Automatic Formal Verification of DSP Software *

David W. Currie†
Mentor Graphics
Billerica, MA
david_currie@mentorg.com

Alan J. Hu
Dept. of Computer Science
University of British Columbia
Vancouver, BC, Canada
ajh@cs.ubc.ca

Sreeranga Rajan
Masahiro Fujita‡
Fujitsu Laboratories of America
Sunnyvale, CA
sree@cad.fla.fujitsu.com
fujita@cad.fla.fujitsu.com

ABSTRACT

This paper describes a novel formal verification approach for equivalence checking of small, assembly-language routines for digital signal processors (DSP). By combining control-flow analysis, symbolic simulation, automatic decision procedures, and some domain-specific optimizations, we have built an automatic verification tool that compares structurally similar DSP assembly language routines. We tested our tool on code samples taken from a real application program and discovered several previously unknown bugs automatically. Runtime and memory requirements were reasonable on all examples. Our approach should generalize easily for multiple DSP architectures, eventually allowing comparison of code for two different DSPs (e.g., to verify a port from one DSP to another) and handling more complex DSPs (e.g. statically-scheduled, VLIW).

1. INTRODUCTION

Software for digital signal processors (DSP) is a particularly promising area for automatic formal verification:

- DSPs have become ubiquitous as countless formerly analog application domains have become digital (e.g., audio, video, telecommunications, industrial control, etc.) and countless new applications emerge. The commercial importance of digital signal processing is enormous.
- Unlike software for desktop and larger computers, hand-written assembly-language programming is still common for DSPs. Even when compilers are used, the generated code is often hand-tuned to meet stringent performance and code-size requirements. In many respects, the situation resembles synthesizing hardware, where post-synthesis modifications and optimizations are common, resulting in a need for equivalence checking.
- Programming DSPs is difficult for both humans and compilers. Typical DSP architectures are highly non-orthogonal, with specialized instructions performing multiple operations in parallel, special purpose registers, and complicated addressing modes. Some recent DSPs even lack pipeline interlocks, forcing the programmer to ensure that the code is scheduled properly. Exploiting the specialized features of the DSP is imperative to attain maximum performance.

Even a very limited tool to check the correctness of small, localized code optimizations would be very useful in practice.

In this paper, we describe such a tool and the underlying formal verification algorithm. The verification method is based on a combination of control flow analysis, symbolic simulation, and co-operating decision procedures for memory, linear arithmetic, and uninterpreted functions with equality. To produce useful results, these techniques are augmented with some DSP-specific optimizations and various simplifying assumptions. As in combinational equivalence checking, the problem is to compare two similar designs (as would occur if checking small optimizations done to compiled/synthesized output), so we use the similarity to simplify the verification problem. As in model checking, the goal is to provide powerful, practically useful debugging support; the tool cannot certify correctness.

We have built a prototype implementation of the tool, targeting the assembly language of the Fujitsu Elixir, a 16-bit fixed-point DSP primarily used in cellular telephones. We chose this DSP as our target because we were able to obtain assembly-language routines for it taken from a real application. Testing the tool on these routines has demonstrated the effectiveness of our new method.

2. BACKGROUND

2.1 Digital Signal Processors

Obviously, a full treatment of digital signal processors is beyond the scope of this paper. Many references are available, and data books on specific DSPs are easily obtained from their manufacturers.

A digital signal processor is essentially a microprocessor that has been designed specifically to execute typical digital signal processing algorithms very efficiently. Common operations that must be

*This work was supported by grants from Fujitsu Laboratories of America and the National Science and Engineering Research Council of Canada.

†Work was performed while at the University of British Columbia.

‡Current affiliation is the Department of Electrical Engineering, University of Tokyo, 7-3-1 Bunkyo-ku, Tokyo 113-8656, Japan.

efficient include fixed-count loops to iterate over data samples, multiplication and multiply-accumulate for various calculations, and data movement between memory and the registers used for computation. The overriding design goal for a DSP is generally the highest performance (on DSP algorithms) for a given price point or the lowest cost for a given performance target, so other design goals typically expected of a general purpose processor (ease of programming, ease of compiler code generation, software compatibility, etc.) are secondary. As a result, DSPs typically have non-orthogonal architectures, with special purpose registers, complex addressing modes, specific support for fixed-count loops, and other specialized features.

