IT 09 006
Examensarbete 30 hp
Mars 2009

Amazing Trace

Wilhelm Äretun

Institutionen för informationsteknologi
Department of Information Technology

UPPSALA
UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

Amazing Trace

Wilhelm Äretun

This paper describes the implementation of an execution history, called "Amazing Trace", that entails decoding and analyzing a recorded instruction trace, and in particular the design and implementation of a data structure which can encapsulate the entire state of the target system, including memory and registers, over time, and to do this efficiently because the trace buffer can contain data for hundreds of thousands of instructions.

The Amazing Trace is also integrated in the IAR Embedded Workbench C-SPY Debugger, such that the history can be traversed both forwards and backwards while inspecting the full target state through the normal debugger windows.

Handledare: Jonas Blomberg
Ämnesgranskare: Sven-Olof Nyström
Examinator: Anders Jansson
IT 09 006
Tryckt av: ITC

Contents

List of Figures

1 Introduction

An important tool for debugging embedded systems is a trace buffer. A trace buffer contains a log of instructions (or rather the addresses of instructions) that have been executed by the processor. It can be used to examine the sequence of events that leads to a given state, such as a malfunction of the program.

More advanced trace buffers also contain a log of accesses to memory, i.e. the address and contents of memory read and write operations. This gives further information that is useful when examining the causes of a program error. However, most trace buffers cannot log the contents of processor registers, and for some processors, much of the most important machine state is encapsulated by these registers.

By examining the sequence of executed machine instructions and matching it with corresponding read and write operations in the trace buffer, it is possible to deduce much of the missing register state. For example, if the instruction trace indicates an instruction that moves data from a certain register to memory, and there is a matching memory write operation in the trace buffer, it can be deduced that the value of the register at that point in the program must be the data value found in the trace buffer.

By processing a full trace buffer in this manner, it is possible to recreate a reasonably complete history of the state of the target system, and by traversing this history; it is possible to examine the execution history including the values of variables, even variables residing in processor registers. This would be a distinctly valuable addition to the debugging arsenal for embedded systems.

In this paper there is a brief introduction to the target system, 78K0, in chapter 3, an overview of the implementation process in chapter 4 and more detailed information about the Trace Buffer, instructions and the time-line-memory in chapter 5, 6 and 7. Chapter 8 is about how the Amazing Trace prototype is tested and the performance is measured in chapter 9. The implementation of the Amazing trace is a prototype, or a proof of concept, and some future work is needed and this is discussed in chapter 10.

2 Related Work

The most significant attempt to implement something similar to the Amazing Trace is Green Hills "Time Machine" [10]. This Time Machine works in the same manner as Amazing Trace, using a trace buffer, with no knowledge about the registers, trying to deduce the registers from there [7].

"Omniscient Debugging" [6] is an interesting idea not too different from Amazing Trace, it collects information from every state change of the program and stores the states in a history, allowing the user to go back in time to see the state of the program it a specific time. It does not store the information on a memory and register level like the Amazing Trace.

Shade is a cross platform, fast and flexible custom trace generator described by Bob Cmelik and David Keppel [1]. Shade achieves its flexibility by using dynamic compilation. Code which simulates and traces the application is dynamically generated and cached for reuse. Instead of storing the trace data it recreates the trace on demand which enables collection and analysis of realistically long traces.

An early idea described by Jacques Cohen and Neal Carpenter [5] is a language for studying the behaviour of programs based upon the data collected while these programs are executed by a computer. The program to debug has to be written in an Algol 60-like language and like the "Omniscient Debugging" the idea is to store every state change in a history. To view the stored data the programmer is allowed to make complex queries about the states of the program at different times.

One of the first debuggers called "DDT" released in 1961 allowed the user to set breakpoints, set trace points and single step the program [6]. There has been little change in the way debuggers work since then, the functionality is the same but with better usability. The development in debuggers has been concentrated to when the program is still running, Amazing Trace allows debugging after the program has executed.

3 Target System NEC 78K0

The 78K0 is a general-purpose 8 bit micro controller unit developed by NEC Electronics. The MCU has 63 instructions, [4] which is a relative small number of instructions. To make this project work, every instruction in the MCU will need to be decoded, and therefore any MCU with few instructions will make a good choice to use when testing the idea of Amazing Trace.

The 78K0 Series program memory map varies depending on the internal memory capacity and each 78K0 series product has internal ROM in the address space. Program and constant data, etc. are stored in the ROM. Normally, this memory space is addressed by the program counter (PC). The program counter is a 16-bit register that holds the address information of the next instruction to be executed. The PC is automatically incremented according to the number of bytes of the instruction to be fetched. When a branch instruction is executed, immediate data and register contents are set. There is also a Program Status Word (PSW) which is an 8-bit register consisting of various flags to be set/reset by instruction execution. [2]

The registers in the 78K0 micro controller unit are a bit special and consists of a memory address area called "General-Purpose Registers" which is mapped at particular addresses (FEE0H to FEFFH) of the data memory. This memory area will be ignored by the Amazing Trace and registers will be treated as if they were normal register, mainly because this is the way they behaves to the user and are seen in the IAR Embedded Workbench C-SPY Debugger. There are 8 registers and 4 register pairs, see Figure 1.

The memory space concerning this particular Amazing Trace prototype will be the 16 bit address space of 8 bit memory, the registers containing 8 registers, the Program Counter (PC) and the Stack Pointer (SP).

