What assembly language programmers get up to: control flow challenges in reverse compilation

Adrian Johnstone
A.Johnstone@rhbnc.ac.uk

Elizabeth Scott
E.Scott@rhbnc.ac.uk

Tim Womack
T.Womack@rhbnc.ac.uk

Department of Computer Science, Royal Holloway, University of London, Egham, Surrey, TW20 0EX, UK.
Tel: +44 (0) 1784 443425. Fax: +44 (0) 1784 439786

Abstract

The analysis of assembly code to provide a high level control flow view in terms of the usual high level looping and selection constructs is of great assistance to high level language programmers who are attempting to understand and port low-level code as part of a system re-engineering project. This paper describes the control flow analyser component of our `asm21toc` reverse compiler from assembly language programs for the ADSP-21xx family of Digital Signal Processors to ANSI-C. We give a brief overview of the class of processors and programs that we have studied so as to motivate the design of our reverse compiler. We describe the merged call-graph/dataflow representation that supports our analyses and the way in which hierarchical structural control flow information is extracted and stored. We give summary statistics showing the usage of various classes of control flow structure along with occurrences of non-disjoint functions, self modifying code and non-reducible control flow constructs.

Keywords: reverse compilation;
interprocedural control flow analysis;
low to high level language translation

1 Introduction

Early designs for programming languages and architectures were informed by the designer's opinion of what programmers did. By the late 1960's researchers had begun to measure what real programmers do. In 1971 Knuth published a widely cited empirical study of FORTRAN programs [Knu71] in which he comments in passing that, in a randomly selected sample of 50 programs analysed, none contained *non-reducible* control flow structures when processed by the Cocke interval reduction scheme [AC76]. A reducible structure is one in which all control flow paths enter *via* a single node and exit to a single successor node. The well known structured programming constructs such as while-do and if-then-else correspond to reducible structures of this type: we give examples of non-reducible structures (which may be constructed using go-to statements) below.

In Knuth's sample, complete reduction of each subroutine was achieved in six or less transformations and on average only 2.75 transformations were required. This observation strongly supports the almost universal use of single-entry/single-exit node control flow structures as the basis of modern structured programming languages. It also shows that nesting depths of control structures were typically quite small for FORTRAN programs. Other similar studies have shown that irreducible control flow structures are very rare in high level programs even when they are rich in go-to statements [Muc97, pp196-197]. Nesting levels seem to be quite low even for programs using structured control flow which might be thought to encourage deep nesting. Cook and Lee found that for Pascal programs sub-program nestings greater than one were very rare and gave limited statistics for the frequency of nesting between different classes of control flow statement [CL82]. (ANSI-C does not allow any sub-program nesting.)

Our interest in this old debate arises from a project to *reverse*-compile assembler code for a small and highly constrained processor into ANSI-C. Our goal is to render assembly programs as equivalent C programs so as to aid the porting of code to new architectures. A key part of this process is the recovery of high level control statement semantics from the conditional and absolute jumps present in the assembler code so as to aid program comprehension. Non-reducible control flow patterns at assembler level must be rendered in C using go-to statements. Although the studies referred to in the previous paragraph indicate that such constructs would be rare, we have found assembler programs in which as many as 21% of the functions in an application contain non-reducible control flow. The incidence of such constructs is very dependent on the programmer concerned, and in our samples is clearly a reaction to resource constraints: large applications that use almost all of the available memory are more likely to show 'tricky' programming techniques.

We use the term 'reverse compilation' in preference to the more common 'de-compilation' to emphasise that we are starting with assembler produced directly by programmers as opposed to code generated by compilers. It is a reasonable (although in detail naïve) to assume that machine level code generated by a non-optimising compiler will only contain control flow structures that have direct equivalents in the high level language. Most other efforts in this area concern attempts to recover source from compiled executables for which the original source is not available. We consider this to be a hard task with potentially little return given the difficulty of comprehending programs that have neither comments nor suggestive variable and function names. Cifuentes' work is perhaps the most developed example of this kind of tool [CG95] and her thesis gives a useful survey of other tools [Cif94].

