A Tool for Formal Verification of DSP Assembly Language Programs

by

David W. Currie

B.S., McGill University, 1997

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

THE FACULTY OF GRADUATE STUDIES

(Department of Computer Science)

we accept this thesis as conforming
to the required standard

The University of British Columbia

August 1999

Abstract

Formal verification has, in recent years, become widely used in the design and implementation of large integrated circuits, but its use in general software verification has been more limited. We have developed a new technique to verify assembly code for digital signal processors (DSPs) that makes significant steps into the realm of software verification and serves as a good building block for future verification efforts. In order to demonstrate the applicability of our approach, which takes inspiration from successful techniques applied in hardware verification, we have implemented a prototype tool to verify assembly code for a Fujitsu DSP chip. The approach we have created is based on symbolic simulation with uninterpreted functions and control flow analysis.

DSP assembly language programs are an attractive target for formal verification. On one hand, DSP assembly language programs must often be modified for size and speed constraints which requires that the code be optimized by taking advantage of the idiosyncrasies of the chip. This optimization can make even small programs hard to reason about and debug. On the other hand, verification of optimized versus unoptimized versions of the same program can be simplified by exploiting the similarities between the two. This combination produces an application domain that is simultaneously challenging yet tractable.

This thesis describes our verification approach and how we were able to successfully implement a prototype tool.

Contents

Abstract ii

Contents iii

List of Tables vii

List of Figures viii

Acknowledgements x

Dedication xi

1 Introduction 1

1.1 Research Overview 2

1.1.1 The Problem 2

1.1.2 Our Solution 4

1.1.3 Contribution 4

1.2 Overview of Thesis 5

2 Background and Related Work 6

2.1 Formal Verification Techniques 6

2.1.1 Theorem Proving 7

2.1.2 Model Checking 8

2.1.3 Equivalence Checking 9

2.2 Control Flow Analysis 11

2.3 SVC - Logic Decidability 13

2.4 Related Work 15

3 Implementation **17**

3.1 Verification Method 18

3.1.1 Summary of Program Execution 18

3.1.2 SVC Interface 19

3.2 Basic Block Analysis 22

3.2.1 Problem 22

3.2.2 Solution 23

3.3 Control Flow Analysis 26

3.3.1 Implementation 26

3.4 Context Stores and Retrieves 28

3.5 Memory 29

3.5.1 Problem 29

3.5.2 Solution 31

3.6 Axioms 33

3.6.1 Problem 33

3.6.2 Solution 34

3.7 Loops 34

3.7.1 Problem 35

3.7.2 Solution 35

3.8 Branching Conditions 36

3.9 Expression Construction and Simplification 36

3.9.1 Opcodes 37

3.9.2 Expression Simplifications 37

3.10 Execution Reports 41

3.10.1 Code Listing and CFG Listing 41
3.10.2 Execution Trace 41
3.10.3 Expression Reporting 42
3.10.4 Uninitialized Variables 42
3.11 Example Results 42

4 Industrial Examples **46**
4.1 Fujitsu DSP 47
4.2 Yhaten 47
4.3 Hup 48
4.4 Kncal 48
4.5 Dt_pow 50
4.6 Common Problems 52
4.7 Performance 52

5 Limitations and Future Work **54**
5.1 Limitations 54
5.1.1 Bit Reasoning 54
5.1.2 Modulo Addressing 55
5.1.3 Memory Simplifications 55
5.1.4 Arithmetic Problems 56
5.1.5 Axiom Handling 56
5.2 Future Work 56
5.2.1 Language Module 56
5.2.2 Interface with Decision Procedure 57
5.2.3 Loop Handling 57
5.2.4 Control Flow Graph Matching 58

6 Conclusion **59**

Bibliography **61**

List of Tables

4.1 Examples of Syntactic Corrections 52
4.2 Tool Performance on Examples 53

List of Figures

1.1 Demonstration of Program Equivalence 3

2.1 Symbolic Simulation 10

2.2 Basic Blocks Divisions 11

3.1 Example of Symbolic Simulation with Uninterpreted Functions . . . 18

3.2 Basic Algorithm for Symbolic Execution 20

3.3 Supporting Functions to Algorithm in Figure 3.2 21

3.4 Pairwise Comparison and Simplification of Equal Values 23

3.5 Tree Representation of CFG for Dt_pow 25

3.6 Example of Data Structure used in Tool 28

3.7 Read and Write Functions 29

3.8 Example of Memory Growth if Stored in a Single Location 30

3.9 Difficulties of Memory Reads and Writes 30

3.10 Memory Storage 31

3.11 Input Code for Simulation 43

3.12 Parsed Code Listing 44

3.13 Given and Desired Equivalencies 44

3.14 Control Flow Graphs 44

3.15 Example Output for Valid Trace 45

3.16 Example Output for Incompatible Branch 45

3.17 Example Output for Invalid Trace 45

4.1 Tree Representation of CFG for Kncal 49

Acknowledgements

I would like to thank my supervisor, Alan Hu, for all his invaluable support.

DAVID W. CURRIE

The University of British Columbia

August 1999

I would like to dedicate this to my parents for their never ending support through the good times and bad.

I would also like to dedicate this to my companion and best friend, Jackie Reis. You have made every moment of the last four years exciting and full.

Chapter 1

Introduction

The use of formal verification methods in the design and implementation of large integrated circuits is growing rapidly[6, 34, 5, 10]. In recent years, it has also begun to prove itself as an indispensable tool, as can be seen by the number of companies engaging in active research in the area, such as Motorola, IBM, and Intel. Delays to market can be crucial and bugs found once the chip has gone to market can be very costly, as Intel was made painfully aware with its floating point division bug.

Size and speed can often be critical to the success or failure of a hardware design, thus the original design is often modified many times (potentially by a number of people who may not have originally created it) in an effort to optimize it as much as possible. The changes can often be subtle, as they make use of the idiosyncrasies specific to that chip, making it hard to verify the correctness of the changes. On the other hand, one reason why formal methods have been so successful in handling practical problems in the field is because the modifications are subtle. By using the fact that the design has changed in only subtle ways, methods have been developed to verify large circuits which might otherwise be infeasible. These methods fall largely under the heading of *Combinational Equivalence Checking*.

Similarly, the software for digital signal processors (DSPs) often needs to be hand coded, or compiled code needs to be optimized to meet size and speed requirements. Many DSPs have specialized instructions that have multiple parallel side effects and can be highly non-orthogonal; that is the instructions operate only on certain arguments or worse, operate differently depending on the arguments given. Fully exploiting the idiosyncrasies (something compilers find hard to do) of each DSP architecture can be crucial to achieving size and speed requirements, making writing and analyzing the program difficult and error prone. For these reasons, as in hardware design, verification of assembly code for DSP architectures is a promising area for automatic formal verification.

Unfortunately, formal verification of software is difficult at best. Fortunately, even a limited but automated tool that helps confirm correctness would be useful in practice: any bug found is important but little user effort is required due to the automated nature of the tool. The benefit of such tools has been demonstrated in the verification of hardware designs where catching bugs can be crucial in the same way that a missed bug in the DSP code running a phone could also be fatal. The rest of this thesis will describe the concepts behind the successful implementation of a tool for verifying DSP assembly code and the details of how it was implemented.

1.1 Research Overview

In order to tackle the problem of software verification in the context of the restrictions described below, we needed to resolve a number of issues. These included the difficulties in reasoning about the complex instructions commonly found in DSP architectures, about complicated arithmetic that often foils Binary Decision Diagram (BDD[7]) based methods and how to reason about memory.

1.1.1 The Problem

The problem that our method attempts to solve is the automatic verification of the functional equivalence of two pieces of DSP assembly code. By functional equivalence we are referring to the equality between the two programs with respect to

```
      Source 1                        Source 2
MOV cx, foo                     MOV DX, 1
ADD cx, 1                       ADD DX, FOO
MOV result, cx                  MOV RESULT, DX
RET                             RET

{= result {+ foo 1}}            {= RESULT {+ 1 FOO }}
```

Figure 1.1: Demonstration of Program Equivalence

the expressions generated for the user-specified output variable pairs given the user-specified input equivalencies. For example, if the user specifies that the programs in fig 1.1 should be equivalent with respect to the value placed in *result* and *RESULT* (given that the value *foo* is equivalent to *FOO*), then the expressions generated by the tool for the *result* locations should be equivalent. We can easily see this is the case.

To carry out the verification on more complex examples, given the general difficulties involved in verifying software, a number of restrictions are placed on the problem. The first restriction is that the control flow graph (CFG) be the same in both programs. This restriction is not so severe in light of the fact that what we are attempting to verify is that small changes made to a piece of code do not change its functionality, making it likely that the CFG will be very similar. In section 5.2 on Future Work we will also discuss ideas for relaxing this constraint somewhat. The problem is also simplified by only verifying a single function at time, that is to say we do not handle subfunctions or interrupts. In practice, we envision the tool being be applied to the bottom-level functions in a large program. To simplify control flow analysis we do not allow arithmetic on the program counter. Most DSP's do not allow this. Recursion is also not allowed in the programs.

We will verify only rough equivalence of functionality. By rough equivalence we mean that we do not consider errors due to rounding or precision issues. These issues often arise when multiplication or shifting is performed and bits are lost due to overflow. For instance, a logical shift left by 1 behaves like a multiplication by two unless there is a 1 in the highest order bit, at which point the 1 is shifted out as an overflow and the shift no longer matches a multiplication by 2.

In some cases two algorithms may compute different functions, but be considered equivalent by the user. For instance, consider an MP3 program which compresses music versus another which does not. For the user, both algorithms function correctly (the music played sounds the same), but determining this equivalence is evidently not within the capabilities of the tool. Instead, all user-specified output values are required to be identical. In other words there is no notion of "similar enough" in our program.

1.1.2 Our Solution

In order to verify the software we chose to use symbolic simulation based on a depth first tracing of the CFG for each program. The symbolic simulation allows us to handle the complex DSP instructions while the uninterpreted functions allow us to handle the complicated arithmetic (i.e. non-linear arithmetic). The tracing allows us to reason about the execution order and branching structure of the programs. The tracing also allows us to decrease the size of the expressions generated since only one trace at a time is considered.

For common optimizations that are not easily reasoned about by our general approach, we applied a more ad hoc solution by special casing them. For instance, fixed-iteration loops are unrolled, and bit shifts are converted to multiplication.

1.1.3 Contribution

The main contribution of our work is the presentation of a method for the formal verification of DSP software. The approach, which is implemented in a prototype tool, uses a combination of techniques that allowed us to verify non-trivial examples of real life code. Other efforts have been made to verify assembly code for DSPs [6] but these, while more accurate (described further in section 2.4 on Related Work), have largely been user intensive and very time consuming to set up. Our tool, however, is almost fully automatic (only requires user to specify initial equivalences and equivalences to verify) and could be easily generalized to other architectures.

The tool, though limited in scope, was able to find a number of errors, some of which were subtle and had been missed by the developers, as discussed in detail in chapter 4. The errors included incorrect multiplication, values being stored in the wrong locations and missing initialization of values.