For understanding this paper, one may consider a DSP to be simply a microprocessor with many unusual features. For a simple example, consider the following code fragment for the Fujitsu Elixir:

```
        msm dx, a0, a1
        bge ok
        add dx, cx
ok:     mov (x0++1), dx
```

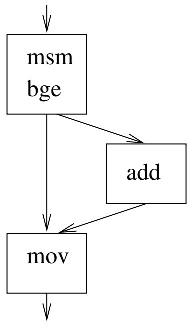
The first instruction multiplies registers `a0` and `a1` and adds the result to register `dx`. This instruction also sets condition codes, so the following branch instruction branches to label `ok` if the result was nonnegative. The `add` instruction adds register `cx` to `dx`. The `mov` instruction stores the contents of register `dx` into the memory location pointed to by index register `x0`, and then increments `x0` to point to the next memory location.

2.2 Control Flow Analysis

The control flow analysis we used is standard and is described in numerous sources (e.g. [1]).

A *basic block* is a maximal contiguous sequence of statements that can be entered only at the first statement and exited only after the last statement. In the simple example above, the `msm` and `bge` instructions form one basic block, the `add` instruction is a second basic block, and the `mov` instruction is a third.

The basic blocks can be combined to form a *control flow graph* (CFG). An arc goes from one basic block to another if the control flow can proceed directly from one to the other. A path through the CFG is called a *trace* and represents one possible flow of control through the program. For our simple example, the CFG is as follows:



2.3 Symbolic Simulation

Symbolic simulation has become a standard technique in hardware verification (e.g., [5, 4]), although the general concept was originally proposed for software. The basic idea for verifying digital circuits is that rather than simulating a circuit for a particular input vector, we can simulate it with symbolic variables at the inputs and compute the circuit behavior as a Boolean function of the symbolic variables.

For software, one could employ the same approach, treating data values as bit-vectors, exactly as the underlying processor would. Symbolic simulation proceeds using vectors of Boolean functions. The advantage of this approach is that the simulation is bit-for-bit accurate. The disadvantage is that the complexity of the generated Boolean functions quickly blows up, especially if the software performs operations like multiplication and division, which DSP programs do. Because of this complexity blow-up, we need a different approach.

Instead, we can treat variables as arbitrary values, rather than as bit-vectors. Some functions, such as addition or if-then-else, can be easily handled using arithmetic and logical operators. For more complicated functions, we can introduce *uninterpreted function symbols* — newly named functions about which we assume nothing except that it's a function. These uninterpreted functions let us abstract away some of the complexity of the program. Memory can be handled by introducing special memory functions *read*, which takes a memory and an address and returns the value of the most recent write to that address, and *write*, which takes a memory, an address, and a value, and returns a new memory which has that address updated to the new value.

A powerful advantage of symbolic simulation is that it is fundamentally simulation. No complex mathematical model of the system needs to be constructed. For switch-level circuit simulation, the symbolic simulator is identical to a normal simulator, except that symbolic values are computed. For DSP assembly language, we need only create a simulator for the instruction set architecture (programmer's view of the processor). Such a simulator consists simply of the programmer-visible state of the processor (registers, condition codes, etc.) and code to perform the effect of each opcode, which can be copied directly from the processor manual. In our project, simulating the Fujitsu Elixir was straightforward; simulating any other DSP should be similarly easy.

Returning to our simple example, if we symbolically simulate the trace that skips the `add`, we find at the end of the trace that:

$$\begin{aligned}
\texttt{dx} &= \mathit{initial_dx} + f(\mathit{initial_a0}, \mathit{initial_a1}) \\
\texttt{x0} &= \mathit{initial_x0} + 1 \\
\texttt{memory} &= \mathrm{write}(\mathit{initial_memory}, \mathit{initial_x0}, \\
&\quad\; \mathit{initial_dx} + f(\mathit{initial_a0}, \mathit{initial_a1}))
\end{aligned}$$

where f is an uninterpreted function used to denote multiplication.

