Binary-to-Binary Translation

Final Report

University of Texas at Austin

Department of Electrical and Computer Engineering

Juan Rubio

Wade Schwartzkopf

May 8, 1998

I. INTRODUCTION 4

II. HISTORY 4

III. DIFFERENCES BETWEEN THE C5X AND C54X PROCESSORS 5

IV. INSTRUCTION TRANSLATION 6

A. SHIFTING ADDRESSES 7

B. PIPELINING 8

C. CONTEXT OF REPEAT INSTRUCTIONS 8

D. "UNTRANSLATABLE" INSTRUCTIONS 8

V. COMPUTER IMPLEMENTATION 9

VI. CONCLUSIONS 10

VII. REFERENCES 11

Binary-to-Binary Translation

Juan Rubio and Wade Schwartzkopf

University of Texas at Austin

Department of Electrical and Computer Engineering

Abstract – There is much code presently written for the Texas Instruments TMS320C5x line of processors. However, the Texas Instruments TMS320C54x processors have surpassed the C5x line in many ways. It would be very useful if one could take the programs written on a C5x and use them on a C54x processor. In this way, one could take advantage of the capabilities of the newer processor without having to spend time and money redeveloping programs already existing on the older platform. But this is not possible because C5x binaries are incompatible with the C54x. In order to use C5x programs on a C54x processor, those programs now must be rewritten. It was our goal to develop the framework for a binary-to-binary translator that can automatically convert binary code from the C5x to the C54x. We have now created a translator that can translate a useful subset of C5x code into C54x code that accurately emulates the behavior of the original code. More importantly, we have built this translator so that it is flexible enough so that other can easily add translations for other instructions and structure of functions.

I. Introduction

Binary translation is the process of directly converting machine code from one processor into code that runs on another. This is often useful when one wants to transfer code from an old machine to a newer one. This is the case with the C5x and C54x processors. The C5x is commonly used, but the C54x family is newer, faster and has more capabilities than the C5x family, especially in the area of wireless communications. Therefore, it would be very useful if one could take the programs running on C5x processors and translate them to use them on the newer C54x processors. If one could do this, he could take advantages of the capabilities of a newer processor without having to rewrite his software.

In this paper, we will briefly review the history of binary translation in Section II. In section III, we will examine the differences between the two processors. We will then discuss the interesting issues we encountered in our binary translation project in Section IV. Section V will cover the computer implementation of our translator, and in Section VI, we will state what we have accomplished and why we believe that this is important.

II. History

One of the first attempts at binary translation was Hewlett Packard's translator for the HP-3000 in 1987 [1, 2, 3]. This translator translated code from the HP-3000 into code for the HP-PA. The code for the HP3000 was fairly simple, and typically there was only one file to translate.

As computers became more complicated so did binary translation. In modern computers, it is common to have several programs that work together. Timing intricacies, parallel processing, exception handling, interactions with operating systems,

and read-write ordering problems have also made the task of binary translation more difficult. The first major successful binary translator to deal with these problems was Digital's VEST (VAX Environment Software Translator) that translated VAX code into code for their new Alpha AXP computers [1]. VEST was completed in 1993, and Digital has since proven themselves the leader in the area of binary translation. Since VEST, Digital has built Freeport express, which translates from SPARC to Alpha, and FX!32, which translates x86 to Alpha.

Many other notable attempts at binary translation have also been made [3], but to date, there have been few if any successful attempts at binary translation for digital signal processors. Translation in digital signal processors is a special challenge because of all the instructions that are architecturally specific. In general purpose processors, translation can be much simpler. Code can often be written for these processors without knowing anything about the architecture of the processors. There is no need to translate any architectural differences. But digital signal processors often have registers that are allocated for a specific purpose. There are registers for shifting, registers for special addressing modes, registers for storing products, as well as others. Instructions on digital signal processors often refer to these specific registers and other architectural structures. In translation, all these architectural differences must be simulated if they do not exist in the target processor.

III. Differences between the C5x and C54x processors

In order to understand how to translate code between these two processors, one must first examine the differences between them. There are a number of differences between these two processors both architecturally and in the instruction sets of the two processors.