4 Implementation of Amazing Trace

An opcode table, where all instruction is stored, is generated. This is done only once, see Appendix A.2.7. This process is explained further in chapter 6.1.

When data from the Trace Buffer is collected, the Trace Buffer is traversed from the beginning to the end. For each instruction the Program Counter and

r2	r1	r0	reg	
0	0	0	R0	X
0	0	1	R1	A
0	1	0	R2	C
0	1	1	R3	B
1	0	0	R4	E
1	0	1	R5	D
1	1	0	R6	L
1	1	1	R7	H

p1	p0	reg-pair	
0	0	RP0	AX
0	1	RP1	BC
1	0	RP2	DE
1	1	RP3	HL

Figure 1: NEC 78K0 registers.

the trace buffer data, see Figure 5, are fetched. The instruction is located in the opcode table and decoded. The instruction is simulated, to behave exactly the same way as in the MCU, and for every read or write performed by that instruction the Amazing Trace time-line-memory is updated. The behavior of this could be described with the pseudo code in Figure 2.

```
for each instruction item in trace buffer do
  Get the PC address of the instruction
  Read the instruction from memory
  Decode the instruction
  for every read and write performed by that instruction do
    Locate the corresponding memory activity in trace buffer
    Get the actual values read or written
    Update Amazing Trace time-line-memory data structure
  end for
end for
```

Figure 2: Iteration through the Trace Buffer and update the Amazing Trace time-line-memory.

The Amazing Trace data structure is implemented as a map from the C++ standard template library containing one map for each memory address stored in the Amazing Trace time-line-memory. The first maps index is the memory address and the other maps' indexes are the time, see Figure 3. The Status of one memory item stored could be either "Known From Here" or "Unknown". Until the time of the first item stored on a memory address the status of that memory address is "Unknown".

When a line in the trace buffer is highlighted by the user, see Figure 13, a read is performed on the Amazing Trace time-line-memory on the highlighted position and the result is displayed. The items in the Amazing Trace time-line-memory are organized after the time that the instruction was executed. In Figure 3 four items are stored in the Amazing Trace time-line-memory at

memory address FB00 at different times. If a search is executed on the Amazing Trace time-line-memory at position 185, the item stored before that position is returned if the status of that item is "Known From Here", otherwise "Unknown" is returned.

Only the memory and registers accessed are stored in the Amazing Trace time-line-memory, some registers might not be accessed yet and are therefore unknown to the Amazing Trace. That is not a problem because when using Amazing Trace when debugging a program, a register or memory address that has not been accessed by the program is not interesting to know about. It's actually rather helpful to be able to exclude those from what might be the problem with the program that is being debugged.

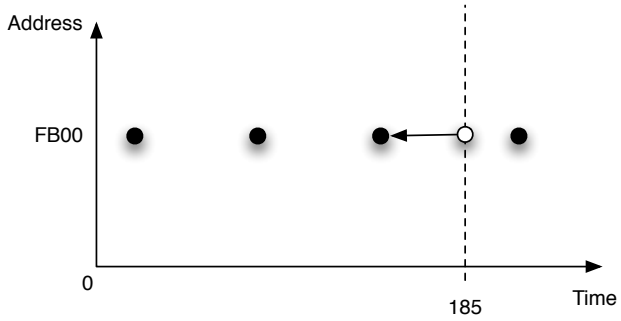


Figure 3: Amazing Trace time-line-memory, four memory states is stored on address FB00 and a search on time 185 returns the nearest memory state stored before time 185.

5 Trace Buffer

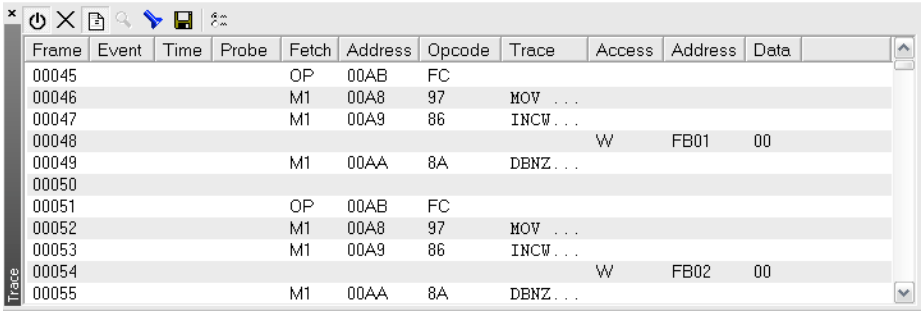
The data stored in the trace buffer is labeled: "Frame", "Event", "Time", "Probe", "Fetch", "Address", "Opcode", "Trace", "Access", "Address" and "Data". See Figure 4 and Figure 5 for a brief explanation of the labels. This stored data can be accessed in the existing architecture of the IAR Embedded Workbench C-SPY Debugger (using the TdEmuExecHandeler member function GetTraceInfo, see Appendix A.2.1).

Information important for Amazing Trace is under "Frame", "Fetch", "Address", "Opcode", "Access", "Address", and "Data".

The data is collected from the Trace Buffer and stored in the Amazing Trace time-line-memory. There is no information about the registers in the Trace Buffer and therefore additional calculations are performed before updating the Amazing Trace time-line-memory, so that the Amazing Trace also can keep track of the registers.