2.4 Automated Decision Procedures

If the logic used in the symbolic simulation is decidable, then the results can be compared using an automated decision procedure. For this work, we used the Stanford Validity Checker (SVC) [6], an efficient decision procedure that handles the combined theories of propositional logic, linear arithmetic (arithmetic with addition and multiplication, but multiplication can only be by constant factors), read/write to memories, and uninterpreted functions with equality (e.g.,$a = b$ implies $f(a) = f(b)$ regardless of what f is). This combination of theories was adequate for our purposes in most cases.

2.5 Related Work

We were inspired by the highly successful work on equivalence checking of combinational circuits (e.g. [7] is a recent survey), although at a technical level there are few similarities with our work. The key insights are that comparison of two very similar designs is an important practical problem and that conservatively exploiting that similarity greatly simplifies verification.

The general software verification literature contains no closely related work. The naive application of successful hardware verification techniques to software is frustrated by theoretical undecidability results, extremely large (considered infinite) state spaces, while loops, recursion, pointers, etc. We will see that low-level DSP software does not exhibit many of the features that complicate software verification in general, making DSP software particularly appropriate for the techniques we describe.

Non-DSP embedded software has some similarities to DSP software, in particular stringent performance and code size requirements, hand-optimization, and the concomitant need for verification. There has been some work in verifying assembly-language programs for embedded systems. For example, Thiry and Claesen [10] proposed a model-checking approach based on BDDs. Balakrishnan and Tahar [2] proposed a similar approach based on the more general multiway decision graph to avoid some BDD size problems. Both were able to verify a mouse controller and find inconsistencies between the assembly code and flow chart specifications.

For verification of DSP software, Brock and Hunt's work on the Motorola Complex Arithmetic Processor stands out [3]. They were able to verify assembly code by specifying the entire processor in ACL2 logic and using the ACL2 theorem-prover to carry out the mechanical proofs. Their verification is bit-for-bit accurate and capable of verifying algorithms involving both hardware and software components. Although much more extensive and accurate than our work, their specification of the chip required eight man-years of effort. Our work, in contrast, trades accuracy for ease-of-use. In an application for which the highest assurance of correctness is needed (and can be cost-justified), Brock and Hunt's approach is clearly superior. Our approach, on the other hand, offers quick and easy (but still reasonably powerful) debugging assistance, for more routine usage.

3. VERIFICATION APPROACH

3.1 Simplifying Assumptions

Software verification in general is notoriously difficult, so we have imposed several simplifying assumptions. The main assumptions are structural:

- We assume we are comparing a single function to another single function, i.e. we do not allow subroutines, co-routines, interrupts, etc. In practice, we envision the tool being applied to bottom-level functions in a much larger program. Such a decomposition could be automated.
- The two routines must have very similar CFGs. Specifically, the traces of the two must encounter corresponding branches. This restriction exploits the fact that we are verifying that small changes made to a piece of code do not change its functionality, making it likely that the CFGs will be very similar.
- To simplify control flow analysis, we do not allow arithmetic on the program counter and self-modifying code. Such programming practices are generally disparaged and are actually impossible on most DSPs.
- To guarantee decidability, we do not permit while-loops or recursion. Fortunately, neither of these are common in low-level DSP routines, in part because of the need for easily bounded runtimes. Fixed-count loops, of course, are very common, and we do allow them.

The other simplifying assumptions regard the notion of equivalence:

- We consider two routines to be equivalent if the user-specified outputs are the same for all traces.
- We ignore rounding and precision. Because we are using uninterpreted constants and function symbols (to avoid complexity blow-up), we cannot model the actual bit precisions of registers and operations. Arithmetic is essentially infininte-precision.
- In practice, extremely powerful optimizations can be based on subjective user testing. A routine might compute a completely different mathematical result, but it's "equivalent" if most users don't notice the difference. Obviously, such a notion of equivalence is beyond what can be expected of our tool.

