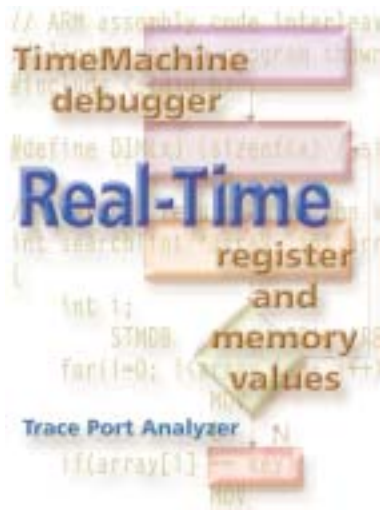


E/E++ Users Journaltm

Advanced Solutions for C/C++ Programmers



Michael Lindahl

Analyzing Real-Time Systems with Hardware Trace

Tracking instructions the microprocessor executes

One of the hardest problems facing embedded developers today is the limited visibility into the workings of their software. This is especially apparent when working on systems with strict real-time constraints because standard debugging techniques generally rely on altering the execution behavior of the software being debugged. For instance, with traditional debuggers, you typically halt the microprocessor to inspect the code paths taken and the state of the system. Alternatively, you may instrument your code to log values to help identify bugs in the system. Both of these debugging techniques cause problems when successful execution of your program relies on specific timing characteristics of your system. Common classes of bugs that these traditional techniques fall short in debugging include race conditions and random glitches. We've all encountered bugs that disappeared when you attached a debugger or added instrumentation because of the minor

changes in system timing that occur. Additionally, systems with strict real-time requirements may have critical sections of code that cannot safely be instrumented or debugged without causing serious errors in the system. However, a technology that has been available to hardware engineers for years is helping software developers solve these problems more quickly and reliably without altering the runtime characteristics of embedded systems.

This debugging technology, called "hardware trace," offers essentially a history of the instructions that a microprocessor executes, sometimes including information about what data values are read and written, when context switches occur, and other information about the state of a microprocessor as it runs. A growing number of embedded processors are implementing hardware trace, which means that the benefits of trace data are becoming available to more and more embedded developers. However, the tools to analyze trace data have traditionally been limited for ordinary software engineers because they focus on the raw trace data—the instructions that the

processor executed—rather than translating this information into the higher level concepts that software developers use to construct software. This has led to hardware trace being used primarily by hardware and very low-level firmware engineers who are used to dealing with assembly code on a regular basis.

Fortunately for embedded software developers, this situation is changing as the tools that analyze trace data improve and give software engineers improved visibility into the workings of their systems. One example of this is the TimeMachine suite of tools available from Green Hills Software (the company I work for), which lets users browse through the information collected from a trace port at a high level. One of the principal components of these tools is the TimeMachine debugger, which lets software engineers debug their traced program as it runs on the system. This debugger even lets you stop and run backwards to help easily track down where a program went astray or where variables were assigned invalid values. The debugger infers register and memory values from the trace data to give a nearly

Michael Lindahl is a software engineer at Green Hills Software. He can be reached at mlindahl@ghs.com.

complete view of the system as it was when the code was actually running. This enables difficult bugs to be captured with hardware trace and analyzed later without interfering with the real-time performance characteristics of the system.

In this article, I explore the information that trace provides as well as some of the underlying techniques that make the TimeMachine debugger possible. These methods often involve simulating the instructions captured with trace, but there are also more advanced techniques that will be discussed. Trace is available on many CPU architectures; however, this article focuses on the ARM instruction set and the techniques that are specific to these instructions.

Introduction to Hardware Trace

Microprocessors that implement hardware trace generally offer a dedicated port that outputs information about the instructions that are executing on every cycle. This information is highly compressed, but it still streams off of the microprocessor at very high speeds. For example, the trace port on an ARM microprocessor outputs between 8 and 20 bits of data per clock cycle. Thus an ARM CPU running at 200 MHz outputs trace data at up to 500 MB/s.