1.2 Overview of Thesis

Chapter 2 describes the background and related work that helps place our tool in context with previous verification efforts as well as some justification for the choices we made in picking the approach we did. Chapter 3 goes into detail about the exact manner in which the tool was implemented. The chapter includes a discussion of the various components of the technique being described, such as the basic blocks, CFG and uninterpreted functions and how they were incorporated into the tool we designed. Chapter 4 describes in detail the industrial examples (provided by Fujitsu) we ran that demonstrate the capabilities of the tool and the bugs that it was able to find in these examples. It also includes some performance statistics on memory and run time. Chapter 5 describes in more detail some of the limitations of the tool and the things we would like to implement in the future. Chapter 6 provides a summary of the work described in this thesis.

Chapter 2

Background and Related Work

There are a number of different techniques that have found supporters in the world of formal verification. Some of these techniques are described briefly in the following sections in order to give a flavor of the types of techniques available today. They are intended to give the reader an overview of some of the methods used in order that we may place our work in context and to explain why we chose our particular style of verification. For a more in depth review of the field and examples of each particular approach see [20, 33].

2.1 Formal Verification Techniques

Formal methods encompass a number of different techniques, but the basic goal behind them is the same. In formal verification, what one attempts to prove is that the implementation matches the specification. In our case, what we are attempting to prove is that two programs are equivalent after one has been altered as would occur in an effort to optimize the code. In this case, the original code can be thought of as the specification, as we assume it to be correct, and the new, modified code can be thought of as the implementation.

The techniques most related to the work we have been doing can be found under the heading of Equivalence Checking. A number of ideas and methods which we drew upon, however, can be found in compiler theory and are also described below.

2.1.1 Theorem Proving

Theorem proving is one of the earliest approaches to formal verification and consists of describing both the specification and implementation in terms of formal logic[1]. A theorem proving engine is then used to show that in the logic, the specification and implementation are suitably related.

The main drawback to this method is the high level of difficulty in deriving the formal proof. A number of semi-automated, application specific, proof checkers have been created such as the Cambridge HOL System[19], and nqthm (the Boyer-Moore Theorem Prover)[32]. Human intervention, even with these tools, is still required to help the proofs along at various points. This requires that the user be an expert in the logic being used, as the original formulation of the specification and implementation in the logic can be very time consuming and difficult.

There are a number of advantages to the approach. The technique has its foundation in an extremely powerful general mathematical logic that can be used in a wide variety of situations. Due to its rigorous mathematical foundation, the notion of "satisfies" is unambiguous and can be mechanically verified once the original proof has been created. The approach is also well suited to a hierarchical attack on the problem, providing opportunities for re-using theorems already proved.

There have also been a number of successes using the theorem proving method for the verification of large and complex systems. Hunt verified a 16-bit microprocessor similar in complexity to a PDP-11[22]. Another good example of the methods application was the verification of the 32-bit Viper microprocessor designed by the British Ministry of Defense and formally verified by the Cambridge

[1]For a good introduction to mathematical logic see [30]

team using HOL[1][2]

Due to the need for an expert user, and the large amount of user intervention required, we ruled this method out early as we wished our method to be largely automated.

2.1.2 Model Checking

Model checking, unlike theorem proving, focuses on the behaviour of the system. Model checking uses temporal logic, an extension of predicate logic, to express qualities that may change over time. The model checking algorithm allows the algorithm to check that the specified properties hold over all executions of the system.

A number of different variations of temporal logic exist, with two of the more popular variations being Linear Temporal Logic (LTL), which was explored by Manna and Pnueli[27], and Computational Tree Logic (CTL), which was first presented by Clarke and Emerson[15].

In order to apply the logic to the desired system, the system must first be described as a state machine. This step is usually fairly simple and can be incorporated into the original design process.

There are two main drawbacks to the approach that are common to most methods for formal verification. The first is the state explosion problem. Due to the size of the state machines for any "interesting" problem, the state space that must be searched is often enormous. The state explosion problem is a very active area of research with one of the most promising areas being symbolic methods[25].

The second drawback is the difficulty of creating the specifications which properly reflect the properties that the user desires to check. It is often unclear and difficult to check if the specification has been found to be correct because the model did in fact have that property or because the specification was not created correctly. It can even be difficult to verify at times that one has not created a

[2]A report written by Brock and Hunt dispute whether this effort was actually successful [5].

tautology because, for instance, the left hand side of an implication is false.

The advantage of model checking is that the decision procedure is completely automated. The user never need be aware of how the inner workings of the procedure function, they need only interact with the model checker to determine that the design satisfies the specification. The logic is also very general and can be applied to both hardware and software systems.

One of the more popular tools available to today is SMV, created by McMillan[29, 28], which uses CTL as its underlying logic and is a good example of how the design procedure can be automated.

We did explore this method briefly before rejecting it. The two main problems were the difficulty in writing the specifications for the properties we wished to verify, which were often almost as complicated as the description of the machine itself, and the size explosion problem. For these reasons we decided not to continue exploring this approach.

2.1.3 Equivalence Checking

Equivalence Checking[3] focuses mainly on combinational circuits and has been very successful in an industrial setting due to its ability to handle very large designs.

Equivalence checking verifies that, given any arbitrary equivalent inputs to the two designs, the output functions of the two circuits are equivalent. This is similar to our problem in that we are trying to verify that given a set of matching inputs, the pairs of outputs are equivalent.

One technique for carrying out the check for equivalence is to convert the output functions of the two designs into a canonical representation of some sort. If the two representations are equivalent, we know that the two designs are equivalent as well. The representation of choice these days is the Binary Decision Diagram (BDD) which was made popular by Bryant[7]. He showed that the BDDs could be

[3]For a good summary paper on Combinational Equivalence Checking see Jain et al.[23]

```
      Code Fragment
1.          MOV   a1, foo
2.          MOV   a2, bar
3.          ADD   x1, a1, a2
4.          RET

x1 = foo + bar
```

Figure 2.1: Symbolic Simulation

constructed, manipulated and compared very efficiently. The drawback to BDDs is that for a number of common functions the size of the BDD is exponential in the number of variables (or bits) being considered (multiplication is an example of such a function[8]).

A second technique focuses on finding equivalent points between two designs. These equivalent points can then be used to segment the design into smaller components that can be verified individually, avoiding some of the size problems. Once the design has been segmented to small enough components these smaller pieces can then be verified as stated above.

This second technique relies on the fact that designs are very similar and are being changed only in small, incremental steps. This again is similar to the problem that we are looking at in that we assume that the two programs are largely equivalent with only small, potentially subtle, changes. Our technique relies on there being a number of equivalent points between the two programs as determined by the control flow graph described below.

Symbolic Simulation

In order to avoid some of the size explosion problems associated with representing variables as specific values, we can instead leave these variables as uninterpreted constants or symbols. This allows the simulation to represent an entire class of

```
      Code Fragment
1.label1:    MOV   a1, 1
2.           MOV   b1, 2
3.label2:    ADD   a1, a1, b1
4.           MOV   x1, a1
5.           CMP   a1, 0
6.           BRZ   label3
7.           MOV   x2, 3
```

Figure 2.2: Basic Blocks Divisions

values in a single simulation run or expression, in our case. This technique is very common in both compiler design[2] and verification[11] (such as Symbolic Trajectory Evaluation[9, 33]).

In the context of our tool, what we are referring to as symbolic simulation is the use of symbols as the expressions are built during a single trace through the control flow graph (CFG). For example, in figure 2.1, the value generated for x1 after line 3 is *foo+bar* where *foo* and *bar* are symbols that could represent any arbitrary value instead of being a specific instance of an integer.

2.2 Control Flow Analysis

Control Flow Analysis is a common technique used to analyze the design structure in both hardware and software [17, 12].

The first step is usually to break the program into basic blocks. A basic block is defined to be[2]:

> ... a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end.

Figure 2.2 has three basic blocks. The first block contains only lines 1 and 2. If the block were to contain line 3 then it would now have two possible entry points (lines 1 and 3). The second block contains lines 3,4,5 and 6. The second block cannot contain line 7, since it would then have the possibility of branching within the block so line 7 is the beginning of a new block.

Once these blocks have been created, a graph can be constructed from the blocks that models the control flow of the program in question. It is described in most basic compiler texts[2] and is defined as follows.

The nodes of the control flow graph consist of the blocks defined above. Since our program has a single entry point then there is a specific node which serves as the root of the graph. Each block is connected by a directed edge to the block which follows it in the code (if the first block is not terminated by an unconditional jump). If the block is terminated by a jump it is also connected by a directed branch to the block to which it jumps.

The CFG can be used to help analyze a program to understand the properties of the program and in order to verify the program. In both cases the graph is useful in dividing the program into smaller sections that can then be handled more easily. The CFG is also a necessary step in the simulation of a program so that the simulator can decide which block of code should be executed next.

A large body of work has been done by Claesen et al.[12, 18] using the flow graph (referred to as Signal Flow Graph (SFG)) to help partition the verification problem in order to avoid the size explosion. The authors attempt to verify a design at different levels of abstraction as generated by the Cathedral silicon compiler. The approach is similar to the equivalence checking approach described earlier in which reference points (in their case signals) are located that can be used to further subdivide the program. After a subgraph has been verified, the outputs of the verified subgraph are replaced by symbolic constants which can then be used in the verification of the next subgraph. The original reference points are the inputs and outputs of the entire graph. This SFG is much like our CFG and is used in a similar manner for the purpose of verifying the equivalence of a hardware design at different levels of abstraction.

2.3 SVC - Logic Decidability

In order to validate the equivalence of expressions generated during the symbolic simulation, a decision procedure was required that was automatic (requiring no user input to verify equivalence or in-equivalence) and flexible (to allow the use of uninterpreted constants, reason about memory, etc).

The Stanford Validity Checker(SVC) [4] admirably fulfills these requirements. SVC is an extended version of the tool described by Jones, Dill and Burch in 1995 [24] which now has congruence closure (i.e. $f(a) \neq f(b) \Rightarrow a \neq b$) and linear arithmetic, among other extensions.

The tool uses a combination of decidable theories that allows the tool to be fully automatic and complete.[4] These decidable theories include:

- Uninterpreted functions with equality. E.g.

$$(FUNC\ a\ b) = (FUNC\ a\ b) \text{ and}$$
$$(\neq (FUNC\ a)\ (FUNC\ b) \Rightarrow (\neq a\ b))$$
$$\text{where } FUNC \text{ is an uninterpreted function.}$$

- Linear arithmetic. E.g.

$$\{+ \ 1 \ 2\} = 3$$
$$\{- \ 3 \ 4\} = \text{-}1$$
$$\{+ \ \text{a a a a}\} = \{* \ 4 \ \text{a}\}$$

- Arrays for reading and writing in memory. E.g.

