Using the Processor Bus to Shorten Software Development

An Application Note

www.arium.com

Abstract:

Designing complex systems under severe time constraints is the rule, not the exception. With today's advanced processors from Intel, low-level software and firmware developers must either understand some basic concepts of the processor's operations or prepare themselves to take an inordinate amount of time to debug their code.

The purpose of this paper is to introduce the reader to processor cycles on the front-side bus (FSB) that can be used to shorten the development and debug cycle. A comparison between the debug functions found using a Test Access Port (TAP) and processor bus cycles will also be discussed. Real world examples that demonstrate the advantage and ease of using the bus cycles in software debug will also be discussed. Finally, timings garnered from the processor bus will be used with a spreadsheet as a profiling tool.

Introduction: The Processor Bus

The processor bus is a generally used term to describe specific signals passed between the processor and the rest of the system. Intel has split bus signals into three different types: data transfer, bus operation and bus cycle.

The *Pentium® Processor Family Developer's Manual: Volume 1* defines the parameters of a bus cycle as beginning "with the Pentium processor driving an address and status and asserting ADS#, and ends when the last BRDY# is returned. A bus cycle may have 1 or 4 data transfers."

There are differences between the bus cycles initiated by the Pentium processor and those initiated by those in the Pentium Pro processor family. Bus cycles initiated by the processor include:

Pentium processor	Pentium Pro processor family
• Special Bus	• Special Bus
• I/O R/W	• I/O R/W
• Memory R/W	• Memory R/W
• Interrupt Acknowledge	• Memory Read, invalidate line
• Fetch or Code Fetch	• Interrupt Acknowledge
	• Deferred Reply

Table 1: Processor Bus Cycles

Special bus cycles include:

Pentium processor	Pentium Pro processor family
• Shutdown	• Shutdown
• Halt	• Halt
• Flush	• Flush Acknowledge
• Branch Trace Message	• Branch Trace Message
• Writeback	• Sync
• Flush Acknowledge	• Stop Clock (Grant) Acknowledge
	• SMI Acknowledge

Table 2: Special Bus Cycles

The remainder of this paper will focus on using the Special Bus, I/O and Memory cycles as tools for software debug.

Branch Trace Messages

Although most of the bus cycles are self-explanatory, Branch Trace Messages (BTMs), require a little explanation.

Whenever a branch occurs (JMP, JC, IRET, CALL, etc.), the processor can place a BTM cycle on the bus. This special cycle, which takes very little overhead, allows a debug tool that can capture and save bus cycles to follow code execution as it occurs inside the processor or cache.

Four other operations besides a branch can cause a BTM cycle - serializing instruction, segment descriptor loads, hardware interrupts and exceptions that invoke a trap or fault handler - but for the purpose of this paper, branch-related BTMs are the most interesting.

The BTM cycle can be translated, in effect, as the processor saying "I am now here, and I came from this code address." By pairing BTM cycles, and matching the first "I am here" with the second "I came from this code address," the beginning and ending location of a code chunk is defined.

There are minor differences between the Pentium processor and Pentium Pro processor BTMs, but the above is a good working interpretation.

Debug tools such as In-Circuit Emulators (ICE) will store the BTMs into trace memory, and pair them to define a block of code. When the processor stops, the ICE will read memory between the beginning and ending code location of the pair, disassemble the memory and -voila!- you see assembly code in your bus trace. Saving a bus cycle is clearly more efficient than storing every line of code the processor executes, especially with a limited amount of storage.

Test Access Port (TAP)

From a software perspective, the Pentium processor implementation of the IEEE 1149.1 Standard Test Access Port (TAP) and Boundary Scan Architecture gives the developer "run-control" of the processor. By placing addresses in the debug registers, the processor can be stopped on I/O or memory accesses, or code execution. Once stopped, a debug tool connected to the TAP can be used to interrogate and modify I/O, memory or registers; single-step or start the processor; and set new "breakpoints" to stop the processor once it has been re-started.

There are two levels of TAP implementation. Level 1 defines a 20-pin AMP connector used to debug single processor systems. Level 2 defines a 30-pin AMP connector used to debug dual processor systems. Level 2 pins 1 through 20 are the same as the Level 1 pins.