Trace data from a processor is collected by specialized hardware called a “Trace Port Analyzer” (TPA). These devices are available from several vendors, including Agilent and Green Hills Software. The compressed trace data is decompressed on a host system and is then analyzed to determine the internal state of the microprocessor as the program ran. However, the trace data from a CPU generally only includes information about the instructions that were executed by a microprocessor and potentially the memory addresses and values that were read and written. The challenge is then to infer additional information from this limited trace stream so that registers

Listing 1

```
// C source code for a simple linear search program
#include <stdio.h>

#define DIM(x) (sizeof(x) / sizeof(x[0]))

/* search() returns 1 if the key is found and 0 if it is not found. */
int search(int *array, int array_len, int key)
{
    int i;
    for(i=0; i<array_len; i++) {
        if(array[i] == key)
            return 1;
    }
    return 0;
}

int main(int argc, char *argv[])
{
    int found = 0;
    int fib_seq[] = { 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 };

    if( search( fib_seq, DIM(fib_seq), 3 ) )
        found++;
    printf("Number found: %d\n", found);
    return 0;
}
```

Listing 2

```
// ARM assembly code interleaved with the C source for the simple
// linear search program shown in Listing 1
#include <stdio.h>

#define DIM(x) (sizeof(x) / sizeof(x[0]))

/* search() returns 1 if the key is found and 0 if it is not found. */
int search(int *array, int array_len, int key)
{
    int i;
    STMDB    SP!, {R8,R9}
    for(i=0; i<array_len; i++) {
        MOV    R3 <i>, 0
        B     0x20(search+0x30 (0xd8))
        if(array[i] == key)
            MOV    IP, R3 <i>
            MOV    IP, IP, LSL 2
            ADD    IP, IP, R0 <array>
            LDR    IP,[IP]
            CMP    IP, R2 <key>
            BNE   0x4(search+0x2c (0xd4))
            return 1;
            MOV    R0 <array>, 1
            B     0xc(search+0x3c (0xe4))
            ADD    R3 <i>, R3 <i>, 1
            CMP    R3 <i>, R1 <array_len>
            BLT   0xfffffd0(search+0xc (0xb4))
        }
    return 0;
    MOV    R0, 0
}
LDMIA    SP!, {R8,R9}
MOV    PC, LR

int main(int argc, char *argv[])
{
    STMDB    SP!, {R4,R5,R8,R9,LR}
    SUB    SP, SP, 40
    MOV    R2, R0

    int found = 0;
    MOV    R4 <found>, 0
    int fib_seq[] = { 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 };
    MOV    IP, SP
    LDR    R5,[PC,92](0x164)
    LDMIA    R5!, {R2,R3}
    STMIA    IP!, {R2,R3}
    LDMIA    R5!, {R2,R3}
    STMIA    IP!, {R2,R3}
    LDMIA    R5!, {R2,R3}
    STMIA    IP!, {R2,R3}
    LDMIA    R5!, {R2,R3}
    STMIA    IP!, {R2,R3}
    LDMIA    R5, {R2,R3}
    STMIA    IP, {R2,R3}
    if( search( fib_seq, DIM(fib_seq), 3 ) )
        MOV    R0, SP
        MOV    R2, 3
        MOV    R1, 10
        BL    0xfffff68(search (0xa8))
        MOV    R1, R0
        CMP    R1, 0
        BEQ   0x0(main+0x60 (0x14c))
    found++;
    ADD    R4 <found>, R4 <found>, 1
    printf("Number found: %d\n", found);
    MOV    R1, R4 <found>
    LDR    R0,[PC,16](0x168)
    BL    0x24(printf (0x180))
    return 0;
    MOV    R0, 0
}
ADD    SP, SP, 40
LDMIA    SP!, {R4,R5,R8,R9,PC}
```

and other processor states can be reconstructed.

An ARM processor implements the ARM instruction set, which is a RISC, load-store architecture. The ARM CPU defines 15 general-purpose registers in addition to the PC. Some of these registers are reserved as the stack pointer, link register, and other special uses, but this is only by convention and is not enforced in the instruction set. In our example, we'll attempt to reconstruct the values of as many of these general-

purpose registers as possible from a sample trace.

Let's examine the sample trace for a simple example program compiled for the ARM processor. The source code for this simple linear search program is in Listing 1. Listing 2 is the same source code with the ARM assembly code interleaved. I will work through this example in this article, showing how you can reconstruct the state of almost all of the registers from a program trace including execution and memory history. The example trace that I use consists of a trace run from the beginning of main through the first call of search() (see Listing 3).

was executed. Figure 1 is a flowchart of this algorithm. Different instructions allow for different information to be inferred. I'll examine some of the implications from the different classes of instructions to illustrate how this information is generated.