We have reported on some static and dynamic dataflow effects handled by our reverse compiler elsewhere [JSW99, JSW00]. Those papers also give some detail on the design of our target processor and the market trends that have made reverse compilation attractive to users. In this paper we give only a brief overview of the class of processors and programs that we have studied so as to motivate the design of our reverse compiler. Our concern here is with two classes of control flow problem that arise in practice: (i) non-reducible control flow, (ii) functions that share code (non-disjoint functions).

In what follows we describe the merged call-graph/dataflow representation that supports our analyses and the way in which hierarchical structural control flow information is extracted and stored. We give summary statistics showing the usage of various classes of control flow structure along with occurrences of non-disjoint functions, self modifying code and non-reducible control flow constructs, and we conclude that cloning of code is a viable approach to resolving these difficulties.

2 Problem domain

The initial stimulus for this project came from users of Digital Signal Processors (DSPs) who had significant intellectual property in the form of assembler source that they wished to migrate to new architectures. DSPs are highly specialised microprocessors with architectures optimised for arithmetic-intensive applications. In particular, DSPs must be able to execute the inner 'butterfly' operation of the Fast Fourier Transform algorithm at high speed so as to allow realtime signal conditioning and analysis: tasks that have traditionally been implemented using analogue electronics. To this end, DSPs usually contain fully parallel (single-cycle) multipliers, barrel shifters, multiple operand spaces and mechanisms for supporting counted loops with the counter increment and test operations being performed in parallel with the main calculation.

In addition to these high performance characteristics, DSPs must be cheap. They are usually used in embedded systems such as mobile telephones in which market acceptance depends critically on low cost and low power consumption. The twin demands of high arithmetic rates and low cost compared to a general purpose desk-top processor require some compromises, and the usual trade-off selected is to severely limit the virtual address space of the DSP. A side effect of this is that the C compiler supplied for such devices produce code which is very poor compared to that produced by assembly language programmers and as a result very few systems are coded in C.

The particular DSP architecture investigated here is the Analog Devices ADSP-21xx family of devices [AD90]. They have a 16-bit datapath and a 24-bit program word, with data and program being held in separate memory spaces. It is possible to read and write program memory, but there is only a maximum of 16K words in each memory space. As a result, programs may only directly address 16K instructions. More recent versions of the processor have some facility for overlay blocks, but these processors clearly have very restricted program spaces even by the standards of 1970's 8-bit microprocessors.

The small memory size has been acceptable on these devices because the very high operand fetch rates found in DSP applications mandate the use of on-chip memory or very high speed (and therefore expensive) off-chip static RAMs. The introduction of new memory technologies and the availability of low cost dies with large memory arrays has led to the introduction of new DSP devices which provide both large on-chip memories and high degrees of parallelism in the computational units. In particular, the TMS320C6x series of devices [TI99] is designed as a next generation DSP but is radically harder to program than traditional DSPs. The TMS320C6x is a VLIW processor in which eight computational units share non-interlocked access to two banks of registers *via* pipelines. There is strong anecdotal evidence that programmers are very poor at reasoning about time dependencies in pipelined systems and the experience of our users is that low level programming of these new devices is not cost-effective.

This change in the DSP market is creating an acute situation: in many ways DSP applications have been the last bastion of assembly level programming but the new processors are almost too difficult to program in assembly language. Traditional DSP programmers are suspicious of potential performance overheads when using high level languages, and high level language programmers find DSP assembly code hard to read. Our goal is to provide translations from assembly to ANSI-C that preserve the assembly semantics and follow the structure of the assembly programs as closely as possible.

3 Sample applications

The statistics presented in this paper have been collected using code drawn from the following three sources.

1. A company producing real-time image processing systems based around uni- and multi-processor configurations of the ADSP-21xx processors. Four different architectures are represented. In each case the application comprises a set of general purpose image processing functions based around small window kernels,
2. A company in the cellular telecommunications sector that uses ADSP-21xx processors to implement modulation schemes and data protocols for mobile telephony. Five different applications are included in the sample.
3. The supplier of the ADSP-21xx processors, who publish source code examples on their Web site.