5.1 Access Trace Buffer

The trace buffer can be accessed from within the existing architecture of the IAR Embedded Workbench C-SPY Debugger using a member function (Get-



Frame	Event	Time	Probe	Fetch	Address	Opcode	Trace	Access	Address	Data
00045				OP	00AB	FC				
00046				M1	00A8	97	MOV ...			
00047				M1	00A9	86	INCW...			
00048								W	FB01	00
00049				M1	00AA	8A	DBNZ...			
00050										
00051				OP	00AB	FC				
00052				M1	00A8	97	MOV ...			
00053				M1	00A9	86	INCW...			
00054								W	FB02	00
00055				M1	00AA	8A	DBNZ...			

Figure 4: The trace window as seen when using the IAR trace debugger.

Indexing	
Frame	Line number.
Not Relevant For Amazing Trace	
Event	The name of the single events that have been triggered by the event conditions.
Time	Time stamp of the trace frame..
Probe	probe value of the trace frame.
Instruction Information	
Fetch	Shows if it is an instruction or operation data.
Address	The memory address of the instruction.
Opcode	Instruction opcode.
Trace	Short info about instruction, readable for humans.
Memory Access Information	
Access	Type of access to the memory, Read or Write.
Address	Address of the access.
Data	Data read or written to the memory address.

Figure 5: Explanation of the Trace Buffer labels [11].

TraceInfo from an instance of the gTdEmuExecHandler class, see Appendix A.2.1) which returns a string from one row from the trace buffer with the items separated by tabs.

When a program has been compiled the opcodes are locked in memory and will not change over time, so it is possible to read those directly from the memory, which is faster than reading from the Trace Buffer. Therefore these items are not read from the Trace Buffer.

5.2 Traversing Trace Buffer in IAR Embedded Workbench C-SPY Debugger

When the user traverses the trace buffer, a line is highlighted in the trace window, see Figure 4, and it is possible to traverse with the arrow keys or by clicking. There are several different views in the IAR Embedded Workbench C-SPY Debugger, see Figure 13, and the disassembly and source code view will also get highlighted at the corresponding place in the program.

In the existing architecture of the IAR Embedded Workbench C-SPY Debugger, all of the views gets the information to display from the same source, which displays the memory. By letting that source return information from the Amazing Trace data structure instead of from the memory the "register view", "memory view" and any other views affected, like the "watch view" for example, will also be changed accordingly when traversing the trace buffer.

6 Instructions

There are a lot of instructions in this micro controller unit and for testing purposes a subset of instructions (42), that are used in a simple program as seen in Figure 6, will be chosen. This simple program will generate the set of opcodes shown in Figure 7.

All instructions have its own class and to keep track of all the instructions an instance of every instruction is stored in a opcode table which implements as a 256 long array. Every instruction is stored in the position of the array that corresponds to the opcode number.

```
#define len 3

char str[len];
char a = 'a';

int main( void )
{
  for(int i=0 ; i < len ; i++)
  {
     str[i] = a;
     a++;
   }
   return 0;
}
```

Figure 6: A simple program.

Opcode	Mnemonic	Operands	Opcode	Mnemonic	Operands
$10110p_1p_01$	PUSH	rp	10010111	MOV	[HL], A
$10110p_1p_00$	POP	rp	10001010	DBNZ	C, $addr16
$00010p_1p_00$	MOVW	rp, #word	10001011	DBNZ	B, $addr16
11111010	BR	$addr16	11101110	MOVW	SP, #word
$11000p_1p_00$	MOVW	AX, rp	10000111	MOV	A, [HL]
01111101	XOR	A, #byte	00000011	MOVW	!addr16, AX
11011010	SUBW	AX, #word	00000010	MOVW	AX, !addr16
10001101	BC	$addr16	10011101	BNC	$addr16
10001110	MOV	A, !addr16	11101010	CMPW	AX, #word
$01110r_2r_1r_0$	MOV	r, A	10101110	MOV	A, [HL+byte]
11001010	ADDW	AX, #word	10001001	MOVW	AX, SP
$11010p_1p_00$	MOVW	rp, AX	10111110	MOV	[HL+byte], A
$01100r_2r_1r_0$	MOV	A, r	10000101	MOV	A, [DE]
10010101	MOV	[DE], A	00100010	PUSH	PSW
$01000r_2r_1r_0$	INC	r	00100011	POP	PSW
10011110	MOV	!addr16, A	$11100p_1p_00$	XCHW	AX,RP
$10000p_1p_00$	INCW	rp	10101011	MOV	A, [HL+B]
10101111	RET		$00110r_2r_1r_0$	XCH	A, r
10011010	CALL	!addr16	$10100r_2r_1r_0$	MOV	r, byte
00000000	NOP		$10010p_1p_00$	DECW	rp
01111011	DI		10101101	BZ	$addr16

Figure 7: Instructions used by a simple program.

6.1 Generating Opcode Table

The opcodes are 8 bit long, which can be any number between 0 and 255, so the table has to be able to store 256 entries. This is done with an array of instances of a subclass to the class DbuInstruction that would represent the actual instruction. An instance of the subclass is placed in the array on the position that has the same number as the opcode. For example the opcode $00010p_1p_00$ (MOVW rp,#word) gets the positions 16, 18, 20 and 22 and $01100r_2r_1r_0$ (MOV A,r) gets the positions 96, 97, 98, 99, 100, 101, 102, 103 and 104. The "p_1p_0" in opcode represents the register pair, and the "$r_2r_1r_0$" represent the register, see Figure 1.

To generate this table a smaller table is created, see Appendix A.2.6, and this one is used to extract the "p_1p_0" and "$r_2r_1r_0$" part of the opcode to create the final 256 opcode table, implemented as a array, using the member functions of an instance of the class TableGenerator see, appendix A.2.7.