The instruction sets of the C5x and C54x are vastly different. In fact, in these two instruction sets, there are no instructions that use the same opcode to accomplish the same function. There are only a few instructions that even keep the same assembly mnemonics. Thus, one cannot just disassemble a C5x program and reassemble it with a C54x assembler.

There are also a number of architectural differences between the two processors. One significant difference between the two processors is their registers. There are a number of registers in the C5x that do not exist in the C54x. Some registers in the C5x (the accumulators, TREG0, etc.) can be mapped to registers in the C54x; that is, we can use registers in the C54x to simulate the behavior of some of the C5x registers. However, for some registers (PREG0, BMAR, DBRM), no register in the C54x can emulate their behavior. These registers must be simulated in the memory of the C54x.

There are other architectural differences between the C5x and C54x as well. The C5x has a 4-stage pipeline. The C54x has a 6-stage pipeline. The multipliers in the two processors are very different as well. The C5x has a product register (PREG0) that holds the last product calculated. Such a register does not exist in the C54x. It is one of the registers that must be simulated in the memory of the C54x. There are also differences in the ALU's of the two processors. All these differences must be accounted for if one is to create an accurate translation from C5x code to C54x code.

IV. Instruction translation

It is important to note that translation can take place on many levels. There are translators that look at the structure of code and the relationships between instructions and translate using this knowledge [1]. Such translators can make useful optimizations in the code, but for the purposes of this project, most of the translation is done by looking at

the individual C5x instructions and replacing those instructions with an individual instruction or a set of instructions that emulate the behavior of the C5x instruction. (The notable exception to this is the handling of the repeat (RPT) instruction, which will be discussed later.)

During translation each source instruction falls into one of four categories. Either

1) Its translation is a single instruction with the same number of words and the same number of cycles
2) Its translation is a single instruction with the same number of words, but a different number of cycles
3) Its translation is a single instruction with a different number of words and a different number of cycles
4) Its translation is a series of instructions.

Fortunately, many of the instructions fall into first category, and only a couple fall into fourth category. The more instructions we can translate whose word length and execution time are equivalent to that of their translation, the less we will have to worry about timing and memory constraints.

Below some important issues are described that must be dealt with for an accurate instruction translation from the C5x to the C54x.

A. Shifting addresses

Because we have translated some instructions from a single word to double words and sometimes a single instruction to multiple instructions, the length of our code and the positions of instructions will change in the translated code. Because of this, the addresses

in branches and repeat instructions must be changed to account for this. For instructions with absolute addresses into program memory (branch unconditionally (B), branch conditionally (BCND), and repeat block (RPTB)), these addresses must be calculated on a second pass through the code. Branches with variable targets (BACC) cause greater problems, but we have not dealt with them here. (Translating branches with variable targets is a very difficult problem and still an open area of research in binary translation. Target addresses generally must be calculated at run time rather than translation time.)

B. Pipelining

One problem that must be considered during binary translation is the affects of the pipelines in the processors. The C5x has a 4-stage pipeline, and the C54x has a 6-stage pipeline. These could affect the behavior of the delayed branches and other delayed instructions. This was a concern when we initially looked into C5x to C54x translation. However, Texas Instruments has created the pipelines so that in both processors two words are fetched and executed after a delayed branch.

C. Context of repeat instructions

The translator currently looks at whether or not a particular instruction is preceded by a repeat instruction. This is important because some instructions change their behavior when preceded by a repeat instruction. Thus, they will have different translations depending on whether or not they are preceded by a repeat instruction.

D. "Untranslatable" instructions

There are instructions in the C5x whose functionality does not exist in the C54x. These are instructions like the table read (TBLR) instruction, which references program memory with a variable address (in the TBLR instruction program memory is referenced

with the address in the accumulator). There is no instruction in the C54x instruction set that can reference program memory with anything but absolute addressing. This means that, in the C54x, addresses into program memory generally cannot be calculated at run time like they can in the C5x, but rather must be determined at the time the program is written.

In order to work around this problem of "untranslatable" instructions, we have translated these instructions into series of instructions that are self-modifying. We use a C54x instruction that reads from a location in program memory addressed by an absolute operand. We then write over this absolute operand with the contents of the lower word of the accumulator and then execute the instruction. By this method of self-modifying code, we can emulate the behavior of TBLR and other similar instructions such as block move (BLDP) and table write (TBLW).