We refer to these samples as the image processing (IP), telecoms (TC) and manufacturer's example (ME) sets. We expected the IP set to display the most 'tricks': each of the IP applications is very severely constrained by available memory and has also been extensively tuned for speed. The TC applications are much less tightly coded, and the ME functions are intended to have pedagogic value, so we would be surprised if the control flow were to be tangled.

Table 1 gives summary statistics for these sets. The number of *modules* is the number of separate compilation units in the sample. After reverse compilation, each module yields one ANSI-C file. A module establishes a scope region for the ADSP-21xx assembler – all identifiers must be unique within a module[1].

ADSP-21xx assembler lines come in two categories: *directives* which include variable and constant declarations and *instructions* which generate executable code. The ADSP-21xx processors support limited programmed parallelism: it is possible to combine a calculation instruction along with either one or two memory-to-register transfers into a single *multifunction* instruction. The final two columns of table 1 show the number of such instructions and the proportion of the overall instruction count that is non-sequential. The proportion of multifunction instructions is quite low for the TC set. We believe that this reflects the reduced pressure on resources for those applications. The IP applications are highly optimised for time and space (using multifunction instructions improves both execution speed and code density), and the ME set is to some extent designed to show off the processor's parallel capability, hence the high proportion (nearly 10%) of multifunction instructions.

The observant reader might notice that only about one quarter of the lines in the IP set and one fifth of the lines in the TC set have either an instruction or a directive. This might be taken to indicate that the code is very heavily commented, but in fact it is merely *well* commented. All of these sets make extensive use of the C pre-processor to support conditional compilation and file inclusion. An artifact of the ADSP preprocessor is that it leaves large runs of blank lines in the output file. The line counts given in Table 1 are lines *after* pre-processing, not original source lines.

4 A note on implementation

Our approach to reverse compilation is to exploit known algorithms developed for parsing, analysis and optimisation in compilers. Reverse compilation is, after all, just another form of translation. Our task does differ from standard compilation in these two key areas.

1. Control flow in our programs is not neatly partitioned into inter- and intra-procedural components. Compilers typically maintain separate *call graphs* which represent the call hierarchy between functions and *flow graphs* which represent the flow of control within individual functions. As will be

[1] There exist special facilities for establishing local symbol names within macros as is usual for assemblers: however none of our applications used the built-in macro facility.

Set	Modules	Lines	Directives	Instructions	Multifunction	Multifunction/ Instructions
IP	273	242,488	13,125	48,692	3822	7.8%
TC	38	23,323	1,579	3,057	44	1.4%
ME	115	13,374	1,365	3,789	370	9.7%

Table 1: Summary statistics for code samples

shown in the next section, assembler programs often intermix calls and jumps to the same parts of the application and so a combined representation is required.

2. Speed of translation is less critical for a reverse compiler than for a compiler. The input to a reverse compiler is assumed to be a fully developed application that will be translated once to the high level. A compiler on the other hand is used in a development loop, and generally compile times of more than a few tens of seconds per compilation unit will be unacceptable. Our reverse compiler typically takes a few minutes to process a realistically sized application because we make use of brute-force methods to perform exact analyses which provide improved translation. (In fact, many of our algorithms are exponential in the worst case although we find in practice that translation times are not much more than linear.)

The starting point for the analyses described in this paper is a hybrid intermediate form constructed as follows. We write a parser for the assembler source language using our `rdp` translator generator [JS98]. This features special annotations that control the construction of a *Reduced Derivation Tree* which is an abstract syntax tree formed from the full derivation using a small set of formally defined transformations. At this point we have a rather flat tree in which each expression in the assembler source generates a sub-tree attached to the root. The nodes in this tree are called *expression nodes*. We then pass over this tree, creating one or more *control flow* nodes *per* expression which are connected so as to display the control flow through the program. Jumps are shown as edges from control flow nodes to control flow nodes, expression evaluation as edges from expression nodes to expression nodes, calls as edges from expression nodes to control flow nodes and returns as control flow nodes with no out-edges that lead to control flow nodes. Figure 1 shows a small example in which all of the expression nodes have been hidden except for the children of the node labeled `mr = mx0 * my0`.