Though beyond the scope of this paper, note that the TAP pinout definition for a Pentium processor is decidedly different than the TAP pinout definition for the Pentium Pro family of processors. See the appropriate Pentium Pro processor manuals for a full description of the implementation.

TAP versus Bus Cycles

Simply stated, the TAP does not have exposure to the bus cycles, and the bus cycles do not have direct exposure to the processor registers.

Although TAP is a useful basic development and debug tool, it won't allow an ICE to:

1. ***Stop the processor on unique data values written to (or read from) memory and I/O addresses.***

For example, the TAP can be used to set up the debug registers to stop the processor on an access of I/O port 80h. The TAP cannot be used, however, to stop the processor <u>only if</u> a 32h is written to I/O port 80h, or <u>only if</u> a 22h is read from I/O port CF8h. The reason is due to the lack of resources in the debug registers; only addresses, not specific values, can be set.

2. ***Trigger breakpoints on specific activities seen on the processor bus, like the Special Bus cycles.***

For example, the TAP cannot stop the processor on an Interrupt Acknowledge, or SHUTDOWN or HALT cycle.

3. ***Save bus transactions for later review.***

Since the TAP does not have exposure to the bus cycles, there's no way to verify what code was executed before the SHUTDOWN. For that matter, it's impossible to see what event occurred in the processor that caused the SHUTDOWN in the first place. Bus transactions, including BTMs, cannot be saved. The TAP cannot "see" code execution without interfering with the processor.

Using only the TAP means is it impossible to know what event caused a SHUTDOWN and what code executed prior to it.

Real World Examples

As you probably know, BIOS firmware is used in every PC to bootstrap the system. When the PC boots, the BIOS runs its POST (Power-On Self-Tests). The POST has two objectives: initialize the hardware and perform a limited hardware test to ensure the basic functionality of the system.

During the POST, values are written to I/O port 80h. These values, used as flags to show the code progress, are displayed on POSTCards (PCBs with seven-segment LEDs).

BIOS developers, and all software developers who port code to multiple platforms, understand the frustration of a frozen machine. The POSTCard can be used to give a rough idea of the code hang, but adding additional flags and reburning PROMs (or reflashing the BIOS) adds, rather than subtracts, to the frustration of the situation.

Low-level operating system developers do not have it much easier. While the hardware appears stable, many steps are required before OS debug tools can be used. Even after the debug tools can be used, it's impossible to guess where the OS will load a DLL.

In the next few pages, a few real world problems will be discussed. Solving problems by using the processor bus will also be discussed.

POST Codes

Oftentimes, embedded developers will temporarily use a modified PC to develop firmware until their system has been built. PC manufacturers also use this technique to develop for new motherboards that have not yet been built. Once the system is built, it is a simple matter to move the code.

The modified PC, with the new (but untested) code and peripherals, may hang. The POST codes in the modified PC may assist in determining the cause of the hang. Understanding those codes, then, may help eliminate the hang before it ever affects the new system.

Using an ICE and the TAP, it's possible to create a macro to find the POST codes. It would start the processor, run until an I/O 80h access, display the AL register, and loop until the PC boots.

One problem, though, is that the pauses caused by processor stoppage between I/O accesses may alter the bootstrap sequence. Another problem is that the pauses may slow down the bootstrap enough to hide any timing related problems seen only by a speedy bootstrap. This "slowdown masks problem" is often seen when troubleshooting communications code.