V. Computer Implementation

The translation program is written entirely in ANSI C and was tested under Solaris 4.2 and Windows 95. It reads a standard ASCII HEX file. The translator is implemented using a table of templates indexed by the corresponding opcode, and a set of eight parsing functions. If an instruction is found in the table, the translator replaces the instruction with the corresponding output code that the table has for that instruction. If the instruction is not found in the table or translation is not possible for some other reason, the translator delivers a message that is intended to help the user translate the code using a different methodology. This table look-up algorithm is simple and runs quickly. Running sample code on a Pentium II processor, we translated 40,000 instructions per second.

The translator then returns output code in C54x assembly. This is so that the user can make any changes he feels necessary to the code. This is very useful when there has been a section of code that the translator cannot translate. When unfamiliar code is encountered, the translator will mark this section is so that user can translate these sections manually. To get the final C54x binary, the user simply needs to assemble the code.

VI. Conclusions

We have built a useful tool that translates a subset of C5x instructions into C54x instructions. To the best of our knowledge, this is the first significant attempt at binary-to-binary translation for two digital signal processors. Our translator can translate all the instructions used by the C5x C compiler (see Table 1). We have also included several other instructions which are not used by the C compiler, but which we thought were important instructions. These instructions were the conditional branch (BCND, BC), the multiply-accumulate (MAC), control bit set (SETC) and clear (CLRC), and the repeat (RPT) instruction. In all, we have translated 64 of the 133 C5x instructions to run on the C54x. This in itself is significant, but our greater contribution is the framework we have built for translation between the two processors. Because the C code of the translator uses a simple table to define translations, it is simple to add new instructions and modify current ones.

ABS	ADD	ADDB	ADDS	ADRK	AND	ANDB	APAC	APL	B	BACC	BANZ
BIT	BLDD	BSAR	CALA	CALL	CMPL	IN	LACB	LACT	LAMM	LAR	LMMR
LT	MAR	MPY	MPYU	NEG	NOP	OPL	OR	ORB	OUT	PAC	PSHD
RET	RPTB	SACB	SACH	SACL	SAMM	SAR	SATH	SATL	SBB	SBRK	SFL
SFR	SPAC	SPH	SPL	SPLK	SUB	SUBS	TBLR	XOR	XORB	XPL	

Table 1: Instructions use by the C5x compiler

VII. References

1. R. Sites, A. Chernoff, M. Kirk, M. Marks, and S. Robinson, “Binary Translation”, *Comm. of the ACM*, vol. 36, pp. 69-81, Feb. 1993.
2. A. Bergh, K. Keilman, D. Magenheimer, and J. Miller, “HP 3000 Emulation on HP Precision Architecture Computers,” *Hewlett-Packard Journal*, pp. 87-89, Dec. 1987.
3. C. Cifuentes and V. Malhotra, “Binary Translation: Static, Dynamic, Retargetable?”, *Proc. Int. Conf. on Software Maintenance*, IEEE-CS Press, Monterey, CA, pp. 340-349, Nov. 1996.
4. V. Zivojnovic, S. Tjiang, and H. Meyr, “Compiled Simulation of Programmable DSP Architectures”, *Proc. IEEE Workshop on VLSI Signal Processing*, Osaka, Japan, pp. 187-196, Oct. 1995.
5. C. Mills, S. Ahalt, and J. Fowler, “Compiled Instruction Set Simulation”, *Software Practice and Experience*, vol. 21(8), pp. 877-889, Aug. 1991.
6. V. Zivojnovic, S. Pees, and H. Meyr, “LISA – Machine Description Language and Generic Machine Model for HW/SW Co-Design”, *Proc. IEEE Workshop on VLSI Signal Processing*, San Francisco, California, pp. 127-136, Oct. 1996.
7. V. Zivojnovic and H. Meyr, “Compiled HW/SW Co-Simulation”, *Proc. Design Automation*, Las Vegas, Nevada, pp. 690-695, June 1995.
8. C. Cifuentes and S. Sendall, “Specifying the Semantics of Machine Instructions”, Technical Report 422, Department of Computer Science, The University of Queensland, Dec. 1997.