[4] By complete we mean that given a legal expression in the logic, the tool can always resolve whether the expression is true or not. A difficulty arises in that the syntax of SVC allows statements to be created that are syntactically correct but not legal in the logic. For instance, a non-linear equation is syntactically possible, but is not a legal expression in the logic that SVC handles.

$$\{\text{read } \{\text{write mem a b}\} \text{ a }\} = \text{b}$$

- Propositional logic such as and, or, not, etc. E.g.

$$(\text{a} \vee \text{b}) \wedge \text{c} \equiv (\text{a} \wedge \text{c}) \vee (\text{b} \wedge \text{c})$$

The primary data structure used by SVC is a directed acyclic graph (DAG) that is very similar to the BDDs[7] used successfully in other tools such as Ever[21] and SMV[28]. They extend the data structure from a boolean domain to one including decidable fragments of first-order logic, which enables the tool to overcome some of the limitations[8] of BDDs while maintaining the desirable automation and efficiency properties.

In order to reason about these structures, SVC uses a hybrid between Boolean case-splitting and syntactic re-writing. SVC determines whether an expression α is true by extracting an atomic formula β from α and asserting it in the current context (i.e. sets β to true). It then simplifies α and repeats the extraction of a new β from the simplified α in this new context. This process of extracting and asserting β repeats until the expression is found to be false or is reduced to true. If the expression is reduced to false then the proof stops and a counter example is generated. If it is found to be true, then the contexts are restored and the original β is negated. Then the whole thing repeats again with the α simplified by the negation.

The syntactic part of the SVC procedure occurs when the tool chooses the β at each recursive step. SVC attempts to pick the "simplest" expression for β by first imposing a total ordering on the expressions. The total order is monotonic with respect to term structure and allows SVC to proceed with substitution and rewriting using the simplest equivalent formula. The ordering is done using a number of steps the first of which is:

Constant expressions are always simpler than non-constant expressions.

For arbitrary boolean, rational and user-defined constants b, r and c, we define $b \prec r \prec c$. For booleans, we simply have $false \prec true$. Rational constants are ordered numerically and user-defined constants are ordered lexicographically.

For a complete description of the remaining steps and a proof that the rules impose a monotonic total order, see [4].

The result of combining the techniques and decidable theories is a tool that is fully automatic and powerful in terms of the number of possible operations allowable.

2.4 Related Work

Much of the work being done in automated formal verification is in the world of hardware verification. Software is notoriously hard to verify formally even for relatively simple examples. This difficulty arises for a number of reasons, not least of which is the tendency of software designers to "code" without writing any specifications. In addition to this lack of specifications, any high level language such as C or C++ has complex features that are hard to analyze and verify. Some examples of these features include recursion, pointers and unbounded data structures. As a result of these difficulties, most of the practical successes in the verification field have focused around hardware verification, where there has been a number of successful applications to real problems, and commercial tools are now available.

One of the few attempts at assembly code verification for DSP chips was done by Brock and Hunt [6] for Motorola's Complex Arithmetic Processor (CAP). Their work involved formally specifying the entire processor in ACL2 logic[26] using the ACL2 theorem-prover to carry out the mechanical proofs. The work done for this chip was bit for bit exact and capable of verifying algorithms (programs) that involved both hardware and software components. Although much more extensive and accurate than the work we are doing, their specification of the chip required eight man years to complete. Our approach (although far less encompassing) is very different from the work done above in that the effort is significantly less, the user need not be an expert in some aspect of formal verification and the user is not required to adopt a new method of coding or designing.

Another approach to low level code verification is presented by Clutterbuck and Carré[13]. The authors present a language called SPADE which has a Pascal-like format and can be translated to assembly. Due to the fact that the code must be verifiable, no optimizations are allowed between the higher level code and the lower level assembly. This approach is much more restrictive than our approach, because the program must be written in SPADE and cannot be optimized. It would also be difficult to generalize the tools written for SPADE to other applications.

Formal verification of embedded software has also been studied. One example of this is the work done by Thiry and Claesen[35] who proposed a method using model checking in SMV[28]. They were able to find inconsistencies between the assembly code and the flow chart specification for a mouse controller. This approach is different from ours in that it takes a model checking approach using CTL. The user must be an expert in CTL, and their approach could be very user intensive.

Balakrishnan and Tahar [3] carried out a similar effort to verify a mouse controller as an example of an embedded system. They also used a model checking approach but they used the recently proposed multiway decision graphs (MDG) and the related decision procedures. The advantage of this approach being that the MDG is supposed to avoid some of the size explosion problems that BDDs encounter.

Chapter 3

Implementation

When implementing the tool we had two goals in mind. One was to keep the tool as conservative as possible. The degree to which a tool or technique is considered conservative is the infrequency with which it answers "Yes, the two programs are equivalent" when in fact they are not. If it can be guaranteed that the tool will never answer "Yes" when there are bugs present, then the tool is said to be strictly conservative. The tool was already not strictly conservative because we had decided that we were not going to deal with issues such as round off errors. However, we wished to minimize the chances that the tool would find the two programs equal when in fact they were not. The second goal was to make the tool general enough to be useful. If a tool is too restrictive, then the number of times it reports a potential bug will overwhelm any useful information that can be derived. There is not much point in implementing a tool if it is so restrictive that it is unable to verify anything interesting.

Unfortunately, these two goals are somewhat in opposition to each other. The more conservative the tool is, the more likely it will report bugs that are not really bugs and the more general it is, the more likely that there will be instances of bugs that are missed. To find a balance between the two we chose to develop the tool in conjunction with real life examples, enabling us to see how weak we could

```
    Source                          Expressions
MOV   a1, foo                    a1 = foo
MOV   b1, bar                    b1 = bar
ADD   b1, 1                      b1 = {+ bar 1}
MUL   dx, a1, b1                 dx = (MULL foo { + bar 1})
RET
```

Figure 3.1: Example of Symbolic Simulation with Uninterpreted Functions

make the tool and still obtain useful results.

These two goals served as the guiding influence behind many of our choices.

3.1 Verification Method

In the previous chapter, we briefly outlined and explained a number of different verification techniques and showed why we chose to pursue the approach we did.

The approach we chose to take was that of symbolically simulating the code in combination with the use of uninterpreted constants (see Fig 3.1)and a decision procedure engine (SVC). This approach allowed us to reason about multiplication and division because we were able to make use of uninterpreted functions to avoid the size explosion problem that often impedes other attempts at verification.

The general steps taken by the program are outlined below.

3.1.1 Summary of Program Execution

The following are the steps in the program execution. The more important steps are elaborated further in the following sections.

1. Initialize all data values and arrays.

2. Read in the source code for the first program.

3. Create the memory relations.

4. Block the program.

5. Create and refine the control flow graph.

6. Repeat steps 2-5 for the second program.

7. Perform symbolic simulation in a depth-first manner of the control flow diagram for each trace. (see figure 3.2)

8. Simplify expressions built during symbolic execution.

9. Compare results using decision procedure (SVC).

10. If the results verify for this trace, return to step 8 and continue for next trace (until all traces have been exercised). Otherwise, exit the program, and give the user a list of the basic blocks that were traversed with the decision taken at each branch (TRUE or FALSE) to reach the error.

11. If simulation reaches the end, then no errors were found.

3.1.2 SVC Interface

The decision procedure that we use is the Stanford Validity Checker (SVC). The logic for this decision procedure includes boolean and uninterpreted functions, linear arithmetic, array operations and linear inequalities.

The syntax is a lisp-like prefix notation where "()" denotes the use of an uninterpreted or boolean function. Some such expressions might be *(MULT a b)* where *MULT* is an uninterpreted function, or *(= expr1 expr2)* as an example of a boolean function, while "{}" denotes an interpreted function such as $\{+\ a\ b\}$ or $\{$*read memory location*$\}$. For a complete description of the syntax see [16].

During the development of this tool, it was our observation that SVC (which is still an experimental tool) was not completely robust. The main indication of this was the fact that SVC's performance would degrade as the number of calls

```
verify(block1,block2)
  symbolicallyExecute (block1)
  symbolicallyExecute (block2)
    if(stoppedOnBranch(block1) & stoppedOnBranch(block2))
      if (verifyBranchEquality(block1,block2))
         pushContext()
         if(flippedBranches(block1,block2))
           verify(false(block1),true(block2))
           popContext()
           verify(true(block1),false(block2))
         else
           verify(false(block1),false(block2))
           popContext()
           verify(true(block1),true(block2))
      else
         CONTROL FLOW DIDN'T MATCH, CAN'T VALIDATE
         exit
    else
      if (stoppedOnRet(block1) & stoppedOnRet(block2))
         if (validate())
           THIS TRACE IS VALID
           popContext()
         else
           TRACE NOT VALID, CAN'T VALIDATE PROGRAM
           exit
      else
         CONTROL FLOW DOESN'T MATCH, CAN'T VALIDATE
         exit
end
```

Figure 3.2: Basic Algorithm for Symbolic Execution

```
symbolicallyExecute(block)
  until end of block is branch
  or return
    build expression tree for block
    goto next block
end

verifyBranchEquality(block1,block2)
  if branch conditions are equal
  or branch conditions are complements
    return TRUE
  else
    return FALSE
  end

pushContext()
  Stores all the current values
  in memory, registers, etc.
end

popContext()
  Restores the values for memory,
  registers, block locations, etc.
end

flippedBranches(block1,block2)
  if branch condition codes are equal
  when one is negated
    return TRUE
  else
    return FALSE
end

validate()
  if user specified equalities
  are true
    return TRUE
  else
    return FALSE
end
```

Figure 3.3: Supporting Functions to Algorithm in Figure 3.2

made to it and the size of the strings passed increased. For that reason, we made a number of design decisions that attempted to minimize the use of SVC. When SVC was needed, every effort was made to minimize the size of the strings being handed to the tool. This led to some design decisions that might not have been the optimal theoretical choices but were dictated more by performance concerns. Some decisions that were made based on these goals were the manner in which the CFG was traversed (depth-first) and the attempt to simplify expressions before handing them to the tool. These tradeoffs are discussed further in the following sections and in section 5.2 on Future Work.

3.2 Basic Block Analysis

There were a number of goals behind breaking the code into basic blocks. The primary goal was to be able to reason about the control flow of the program. In addition, one can cut off the calculation at the basic block boundaries and do a pairwise comparison on all known quantities to find equalities. These equalities can then be used to simplify the expressions. The later capability was implemented and tested in the tool but was eventually put on hold, as we discovered that the examples could be carried out without this step.

3.2.1 Problem

The concept behind basic blocks is explained in section 2.2 on Control Flow Analysis.

As was mentioned, dividing the program into basic blocks allows us to calculate the control flow of the program, which is a necessary step before any symbolic execution can be carried out.

The more interesting property of dividing the program into basic blocks is the potential for cutting off the symbolic execution at a given point and performing a pairwise comparison of all known values to find equalities. These equalities can then be used to prune the data by replacing the values with a unique uninterpreted value

If $cx = (a1 * foo) + bar$ and $dx = (a1 * foo) + bar$ then $cx = dx$
Replace value of cx and dx by a unique constant.
$cx = dx = uniqueConstant$

Figure 3.4: Pairwise Comparison and Simplification of Equal Values

(see Fig 3.4). This step would most likely be necessary in larger programs if the generated expressions grew to an untenable size. This blocking of the program could also be used to break a program into pieces that could be verified independently, again with the intent of making larger programs more manageable.

The ability to evaluate equalities at the beginning of a non-fixed count loop could also be used to calculate a type of "loop invariant" (see section 3.7 on Loops).