Loads and Stores

The ARM instruction set contains several load and store instructions. The first instruction in Listing 3 (line 0) is an STMDB instruction, which stores the values from several registers to memory. In this case, the load instruction stores general-purpose registers R4, R5, R8, R9, and the link register to memory, shown in Listing 3. You can look at the values stored to memory and determine the value of each of these registers. For instance, since the first memory value stored is 0x0, you know that the register R4 contains the value 0. Also, since you are storing these values into memory, you say that these values are "valid backward" because the values that you determined must have been the values prior to the execution of this instruction.

The same analysis can be done for load instructions, such as the LDMIA instruction, which loads multiple values from contiguous memory locations into multiple registers, found at line #6 of the example trace run. In this specific case, the LDMIA instruction loads from memory into registers R2 and R3. The value from the trace buffer for these registers is 0x1 for both, so you know that after instruction #6, these registers have the value 0x1. Because you are loading into these registers, you cannot learn the value of R2 and R3 prior to this instruction. In

Register Reconstruction Techniques

After collecting a trace stream from an ARM processor, which includes instructions and associated memory accesses, analysis techniques allow for additional information that is not in the trace stream to be inferred. This information usually includes register and memory values over time so that source-level variables can be known. The process of reconstructing register values starts at the beginning of the trace buffer with all values marked as unknown. Then, each instruction that was executed is examined and any additional information that can be determined is incorporated into the reconstructed register values. Some instructions will increase the number of register values known after they execute and others may actually decrease the number of known register values. Additionally, some instructions may even discover register values that were valid before that instruction

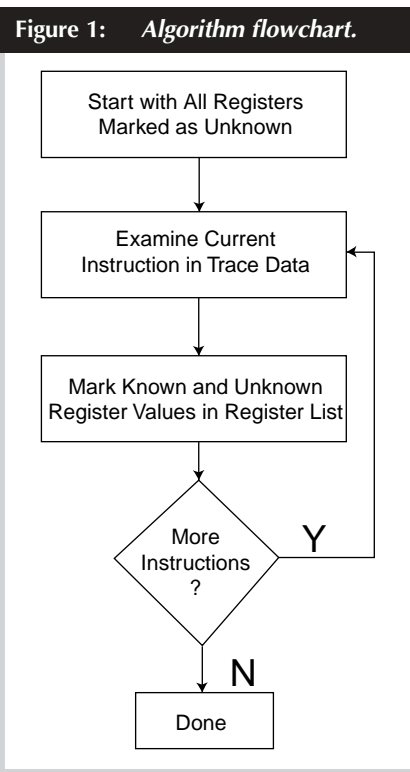


Figure 2: The register values after Line 0 in Listing 3. Unknown register values are marked with "???".

```

R0 = ???
R1 = ???
R2 = ???
R3 = ???
R4 = 0x0
R5 = 0x0
R6 = ???
R7 = ???
R8 = 0x182310
R9 = 0x182308
R10 = ???
FP = ???
IP = ???
SP = 0x182264
LR = ???
PC = 0xf0
    
```

Figure 3: The register values before the call to search() at line 19 in Listing 3.

```

R0 = 0x18223c
R1 = 0xa
R2 = 0x3
R3 = 0x37
R4 = 0x0
R5 = 0x1c30
R6 = ???
R7 = ???
R8 = 0x182310
R9 = 0x182308
R10 = ???
FP = ???
IP = 0x18225c
SP = 0x18223c
LR = 0x13c
PC = 0xa8
    
```

Figure 4: The register values after the call to search() at line 61 of Listing 3.

```

R0 = 0x1
R1 = 0xa
R2 = 0x3
R3 = 0x3
R4 = 0x0
R5 = 0x1c30
R6 = ???
R7 = ???
R8 = 0x182310
R9 = 0x182308
R10 = ???
FP = ???
IP = 0x3
SP = 0x18223c
LR = 0x13c
PC = 0x13c
    
```