6.2 Decode Instructions

The decoding of instruction and update to Amazing Trace data structure is done in the class DbuInstruction, see appendix A.2.2, which has a few helper functions for the decoding process. Each instruction has its own class, with member functions for decoding and update, which override the decode and update functions from the main class DbuInstruction.

DbuInstruction has member variables for memory address, register address and register pair address. One instruction can read a maximum of 4 bytes [4], so there are also member variables for the value of each of the 4 bytes. Each member variable has a corresponding setter member function that updates the member variable. If there are any "p_1p_0" or "$r_2r_1r_0$" in the opcode that

represent a register pair or register address, as seen in Figure 1, the register or register pair is determine in the register pair or register setter member function and updated to the corresponding member variable.

6.3 Update Amazing Trace Time-Line-Memory

There is no information about the registers in the trace buffer and therefore when updating the Amazing Trace time-line-memory there will be a bit of detective work to determine what the instruction read from, or write to the register. For example, MOVW rp #word moves a word from memory to register and make it possible to deduce the value of that particular register until the next write to that register.

The behavior of the instruction needs to be simulated before updating the Amazing Trace time-line-memory. For example if the instruction does some sort of calculation, like "XOR A,r", it is also calculated by the update member function.

7 The Time-Line-Memory

The 78K0 MCU, which is an 8 bit micro controller unit, has a 16 bit address space. That means that the memory has 65536 addresses. To store the value of every address for every time event would lead to a memory consumption which would get very quickly out of hand. 100 time events would be 6.25 MB, without storing any additional data. And this micro controller unit uses only 16 bit address space. To make this a bit more general, so the time-line-memory structure also could be implemented in a target system with for example 32 bit address space you need something that consumes less memory.

The memory occupying an address in the Amazing Trace data structure could be in two different states that also need to be stored. The states of the memory address could be "Unknown" or "Known From Here". For example, XOR with a "Known From Here" register and a "Unknown" register would turn the register with a "Known From Here" state into an "Unknown" state.

7.1 Data Structure For Testing

The first implementation of the time-line-memory data structure is an atEntity, see Figure 8, 2D array, SimpleStructure[65536][500], this will cover the needs for testing purposes for the first 500 states in the program. The data stores an atEntity in the 2D array at position [memory address][time].

Because of the memory consumed by this structure this is not a useful way to store the time-line-memory, but before the final Amazing Trace time-line-memory data structure is finished it is a good way of testing the idea and also to confirm that the final, slightly more complicated, data structure is working as intended.

7.2 The Final Data Structure

The final Amazing Trace time-line-memory data structure needs to be space conservative and not involve to mush traversing through the representation of

```
enum MemoryStatus
{
  Unknown,
  KnownFromHere
};
struct atEntity
{
  char Value;
  MemoryStatus Status;
};
typedef atEntity SimpleStructure[65536][500];
```

Figure 8: Simple data structure for testing purposes.

the memory states, because there could be extremely many states and traversing through them could take more time than the user might be willing to accept.

The original thought was a map from the C++ Standard Template Library containing one binary search tree for each memory address accessed by the trace buffer. Searching for the memory state in a specific time was to be done with a modified search in the binary search tree that returns the nearest state stored before, if the time not stored in the tree or exactly on the time if the time is stored in the tree. This could be done in O(log n) time [12] for each memory address.

There are many insertions, and the insertions occurs one after the other ordered by time. The index of the map is also the time, that means every insertion will be in order [1]. In this case a self balancing binary search tree, like a AVL-tree, is the optimal tree structure to use.[8]

A map is implemented in Microsoft Visual Studio as a Red Black tree and can do exactly as what was intended with the self balancing binary search tree in O(log n) time using a combination of the lower_bound, which returns an iterator to the first element greater than or equal to a certain value, and upper_bound, which returns an iterator to the first element greater than a certain value, member functions. There are two cases, see Figure 9.

1. If lower_bound is not equal to upper_bound the item searched for is where the lower_bounds iterator points.

2. If lower_bound is equal to upper_bound, the item searched for is where the lower_bounds iterator -1 points.

The member function equal_range returns a pair of lower_bound and upper_bound in O(log n) time [3] and it's that member function that is actually used, see appendix A.2.5.

The final data-structure is an instance of a class with a map where the index represents all the memory addresses and contains another map with the time as its index, containing all the states that particular memory address has been in during the time that the trace has executed. And the execution time for finding each memory address state at a particular time would still be O(log n) [3]. The class also includes some functions for access and adding data to the memory map, see Appendix A.2.4.

[1] Ordered insertion into a binary search tree turns the tree into a list.

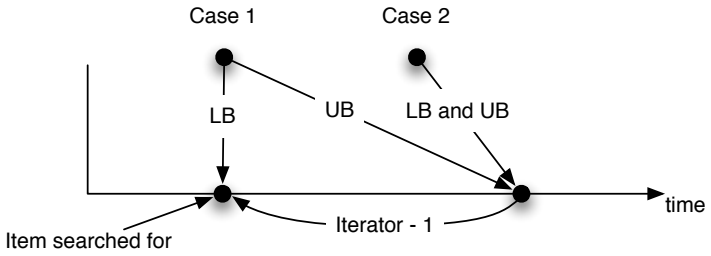


Figure 9: lower_bound and upper_bound.