3.2.2 Solution

In the initial step the program creates a block at every instruction (or label) where the program could deviate from the "next statement" execution. This includes all labels, branches, loops, returns and calls. It is assumed that there can be no arithmetic on the program counter, nor interrupts that could alter the control flow of the program. The intial blocking is very crude, as it contains many unecessary blocks such as unused labels and branches with no condition codes. These unecessary blocks are eliminated during two subsequent passes over the blocks to create the control flow diagram as described in the following section.

Upon initially constructing the tool, we believed the bottleneck was likely to be found in the size of the expressions being generated during the symbolic simulation. To deal with this possible size problem, we implemented the concept described above of truncating the phrases. Upon running the tool, on the industrial examples, however, it became clear that the bottleneck was in the expressions being generated due to memory accesses. After some basic optimizations and simplifications (see section 3.5on Memory), we found that the expression size could be kept to a reasonable length without truncating. We also found that even though the truncation is conservative (will not cause it to verify something that is not equal), if it was performed too often, say after every block, the loss of information rendered the verifier too conservative to be useful.

For instance, if the register *x1*,*x2*, and *x3* have values *foo+1*,*foo+1*, and *foo* at the end of one block, and the truncation is performed, then *x1* and *x2* would be assigned a unique constant *const*, while *x3* would retain the value *foo*. If, in the next block, *x1* and *x2* are not altered and *1* is added to *x3*, then if the truncation had not been performed, it would be equivalent to *x1* and *x2*. Since the value of *x1* and *x2* is now *const*, we would be unable to verify that *x3* is equivalent.

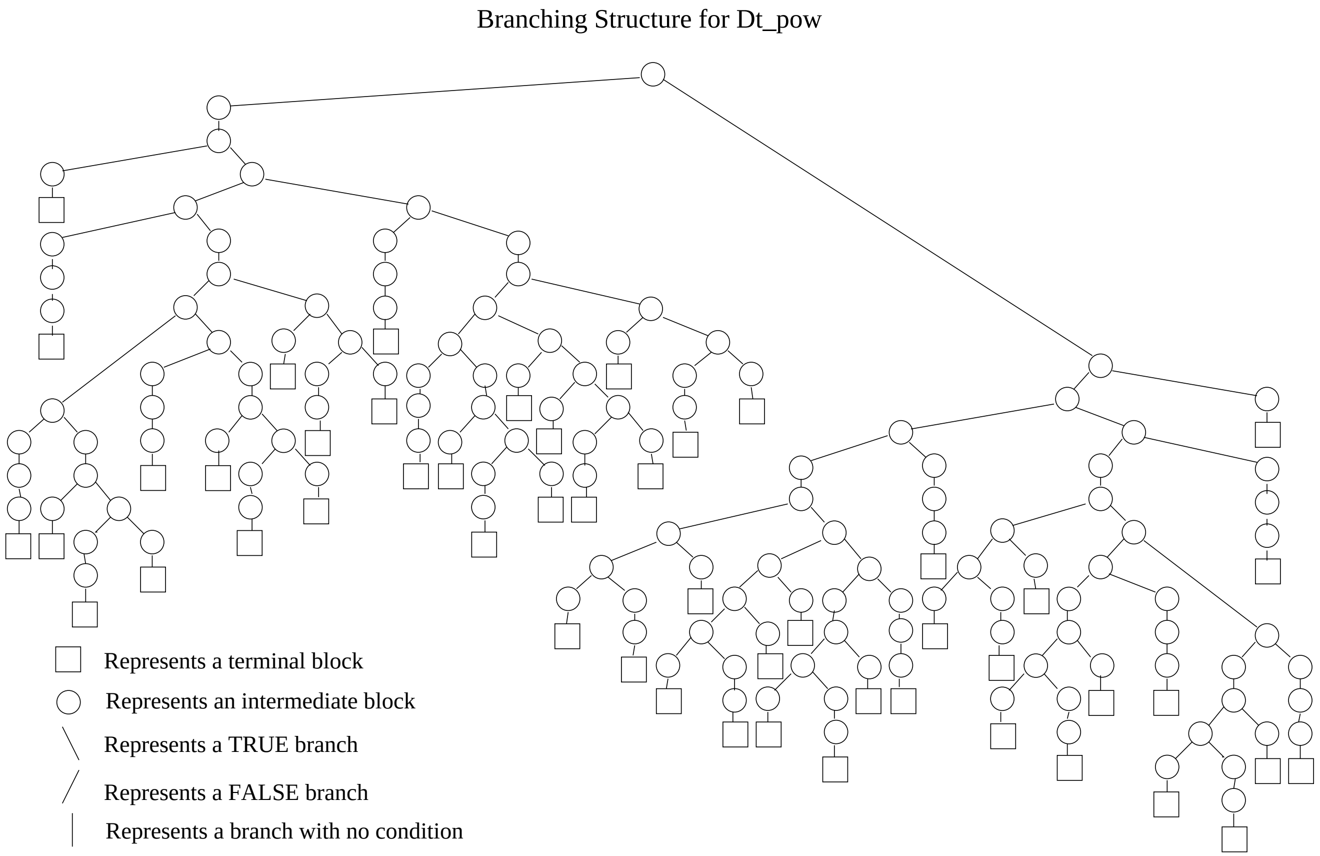

Figure 3.5: Tree Representation of CFG for Dt_pow

3.3 Control Flow Analysis

The control flow analysis allows the program to simulate the code by doing a depth-first traversal of the graph (actually the tree representation of the graph) constructed from the assembly code. It is this graph which also determines whether the structure of the two programs is the same. The graph is directed and potentially cyclic. In the examples that we had, however, the graph was a DAG (directed acyclic graph) even though the examples contained fixed count loops that create a cycle. By using the concept of loop unrolling, which will be explained in more detail in section 3.7 on Loops, we were able to eliminate the cycles created by the loops. The possible paths through the graph, even when acyclic, can quickly become fairly large and beyond the "practical" ability of a user to manually construct or simulate (see Figure 3.5 for the tree representation of the CFG for the industrial example Dt_pow).

Since the definition and concepts behind the Control Flow Analysis were described in Chapter 2, we will only discuss how it was used in our tool.

3.3.1 Implementation

Initially, we intended to use the graph to divide the program into smaller segments and to use it to simulate the code. During the simulation, as each program reached a branch where both the TRUE and FALSE branches were possible, the simulation would stop. The condition upon which each branch depended would then be tested for equality. If the conditions were equal to each other or equal after one has been negated, the program would then perform a pairwise comparison of all registers and memory in an attempt to find pairings of equal values that could be replaced by a unique constant, reducing the amount of space required for the simulation to continue. This truncation proved unecessary and was subsequently dropped.

The approach we eventually settled on was to simulate the graph in a depth-first manner, storing the context at each branch and continuing until a terminal branch (return) was reached. At this point the equalities (given by the user) were checked. If they were found to be valid, the program would back up one level until an unexplored branch was found. The context for this level would then be restored and the simulation would continue.

Three passes are used to construct the control flow graph. The first pass does the following:

- All labels are marked as unused initially. During the first pass, as each label is referenced, it is marked as used. This indicates that the labels are referenced by a branch or a DO loop (calls and jumps are not supported at this time although the program will properly construct the graph for these instructions).
- Each block is connected to the following block (except where the block is terminated by a return).
- The blocks terminated by a branch are also connected to the block where they branch to if the condition code is found to be satisfied.
- The blocks on which a pairwise comparison can be performed are marked.

The second pass is used to eliminate all the blocks that contain only an unused label, non-conditional branches and blocks created around fixed-count loops which can be unrolled.

The third pass is used to further clarify which blocks should have the pairwise comparison performed after having executed them. This pass is not currently being used, because we do not do the pairwise comparison.

The control flow graph is linear in size with respect to the size of the program. However, the number of possible paths through the graph (which can be represented as a tree, see fig 3.5) can quickly become very large as the number of branches increase. The tree representation of the graph (i.e. the number of paths through the CFG) expands at an exponential rate in the number of conditional branches.

3.4 Context Stores and Retrieves

As the symbolic simulation proceeds, choices need to be made when a branch is encountered. At this point, the program stops the simulation and ensures that both programs have encountered a similar choice. If the program finds that this is the case then the context is stored. This allows the program to recursively explore one branch of the control flow graph. Once this branch has been explored and verified, the program will return to this branch and restore the context. After having done this, it can then explore the next branch.

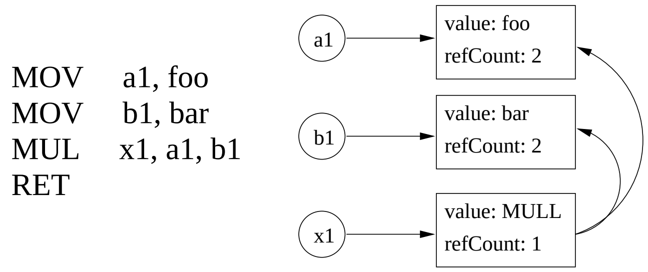

Figure 3.6: Example of Data Structure used in Tool

The context itself consists of all the values in registers and memory as well as a pointer to the block that should be executed next from that point in the control flow graph. The values stored in memory consist of a pointer to the value stored. No replication of data is done, as can be seen in figure 3.6.

The contexts are stored in a stack and a new context is pushed onto the stack for each branch that has had only one branch explored. The top context is popped when a terminal branch (return) is reached (the usual manner in which a tree is traversed depth-first [14]).

```
{read mem1 foo}      where mem1 is the memory
                     foo is the location being read
                     returns the last value written to location foo
{write mem1 foo bar} where mem1 is the memory
                     foo is the location
                     bar is the value being written
                     returns a new memory with location foo updated
                        to value bar
```

Figure 3.7: Read and Write Functions

3.5 Memory

Memory proved to be one of the trickier and more crucial components of the simulation due to its tendency to explode in size. Memory is often modeled using read and write axioms (see figure 3.7).

3.5.1 Problem

A number of requirements needed to be satisfied in order to simulate the memory for a DSP chip.

1. We needed to be able to reason about the declared memory location's addresses relative to each other. A very common practice (used often in our examples) is to move an address into a register and use this to access memory. Then the value in the register is simply incremented or decremented to access other locations.

2. The memory could not grow beyond a reasonable size. One can see how this would occur if, for instance, you had a loop in which the memory was accessed and the result added to the value calculated in the previous iteration (figure 3.8). If memory were stored in a single location (without simplification) then the resulting growth would be sizeable.

```
        DO    endLoop, 512   ;; repeat to label endLoop
                             ;; 512 times
        MOV   a0, (x1++1)    ;; move value of memory location
                             ;; and incremement
        MOV   (x2++1), a0    ;; store a0 in location x2 and
                             ;; increment
endLoop:

Value of Memory after iteration:
0:  Empty
1:  {write Empty {read Empty x1} x2}
2:  {write {write Empty {read Empty x1} x2} x2+1 {read
        {write Empty {read Empty x1} x2} x1+1 } }
...
```

Figure 3.8: Example of Memory Growth if Stored in a Single Location

```
1.  MOV knownLoc1, 1
2.  MOV knownLoc2, 2
3.  MOV unknownLoc1, 3
4.  MOV x1, (knownLoc1)
5.  MOV knownLoc1, 4
```

Figure 3.9: Difficulties of Memory Reads and Writes

```
MOV    (foo), 1        {write empty foo 1}
MOV    (bar), 2        {write {write empty foo 1} bar 2}
RET
```

Figure 3.10: Memory Storage

3. The memory had to be flexible enough to handle reads and writes to unknown locations. This requirement was more subtle than it first appeared, since a write to an undeclared location might very well overwrite a value at a known location, or similarly, a read might access a location which had been written to already. Thus, if one were to implement the memory as seperate locations, one would need to remember when each of the writes occured.