Listing 3

```
// A portion of the instruction and data trace for the program shown in Listings 1 and 2. This information is all that is required
// to infer register and memory values for the TimeMachine Debugger
```

Line	Address		Instruction	Data Address	Access	Value
0	0x000000ec	main	STMDB SP!, {R4,R5,R8,R9,LR}	0x00182264 (.stack)	W	0x00000000
				0x00182268 (.stack)	W	0x00000000
				0x0018226c (.stack)	W	0x00182310
				0x00182270 (.stack)	W	0x00182308
				0x00182274 (.stack)	W	0x00001b1c
1	0x000000f0	main+0x4	SUB SP, SP, 40	-	-	-
2	0x000000f4	main+0x8	MOV R2, R0	-	-	-
3	0x000000f8	main+0xc	MOV R4 <found>, 0	-	-	-
4	0x000000fc	main+0x10	MOV IP, SP	-	-	-
5	0x00000100	main+0x14	LDR R5,[PC,92](0x1	0x00000164 (.text)	R	0x00001c10
6	0x00000104	main+0x18	LDMIA R5!, {R2,R3}	0x00001c10 (.rodata)	R	0x00000001
				0x00001c14 (.rodata)	R	0x00000001
7	0x00000108	main+0x1c	STMIA IP!, {R2,R3}	0x0018223c (.stack)	W	0x00000001
				0x00182240 (.stack)	W	0x00000001
8	0x0000010c	main+0x20	LDMIA R5!, {R2,R3}	0x00001c18 (.rodata)	R	0x00000002
				0x00001c1c (.rodata)	R	0x00000003
9	0x00000110	main+0x24	STMIA IP!, {R2,R3}	0x00182244 (.stack)	W	0x00000002
				0x00182248 (.stack)	W	0x00000003
10	0x00000114	main+0x28	LDMIA R5!, {R2,R3}	0x00001c20 (.rodata)	R	0x00000005
				0x00001c24 (.rodata)	R	0x00000008
11	0x00000118	main+0x2c	STMIA IP!, {R2,R3}	0x0018224c (.stack)	W	0x00000005
				0x00182250 (.stack)	W	0x00000008
12	0x0000011c	main+0x30	LDMIA R5!, {R2,R3}	0x00001c28 (.rodata)	R	0x0000000d
				0x00001c2c (.rodata)	R	0x00000015
13	0x00000120	main+0x34	STMIA IP!, {R2,R3}	0x00182254 (.stack)	W	0x0000000d
				0x00182258 (.stack)	W	0x00000015

general, since load instructions modify register values, no register values learned from data loads are valid backward.

The other register value that can be learned from load and store instructions is the base register of the memory access. The load and store multiple instructions in the ARM instruction set use a register to load from or store to. There are several different addressing modes for these instructions, which determine how the processor interprets the base register. The details of this process are complicated and specific details on the ARM instructions can be found in the documentation for the ARM core [1]. In this case, the base register is known to have the value 0x182264, as this is the value of the first memory access. Thus, after the first instruction, you know the value of the stack pointer because the stack pointer acts as the base register for this instruction. Figure 2 shows the state of the registers after doing these analyses on the first instruction in Listing 3.

In addition to learning register values from load and store instructions, you also propagate memory values forward and backward so that known memory reads required by the debugger can be satisfied. For instance, you know the values at the addresses written by store instructions and read by load instructions. However, the instructions that cause memory values to be valid backward are reversed from register values. Since store instructions modify the memory location, the value at that memory location cannot be valid backward. Load instructions are just the opposite because they simply monitor

a memory location without modifying it. Thus, memory values learned from a load instruction are valid backward.

There are also other load and store instructions in the ARM instruction set. However, I leave register and memory reconstruction for these instructions as exercises because the process is the same even though the details differ.

Arithmetic Instructions

Another class of instructions that can be analyzed to determine additional processor state includes arithmetic instructions. The first example of this in our example trace listing is the subtract at line 1:

```
SUB SP, SP, 40
```

This instruction subtracts the decimal number 40 from the stack-pointer register and stores the result back to the SP register. Since you learned the value of the stack-pointer register on line 0, you know that its value prior to line 1 is 0x182264. Thus, after this instruction, you know that its value is $0x182264 - 40 = 0x182264 - 0x28 = 0x18223c$.