7.2.1 Memory Consumption

Because of how the C++ compiler interprets enum, in the atEntity from Figure 8, the memory needed to store one atEntity is 32 + 8 + 24 bits. One integer for the enumerated item, the value of the memory address and 24 bit padding. Many atEntitys will be stored and a waste of space with 32 bits where only a 8 bit boolean is actually needed for the memory status is not acceptable. Since there are only two options in the status, "Unknown" and "KnownFromHere", a bool named "KnownFromHere" is replacing the enum type "Status" in the final data structure. So the memory needed for the new atEntity will be 8 + 8 + 16 bits. A Boolean for the status, the value of the memory address and 16 bit padding. This will half the memory consumption of the atEntity. see figure 10

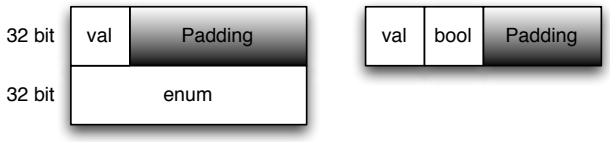


Figure 10: Difference in memory consumption when using enum or bool to reprecent the status of an atEntity.

The IAR Embedded Workbench C-SPY Debugger is developed using Microsoft Visual Studio and a map in the C++ Standard template library is implemented by Microsoft Visual Studio as a red black tree and it uses three 32 bit pointers per node, one pointer for the parent and two for the left and right node. [9]

- For each memory item accessed by the program, 16 bits (8 for memory and 8 for the boolean KnownFromHere) for the atEntity is stored in the Amazing Trace data structure.
- All maps for each memory address and each register address can be approximated to a single map with the size of all maps combined. This map uses three 32 bit pointers, 32 bit unsigned int as key and 16 bit atEntity as value. This gives: $(3*32)+32+16 = 144bit$. The node will also occupy

a additional 16 bit because of the 32 bit word length on a standard 32 bit Windows machine, see figure 11, to a total of 160 bit = 20 bytes.

- There is also a map containing all maps for each memory address with max 65536 nodes and a map for the registers with max 16 items. These maps use unsigned short for key and the previously calculated maps for value. This gives: $(16 + 3 * 32) * 65536 = 7340368bit = 917546byte$ and $(16+3*32)*16 = 1792bit = 224byte$. These values are constant and rather small so they can be disregarded when calculate the memory consumption.

Where n is the number of nodes stored in the size of the Amazing Trace time-line-memory is:

$$size(n) = 20 * n + 917546 + 224\ byte$$

When running the simple program in Figure 6, all access to the memory and register stored by the Amazing Trace time-line-memory, divided with the number of instructions used in the simple program gives the he average memory stored for one instruction. It is approximately 5. So when i is the number of instructions executed by a program the size of the Amazing Trace time-line-memory is approximately:

$$size(i) = 20 * 5 * i = 100 * i\ byte$$

The memory consumption will approximately be the number of instructions times 100. The simple program executes 103 instructions from beginning to the end. This gives the size of Amazing Trace time-line-memory when running the simple program to approximately $100 * 103 = 10300\ byte$.

A longer program with 10000 instructions: $100 * 10000 = 1MB$.

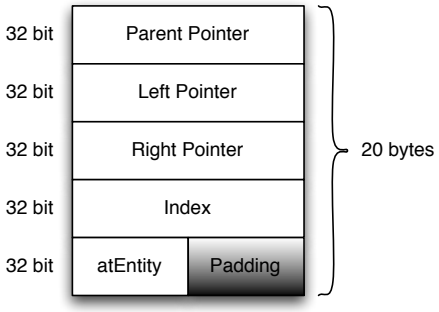


Figure 11: The memory consumed by each node in the red black tree.

8 Testing

The code has been tested by single stepping through the test program in the IAR Embedded Workbench C-SPY Debugger and taking notes of the state of memory and registers in each step, and later comparing to stepping through the program and using the Amazing Trace. A level of testing has also already been made

during the development state when writing the decoding for each instruction. Only the register contents have been tested this way, but the registers depend on the memory to be correct, so if the registers show the correct values it is enough to pass the test.

9 Performance

Performance is a critical issue for the user experience and in this prototype, there are essentially two critical moments where performance is important. 1) Storing the data in the Amazing Trace time-line-memory and 2) reading the data from the Amazing Trace time-line-memory, this is done when traversing the trace buffer in the IAR debugger.

9.1 Storing Data in Amazing Trace Time-Line-Memory

Every time the trace buffer changes the Amazing Trace time-line-memory is completely rewritten, this was supposed to be a temporary solution but the time of this project run out before fixing this.

Rewriting the time-line-memory takes longer time the more items are in the trace buffer, so when single stepping in the IAR debugger is not a pleasant experience when there are too many (2000 lines and above) items in the trace buffer. This is not a problem when running the whole program at once, without any single stepping in the IAR Embedded Workbench C-CPY Debugger, but when going from this prototype to a real implementation this must to be solved by updating the Amazing trace time-line-memory incrementally. As seen in Figure 12 the time for running trace buffer and adding data to the Amazing Trace time-line-memory increase in the same rate as the length of the trace buffer. Approximately, doubling the number of lines in the trace buffer also doubles the time it takes to add data to the Amazing Trace time-line-memory.

9.2 Reading Data From Amazing Trace Time-Line-Memory

Traversing the data structure after adding data performs as intended, a nice user experience without noticeable difference in lag even with a lot of data stored in the Amazing Trace time-line-memory.

Lines	Run program and add data
495	1.3 sec.
1028	2.6 sec.
2012	5.5 sec.
4062	12.3 sec.
7957	30.5 sec.