Of the simplifying assumptions, only the rounding and precision issue is particularly troubling. The other assumptions are trivial to assure, or are conservative — the tool may erroneously declare two routines inequivalent if an assumption is violated, but will not declare equivalence of inequivalent routines. The inability to model accurately at the bit-level, on the other hand, could allow the tool to incorrectly declare equivalence in some cases. This is the most serious weakness of our verification method. If we consider the tool a debugging aid rather than a certification of correctness, however, and given that the verification is completely automatic, the tool should still be useful despite this theoretical limitation.

3.2 Verification Overview

Given the simplifying assumptions, the overall verification approach is straightforward. The user specifies input equivalences and which outputs are supposed to be equivalent. The tool performs a depth-first traversal over the CFGs of the two routines being compared, symbolically simulating along the way. At each branch, the branch conditions (as a function of the inputs) are checked for compatibility: identical branch conditions or complementary branch conditions maintain correspondence between the two routines. At the end of each trace, the outputs are checked for equivalence. If a miscompare is found, the tool produces an error trace showing the flow of control to the problem.

An obvious alternative would be to build a single expression for the outputs over all possible traces, rather than comparing traces individually. Unfortunately, we encountered some performance issues with SVC, so we chose the approach described here.

3.3 Verification Details

The overall approach may be simple, but creating a useful tool, not surprisingly, requires addressing many practical details. Our improvements to the basic method can be grouped into three general categories: handling specific architectural features, increasing efficiency, and strengthening the decision procedure.

Loop Unrolling: As mentioned, low-level DSP code contains numerous fixed-count loops. These loops are handled by unrolling them completely, as if they had been straight-line code. No loop invariants or fixed-point iterations are needed. Many DSPs, including the Fujitsu Elixir, have special instructions and hardware support for fixed-count loops, so identifying these loops in the code is especially easy.

Condition Codes: Condition codes can be handled in symbolic simulation just as any other register is. Branch instructions simply refer to the condition codes. No special modeling is needed for comparison instructions.

Modulo Addressing: Many DSPs, including the Fujitsu Elixir, have special modulo or circular addressing modes, which compute auto-increment and auto-decrement of index registers modulo a specified modulus. These addressing modes ease accessing circular data buffers. We handle these addressing modes with a simple if-then-else. For example, an auto-increment on `x0` would be treated as $(\text{ite } (\mathtt{x0}+1 \geq \mathtt{md})\ (\mathtt{x0}+1-\mathtt{md})\ (\mathtt{x0}+1))$, where `md` is the modulus. Note that the tool is not actually computing the mathematical modulo operator (e.g., if `x0` were more than twice `md`), but fortunately, real DSPs typically perform the same computation as our tool for these addressing modes, rather than true mathematical modulo.

Memory: A key efficiency improvement concerns our modeling of the DSP's memory. Memory was implemented as a single array with all reads and writes directed at this array using the read/write functions described earlier. This strategy was chosen because it correctly handles reads and writes to unknown locations, as well as relative addressing (using address arithmetic on index registers based on knowing the layout of data in memory). Handling memory accurately is imperative for verifying real programs. This modeling strategy, however, has the problem that the symbolic expression denoting the contents of memory can grow too large, especially when simulating unrolled loops that repeatedly write to memory.

The problem was largely contained by using three simplifications during symbolic simulation. The first simplification checks when a write is performed whether it overwrites a previous write to the same location, the contents of which can then be eliminated. The second simplification is done when a read is performed to check if the location sought has been written to, and returns the associated value if it has. The third simplification is performed in conjunction with the first to establish the earliest write for which every following write's location can be proven not to be the location currently being sought. This simplification incurs a time penalty, as it requires repeated calls to the decision procedure to prove that writes are to different locations. Combined, these simplifications greatly reduce the size of the symbolic expression for the contents of the DSP's memory.