5 Function identification

Most high level languages provide a function abstraction that allows only a single entry point, and it would perhaps be surprising if a compiler generated code that included functions with more than one entry point. To be more precise, in compiler-generated code we might expect all upwards paths from a particular return-from-subroutine statement to converge on a single call target label.

A casual examination of hand-written assembler code shows that it is not safe to assume that all functions are single entry. One common device is the *shared error handler* wherein a single error reporting routine is *jumped to* (not called) from within many functions. A terminating return-from-subroutine instruction at the end of the error handler is shared between all the error-calling functions (see Figure 2). The purpose of this device is to save the overhead of calling and returning from the error handler at the cost of one extra jump. One might speculate on whether this saving is worthwhile, given that error conditions normally imply that program flow is broken, but our purpose here is to describe actual practice rather than to propose a discipline of programming. It turns out that speed is not the key concern. When we queried the use of this kind of error handler the programmers pointed out that one of their systems had only 1,024 program words in which a kernel had to reside, and the shared error handler saved one program word (a return from subroutine instruction) *per* call to the error handler.

We suspect that the single entry assumption might turn out to be simplistic even for compiler-generated code when considering highly optimising compilers for languages that include exception handling, but we also consider reverse compilation of such code to be unlikely to generate useful (that is easily comprehensible) code and know of no attempts to write such a tool, although debuggers for optimising compilers face similar challenges when attempting to produce a readable display of the run-time program and data structures. Of

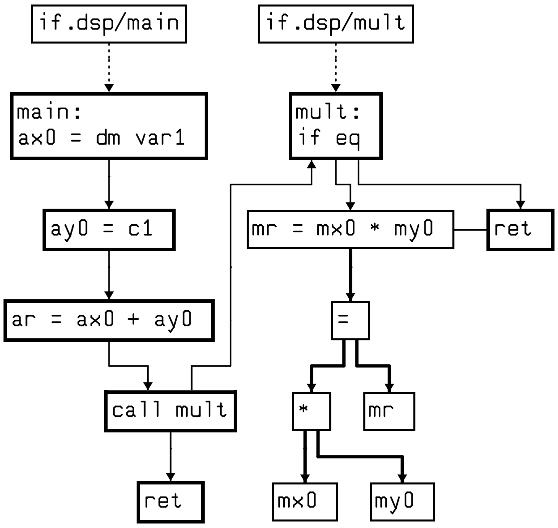

Figure 1: Intermediate form

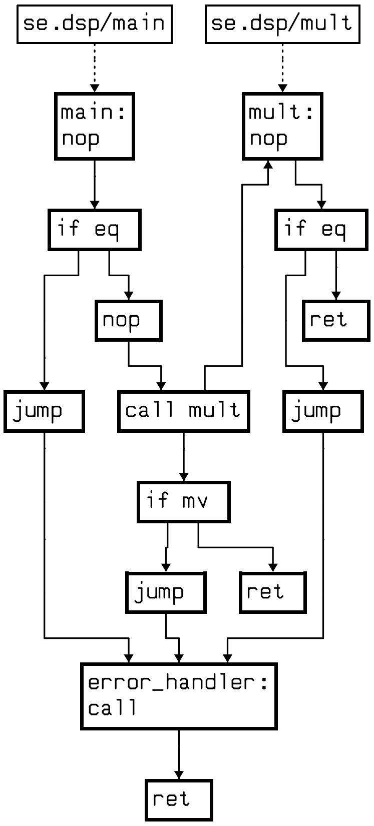

Figure 2: A shared error handler

Set	Modules	Functions	Share points
IP	273	264	26
TC	38	213	1
ME	115	110	8

Table 2: Behaviour of functions

course, a debugger is usually designed in tandem with the compiler and so hints can be left in the executable to help the debugger untangle code-shifting optimisations: a decompiler can expect no such assistance.