   For instance, in figure 3.9 the move on line 4 would no longer retrieve a value of 1 from the location `knownLoc1` since it is possible that the `MOV` that occured on line 3 overwrote the value in `knownLoc1`. After line 5, we can now retrieve a value from `knownLoc1` but we still cannot from `knownLoc2`.

3.5.2 Solution

We decided to implement the memory as an array with read and write functions (fig 3.7). A single memory is used for each program, and all memory reads and writes are directed at this memory.

This meets the third requirement listed above, since each write is wrapped around the existing memory. Intuitively, when a read is performed, it accesses the memory from the outside write and moves inwards, towards the older writes (see figure 3.10 for an example of how memory writes occur). If, at some point, the read encounters a write to an address that cannot be determined different from the address it is looking for (nor can it be determined to be equal), then its stops at this point and returns the entire contents of the memory. (In practice, the memory functions are directly supported by the decision procedure.)

Our solution could violate the second requirement of avoiding rapid memory growth. Fortunately, with the optimizations described below the rapid growth is largely avoided.

To handle the first requirement, we generated a number of *memory axioms* that allowed us to reason about the relative position of each declared memory location. For instance, if foo is a memory location of size 5, and *bar* is the next declared position, a memory axiom is generated that declares *assert (= bar {+ foo 5})*.

Three memory simplifications are needed to help avoid the memory explosion problem. The first of the three is performed every time a write is done. The second and third are done every time a read is performed.

1. When a write is perfomed, the existing memory is scanned to check if the address being written to has already been used. Since this overrides the previous write, that write can then be eliminated from memory. For example,

   $$\{\text{write } \{\text{write mem1 foo bar}\} \text{ foo value}\}$$
   $$\equiv \{\text{write mem1 foo value}\}$$

   This simplification is extremely useful in loops that use a temporary register that is written to in each iteration, such as in the industrial example yhaten. Without such a simplification, a rapid growth would occur, because the entire memory contents are returned instead of the value in the location that is to be retrieved. Due to the relatively slow performance of SVC, this check for equality is performed syntacticaly (a simple string comparison). This check may miss some possible simplifications but it is quicker, and conservative. Even if some simplifications are missed, the new value will be returned regardless, due to the fact that all reads start at the outside write and move inwards(figure 3.10), returning the most current value written to that location.

2. When a read is performed, we search for the location in the existing memory to see if it has been written to. For example,

$$\{\text{read}\ \{\text{write}\ \{\text{write mem1 foo 3}\}\ \text{bar 4}\}\ \text{foo}\ \} \equiv 3$$

   if bar $\neq$ foo. If the memory location is located then the value is retrieved and returned. SVC must be used for this simplification, since if a write to that location is missed or if a location which cannot be determined to not be the desired location is skipped, an incorrect value could be returned.

3. The third simplification is done simultaneously with the second simplification. If the exact location cannot be found, then the size of the memory returned in the read may be reduced by finding the last write that occured for which the location written to cannot be determined to be different than the location sought. For example,

$$\{\text{read}\ \{\text{write}\ \{\text{write mem1 unkown 2}\}\ \text{bar 3}\}\ \text{foo}\} \equiv$$
$$\{\text{read}\ \{\text{write mem1 unkown 2}\}\ \text{foo}\}\ \text{if foo} \neq \text{bar}.$$

   It is often the case that this simplification is sufficient to reduce the memory being used to its initial value, greatly reducing the rate at which the expressions being built expand.

3.6 Axioms

Due to the extensive use of uninterpreted functions the ability to declare axioms was required.

3.6.1 Problem

SVC does not support non-linear arithmetic, rendering it unable to handle some very simple multiplications. An uninterpreted function was used in lieu of the interpreted multiplication to circumvent this problem. Since the multiplication is uninterpreted one needs to be able to assert that *(MULT a b)* $\equiv$ *(MULT b a)* for any value of a and b. Similarly one would like to be able to assert that *(MULT a 2)* $\equiv$ *(ASL a 1)* for any value of a, and where ASL is an arithmetic shift left.

3.6.2 Solution

Unfortunately, SVC does not support the ability to declare axioms which hold for all values of its argument. That is to say, if one declares that *(MULT a 2)* $\equiv$ *(ASL a 1)*, this does not enable SVC to determine that *(MULT b 2)* $\equiv$ *(ASL b 1)* if $a \neq b$.

To solve this problem the program allows axioms to be declared in a file called "axioms.txt". The axioms have the form

ASL assert (= (ASL argASL1 1) (MULT argASL1 2))

When an expression is going to be sent to SVC it is pre-processed by a procedure which locates any instances of ASL in the expression. If an instance of ASL is located, such as:

(ASL foo 1)

then two expressions are generated that assert:

(= argASL11 foo) and (= (ASL argASL11 1) (MULT argASL11 2))

where argASL11 is a unique indentifier. Both of these new expressions are also given to SVC, allowing it to now reason about the ASL and MULT.

All of the axioms are expanded similarly.

3.7 Loops

The presence of loops is an inherently difficult problem. Given that (in general) it cannot be decided whether a while loop terminates, the best one can hope for is a heuristic that works "in most cases".

3.7.1 Problem

For loops with a fixed count, the easiest solution is to unroll them and execute the code contained within the loop the appropriate number of times. This unrolling might not always be possible, as the loop count may be extremely large or the code contained in the loop may cause expressions to grow too rapidly.

An even more difficult case arises when the loop has a conditional exit value. In this case the exit condition would need to be checked for equality between the two loops.

In both cases the loops could potentially be handled by the calculation of a fixed point. The idea of a fixed point is a well known technique used in many different areas. It generally involves repeating a set of calculations until some set of values or implications has ceased to change. In our case, a set of equivalencies could be calculated between variables in the two programs before executing the loop. After each iteration of the loop, the equivalencies could be re-checked. If no new equivalencies have been created that could not be implied from the equivalencies already existing before the loop then a fixed point has been reached, producing a loop invariant.

3.7.2 Solution

None of the industrial examples contained non-fixed count loops[1]. Therefore, the technique used was to unroll the loops. With the simplifcations to memory described earlier, the simulation was able to proceed without the expressions exploding in size. One drawback that was discovered was that by loop unrolling and doing the memory simplifications, the performance suffered.

[1] It is not unusual to find that most loops in DSP algorithms are fixed-count loops due to the need for a bounded running time to meet performance requirements.

It is unclear whether the fixed point calculation would prove useful, since it was not implemented and is, most likely, limited in its applicability.

3.8 Branching Conditions

The branching conditions are treated in the same way as registers, in that a slot exists for every register and for every condition code. The values contained in the condition codes are the expressions that were generated by the last opcode to be executed that affected the condition code.

These expressions can then be used to verify the equality of the branches between the programs.

3.9 Expression Construction and Simplification

The expressions used in the tool are generated by symbolically simulating the opcodes in a serial manner. That is to say all the blocks of code are simulated in the order in which they appear in the CFG, or more accurately, a given branch of the tree representation of the graph.

The expressions are represented by a tree-like data structure where the node is the operation and its children the arguments. This data structure allows the built expressions to be shared so that no values need to be duplicated. For example, if the register cx is set to the product of registers $x1$ and $x2$, then the new node associated with cx will have the label "MULT" and its children will be pointers to the values contained in $x1$ and $x2$ (figure 3.6).

Representing the data in this fashion helps keep the amount of memory usage down, since no data need be duplicated. It does require, however, a fairly careful management of reference counts to be able to decide when the memory should be freed.

3.9.1 Opcodes

Each opcode has a separate procedure in which the behaviour of that opcode is specified. In this procedure the appropriate nodes are created and assigned. If the opcode has a parallel functionality, temporary locations are created to keep the values that need to be remembered until the full set of operations is complete. For instance, in a parallel move such as:

MOV a0, b0, b0, a0

the value *b0* is stored in a temporary location until it is used in the move to ensure that it is not overwritten by the second part of the move.

3.9.2 Expression Simplifications

As was discussed in section 3.6 on axioms, the need to use uninterpreted functions for multiply and divide had some undesired side effects, the most serious one being the loss of ability to reason about basic properties such as commutivity and distributivity. Axioms can be specified to handle simple cases such as:

(MULT a b) = (MULT b a)

but as the number of arguments increases and the $MULT$s are nested, the ability to write axioms quickly becomes inadequate for handling these cases. To handle this, a number of simplification heuristics and re-ordering steps are carried out on the expression in order to increase the probability that the expressions will be found equivalent. These operations are listed and described below in the order in which they are performed.

1. collapse adds: This stage involves accumulating all the constants involved in an add to a single location. This simplifies the expression considerably in terms of length when the auto-increment or decrement feature has been used extensively. In two examples the auto-increment feature is used inside a loop of

512 iterations, making the above simplification necessary to reduce the length of the string handed to SVC.

An example of the above simplification might be:

$$\{+\ x1\ \{+\ \{+\ 1\ x2\}\ \{+\ 1\ \{+\ 1\ \{+\ 1\ x3\}\ \}\ \}\ \}\ \} \equiv \{+\ x1\ \{+\ x2\ \{+\ x3\ 4\}\ \}\ \}$$

2. convert memory locations: This stage is necessary to better increase the probability that the re-ordering will place the arguments in the same order in both programs (see the step on ordering below for an explanation of why). An example of this might be if $mem3$ has size 2 and $mem3 = mem2 + 2 = mem1 + 4$ then:

$$\{+\ \text{mem1}\ 5\} \equiv \{+\ \text{mem3}\ 1\}$$

3. convert shift functions (ASL, LSL) to MULT: This stage is again to increase the probability of the ordering being equivalent in the two programs, and it allows the program to reason about the equivalence of MULT and LSL opcodes. The conversion is:

$$(\text{ASL cx y}) \equiv (\text{ MULT cx } 2^{y}\ )$$

4. move DSTP (divisions) outside of MULT: All divisions are moved to the outside of any multiplications. This move will better allow the program to re-order the arguments of the multiplications. It also improves the simplification of the real multiplications described in the following step. The simplification is repeated until convergence, meaning in each step, the division is moved outwards one position (or multiplication) at a time until it cannot be move outwards any farther. For example:

$$(\text{MULT (MULT (DSTP x2 2) x3) x4) } \equiv \text{(DSTP (MULT (MULT x2 x3) x4) 2)}$$

In reality, this re-arranging might not be valid since it could result in rounding or truncation errors. However, in our more restrictive model, this is not a problem, since we explicitly state that we are not dealing with rounding and truncation errors.