An ICE with bus trace memory can be easily configured to capture only writes to I/O 80h without affecting the processor, as seen in Screen 1. Also, by filtering the bus cycle collection (a process known as "data qualification") to include only writes, not reads, irrelevant data is eliminated. Finally, the timing information can aid in bootstrap profiling.

```
Trace                                                                    _ □ ×
STATE STS    ADDR     DATA     DATA     BE EXT STS1 STS2 STS3    TIMESTAMP   ▲
-00023 I/O WR 00000080 -------- ------0C FE FFFF 2DBF 373  027    +1.334 sec
-00022 I/O WR 00000080 -------- ------0D FE FFFF 2DBF 373  027    +9.230 us
-00021 I/O WR 00000080 -------- ┤-----BF FE FFFF 2DBF 373  027   +58.390 us
-00020 I/O WR 00000080 -------- ------0D FE FFFF 2DBF 373  027  +231.850 us
-00019 I/O WR 00000080 -------- ------0E FE FFFF 2DBF 373  027    +2.126 sec
-00018 I/O WR 00000080 -------- ------0F FE FFFF 2DBF 373  027  +533.083 ms
-00017 I/O WR 00000080 -------- ------10 FE FFFF 2DBF 373  027   +11.160 us
-00016 I/O WR 00000080 -------- ------11 FE FFFF 2DBF 373  027    +5.620 us
-00015 I/O WR 00000080 -------- ------12 FE FFFF 2DBF 373  027    +5.870 us
-00014 I/O WR 00000080 -------- ------13 FE FFFF 2DBF 373  027    +5.700 us
-00013 I/O WR 00000080 -------- ------14 FE FFFF 2DBF 373  027    +5.860 us
-00012 I/O WR 00000080 -------- ------15 FE FFFF 2DBF 373  027   +35.220 us
-00011 I/O WR 00000080 -------- ------16 FE FFFF 2DBF 373  027    +1.779 ms
-00010 I/O WR 00000080 -------- ------17 FE FFFF 2DBF 373  027    +1.777 ms
-00009 I/O WR 00000080 -------- ------18 FE FFFF 2DBF 373  027    +5.860 us
-00008 I/O WR 00000080 -------- ------19 FE FFFF 2DBF 373  027   +46.220 us
-00007 I/O WR 00000080 -------- ------1A FE FFFF 2DBF 373  027    +5.740 us
-00006 I/O WR 00000080 -------- ------1B FE FFFF 2DBF 373  027    +5.710 us
-00005 I/O WR 00000080 -------- ------1C FE FFFF 2DBF 373  027    +4.770 us
-00004 I/O WR 00000080 -------- ------1D FE FFFF 2DBF 373  027    +4.740 us
-00003 I/O WR 00000080 -------- ------1E FE FFFF 2DBF 373  027    +4.740 us
-00002 I/O WR 00000080 -------- ------30 FE FFFF 2DBF 373  027    +4.900 us
-00001 I/O WR 00000080 -------- ------30 FE FFFF 2DBF 373  027    +5.860 us
  TRIG I/O WR 00000080 -------- ------31 FE FFFF 2DBF 373  027    +2.740 ms  ▼

[-00021]  [Mixed    ▼]  [Display Settings...] [Zero Timestamp] [STS Bit Help]
```

Screen 1: POST Code Collection

After I/O writes to 80h are collected, it is easy enough to determine the POST that caused the hang. Most PC manufacturers have detailed explanations of POST codes in their reference manuals; the technical support department of all major PC manufacturers can help if the POST codes are not documented.

If the POST that caused the PC hang transferred control to embedded firmware on your peripheral board, other techniques and tools may be employed to find the root cause of the problem.

If the POST that caused the PC to hang was in hardware initialization or verification, you may be looking at a hardware conflict problem.

It is interesting to note that in Screen 1, the processor - and subsequently, bus cycle collection - was stopped when a value of 31h was written to I/O 80h. This is another advantage of the processor bus over the TAP. The debug registers do not allow processor stoppage on a specific data value.

Debug is an important, but time-consuming, task. Imagine how much easier and faster it would be letting the code run up to the problem POST (like 31h) over stopping and starting the processor until finally getting to your final destination.

Other Buses

Though the focus of this document is upon the processor bus, it would be foolhardy to ignore other buses in the system. The PCI bus is a good example of another system bus that can cause problems, particularly when interacting with the system bus.

In particular, certain values are written by code executing in the processor to the PCI registers via I/O ports, including CF8h. Tracing information written to, or read from, these I/O ports is no more difficult than in the previous example.

Consider that during a bootstrap sequence, thousands of values are being written to CF8h - and just one out of order value may cause the system to hang.