We investigated the use of the function call mechanism in our samples. Our internal representation denotes calls as an edge from an expression node to a control flow node. The reverse compiler constructs a table of functions by locating every control flow node that is the target of such an edge and then performing a depth first search of the control flow nodes in the intermediate representation. Return nodes have no out edges to control nodes, so this search automatically terminates at the end of a function. A list of return points is attached to the symbol table entry for each function.

After function entry and return points have been built, a further pass attempts to uniquely label every control flow node with a pointer to its function entry point. Functions that share code (or equivalently, a function with two entry points) will result in nodes with more than one such label. Such nodes are called *share points*. Depth first searching of the control flow nodes ceases at share points, so no further analysis is performed. There may be two or more functions with control flow paths through a particular node: in the case of the shared error handler the first node of the error handler will be shared between all functions using the error handler.

Table 2 shows the results of our function analyses. The number of functions shown is the number of control flow nodes with incoming edges from expression nodes, as described above. In each set, some modules contain only declarations which accounts for the lower number of functions than modules in the ME set.

We term the set of functions calling a share point the *share set*. The majority of share points involved only two functions. The only instances of share sets with more than two members were shared error handlers.

When performing translation to ANSI-C we in general try to keep as close to the original assembler as possible. In the case of share points, however, we are forced to rework the code. ANSI-C does not allow general jumps between functions since the `goto` statement in C is restricted to jumps within a function. The `setjmp` and `longjmp` system calls provide a limited facility for inter-procedural jumps in which a call to `setjmp` executed within a function saves the control flow state in a named block that may be activated by a subsequent `longjmp`. The effect of this is to allow a program to globally jump to somewhere it has already visited but to not allow jumps to code which has not already been executed. Note that this is *not* the same as disallowing forward jumps! Nevertheless, it does not constitute a sufficiently general mechanism to allow direct implementation of inter-functional jumps in the assembly source. In our reverse compiler we *clone* share functions by making a new copy of the code in the shared part for each function in the share set. This has the effect of increasing code size and also has implications for functions with local variables which would then need to be shared between the clones. In fact our examples have no examples of share points that cross module boundaries. Since all share sets contain functions that are in the same scope region, there can be no local variable conflicts.

6 Control structure identification

A variety of techniques have been described in the literature for transforming flow graphs by locating subgraphs corresponding to particular control flow constructs and replacing them with a super-node, which may itself then take place in further transformations. Since the number of nodes in the graph gets smaller at each step the transformations are usually called *reductions*. The goal is to reduce the flow graph for each function to a single node. For a given set of transformations, a flow graph is said to be *reducible* if it may be transformed to a single node and *irreducible* if not.

Knuth's paper on FORTRAN statistics used Cocke's procedure [AC76] which was able to completely reduce all the programs in his sample. Sharir's algorithm [Sha80] is more powerful in two ways: firstly it can detect a wider range of control structures and secondly when it encounters a non-reducible node it generates a supernode that contains the smallest non-reducible region surrounding that node. In this way, Sharir's algorithm is able to 'recover' from local irreducibility and continue the hierarchical reduction, rather than simply abandoning function reduction as soon as an irreducible node is encountered.

Reductions are of two basic kinds: the reduction of sequences and self loops to single nodes and the reduction of regions around conditional statements that have more than one control flow out-edge. In Table 3 we show the frequency of instructions that may generate control structures in this second class. In ADSP-21xx assembly source most instructions may be preceded with an optional condition which must be true for the instruction to be executed. Branching is implemented in this way using conditional jumps. In addition, the ADSP-21xx provides a special `do-label-until condition` instruction that allows *zero-overhead* looping. The condition (which might be a comparison to a counter value) and the target label are loaded into a stack of special hardware registers and then whenever the program counter equals the value in the counter a conditional jump is automatically executed. This saves instruction fetch-execute cycles and program space. These loops may only be nested up to four levels deep because the stack of branch/condition registers is only four deep.