Figure 12: Performance of Amazing Trace in the IAR Embedded Workbench C-SPY Debugger running on an Intel QuadCore 2.4GHz.

10 Future Work

Single stepping when in the IAR Embedded Workbench C-SPY Debugger is to slow to be useful as Amazing Trace works today.

The instructions implemented in this prototype are only the ones used by the two programs, "Simple program" in Figure 6 and another program that calculates the first Fibonacci numbers. There are a few opcodes left to implement.

The hardware trace buffer has an upper limit of 8000 lines and when exceeding that limit it will create gaps in the Amazing Trace time-line-memory.

To avoid the problem with the single stepping and the gaps when buffer exceeds 8000 steps the data Amazing Trace must not read the whole trace buffer when adding the data.

11 Discussion

This prototype stores some parts in the Amazing Trace time-line-memory thats not necessary to store in the final application. For example, the opcodes that are read directly from memory might be unnecessary to store in the data structure, because they will not change over time and it's probably better to read those directly from the memory. However, it is hard to know what part of the memory to store or not to store, so the safe way is to store those as well. The Amazing Trace time-line-memory is quite memory effective and fast enough to access. A solution that is a bit more elegant might be to read those items from the memory.

One possible feature that might be interesting to investigate in the future would be to make it possible to traverse the history back in time with Amazing Trace and write all the things Amazing Trace know of the memory and registers and continue the real program from that point in time. Moreover, maybe be able to change the parameters that one might think is the problem when debugging the program.

12 Conclusion

The Amazing Trace is a helpful addition to the Trace Buffer, this project provides a proof of concept that it's possible to access the Trace Buffer and extract the data needed and even as there is no information about the registers in the Trace Buffer, it's often possible to deduce the value about the registers with the given information.

References

[1] Bob Cmelik and David Keppel. Shade: a fast instruction-set simulator for execution profiling. *Measurement and modeling of computer* , 1994.

[2] NEC Corporation. *User's Manual, Instructions Common to 78K/0 Series.* U12326EJ4V0UM00, forth edition, 2001.

[3] David R. Musser, Atul Saini. *STL Tutorial and reference Guide.* Addison-Wesley, first edition, 1996.

[4] NEC Electronics (Europe) GmbH. 78K0/KC2 8-bit microcontroller. *78K Series Product Letter*, 2005.

[5] Jacques Cohen and Neal Carpenter. A language for inquiring about the run-time behaviour of programs. *Software-Practice and Experience*, 1977.

[6] Bil Lewis. Debugging backwards in time. *Lambda Computer Science*, 2003.

[7] Michael Lindahl. TimeMacines: the future of debuggers. *Green Hills Software*, 2009.

[8] Ben Pfaff. Performance analysis of bsts in system software. *Joint International Conference on Measurement and Modeling of Computer Systems*, 2004.

[9] P.J. Plauger. xtree internal header. *Microsoft Visual Studio header file*, 1995.

[10] Green Hill Software. In-memory timemachine and traceedge. *Green Hill Whitepaper*, 2006.

[11] IAR Systems. 78K IAR C-SPY® hardware debugger systems. *User Guide*, 2008.

[12] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introductions to Algorithms*. MIT Press, second edition, 1990.

A Appendix

A.1 Screenshots

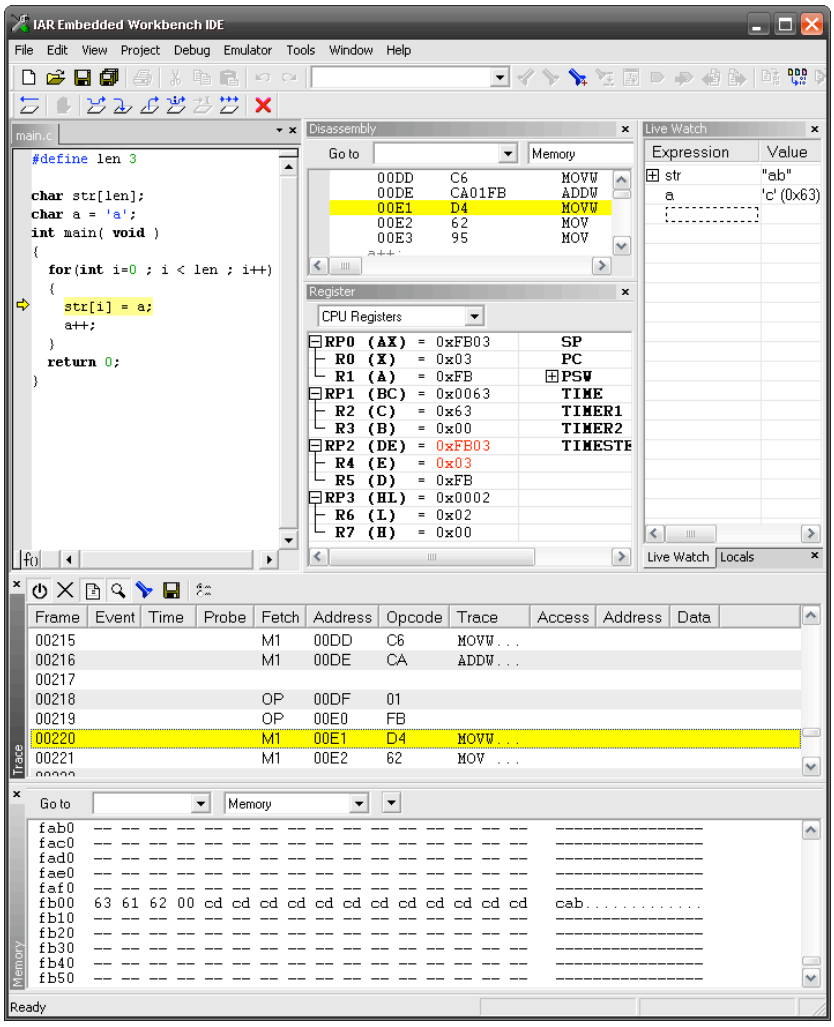


Figure 13: IAR Embedded Workbench C-Spy Debugger screenshot with Amazing Trace.

