registered trademarks of the International Business Machines Corporation.

IBM will continue to enhance products and services as new technologies emerge. Therefore, IBM reserves the right to make changes to its products, other product information, and this publication without prior notice. Please contact your local IBM Microelectronics representative on specific standard configurations and options.

IBM assumes no responsibility or liability for any use of the information contained herein. Nothing in this document shall operate as an express or implied license or indemnity under the intellectual property rights of IBM or third parties. NO WARRANTIES OF ANY KIND, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE ARE OFFERED IN THIS DOCUMENT.

---

[1] RISC - Reduced Instruction Set Computer

[2] ASIC - Application Specific Integrated Circuit

[3] JTAG - Joint Test Action Group, as defined in the IEEE 1149.1 standard