5. simplify interpreted multiplication: This step basically simplifies the multiplications by the real number -1. All multiplications are represented by the uninterpreted function MULT except for multiplication by -1. Multiplication by -1 occurs whenever a negation or subtraction is performed (SVC does not support an explicit subtraction). The multiplications by -1 are pushed to the outside of all multiplications and simplified as pairs are found. The movement is done one term at a time and is repeated until convergence.

   {* -1 (MULT a {* -1 b}) } ≡ (MULT a b)

6. accumulate constants in MULTs: This step is much like the collapse add step except that in this case, the multiplication by constants is being collapsed. The end goal is, once again, to aid in the re-ordering of the arguments of the multiply. This simplifcation will also aid in reducing the size of an argument if the multiplication is done repeatedly in a loop.

   (MULT (MULT a 3) 4) ≡ (MULT a 12)

7. simplify MULT's to be in format (MULT (MULT (MULT a b) c) d): The benefit of this step is largely to simplify the code needed to properly re-order the arguments in the MULT. It also brings all multiplications to a common format which has a higher probabililty of being equal in the two programs.

   (MULT (MULT a b) (MULT c d) ≡ (MULT (MULT (MULT a b) c) d)

8. re-order arguments of MULT: This step is the most crucial, as it is this re-ordering that will hopefully allow the program to verify the equivalence of

the multiplicative expressions generated in each program. The heuristic for ordering the arguments is based on the order in which the memory locations were declared. This allows the corresponding arguments in both programs to be sorted compareably.

Each argument is ranked according to the following heuristic:

- Each memory location is given a value corresponding to a list of prime numbers. It was hoped that, by using prime numbers, it would make it less likely that the sum of two different pairs of locations would be equal.
- Each uninitialized location is given a large value so that they get pushed to the outside.
- The rank given each argument is the sum of the values generated above as they occur in an expression. For example, if foo has a value of 3 and bar has a value of 7, then the argument expression {+ foo bar} will have a rank of 10.

Then the arguments are sorted such that the outermost argument in a multiplication will have the smallest value and the arguments will increase in value as they move inward. For example, if $arg1$ has value 5, $arg2$ has value 10 and $arg3$ has value 7, then the expression:

(MULT (MULT (MULT arg3 4) arg2) arg1)

becomes:

(MULT (MULT (MULT arg2 arg3) arg1) 4)

Although the above ordering and simplifications do not hold in all cases (in fact, it is quite easy to think of examples that would break the re-ordering heuristic) it did prove powerful enough to work in all of the examples that we tackled. It should be noted that the cases where the order of the arguments was different in the two programs occured in 4 different instances. The re-ordering was powerful enough to handle all four different cases and was not merely tuned to one specific example.

It should also be noted that within the constraints of the problem we were attempting to solve, the above simplifications and re-orderings are conservative because the resulting expressions should be equivalent.

3.10 Execution Reports

During the simulation run on the assembly code a number of reports are generated to facilitate locating any potential errors detected by the tool. A short example (which we wrote), with the corresponding output, is given at the end of this chapter.

3.10.1 Code Listing and CFG Listing

After the code is parsed and the blanks lines and comments are removed, it is printed with the corresponding line numbers. This listing can then be used to check which lines of code were executed in a particular trace.

The CFG is printed as a list of blocks with the corresponding line numbers indicating which lines of code are contained in that block. Each block in the control flow graph also indicates the lines of code to which it can branch.

3.10.2 Execution Trace

The execution trace is a list of the blocks (and corresponding lines of code) which were symbolically executed in that particular trace. The trace can be terminated by either reaching a return instruction or encountering a discrepancy in the correspondance in the control graph.

If the trace terminates due to a discrepancy, the trace, the suspicious branch and the expression corresponding to the condition codes on which the branches are dependent are printed.

If the trace terminates because of a return, then the equalities are checked. If the equalities are validated, the trace is printed along with a message indicating that this trace was found to be correct. If the equalities are not validated, the trace is printed along with the equality and corresponding expressions which could not be verified.

3.10.3 Expression Reporting

The length of many of the expressions generated in the tool are large. For this reason, the only time an expression is printed is if it is involved in a problem. It is printed in the case where a branch does not match the corresponding branch or if an equality cannot be verified.

3.10.4 Uninitialized Variables

It is often unclear from a manual inspection of the assembly code which values are going to be accessed and if they all correspond to values that have been initialized. To aid the user, a report is printed at the end of the symbolic simulation run that lists all values that were accessed without having been initialized. The user is then free to decide if this is a problem or if the equality between uninitialized variables in the two programs should be added to the list of known equalities.

3.11 Example Results

The following example was created to demonstrate some of the capabilites of the tool. Figure 3.11 is a listing of the two source code files. Figure 3.12 is a listing of the code as it is parsed by the simulation tool. Figure 3.13 is a list of the equivalencies that can be assumed by the program and the equivalencies that are to be checked. Figure 3.14 is a listing of the control flow graphs that are constructed for each source

```
   Source 1
   MOV   dx, 42
   MOV   a0, (X:arg1)
   MOV   b0, (X:arg2)
   MUL   cx, a0, b0
   CMP   cx, dx
   BRLT  exit
   SUB   cx,dx
exit:
   MOV   result, cx
   RET

.AMEM
.mem arg1, 1
.mem arg2, 1
.BMEM
```

```
   Source 2
   MOV   x0, X:arg1
   MOV   a0, (x0++1), a1, (x0)
   MOV   dx, -42
   MOV   cx, dx
   NEG   cx
   MSM   dx, a0, a1
   BRGE  exit
   ADD   dx, cx
exit:
   MOV   result, dx
   RET

.AMEM
.mem arg1, 1
.mem arg2, 1
.BMEM
```

Figure 3.11: Input Code for Simulation

file. It is this graph that is used to decide which block is executed next, or in the case of a branch, which blocks are possible. Figure 3.15 is an example output trace for a valid program listed in Figure 3.11. Figure 3.16 is an example output trace that is invalid due to a discrepancy between the control flow graphs of the two programs. This error was created by changing the BRGE to a BRGT in the second source file. Figure 3.17 is an example output trace where the equality of the two outputs could not be validated. This error was created by adding the line *ADD dx, 2* to the second source file after the line *ADD dx,cx*.

Source 1	Source 2
0:MOV 2:dx 3:42	0:MOV 2:x0 3:Xarg1
1:MOV 2:a0 3:(Xarg1)	1:MOV 2:a0 3:(x0++1) 4:a1 5:(x0)
2:MOV 2:b0 3:(Xarg2)	2:MOV 2:dx 3:-42
3:MUL 2:cx 3:a0 4:b0	3:MOV 2:cx 3:dx
4:CMP 2:cx 3:dx	4:NEG 2:cx
5:BRLT 2:exit	5:MSM 2:dx 3:a0 4:a1
6:SUB 2:cx 3:dx	6:BRGE 2:exit
7:exit	7:ADD 2:dx 3:cx
8:MOV 2:result 3:cx	8:exit
9:RET	9:MOV 2:result 3:dx
10:.AMEM	10:RET
11:.mem 2:arg1 3:1	11:.AMEM
12:.mem 2:arg2 3:1	12:.mem 2:arg1 3:1
13:.BMEM	13:.mem 2:arg2 3:1
	14:.BMEM

Figure 3.12: Parsed Code Listing

Given Equivalencies	Desired Equivalencies
Xarg11 Xarg12	{read memory1 result} {read memory2 result}
Xarg21 Xarg22	

Figure 3.13: Given and Desired Equivalencies

```
Line Number (begin block, end block) - (leftchildBegin,leftChildEnd)
               (rightChildBegin,rightChildEnd)
```

Source 1	Source 2
1(0-5) - (5-7) (7-9)	1(0-6) - (6-8) (8-10)
2(5-7) - (7-9)	2(6-8) - (8-10)
3(7-9)	3(8-10)

Figure 3.14: Control Flow Graphs

```
THE TRACE WAS VALID
Depth 0, block1:  0, 5 FALSE , block2:  0, 6 TRUE

Depth - how many branches were taken up to this point.
block - what lines of code were contained in this block of code.
FALSE/TRUE - what branching choice was made at the end of this block
```

Figure 3.15: Example Output for Valid Trace

```
Got two branches in the verify with BRLT BRGT
assert (= N1 { + ( MULT {read EMPTY1 Xarg11 } {read EMPTY1 Xarg21 } )
                                              { * -1 42 } } )
assert (= N2 { + -42 ( MULT {read EMPTY2 Xarg12 } {read EMPTY2
                                              {+ 1 Xarg12 } } ) } )
BRANCHES NOT EQUAL, CONTROL STRUCTURE INVALID
```

Figure 3.16: Example Output for Incompatible Branch

```
THE WHOLE THING IS NOT VALID due to check_valid (= cx1 dx2)
Depth 0, block1:  0, 5 TRUE , block2:  0, 6 FALSE
assert (= cx1 ( MULT {read EMPTY1 Xarg11 } {read EMPTY1 Xarg21 } ) )
assert (= dx2 { + ( MULT {read EMPTY2 Xarg12 } {read EMPTY2 Xarg22 } )
                                               { * -1 -42 } -40 } )
```

Figure 3.17: Example Output for Invalid Trace

Chapter 4

Industrial Examples

The goal of the software developed was to verify the correctness of two structurally similar programs such as would occur when a program has been optimized or when the code is generated from a higher level specification.

To test that our tool was able to do the above verification, we obtained a set of DSP benchmarks taken from part of a large Fujitsu cellular telephone application. The benchmarks consist of four examples: hup, kncal, yhaten and dt_pow. Each example has a hand-coded version and a generated version that was believed to be equivalent. The generated version is created by the SPAM-based Elixir compiler [31]. The compiler uses a C language source code that is semantically equivalent to an algorithmic description of the program included with the hand-coded version of the program. The examples range in size from 37 lines to 190 lines of code for the generated version, including memory declarations. These examples were obtained under a non-disclosure agreement with Fujitsu Laboratories of America so we are not able to give a full code listing or detailed descriptions of the functionality.

Since the hand-coded version was the code actually being used in the Fujitsu application, we assumed that if a discrepancy occurred between the two programs then the error was likely to be in the generated code. The program did, however, locate one error that we believe is an error in the hand-coded version.

The following sections describe in detail the DSP being simulated and some general things that needed to be fixed in order for the code to verify correctly. We emphasize that all bugs were found fully automatically. Finally, a table is presented with some performance statistics for each example.

4.1 Fujitsu DSP

The Elixir is a proprietary DSP used in cellular phones by Fujitsu. It is composed of a 16-bit digital processor core, program and data memories, and supporting peripherals.

The chip contains two 40-bit accumulators (CX and DX), four 16-bit general-purpose data registers (A0,A1,B0 and B1) and eight 16-bit address registers (X0,...,X8).

The data memory banks are divided into two banks, ARAM and BRAM, which can be accessed in parallel.

The Elixir ISA supports three addressing modes: *Absolute addressing, Register-indirect addressing*, and *Modulo addressing.*

The chip also has the capacity for a significant amount of instruction-level parallelism between the ALU and data memory banks.

For a more complete description, see [31].