A.2 Source Code

There are four files used concerning this project, two already existing files that have been modified, "TdEmuTraceDataProvider.cpp" and "TdEmuTraceDataProvider.h", and two new files, "TdEmuAmazingTraceDataProvider.cpp" and "TdEmuAmazingTraceDataProvider.h". In this appendix code relevant to Amazing Trace will be explained.

A.2.1 Set up AmazingTrace Data

```
bool gTableGenerateNotDone = true;
/*
Set up Amazing Trace data
*/
void
TdEmuTraceDataProvider::
AmazingTrace()
{
  TdTraceData traceData;
  uint32_t index = NumberOfRows();
  uint32_t pc;
  int accessUntilFrame = 0;

  //Setup Decode Table, only once
  if(gTableGenerateNotDone)
  {
    SetupDecodeTable(dtab);
    gTableGenerateNotDone = false;
  }

  //clear MemoryMap
  gAtMemoryMap.clear();

  //itterate trough tracebuffer and use decodetable to decode and also update the datastructu
  for(uint32_t i=0; i<index; i++)
  {
    char * row = gTdEmuExecHandler->GetTraceInfo( i, TRACEINFO_WINDOW, &pc );
    traceData.ExtractTraceData(row);
    if(traceData.fetch == "M1")
    {
      dtab[traceData.opcode]->Decode(traceData.opcode,(uint16_t)pc);
      dtab[traceData.opcode]->Update(gAtMemoryMap,i,accessUntilFrame);
    }
  }
}
```

Main function for Amazing Trace that is called from another member function in the already existing TdEmuTraceDataProvider class when the trace buffer reports to the existing architecture how many entries added or removed. From here functions for set up the decode table, decoding and update the Amazing Trace data structure are called.

A.2.2 DbuInstruction

```
class DbuInstruction
{
public:
  /*
  Decode the instruction...
  accessUntilFrame is an index for how far in the trace buffer
```

```
  GetNextReadOrWrite has looked.
  */
  virtual void Decode(uint8_t opcode, uint16_t pc) = 0;

  /*
  Update result of the Decode to the time-line-memory atMemoryMap
  */
  virtual void Update(AmazingTraceMemoryMap &atMemoryMap, int time,
              int &accessUntilFrame) = 0;

protected:
  void SetB1Value(uint8_t opcode);
  void SetB2Value();
  void SetB3Value();
  void SetB4Value();
  void SetRPAddress(uint8_t opcode);
  void SetRegisterAddress(uint8_t opcode);
  void SetMemoryAddress(uint16_t addr);

  //Returns the value stored in a 8 bit Address
  uint8_t ReadMemory8Bit(uint16_t addr);

  //Access the tracebuffer and get next write or read
  TdTraceData GetNextReadOrWrite(int &accessUntilFrame, int time);

  uint8_t rpAddress,registerAddress,b1Value,b2Value,b3Value,b4Value;
  uint16_t memoryAddress;

private:
  //translate the pp in opcode to a int
  int ReadPP(uint8_t opcode);

  //translate the rrr in opcode to a int
  int ReadRRR(uint8_t opcode);
};
```

The DbuInstruction is the main class for all instructions containing a few helper functions and some member variables that are used a lot by the instructions. Each instruction has its own class that inherit and overrides the member functions decode and update from DbuInstruction.

A.2.3 An Instruction

```
class xor_a_byte_t : public DbuInstruction
{
  void Decode(uint8_t opcode, uint16_t pc)
  {
    SetMemoryAddress(pc);
    SetB1Value(opcode);
    SetB2Value();
  }
  void Update(AmazingTraceMemoryMap &atMemoryMap, int time,
          int &accessUntilFrame)
  {
    atEntity A;

    A = atMemoryMap.GetLatestWriteToRegisterAddress(kTdRegR1);
    A.value = b2Value^A.value;

    atMemoryMap.WriteMemory(time,true,memoryAddress,b1Value);
    atMemoryMap.WriteMemory(time,true,memoryAddress+1,b2Value);

    atMemoryMap.WriteRegister(time,A.knownFromHere,kTdRegR1,A.value);
```

```
    atMemoryMap.WriteRegister(time,true,kTdReg_PC,memoryAddress+2);
  }
};
```

This is XOR A #byte instruction that inherits from DbuInstruction and overrides the decode and update functions. The Decode decodes the instruction and stores the data in the member variables. Update do the calculations needed and update the Amazing Trace data structure.