In Screen 1, I/O 80h writes were collected; it would take minimal effort to change "80h" to "CF8h". Further, upon a hang, viewing the trace of the I/O writes will point to the last value written to CF8h prior to the hang. Interpretation of the value, in consideration of other system indicators, may be of great benefit.

Using I/O 80h To Find A DLL

When troubleshooting low-level software, especially DLLs, the first step <u>must</u> <u>be</u> to find the location of the suspect code. Since the load location is at the whim of the operating system, this first step may be more difficult than at first glance. Borrowing from the tools of the BIOS developer, though, we can quickly find the code.

As we have seen, it is a simple matter to stop the processor when a specific value is written to a known memory or I/O address. The value is sent to its destination via the processor bus, and we can stop the processor if a known value is going to a known location.

It is a fact that I/O 80h is allocated for POST codes. It is also a fact that all the POST codes for a PC can be found using Qualifiers. Upon observation of Screen 1, it is seen that an **AAh** is not a valid POST code.
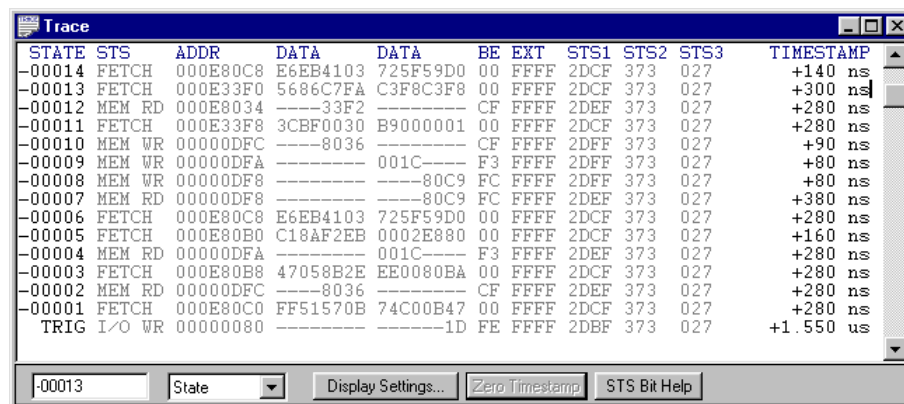
If the first line of the DLL were to write an AAh to I/O 80h, and a trigger were set in the ICE to stop the processor on an I/O 80h write of AAh, the DLL can quickly be found.

Why can't the TAP be used? Certainly it's possible to stop the processor just before the operating system is loaded, and set a breakpoint for an I/O 80h access. However, some operating systems are known to treat the debug registers as scratch registers. The result is that the breakpoint is overwritten by the operating system and will never stop the processor.

Triggers set in an ICE to stop the processor, however, cannot be influenced by software running on the target system.

More on Qualifiers

It may be advantageous to collect bus cycles between two specific events. Screen 2 demonstrates the capture of all bus transactions between an I/O 80h write of 1Ch and an I/O 80h write of 1Dh.

```
Trace                                                              _ □ ×
STATE  STS     ADDR      DATA      DATA     BE EXT  STS1 STS2 STS3    TIMESTAMP  ▲
-00014 FETCH   000E80C8  E6EB4103  725F59D0 00 FFFF  2DCF 373  027      +140 ns
-00013 FETCH   000E33F0  5686C7FA  C3F8C3F8 00 FFFF  2DCF 373  027      +300 ns
-00012 MEM RD  000E8034  ----33F2  -------- CF FFFF  2DEF 373  027      +280 ns
-00011 FETCH   000E33F8  3CBF0030  B9000001 00 FFFF  2DCF 373  027      +280 ns
-00010 MEM WR  00000DFC  ----8036  -------- CF FFFF  2DFF 373  027       +90 ns
-00009 MEM WR  00000DFA  --------  001C---- F3 FFFF  2DFF 373  027       +80 ns
-00008 MEM WR  00000DF8  --------  ----80C9 FC FFFF  2DFF 373  027       +80 ns
-00007 MEM RD  00000DF8  --------  ----80C9 FC FFFF  2DEF 373  027      +380 ns
-00006 FETCH   000E80B0  E6EB4103  725F59D0 00 FFFF  2DCF 373  027      +280 ns
-00005 FETCH   000E80B0  C18AF2EB  0002E880 00 FFFF  2DCF 373  027      +160 ns
-00004 MEM RD  00000DFA  --------  001C---- F3 FFFF  2DEF 373  027      +280 ns
-00003 FETCH   000E80B8  47058B2E  EE0080BA 00 FFFF  2DCF 373  027      +280 ns
-00002 MEM RD  00000DFC  ----8036  -------- CF FFFF  2DEF 373  027      +280 ns
-00001 FETCH   000E80C0  FF51570B  74C00B47 00 FFFF  2DCF 373  027      +280 ns
  TRIG I/O WR  00000080  --------  ------1D FE FFFF  2DBF 373  027     +1.550 us
                                                                            ▼
[-00013]   [State    ▼]   [Display Settings...] [Zero Timestamp] [STS Bit Help]
```