The ADSP-21xx, in common with other processors, allows the target of a jump or call to be held in a register: a so-called *indirect jump*. Such instructions are potentially the source of great difficulty when reverse compiling because they allow the run-time computation of a control flow target. We intend in the future to implement a program slicing algorithm to allow static calculation of the set of potential indirect jump targets. In the present reverse-compiler we require programmer annotations in the assembler source to mark the potential targets. Fortunately, as shown in the table, indirect jumps and calls are quite rare, and where they do occur they normally take the form of `switch` statements that can be easily understood by the user.

Our reverse compiler recursively searches the control flow graph for structures that correspond to the familiar `if-then`, `if-then-else`, `while-do` and `do-while` statements along with a variety of self-loops. Our merged intermediate representation includes function returns, and these along with nested self-loops act so as to create control flow nodes and super-nodes from which control never returns to the function being analysed. We call these nodes *sink nodes* and our reduction algorithm recognises them in a separate phase.

In detail, our reduction procedure also distinguishes between 'left' and 'right' handed control constructs: the standard `if-then` and its pair `if-else` for instance, the difference being that the body of the conditional statement may be attached to either the TRUE or FALSE out edges. When rendered as ANSI-C, we invert the sense of the condition for `if-else` statements and output as an `if-not-then`. Loops are treated similarly. For clarity, we have conflated these asymmetric structures into single classes in the statistics.

Table 4 shows the number of functions containing irreducible structures in each of our samples. Such functions are unfortunately not rare in optimised handwritten assembly code. Techniques for dealing with non-reducible functions fall into three categories:

1. do nothing: render the function using `goto` statements and labels;

2. clone the well-behaved part of the structure making new copies for each exit node, by analogy with our algorithm for shared functions (this procedure is called *node-splitting* in the literature);

3. improve the recognition power of the reduction algorithm.

At present we use only option 1: we justify this in terms of keeping the translated code as close to the original assembler as possible so as to ease the understanding of ported code. Option 2 is attractive as a general mechanism: in the worst case node splitting can cause an exponential increase in code size but examination of our sample programs show that such explosions will not occur in this case. Option 3 merits further research: the ANSI-C `break` within a loop statement produces control flow diagrams that meet the 'reducibility' criterion of a single entry and single exit node. However, our reduction algorithm at present can not detect such flow, and so if an assembler programmer builds such a structure we will render it using `goto`'s rather than using the break statement. We have some anecdotal evidence (although no firm statistics) that a significant proportion of non-reducible functions contain loops that effectively contain `break` or `continue` statements.

Figure 3 shows three examples of non-reducible structures detected in our samples that are candidates for translation using each of these three options.

The first is a genuinely tangled pair of loops in which multiple exit conditions exist. Whilst illustrating an impressive ability to optimise control flow, the structure is hardly natural, especially for a high level language programmer, but we feel that rendering using `goto` is less likely to cause confusion than node splitting. The second example shows two `if` statements that share a body, and this is a classic candidate for node-splitting. The final example shows a `do-while` loop containing a `break` statement.

Table 5 gives the distributions of control constructs and their nestings in our samples. We saw no nestings deeper than six. Of the 1,125 structures detected,