Another simple arithmetic instruction in Listing 3 is the MOV instruction, which moves one register into another. For instance, at line 2 you move R0 into R2. However, you do not know the value of R0 at this point. Thus, you must assign the value “Unknown” to register R2. In this example, you did not know R2 prior to this instruction, but even if you had, after this instruction you cannot know the value of R2. This is an

example of an instruction that causes you to know fewer register values after it executes than you did prior to its execution.

The MOV instruction at line 3 of Listing 3 moves the value 0 into register R4. After this instruction, you obviously know that R4 contains 0. Finally, another instance of the MOV instruction is found at line 4, where the program moves the SP register into the IP register. Because you know the value of the SP register going into this instruction, you can set the IP register to this value after line 4.

The ARM instruction set contains many more arithmetic instructions, and the analysis techniques for these instructions is similar to those outlined here. These will also be left as exercises. The main caveat with these instructions is how to handle them if an operand register value is unknown. For instance, if you have something like:

```
ADD    R3, R4, R5    ; R3 = R4 + R5
```

and either R4 or R5 is unknown, then after this instruction, R3 must be marked as unknown.

Listing 3: *continued*

14	0x00000124	main+0x38	LDMIA	R5, {R2,R3}	0x00001c30 (.rodata) 0x00001c34 (.rodata)	R	0x00000022 0x00000037
15	0x00000128	main+0x3c	STMIA	IP, {R2,R3}	0x0018225c (.stack) 0x00182260 (.stack)	W	0x00000022 0x00000037
16	0x0000012c	main+0x40	MOV	R0, SP	-	-	-
17	0x00000130	main+0x44	MOV	R2, 3	-	-	-
18	0x00000134	main+0x48	MOV	R1, 10	-	-	-
19	0x00000138	main+0x4c	BL	0xfffff68(search (0xa8))	-	-	-
20	0x000000a8	search	STMDB	SP!, {R8,R9}	0x00182234 (.stack) 0x00182238 (.stack)	W	0x00182310 0x00182308
21	0x000000ac	search+0x4	MOV	R3 <i>, 0	-	-	-
22	0x000000b0	search+0x8	B	0x20(search+0x30 (0xd8))	-	-	-
23	0x000000d8	search+0x30	CMP	R3 <i>, R1 <array_len>	-	-	-
24	0x000000dc	search+0x34	BLT	0xfffffd0(search+0xc (0xb4))	-	-	-
25	0x000000b4	search+0xc	MOV	IP, R3 <i>	-	-	-
26	0x000000b8	search+0x10	MOV	IP, IP, LSL 2	-	-	-
27	0x000000bc	search+0x14	ADD	IP, IP, R0 <array>	-	-	-
28	0x000000c0	search+0x18	LDR	IP,[IP]	0x0018223c (.stack)	R	0x00000001
29	0x000000c4	search+0x1c	CMP	IP, R2 <key>	-	-	-
30	0x000000c8	search+0x20	BNE	0x4(search+0x2c (0xd4))-	-	-	-
31	0x000000d4	search+0x2c	ADD	R3 <i>, R3 <i>, 1	-	-	-
32	0x000000d8	search+0x30	CMP	R3 <i>, R1 <array_len>	-	-	-
33	0x000000dc	search+0x34	BLT	0xfffffd0(search+0xc (0xb4))	-	-	-
34	0x000000b4	search+0xc	MOV	IP, R3 <i>	-	-	-
35	0x000000b8	search+0x10	MOV	IP, IP, LSL 2	-	-	-
36	0x000000bc	search+0x14	ADD	IP, IP, R0 <array>	-	-	-
37	0x000000c0	search+0x18	LDR	IP,[IP]	0x00182240 (.stack)	R	0x00000001
38	0x000000c4	search+0x1c	CMP	IP, R2 <key>	-	-	-
39	0x000000c8	search+0x20	BNE	0x4(search+0x2c (0xd4))-	-	-	-
40	0x000000d4	search+0x2c	ADD	R3 <i>, R3 <i>, 1	-	-	-
41	0x000000d8	search+0x30	CMP	R3 <i>, R1 <array_len>	-	-	-
42	0x000000dc	search+0x34	BLT	0xfffffd0(search+0xc (0xb4))	-	-	-
43	0x000000b4	search+0xc	MOV	IP, R3 <i>	-	-	-
44	0x000000b8	search+0x10	MOV	IP, IP, LSL 2	-	-	-
45	0x000000bc	search+0x14	ADD	IP, IP, R0 <array>	-	-	-
46	0x000000c0	search+0x18	LDR	IP,[IP]	0x00182244 (.stack)	R	0x00000002
47	0x000000c4	search+0x1c	CMP	IP, R2 <key>	-	-	-
48	0x000000c8	search+0x20	BNE	0x4(search+0x2c (0xd4))-	-	-	-
49	0x000000d4	search+0x2c	ADD	R3 <i>, R3 <i>, 1	-	-	-
50	0x000000d8	search+0x30	CMP	R3 <i>, R1 <array_len>	-	-	-
51	0x000000dc	search+0x34	BLT	0xfffffd0(search+0xc (0xb4))	-	-	-
52	0x000000b4	search+0xc	MOV	IP, R3 <i>	-	-	-
53	0x000000b8	search+0x10	MOV	IP, IP, LSL 2	-	-	-
54	0x000000bc	search+0x14	ADD	IP, IP, R0 <array>	-	-	-
55	0x000000c0	search+0x18	LDR	IP,[IP]	0x00182248 (.stack)	R	0x00000003
56	0x000000c4	search+0x1c	CMP	IP, R2 <key>	-	-	-
57	0x000000c8	search+0x20	BNE	0x4(search+0x2c (0xd4))-	-	-	-
58	0x000000cc	search+0x24	MOV	R0 <array>, 1	-	-	-
59	0x000000d0	search+0x28	B	0xc(search+0x3c (0xe4))-	-	-	-
60	0x000000e4	search+0x3c	LDMIA	SP!, {R8,R9}	0x00182234 (.stack) 0x00182238 (.stack)	R	0x00182310 0x00182308
61	0x000000e8	search+0x40	MOV	PC, LR	-	-	-
62	0x0000013c	main+0x50	MOV	R1, R0	-	-	-
63	0x00000140	main+0x54	CMP	R1, 0	-	-	-
64	0x00000144	main+0x58	BEQ	0x0(main+0x60 (0x14c))-	-	-	-
65	0x00000148	main+0x5c	ADD	R4 <found>, R4 <found>, 1	-	-	-