Screen 2: Recording Between Two Parameters

There are typically two methods that are used to trace the bus cycles. The first method is to run until a breakpoint previously set on the first event. After the processor has stopped at the first event, run until a previously set breakpoint is triggered by the second event. Only the bus cycles between the two events will be collected in trace memory.

The second method is to use "qualifiers." Similar to triggers that stop the processor on events, qualifiers are indicators to the ICE to activate collection of bus cycles.

Suppose you notice that your target system hangs after a 21h is written to the PCI register at CF8h, but it never happens after a 54h is written. One hypothesis is that the 21h seems to set up the system for failure, but the 54h appears to clear the set up.

Collecting all bus cycle information is overkill. Needed is a way of turn **on** the bus cycle trace at a write of a 21h to CF8h, and turn **off** the trace at a write of 54h.

Selecting the I/O CF8h write of a 21h as qualifier #1, or Q1, and the I/O CF8h write of a 54h as qualifier #2, or Q2, is the answer. At that point, the ICE can be configured to begin collection upon seeing a Q1 on the processor bus and end collection upon seeing a Q2 on the processor bus.

This method allows the processor to run at full speed. No pauses will occur because it is not requested that the processor be stopped, only that the ICE begin collecting data on one bus cycle and end the collection on another.

There are two additional items of interest. First, by configuring the ICE to collect only BTMs, an enormous amount of information can be collected. Recall that only a pair of BTM cycles can determine a large amount of code. If we conservatively estimate that 5 assembly instructions occur on average before a jump generates another BTM, an ICE with 64K of memory can trace **over 325,000 lines of disassembled code on a Pentium processor, and 650,000 lines on a Pentium Pro processor**!

Second, it is more than possible that the sequence of Q1 followed by Q2 occurs multiple times. Using qualifiers in this manner captures every occurrence of the sequence, and all bus cycles between the qualifiers, much more rapidly than with manual intervention.

Mailboxes and Semaphores

Mailboxes and semaphores are used for data transfer and locks, respectively. Complex programs interacting with other complex programs, and the operating system, make it difficult to find the unauthorized or unexpected code that modifies those predefined memory locations.

```
Trace                                                                    _ □ ✕
STATE STS     ADDR      DATA      DATA      BE EXT  STS1 STS2 STS3   TIMESTAMP
-00013 FETCH  00075040  47048864  058A651D  00 FFFF 2DCF 373  027      +160 ns
-00012 MEM WR 00075040  47048864  058A651D  00 FFFF 2DFF 370  025       +80 ns
-00011 MEM WR 00075048  74003D83  65F6E246  00 FFFF 2DFF 378  025       +20 ns
-00010 MEM WR 00075050  47148864  158A650A  00 FFFF 2DFF 378  025       +20 ns
-00009 MEM WR 00075058  8964158B  65F0EB46  00 FFFF 2DFF 378  025       +20 ns
-00008 FETCH  00075040  47048864  058A651D  00 FFFF 2DCF 373  027      +660 ns
-00007 FETCH  00075048  74003D83  65F6E246  00 FFFF 2DCF 373  027      +140 ns
-00006 MEM RD 00090000  --------  ------A5  FE FFFF 2DEF 373  027      +290 ns
-00005 FETCH  00075050  47148864  158A650A  00 FFFF 2DCF 373  027      +280 ns
-00004 MEM RD 00090000  --------  ------A5  FE FFFF 2DEF 373  027      +280 ns
-00003 FETCH  00075058  8964158B  65F0EB46  00 FFFF 2DCF 373  027      +280 ns
-00002 SPBUS  00075045  10000000  00000000  DF FFFF 2D9F 373  027       +80 ns
              00075041  658A05              MOV       AL,byte ptr GS:[DI]
-00001 SPBUS  00075045  98000000  00000000  DF FFFF 2D9F 373  027       +80 ns
  TRIG MEM WR 00000000  --------  ------A5  FE FFFF 2DFF 373  027       +80 ns

 +00002     Mixed    ▼     Display Settings...   Zero Timestamp   STS Bit Help
```