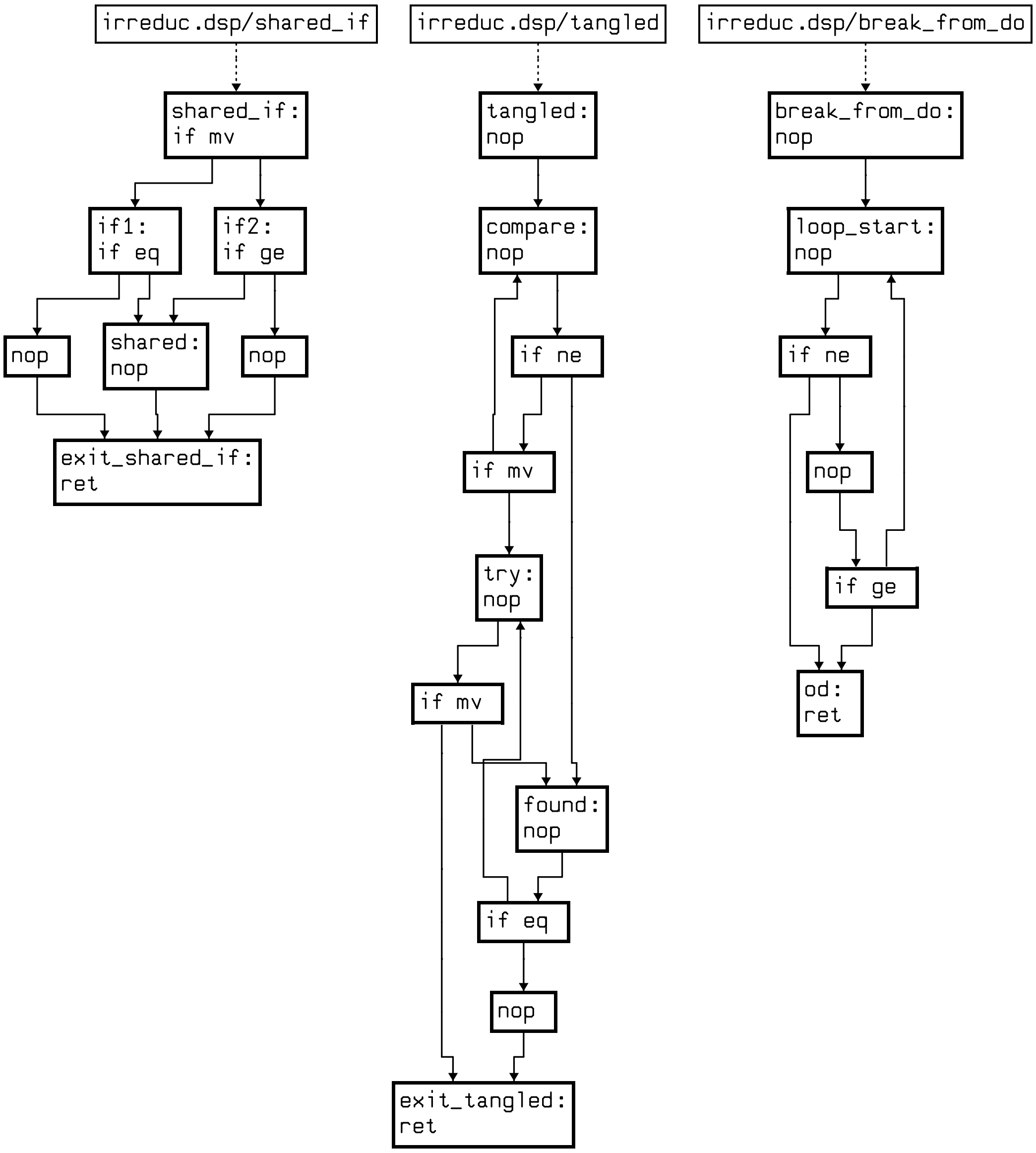

Figure 3: Three irreducible structures

Set	Instructions	Conditionals	Do statements	Indirect calls	Indirect jumps
IP	3,057	300	46	1	0
TC	48,692	3,114	1,602	59	60
ME	3,789	169	45	16	4

Table 3: Distribution of intra-functional control flow changing instructions

Set	Functions	Irreducible functions	Proportion irreducible
IP	264	57	21.6%
TC	213	3	1.4%
ME	110	18	16.4%

Table 4: Proportion of irreducible functions

58.76% were level 1 (the outermost level), 28.6% were level 2, 7.5% were level 3 and 3.9% were level 4. Our average nesting was only 1.6 which is significantly lower than Knuth's figure of 2.75 for FORTRAN programs. This indicates that programmers are more conservative in their use of control flow constructs when they need to keep track of the high level flow themselves rather than having syntactic support from their language. This is not a surprising result.