Remaining Instructions

There are other instructions that have not been covered here, including branches and comparison instructions. The register reconstruction algorithms must handle the side effects of these instructions, such as branches that modify the link register. The details are relatively straightforward, but must be handled correctly to maximize the number of registers known correctly.

Applying the entire register reconstruction algorithm to the example trace buffer gives the register values in Figures 3 and 4 before and after the call to `search()`. See if you can figure out how to derive these register values from the trace data in Listing 3.

You can see that the algorithms presented here let you reconstruct the state of most registers prior to the call to `search()`. In fact, you know 12 of the 16 registers, and the four register values that are unknown are never encountered in the trace buffer. In general, these algorithms are able to reconstruct a large percentage of register and memory

values, especially with large traces, which makes the TimeMachine debugger very useful for debugging software problems.

Conclusion

The register and memory values that are interpolated from trace data may seem like a tedious and academic task with no real-world applications. However, this knowledge allows the TimeMachine debugger to display information at the source-code level because the debugger contains a mapping of program variables to register and memory locations. It can display the values of both global and local variables as well as additional information about the state of the system as it was running. The low-level trace information is thus translated into the high-level concepts that an application-level software engineer uses to write embedded software. Therefore, an ordinary and familiar debugging interface can be used to debug race conditions and other timing-related

failures without altering the runtime characteristics of the system.

Because of tools such as the TimeMachine debugger, hardware trace is a useful debugging tool for embedded software engineers. By increasing visibility into systems and providing debugging information without altering the runtime characteristics of the system being debugged, trace offers a unique and powerful method for debugging problems, both simple and difficult. Since debugging is such a large component of the cost of software development, tools such as these can make a huge difference in the quality and cost of embedded devices. And as trace ports and trace tools become more pervasive, it will have a larger and larger effect on embedded software development in the future.

References

- [1] *ARM Architecture Reference Manual*, Addison-Wesley, 2000. □