Screen 3: Stopping On An Address Write of A5h

As we've seen in earlier examples, stopping the processor on specific bus cycles can make all the difference between working over the weekend and finding the culprit in a few minutes.

In Screen 3, a trigger was set to stop the processor when A5h was written to address 0. By scrolling up a few lines, the code stopping the processor can be seen. From there, it is a simple matter to review the trace memory for the program flow that caused the problem, single-step the program to watch the progress of the code, or run the processor at full speed and let it stop at the next write of A5h.

Troubleshooting Interrupts

It is relatively easy to stop execution of Interrupt code by stopping the processor when the code writes a known value to I/O port 80h. Another method is by setting a breakpoint on code execution at a specific address - if you know where the code is located, and the interrupt vector isn't taken over by other code.

A better method, and one that can be employed with protected mode programs, is to set a breakpoint of a memory read at the location of the real mode Interrupt Service Vector (ISV) or the protected mode Interrupt Descriptor Table (IDT).

Without getting into too much detail, when an interrupt occurs in real mode, the processor:

- saves the location of the currently executing code on the stack,

- determines the interrupt number,

- multiples the number by 4,

- uses the value as an offset from the 0th memory location,

- loads and uses the 4 bytes at the offset as the location of the new code it should execute.

The location of the code that handles Interrupt 4, then, can be found from bytes 16 through 19 (decimal) in memory. (Interrupt 4*4 bytes) If you were interested in stopping the processor any time Interrupt 4 occurred, set a breakpoint on a memory read at address 16. Whenever the processor was interrupted by an Interrupt 4, it would be stopped as soon as it read the ISV for the interrupt.

In Screen 4, an Interrupt Acknowledge bus cycle (IACK) was used as part of a "multi-trigger event." A multi-trigger event is the equivalent of an "**if…then**" code statement. In Screen 4, any I/O write to address EBh was set as trigger 1 (T1), and any IACK was set as trigger 2 (T2).

The ICE was then configured to stop the processor **if** an occurrence of T1 was seen, **then** an occurrence of T2 was seen. The amount of time transpiring between T1 and T2 is irrelevant; any T1 followed by T2 will stop the processor.

```
Trace                                                                    _ □ ×
STATE STS      ADDR      DATA      DATA     BE EXT  STS1 STS2 STS3    TIMESTAMP  ▲
-00012 SPBUS   000FE855 50000000 00000000 DF FFFF  2D9F 373  027       +80 ns
               000FE855 FB                   STI
               000FE856 E6EB                  OUT        EB,AL
               000FE858 FA                   CLI
               000FE859 8B04                  MOV        AX,word ptr [SI]
               000FE85B 3B4402                CMP        AX,word ptr [SI+02]
               000FE85E 74F5                  JE         000FE855L
-00011 SPBUS   000FE855 B8000000 00000000 DF FFFF  2D9F 373  027       +80 ns
-00010 I/O WR  000000EB --------  1E------ F7 FFFF  2DBF 373  027     +1.810 us
-00009 FETCH   000FE858 F5740244 3B048BFA 00 FFFF  2DCF 373  027      +140 ns
-00008 MEM RD  0000041A -------- 001E----- F3 FFFF  2DEF-373  027      +380 ns
-00007 MEM RD  0000041C ----001E -------- CF FFFF  2DEF 373  027      +160 ns
-00006 FETCH   000FE850 EBE6FB15 CD9002B8 00 FFFF  2DCF 373  027      +290 ns
-00005 SPBUS   000FE855 50000000 00000000 DF FFFF  2D9F 373  027       +80 ns
               000FE855 FB                   STI
               000FE856 E6EB                  OUT        EB,AL
               000FE858 FA                   CLI
-00004 SPBUS   000FE855 B8000000 00000000 DF FFFF  2D9F 373  027       +80 ns
-00003 I/O WR  000000EB -------- 1E------- F7 FFFF  2DBF 373  027     +1.800 us
-00002 FETCH   000FE858 F5740244 3B048BFA 00 FFFF  2DCF 373  027      +220 ns
-00001 IACK    00000004 ------44 -------- EF FFFF  2D0F 373  027      +200 ns
 TRIG  IACK    00000000 -------- ------08 FE FFFF  2D0F 373  027      +550 ns

 -00003          Mixed    ▼     Display Settings...  Zero Timestamp  STS Bit Help
```