A.2.4 Amazing Trace Data Structure

```
typedef std::map< uint32_t,atEntity > AmazingTraceMemoryItem;
typedef std::map< uint16_t,AmazingTraceMemoryItem > AmazingTraceMap;

class AmazingTraceMemoryMap
{
public:
  //Cler the Memory and Register
  void clear();

  //Call from SetRemapperTo to Set index to the same
  //as trace buffer.
  void SetIndex(int i);

  //Update Memory at time
  void WriteMemory(int time, bool knownFromHere,
              uint16_t addr, uint16_t value);

  //Update Register at time
  void WriteRegister(int time, bool knownFromHere,
              uint16_t addr, uint16_t value);

  //Updates &high and &low to the the value stored in
  //16 bit pair, splitted to a high and low big endian address pair.
  void SplitPair(long pair, uint8_t &high, uint8_t &low);

  //Returns the 16 bit value stored in the two 8 bit big
  //endian high and low.
  uint16_t HighAndLowToPair(short high, short low);

  //Returns Memory Item from the AmazingTraceMap
  //Register at address addr and the time index
  atEntity GetKnownRegister(uint16_t addr);

  //Returns Memory Item from the AmazingTraceMap
  //Register at address addr and the time index
  atEntity GetKnownMemory(uint16_t addr);

  //Returns the latest write to Register addr
  atEntity GetLatestWriteToRegisterAddress(uint16_t addr);
private:
  AmazingTraceMap atMemory;
  AmazingTraceMap atRegister;
  int index;
};
```

The final datastructure, a map containing another map for each memory address that is accessed by the trace buffer.

A.2.5 Read From the Amazing Trace Data Structure

```
atEntity
AmazingTraceMemoryMap::
```

```
GetKnownRegister(uint16_t addr)
{
  atEntity entity = {0,false};
  if(atRegister.find(addr) != atRegister.end())
  {
    AmazingTraceMemoryItem &m = atRegister.find(addr)->second;
    int mSize = m.size();
    int mBegin = m.begin()->first;

    if(mSize!=0 && mBegin<=index)
    {
      std::pair<AmazingTraceMemoryItem::iterator,
      AmazingTraceMemoryItem::iterator> p = m.equal_range(index);
      if(p.first == p.second)
      {
        p.first--;
        entity = p.first->second;
      }
      else
      {
        entity = p.first->second;
      }
    }
  }

  return entity;
}
```

This is a member function in the AmazingTraceMemoryMap that returns the value of a address in the register stored in the Amazing Trace data structure. The GetKnownMemory function works exactly the same but reads from the member Memory instead of from the member Register.

A.2.6 Instruction Descriptor Table For Generating the Opcode Table

```
InstructionDescriptor i78k0tab[] = {
  { "10110pp1", &push_rp },
  { "10110pp0", &pop_rp },
  { "00010pp0", &movw_rp_word },
  { "11111010", &br_$addr16 },
  { "11000pp0", &movw_ax_rp },
  { "01111101", &xor_a_byte },
  { "11011010", &subw_ax_word },
  { "10001101", &bc_$addr16 },
  { "10001110", &mov_a_addr16 },
  { "01110rrr", &mov_r_a },
  { "11001010", &addw_ax_word },
  { "11010pp0", &movw_rp_ax },
  { "01100rrr", &mov_a_r },
  { "10010101", &mov_de_a },
  { "01000rrr", &inc_r },
  { "10011110", &mov_addr16_a },
  { "10000pp0", &incw_rp },
  { "10101111", &ret },
  { "10011010", &call_addr16 },
  { "00000000", &nop },
  { "01111011", &di },
  { "10101101", &bz_$addr16 },
  { "10010111", &mov_hl_a },
  { "10001010", &dbnz_c_$addr16 },
  { "10001011", &dbnz_b_$addr16 },
```

```
  { "11101110", &saddrp_word },
  { "10000111", &mov_a_hl },
  { "00000011", &movw_addr16_ax },
  { "00000010", &movw_ax_addr16 },
  { "10011101", &bnc_$addr16 },
  { "11101010", &cmpw_ax_word },
  { "10101110", &mov_a_hl_byte },
  { "10001001", &movw_ax_saddrp },
  { "10111110", &mov_hl_byte_a },
  { "10000101", &mov_a_de },
  { "00100010", &push_psw },
  { "00100011", &pop_psw },
  { "11100pp0", &xchw_ax_rp },
  { "10101011", &mov_a_hl_b },
  { "00110rrr", &xch_a_r},
  { "10100rrr", &mov_r_byte},
  { "10010pp0", &decw_rp},
  { "01100001", &dbu_01100001 }, //OBS! must be after mov_a_r,
  { "00110001", &dbu_00110001 }, //OBS! must be after xch_a_r,
  { 0, 0 }
};
```

A smaller table with the opcode string and a instance of the instruction class for generating the final opcode table implemented as a 256 long array.

A.2.7 Table Generator

```
class TableGenerator
{
public:
  //Generate Decode Tabel
  void Generate(DecodeTable &dtab);

  //Translate an 8 bit string c to an int array of length 4
  //containing all possible kombination of the pp in the string.
  void ExtendByteStringPPToInt(const char *c, int t1[]);

  //Translate an 8 bit string c to an int array of length 8
  //containing all possible kombination of the rrr in the string.
  void ExtendByteStringRRRToInt(const char *c,int t2[]);

  //Returns an 8 byte string c as an integer where every bit
  //exept '1' is zero
  int ByteStringToInt(const char *c);

  //returns the char that c contains, 'p' or 'r'
  char ContainsChar(const char *c);
};
```

Generates a decode table that is used to store a DbuInstruction at the position in the table that has the sama number as it's opcode. for example 00010010 (00010$p_1p_0$0 MOV rp,#word) gets the number 18 in the table generated. The p_1p_0 translates to 01 in this instruction. The register pair moved in this case is BC, see Figure 1.