4.2 Yhaten

The Yhaten example was the smallest of the examples at 37 lines. The code contained no branches and one fixed-count REP instruction.

The example also contained no errors other than minor syntax errors in the compiler-generated code.

4.3 Hup

The Hup example was the next largest at 47 lines. The code contained only one branch to check a rounding flag and set the appropriate register and one fixed-count DO loop to calculate a sum and multiplication.

Problems:

- The generated code lacked the aforementioned branch. The tool caught this readily apparent problem before even finishing a trace of the CFG, because the branching structure was not the same.

  To correct the problem, code was added to duplicate what was present in the hand-coded version.

- The generated code explicitly set a register used in the modulo addressing mode. In the hand-coded version, the register was set from a value stored in a memory location. This problem was caught, since the expressions constructed after simulating the code were not equivalent.

  To correct the problem, the MOV in the generated code was changed to take its value from an equivalent memory location.

4.4 Kncal

Kncal was the second largest example at 69 lines. The code contained a much more interesting branching structure (figure 4.1). The code also contained a fixed-count REP loop to calculate a division, as well as a much more interesting array of operations, such as shifting, negating and logical anding.

Problems:

- The generated code used LSL (logical shift lefts) where the hand code was using ASL (arithmetic shift lefts). Depending on the values of the input, this

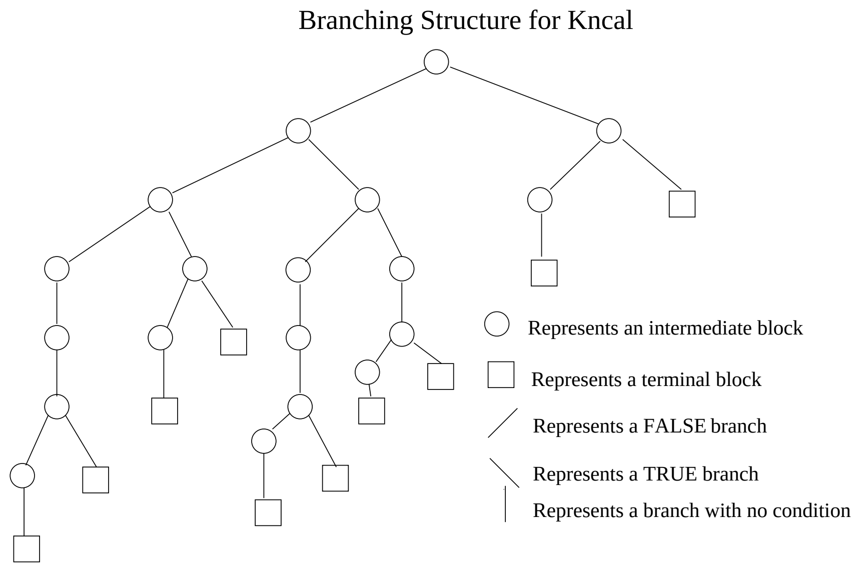

Figure 4.1: Tree Representation of CFG for Kncal

could create a bug since, in the ASL the top bit of the register is not shifted out, while in the LSL it is.

To fix this problem the LSLs were changed to ASLs.

- The generated code made use of temporary memory locations set during the normal flow of the program. Given one set of possible branching choices, however, a temporary memory location was accessed without having been set.

  This was fixed by inserting an extra MOV operation to set the value of the temporary memory location before entering the branches that accessed that particular address.

- One potential problem that was found was the use of an LSL followed by an ASR of a register. The corresponding piece of code in the hand-coded program was a MOV of a subset of bits in the register back into the register. In this particular DSP, this type of MOV has the effect of zeroing out the rest of the register and just leaving the bits that were moved.

Unfortunately, this "bug" also illustrates a significant limitation in the tool as it is implemented currently, namely the difficulty in reasoning about register sizes and some individual bit operations. This limitation is discussed further in the following chapter.

The fix used to get past this problem was to change the LSL, ASR code to the equivalent code used by the hand-coded program.

4.5 Dt_pow

Dt_pow was the largest example at 190 lines. Due to its size and complexity, it also yielded the most interesting results, including a discrepancy that we believe may be an error in the original production code.

The branching structure is quite complex (figure 3.5) and would be very difficult to analyze by hand. Further complicating this example was the extensive use of auto-incrementing and decrementing in the hand-coded example. Thus, at many points in the program it was difficult to determine which memory address was actually being accessed by visual inspection alone.

The program contained numerous branches, a fixed-count REP loop to calculate a division and a wide spectrum of operations.

Problems:

- In the hand-coded version of the program, a register is used that is assumed to contain a value from a memory location called act1. However, in one particular flow of control, this register is not initialized. We believe this is an error in the hand code, since the algorithm describing the function of the program indicates that act1 should have been the value of the register at this point in the program. In addition, the generated version also properly sets the register at the equivalent point in its execution.

  To fix this problem, a line was inserted in the appropriate location to set the register to act1.

- As in the Kncal example, a large number of temporary memory locations are used in the generated code. Both programs contain a branch that allows the program to skip the initialization of a number of memory addresses. However, in the hand-coded program, these addresses would then have a default value that was originally assigned. In the generated code, the temporary memory locations would be initialized to the same location as in the hand code if the initialization stage was not skipped. If this section is skipped, then these temporary locations are not set to anything and would not be equivalent to the hand-coded version.

  To solve this problem, a number of lines were inserted to initialize these temporary registers to the same default value as in the hand-coded version.

- Another initialization problem became apparent in certain branches. Two registers were being used that had not been initialized in the generated code.

  To solve this two lines were inserted to do the required initializations.

- A multiplication error was found in the generated code that was also present in the C description. In the generated code, a sum was multiplied by two, where as in the hand-coded version, only one of the arguments of the sum was multiplied by two.

  The problem was solved by moving the location of the multiplication by two in the generated code, to match that found in the hand code.

- A number of memory locations in the hand code are set in to specific values at the end of the program. In the generated code the memory locations are set, but to a different value.

  The problem was fixed by altering the generated code to set these memory locations before terminating (this is a known bug in the SPAM compiler).

original syntax	modified syntax
MOV a0, X:hupkn	MOV a0, (X:hupkn)
MOV a0, (B:tapx[0])	MOV bv, B:huptapx MOV a0, [bv+x7]

Table 4.1: Examples of Syntactic Corrections

4.6 Common Problems

The Elixir compiler had a number of syntactic problems that needed to be corrected before the examples could be compared (table 4.1). One discrepancy was the lack of brackets around memory locations to indicate that the value contained at that location was being retrieved and not that the address value was being moved. Another problem was the notation used for modulo addressing. The Fujitsu DSP allows a specialized modulo addressing mode to support the use of circular data buffers.

4.7 Performance

The performance for each example is given in table 4.2. The first column is the number of lines of code in the compiler-generated version of the assembly code. The total time is the total clock time taken by the program to verify the assembly code, given no errors were found. The third column gives the clock time that the program spent in SVC. The fourth column gives the maximum memory used during the verification and the last column indicates the largest expression that was generated during the verification process for a valid example.

The memory usage was shared between SVC and our program, but a large percentage of the time (table 4.2) for execution was spent in SVC. Most of the examples ran in a reasonable amount of time with the greatest slowdown occurring during loops due to the large amount of memory simplifications that needed to be performed.

Example	Code Size (lines)	Total Time (seconds)	SVC Time (seconds)	Mem. Usage (megs)	Max. Str. Len. (characters)
Dt_Pow	190	245	243	8	1029
Kncal	69	5	5	3	708
Yhaten	37	254	121	77	63629
Hup	47	15h	14h	120	122854

Table 4.2: Tool Performance on Examples

The tests were run on a Sun Ultra 60 (360mhz) with 750 megs of memory.

Chapter 5

Limitations and Future Work

The following sections touch on some of the limitations of the tool as it exists currently. The limitations will be discussed with respect to the techniques used and the limits of the tool. We will also discuss some things that we would like to try and some future challenges.

5.1 Limitations

Many of the limitations of the tool have been mentioned already. The following sections summarize them.

5.1.1 Bit Reasoning

The lack of ability to reason about numbers at the bit level is one of the more serious limitations of the current approach. As a result of this limitation, we were forced to implement axioms and simplification rules to handle such operations as shifting and anding in order to be able to compare the two examples. It is this limitation which is largely responsible for the non-conservative nature of the tool. For instance, by modeling the shift as a multiplication by two, the tool does not consider possible overflows.

The worst instance of where the tool fails is when the MOV instruction is used to move a subset of bits out of a register and then back into the register. This has the effect of zeroing out the low and high order bits, depending on the flags set. In its current state, the tool handles this by treating the larger registers as smaller ones and clearing the registers representing the high and low order bits.

There is a type in SVC which allows bit vectors to be declared and manipulated, but the bit vectors do not allow uninterpreted functions to be used. As soon as an uninterpreted function is used on the bit vector, it is no longer possible to reason about the individual bits.

5.1.2 Modulo Addressing

The Fujitsu chip is equipped with a modulo addressing mode that uses the value in a modulo register (MD). The way it is implemented currently is to use an if-then-else statement:

$$\text{(ite } (\geq \text{ foo MD) (- foo MD) foo)}$$

Unfortunately this is only true if $foo < 2 * MD$. This problem also occurs even if $foo < MD$ when foo is incremented forward inside a loop.

$$\text{(ite } (\geq \{+ \text{ foo inc}\} \text{ MD) (- } \{+ \text{ foo inc}\} \text{ MD) } \{+\text{foo inc}\})$$

since $\{+$ foo inc$\}$ could be greater than $2 * MD$. This second case could be handled by nesting the *ite*s during the loop.

$$\text{(= foo (ite } (\geq \{+ \text{ foo inc}\} \text{ MD) (- } \{+ \text{ foo inc}\} \text{ MD) } \{+ \text{ foo inc}\}))$$
$$\text{then (= foo (ite } (\geq \{+ \text{ foo 1}\} \text{ MD) (- } \{+ \text{ foo 1}\} \text{ MD) } \{+ \text{ foo 1}\} \text{))}$$

This still does not handle the first case.

5.1.3 Memory Simplifications

The memory simplification process can become too slow. As the memory grows, each read has to potentially search through the entire memory. In a loop that writes to a new memory location, the number of compares is quadratic in the number of iterations of the loop.

This slowdown is particularly obvious in the Hup example, as it contains a loop of 512 iterations that writes twice to memory in every loop.

5.1.4 Arithmetic Problems

Currently SVC does not support non-linear arithmetic. It is a known hard problem to do equality on polynomials. For this reason, the re-ordering heuristic was implemented. The heuristic is not foolproof and it would be relatively easy to concoct examples that would break the re-ordering. However, the heuristic was "good enough" on the examples and may in fact be powerful enough in general.

5.1.5 Axiom Handling

SVC does not currently handle axiom declarations (or quantified statements) which contain variables such as:

$$\forall \mathrm{x}\ \forall \mathrm{y}\ (= (\mathrm{LSR\ x\ y})\ (\mathrm{MULL\ x}\ 2^{y}))$$

A fix for this problem would need to be located in the decision procedure engine itself.