Screen 4: I/O Write to EB followed by any IACK

Profiling

Information gathered in trace memory by an ICE should include the "delta" timing, or difference in time between bus events. In Screen 4, the bus cycle trace of the first 10 POST tests was saved to disk and imported into a spreadsheet.
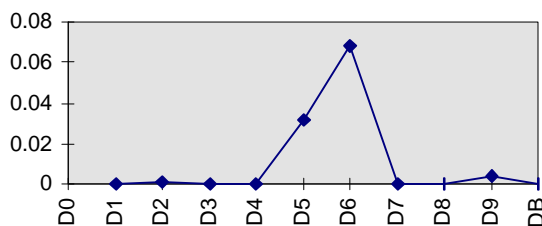
First 10 POSTs Timings

Figure 1: POST Profiling

It is easily seen that POST D6 took an inordinate amount of time to execute, and is a prime candidate for tuning.

If you review the trace of any of the previous screens, you may note that the delta timings contain a mixture of units, including seconds, ms, ns and us. To create a useful graph, each of the timings must be converted to use the same unit measure. When the data was imported into the spreadsheet, times were separated from units and a separate row was created with the following algorithm:

$$\text{"=IF(G24 = "sec",F24,IF(G24 = "ms",F24/10^3,IF(G24 = "us",F24/10^6,F24/10^9)))"}$$

This algorithm effectively converts all times into seconds by raising the time to the appropriate power of 10 based on the unit measurement.

Final Concept

The ICE may be able to start or stop trace collection, or stop code execution, when specific processor pins are active or inactive. The dialog box pictured in Screen 5 shows pin states for a data write. By gaining a better understanding of the processor pins, and the capabilities of your other tools, you'll be able to fully utilize the processor and bus cycles so as to minimize your time debugging your code.
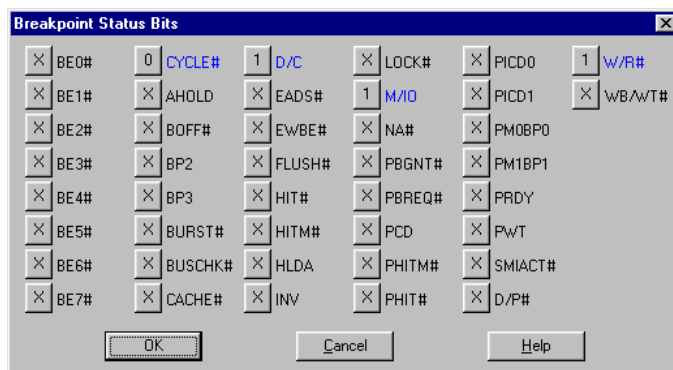
Screen 5: Pentium Processor Pins on a Data Write

Conclusion

In today's complex design environment, only two options are available: use better tools or make no mistakes. Though the latter is attractive, it is also highly impractical. By leveraging the capabilities of the processor through the use of better tools, the inevitable debug portion of the development cycle can be dramatically shortened.

14281 Chambers Road
Tustin, CA 92780
Voice: 714-731-1661
Fax: 714-731-6344
Web: www.arium.com
E-mail: info@arium.com