Context Management: The verification method is based on a depth-first traversal of the CFG. For the tool to run efficiently, it must quickly recall the previous state of the partially simulated trace during backtracks. We call the state of the symbolic simulation at a given point in the program a *context*. The tool maintains a stack of contexts. When a branch in the CFG is encountered during the depth-first traversal, the current context is pushed onto the stack. When backtracking, the symbolic simulation can continue from this point by popping the previously saved context.

Axioms: The ability to specify axioms was crucial to strengthen the decision procedure enough to produce useful results. The verification tool makes extensive use of uninterpreted functions to model most operations on data (almost everything except addition and subtraction). Using uninterpreted functions abstracts away the specific operation being performed, greatly reducing the complexity of verification. Unfortunately, the theory of uninterpreted functions with equality is too weak to capture many properties we want. For example, suppose we use an uninterpreted function symbol `MULT` to denote multiplication. Given that $b = c$, SVC is able to prove that $(\mathtt{MULT}\ a\ b) = (\mathtt{MULT}\ a\ c)$, but it cannot prove that $(\mathtt{MULT}\ a\ b) = (\mathtt{MULT}\ b\ a)$, because it does not know that we are using `MULT` to denote a commutative operator. To address this problem, the tool supports user-defined axioms, which are used to rewrite all assertions to SVC. For example, the axiom

```
MULT assert (= (MULT argMULT1 argMULT2)
               (MULT argMULT2 argMULT1))
```

specifies that `MULT` is commutative. Fortunately, only a few axioms were needed for our experiments (e.g., commutativity of multiplication, equivalence of shift to multiplication by 2, etc.).

Expression Normalization: This optimization can be considered a special-case, extra powerful axiom. The basic problem is that arithmetic with multiplication is undecidable, so we cannot expect SVC to handle arbitrary arithmetic expressions. On the other hand, programs often generate complex arithmetic expressions, so we want a more powerful means to check equivalence. The solution is to rewrite arithmetic expressions heuristically into a fixed normal form, to increase the likelihood that mathematically equivalent expressions can be proven equivalent by SVC.

The heuristics consist of a number of transformations and simplifications in an attempt to bring the two expression to a reduced form. Constants in additions and in multiplications are evaluated as much as possible. Address expressions are simplified based on the memory layout (e.g., if memory location $a+3$ is being accessed, and we know that $a+3 = b$, then replace the reference to $a+3$ by b). In a sequence of multiplications and divisions, the divisions are moved to the end. Most importantly, the arguments of multiplications are reordered according to a fixed order. Although these transformations are obviously not complete, they are sound and were sufficiently powerful to handle all the instances that arose in the industrial examples.

4. EXPERIMENTAL RESULTS

Given the heuristics and abstractions the verification method employs, the true test of the method is in its ability to find bugs in real code. We were fortunate to obtain a set of four matched pairs of code.[1] Each pair consists of two subroutines that were believed to be functionally equivalent [9]. One subroutine was production code taken from a cellular telephone application, hand-written by experts. The other was compiled from an allegedly equivalent C program, using a highly optimizing compiler [9]. The examples range in size from 37 lines to 190 lines of code for the compiled version.

[1] The examples were obtained under a non-disclosure agreement with Fujitsu Laboratories of America so we cannot give code listings or detailed descriptions of functionality.

All errors discovered were found completely automatically. When an error was discovered, we generally considered the hand-written code to be "golden", modified the other program to fix the problem, and repeated the verification.

4.1 Yhaten

Yhaten was the smallest of the examples and was found to have no errors other than minor syntax problems in the compiler-generated code.

4.2 Hup

The Hup example contained only one branch to check a rounding flag and set the appropriate register, followed by a fixed-count loop to calculate a sum and multiplication.

The tool found two errors. The first is that the compiled code lacked the aforementioned branch to check the rounding flag. The tool detected this error due to discrepancies between the two CFG. The second error was that a register used in modulo addressing was set to a fixed value in the generated version while in the hand-coded version it was set from a memory location.