5.2 Future Work

In addition to the above limitations, the following things would also increase the power of our method (tool). Some would be mostly a matter of finding time, while others would require some rethinking on how to verify the code.

5.2.1 Language Module

Currently the program is only able to handle the Fujitsu Elixer ISA. In order to increase the applicability of the tool, it would be nice to remove the chip specific information contained in the program and replace it with a more general design.

A language, of sorts, could then be used to specify the architecture of the chip and the functionality of the opcodes available on that specific chip. The description of the architecture could then be mapped to a more general architecture on which the tool runs (as is done in many existing hardware verification tools).

We do not see this as a major problem but as more of an engineering exercise. The use of symbolic simulation makes it easy to model complex DSP instructions and would allow the tool to be extended to other DSP architectures.

5.2.2 Interface with Decision Procedure

The current interface with the decision procedure, SVC, is by string-passing of the completed logic expressions. This string passing is cumbersome and unnecessary. It would be more logical and easier to build the expressions directly in SVC.

Due to the simplifications that were required to keep the expressions from exploding and due to the unpredictable behaviour observed in SVC, this option was abandoned after being the method of interface earlier in the project).

An obvious future improvement is to use a more stable decision procedure[1] and to build the expressions directly in this tool.

5.2.3 Loop Handling

Currently the only method of loop handling is by unrolling which allows the tool to only handle fixed count loops.

An idea that has been proposed is to use fixed points. The fixed point would be calculated by obtaining equalities before entering the loop, doing a pairwise comparison of all values currently in use in the two programs. The loop could then be executed once and the equalities re-checked. If the new equalities can be derived from the assumptions found in the previous loop, then the fixed point has been

[1]It should be noted that in general, SVC behaved admirably and is currently in the process of being improved further.

found. Otherwise the assumptions would need to be relaxed and another iteration carried out.

It is not clear if this technique would be powerful enough to handle most cases.

5.2.4 Control Flow Graph Matching

The tool currently requires that the Control Flow Graphs be equivalent. This is unduly restrictive and could be relaxed.

One possibility for relaxing this restriction is to allow the order in which the branches appear to be different in the two programs. Then, instead of pausing when a branch is encountered and verifying that the other program has hit a similar branch, one would execute the entire trace in one program, building up a list of branches and the conditions upon which they depend. The second program could then be simulated. Each time a branch is encountered, the list of branches and conditions that were built previously could be searched for a corresponding branch. That branch could then be marked as "used", and the simulation would continue. This would allow the two programs to be organized differently but still have the same basic control flow graph.

Another idea, which would allow the programs to have completely different control flow graphs, is to build up one large expression for the entire simulation. At each branch the two possibilities would be connected by an if-then-else statement where the *if* condition would be the branch condition. The resulting expression of this breadth first construction could then be handed to the decision procedure along with the corresponding expression from the other source and verified. This approach was problematic in that SVC did not always behave nicely when handed very large strings. The other problem with this approach would be the explosion in the size of the expression.

Chapter 6

Conclusion

This thesis has presented a new method for the formal verification of DSP assembly code. By combining a number of theories, we have been able to construct a tool that has proven to be effective at finding bugs in real production code.

The tool presented constructs a control flow graph by first breaking the code into basic blocks and carrying out a number of sweeps over the blocks (and initial CFG) to construct and refine the CFG. The refined graph is then traced in a depth first manner to simulate the execution of the programs being verified. We have demonstrated that this technique allows the program to be simulated while avoiding some of the blow up that is often a bottle neck in many methods. It has also been shown how the CFG could be used to further subdivide the program into smaller chunks, again with the purpose of avoiding blow up.

By using uninterpreted functions, the program is also able to handle multiplication (an operation that is often problematic for BDD-based methods) and a number of other complex operations. These operations include shifting, anding and negating.

In order to verify the expressions generated during the symbolic simulation of the program, the tool has been successfully integrated with the Stanford Validity Checker. By using SVC, we have access to a powerful decision procedure engine that allows us to use uninterpreted functions and constants, first order predicate logic, read and write array axioms and linear arithmetic. These theories combined provided enough power and flexibility to carry out the verification that was required.

The tool presented could be easily expanded to handle larger programs, as the performance indicates that both time and memory were not serious problems. In addition, it could also be generalized to handle different and potentially more complex architectures for different DSPs.

This thesis has demonstrated that not only is verification of DSP assembly code possible, but extremely useful. The method requires little interaction from the user and yet is still able to find bugs.

A number of avenues for future work have been discussed in the previous chapter, but even in its current form, we believe that our approach represents a good first step in the quest for a method that can be fully automated and still provide useful results in the difficult world of software verification.

Bibliography

[1] A. Cohn. The notion of proof in hardware verification. *Journal of Automated Reasoning*, 5(4):127–139, 1989.

[2] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Publishing Company, Don Mills, Ontario, March 1988.

[3] S. Balakrishnan and S. Tahar. On the formal verification of embedded systems using multiway decision graphs. Technical Report TR-402, Concordia University, Montreal, Canada, 1997.

[4] C.W. Barret, D.L. Dill, and J.R. Levitt. Validity checking for combinations of theories with equality. In *Formal Methods In Computer-Aided Design*, Palo Alto, CA (USA), November 1996. Springer Verlag. Lecture Notes in Computer Science 818.

[5] B. Brock and W.A. Hunt. Report on the formal specification and partial verification of the VIPER microprocessor. In *Compass '91: 6th Annual Conference on Computer Assurance*, pages 91–98, Gaithersburg, Maryland, 1991. National Institute of Standards and Technology.

[6] B.C. Brock and W.A. Hunt, Jr. Formally specifying and mechanically verifying programs for the Motorola complex arithmetic processor DSP. In *International Conference on Computer Design: VLSI in Computers and Processors (ICCD '97)*, pages 31–36, Washington, October 1997. IEEE.

[7] R.E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):667–691, August 1986.

[8] R.E. Bryant. On the complexity of VLSI implementations and graph representations of boolean functions with application to integer multiplication. *IEEE Transactions on Computers*, 40(2):205–213, February 1991.

[9] R.E. Bryant and C. Seger. Formal verification of digital circuits using symbolic ternary system models. *Lecture Notes in Computer Science*, 531:33–38, 1991.

[10] J.R. Burch and D.L. Dill. Automatic verification of pipelined microprocessor control. *Lecture Notes in Computer Science*, 818:68–85, 1994.

[11] W.C. Carter, W.H. Joyner, Jr., and D. Brand. Symbolic simulation for correct machine design. In *16th Design Automation Conference Proceedings*, pages 280–286, New York, USA, June 1979. IEEE Computer Society Press.

[12] L. Claesen, M. Genoe, F. Proesmans, E. Verlind, and H. De Man. SFG-Tracing: A methodolgy for the automatic verification of MOS transistor level implementations from high level behavioral specifications. *Lecture Notes in Computer Science*, 1991.

[13] D.L. Clutterbuck and B.A. Carré. The verification of low-level code. *IEE/BCS Software Engineering Journal*, 3(3):97–111, May 1988.

[14] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. McGraw-Hill Book Company, 13 edition, 1994.

[15] E.M. Clarke and E.A. Emerson. Design and Synthesis of Synchronization Skeletons using Branching Time Temporal Logic. In D. Kozen, editor, *Proceedings of the Workshop on Logics of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71, Yorktown Heights, New York, May 1981. Springer-Verlag.

[16] D.L. Dill et al. SVC home page. <URL:http://sprout.Stanford.EDU/SVC/>, 1999.

[17] F.H.M. Franssen, F. Balasa, M. van Swaaij, F.V.M. Catthoor, and H.J. De Man. Modeling multidimensional data and control flow. *IEEE Transactions on very large scale integration (VLSI) systems*, 1:319–327, September 1993.

[18] M. Genoe, L. Claesen, E. Verlind, F. Proesmans, and H. De Man. Automatic formal verification of Cathedral-II circuits from transistor switch level implementation up to high level behavioral specifications by the SFG-tracing methodolgy. In IEEE, editor, *Proceedings of the European Conference on Design Automation*, pages 54–58, Washington, March 16–19 1992. IEEE Computer Society Press.

[19] M. Gordon. HOL:A machine oriented formulation of higher order logic. Technical Report TR-68, Cambridge University, Cambridge, England, 1985.

[20] A. Gupta. Formal hardware verification methods: A survey. *Formal Methods in System Design*, 1:151–238, 1992.

[21] A.J. Hu, D.L. Dill, A.J. Drexler, and C. Han Yang. Higher-level specification and verification with BDD's. In *Computer-Aided Verification: Fourth International Workshop*, July 1992.

[22] W.M. Hunt. *FM8051: A Verified Microprocessor*. PhD thesis, University of Texas at Austin, 1985.

[23] J. Jain, A. Narayan, M. Fujita, and A. Sangiovanni-Vincentelli. A survey of techniques for formal verification of combinational circuits. In *International Conference on Computer Design: VLSI in Computers and Processors (ICCD '97)*, pages 445–454, Washington, October 1997. IEEE.

[24] R.B. Jones, D.L. Dill, and J.R. Levitt. Efficient validity checking for processor verification. In *IEEE/ACM International Conference on Computer Aided Design*, 1995.

[25] J.R. Burch, E.M. Clarke, K.L. McMillan, and D.L. Dill. Sequential Circuit Verification Using Symbolic Model Checking. In *Proceedings of the 27th ACM/IEEE Design Automation Conference*, pages 46–51, Los Alamitos, CA, June 1990. ACM/IEEE, IEEE Society Press.

[26] M. Kaufmann and J.S. Moore. ACL2: An industrial strength version of nqthm. In *Compass'96: Eleventh Annual Conference on Computer Assurance*, page 23, Gaithersburg, Maryland, 1996. National Institute of Standards and Technology.

[27] Z. Manna and A. Pnueli. Verification of concurrent programs: The temporal framework. In R. S. Boyer and J. S. Moore, editors, *The Correctness Problem in Computer Science*. Academic Press, New York, NY, 1981.

[28] K. McMillan. Ken McMillan's home page at UCB. <URL:http://www-cad.eecs.berkely.edu/~kenmcmil>, 1999.

[29] K.L. McMillan. *Symbolic Model Checking*. Kluwer, 1993.

[30] E. Mendelson. *Introduction to Mathematical Logic*. Van Nostrand, New York, 1964.

[31] S.P. Rajan, M. Fujita, A. Sudarsanam, and S. Malik. Development of an optimizing compiler for a Fujitsu fixed-pointed digital signal processor. 1999. Accepted for publication in 1999 International Workshop on Hardware/Software Codesign.

[32] R.S. Boyer and J.S. Moore. *A Computational Logic Handbook*. Academic Press, 1988.

[33] C. Seger. An introduction to formal hardware verification. Technical Report TR-92-13, Department of Computer Science, University of British Columbia, June 1992. Tue, 22 Jul 1997 22:21:30 GMT.

[34] M. Srivas and M. Bickford. Formal verification of a pipelined microprocessor. *IEEE Software*, 7(5):52–64, September 1990.

[35] O. Thiry and L. Claesen. A formal verification technique for embedded software. In *IEEE International Conference on Computer Design*, pages 352–357, New York, USA, 1996. IEEE Computer Society Press.