The distribution of loops in table 5 bears some explanation. Initially we considered the complete absence of `while-do` loops to indicate an error in our analyser. However, we wrote a toy compiler for a subset of ANSI-C to ADSP-21xx that implemented loops and statements using conditional jumps and labels. The output of our reverse compiler matched the input C program, and indeed `while-do` statements were counted correctly. The main reason ADSP-21xx programmers eschew the `while-do` statement is that processor has a built-in `do-until` instruction that allows extremely efficient looping. Our reverse-compiler front end translates all such instructions into `while-not-do` structures. No nestings greater than four were observed for these loops: recall that the `do-until` hardware only supports a maximum of four levels. These loop statistics, then, are essentially an artifact of the ADSP-21xx's idiosyncratic architecture and should in no way be taken as representative of assembly language programs in general.

7 Summary

We have studied three kinds of assembly language programs: heavily optimised real-time image processing code, lightly optimised telecommunications code and example functions designed to show off the abilities of the processor. We have shown that the incidence of non-reducible control structures in these kinds of applications is much higher than indicated by previous studies using high level languages, and we have described approaches to the rendering of such structures. We have also examined nesting depths for reducible structures and concluded that assembly language programs are rather flatter than even FORTRAN programs rich in `goto` statements. We have also found that the function call structure of these programs is more complex than allowed by most high level languages, but we have demonstrated that function cloning may be used to successfully preserve low level semantics when translating to a high level language.

Acknowledgements

The authors would like to acknowledge the support of the directors of Image Industries Ltd who provided much of the source code used in our statistical sample, and Georg Sänder whose VCG tool was used to produce Figures 1 and 3.

References

[AC76] F.E. Allen and J. Cocke. A program data flow analysis procedure. *Communications of the ACM*, 19(3):137–147, March 1976.

[AD90] *ADSP 2101 User's manual (architecture)*. Analog Devices, 1990.

[CG95] C. Cifuentes and K. J. Gough. Decompilation of binary programs. *Software — Prac-*

Set	self loop				if-then						if-then-else						while-do				do-while			
	1	2	3	4	1	2	3	4	5	6	1	2	3	4	5	6	1	2	3	4	1	2	3	4
IP	44	4	4	3	99	124	49	24	12	1	14	14	6	0	1	0	0	0	0	0	144	59	11	9
TC	16	0	0	0	236	58	6	0	0	0	2	0	0	0	0	0	0	0	0	0	26	6	0	0
ME	11	0	2	0	44	47	5	7	0	0	5	3	0	0	0	0	0	0	0	0	20	7	1	1

Table 5: Nesting of structures

tice and Experience, 25(7):811–829, July 1995.

[Cif94] Cristina Cifuentes. *Reverse compilation techniques*. PhD thesis, Queensland University of Technology, July 1994.

[CL82] Robert P. Cook and Insup Lee. A contextual analysis of Pascal programs. *Software — Practice and Experience*, 12:195–203, 1982.

[JS98] Adrian Johnstone and Elizabeth Scott. `rdp` – an iterator based recursive descent parser generator with tree promotion operators. *SIGPLAN notices*, 33(9), September 1998.

[JSW99] Adrian Johnstone, Elizabeth Scott, and Tim Womack. Reverse compilation of Digital Signal Processor assembler source to ANSI-C. In *Proceedings of IEEE International Conference on Software Maintenance*, pages 319–325. IEEE Computer Society, 1999.

[JSW00] Adrian Johnstone, Elizabeth Scott, and Tim Womack. Reverse compilation for Digtial Signal Processors: a working example. In *Proceedings of the 33rd Hawaii International Conference on System Sciences*. IEEE Computer Society, 2000.

[Knu71] Donald E. Knuth. An empirical study of FORTRAN programs. *Software — Practice and Experience*, 1:105–133, 1971.

[Muc97] Steven S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann, San Francisco, California, 1997.

[Sha80] Micha Sharir. Structural analysis: a new approach to flow analysis in optimising compilers. *Computer Languages*, 5(3/4):141–153, 1980.

[TI99] *TMS320C6000 CPU and instruction reference guide*. Texas Instruments, 1999.