4.3 Kncal

Kncal contained a number of branches, a fixed count loop to calculate a division as well as a much more interesting array of operations, such as shifting, negating and logical ANDing.

The first error located was that the compiled code used LSL (logical shift left) where the hand code was using ASL (arithmetic shift left). Depending on the value of the input this could generate different results.Another error occurred when, along a particular trace through several branch choices, the compiled code attempted to access a temporary memory location that had not been set properly. Errors of this sort (missed initialization along one particular trace) are common and can be extremely difficult to detect if the branching structure is complex.

4.4 Dt_pow

Dt_pow was the largest example and also the most complex in terms of branching structure (Figure 1). Further complicating this example was the extensive use of auto-incrementing and decrementing in the hand-coded version. Thus, at many points in the program, it was difficult to determine which memory address was actually being accessed by visual inspection alone.

The tool found several errors in this example. There is a trivial computation error (order of operations), apparently resulting from a typo in the C code. In the compiled code, several computed results were not being stored properly. More interestingly, there are several distinct cases of missed initializations along particular traces. For example, both programs contain a branch to allow the program to skip initialization of several memory locations. In the hand-written code, if this branch is taken, those addresses would have default values instead. In the compiled code, however, if the branch is taken, default values are not properly supplied. The most interesting bug is one that we believe is a missed initialization along an involved sequence of branch choices in the expert hand-written code.

In summary, we emphasize again that in all examples, all bugs were discovered automatically using the tool. The examples were fairly small, but were sufficiently tricky that the bugs had eluded previous detection. We also note that most bugs resulted from errors in the C program, and not the compiler.

Example	Size	Time	SVC Time	Mem
Yhaten	37	254s	121s	77MB
Hup	47	15h	14h	120MB
Kncal	69	5s	5s	3MB
Dt_Pow	190	245s	243s	8MB

Table 1: Tool Performance on Examples

4.5 Performance

The performance for each example is given in Table 1. The Size column is the number of lines of code in the compiler-generated version. Time is the total clock time taken by the program to verify the assembly code, given no errors were found. SVC Time is the time that the program spent in SVC. Mem gives the maximum memory used during the verification. The memory usage was shared between SVC and our program, but a large percentage of the execution time was spent in SVC. Most of the examples ran in a reasonable amount of time, but Hup slowed down significantly because of the memory simplifications that needed to be performed. The tests were run on a Sun Ultra 60 (360mhz) with 768MB of memory.

5. CONCLUSION

This paper has presented a new method for verification of DSP assembly language routines. The method has been implemented in a prototype tool which has already been successful in finding discrepancies in real production code. Performance was reasonable.

One direction for future work is to improve the power of the verification method — in capacity, in accuracy, and in the types of programs that can be verified. For improved capacity, we have already implemented an automated abstraction mechanism that attempts to find equivalences between the two programs at basic block boundaries. Our initial implementation was too conservative, but more research along these lines is warranted. For better accuracy, the challenge is how to allow bit-level modeling without losing the powerful abstraction provided by uninterpreted functions. To handle a wider range of programs, more general CFGs must be handled. While-loops, for example, might be handled by a fixed-point iteration (cf. [8]), or by heuristics to seek loop invariants. With a faster decision procedure, considering all traces in a single expression, rather than a single trace at a time, would allow additional CFG flexibility.

A more immediate direction for future work is to broaden the applicability of the tool to more DSPs. The tool is already useful, albeit only for the Fujitsu Elixir. The verification method, however, was easy to implement and should be easy to extend. Because of the simplicity of symbolic simulation, implementing new opcodes or features consists mostly of copying algorithmic descriptions from the DSP manual. Natural extensions would handle other DSP architectures, allowing comparison of code for different DSPs (e.g., to verify a port from one DSP to another), and support more complex DSPs (e.g. statically-scheduled, VLIW).

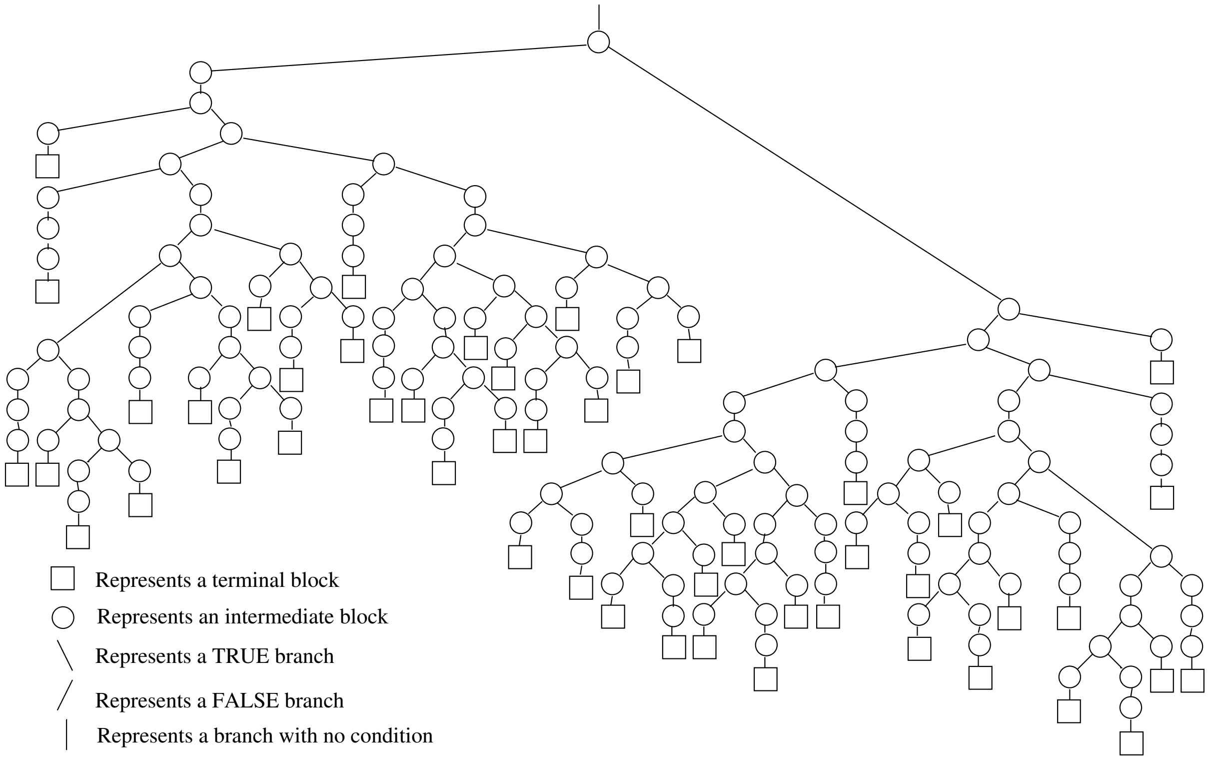


Figure 1: Branching Structure for Dt_pow. This diagram is the CFG unfolded into a tree, showing all possible paths through the routine.

6. REFERENCES

[1] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1988.

[2] 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.

[3] 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.

[4] Randal E. Bryant. A methodology for hardware verification based on logic simulation. *Journal of the ACM*, 38(2):299–328, April 1991.

[5] 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.

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

[7] Jawahar Jain, Amit Narayan, Masahiro Fujita, and Alberto Sangiovanni-Vincentelli. Formal verification of combinational circuits. In *International Conference on VLSI Design*, 1997.

[8] Shin-ichi Minato. Generation of BDDs from hardware algorithm descriptions. In *International Conference on Computer-Aided Design*. IEEE/ACM, 1996.

[9] A. Sudarsanam, S. Malik, S. Rajan, and M. Fujita. Development of a high-quality compiler for a Fujitsu fixed-point digital signal processor. In *Proceedings of the Seventh International Workshop on Hardware/Software Codesign*, pages 2–7, Rome, May 1999. ACM SIGDA.

[10] 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.