Automatic Formal Verification for Scheduled VLIW Code

by

Xiushan Feng

B.E., Harbin Institute of Technology, 1997

M.E., Institute of Computing Technology, Chinese Academy of Sciences, 2000

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 2002

© Xiushan Feng, 2002

In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Department of Computer Science

The University of British Columbia
Vancouver, Canada

Date Dec 12, 2002

Abstract

VLIW processors are attractive for many embedded applications, but VLIW code scheduling, whether by hand or by compiler, is extremely challenging. In this paper, I extend previous work on automated verification of low-level software to handle the complexity of modern, aggressive VLIW designs, e.g., the exposed parallelism, pipelining, and resource constraints. I implement these ideas into two prototype tools for verifying short sequences of assembly code for TI's C62x family of VLIW DSPs and Fujitsu's FR500 VLIW processor, and demonstrate the effectiveness of the tools in quickly verifying, or finding bugs in, several difficult-to-analyze code segments.

Contents

Abstract	ii
Contents	iii
List of Tables	vi
List of Figures	vii
Acknowledgments	viii
Dedication	ix
1 Introduction	1
1.1 Research Overview	2
1.1.1 Problem	2
1.1.2 Solution	2
1.1.3 Contribution	3
1.2 Thesis Outline	4
2 Background	5
2.1 Previous Work	5

2.2	VLIW Processors	7
2.2.1	Texas Instruments' TMS320C62x VLIW DSP	9
2.2.2	Fujitsu's FR500	14
3	Verification Method	19
3.1	Basic Symbolic Simulation Approach	20
3.1.1	Symbolic Simulation	21
3.1.2	Uninterpreted Functions	22
3.1.3	Memory	24
3.2	My Extensions	25
3.2.1	Parallel Execution	25
3.2.2	Predication	26
3.2.3	Control Flow Analysis	27
3.2.4	Branch Delay Slots	32
3.2.5	Read-After-Write Delay Slots	33
3.2.6	Resource Conflicts	35
3.2.7	Condition Code	37
4	Examples	39
4.1	TI TMS320C6x	39
4.1.1	Software Pipelining Example	39
4.2	Fujitsu FR500	45
4.2.1	Two pieces of equivalent FR500 code	45
5	Limitations, Conclusions and Future Work	47
5.1	Limitations	47
5.1.1	Bit Reasoning	48

5.1.2	Uninterpreted Functions and Axiom Handling	49
5.1.3	Memory Size	49
5.2	Future Work	50
	Bibliography	51
	Appendix A Two equivalent FR500 programs	54
A.1	C Functionality	54
A.2	convolv.asm (with no optimization)	57
A.3	convol_three.asm (-03)	60

List of Tables

2.1 C62x Processor Pipeline	12
---------------------------------------	----

List of Figures

2.1	Packet example	9
2.2	TI C62x Datapath Block Diagram	10
2.3	Branch Delay Slot Example	13
2.4	Fujitsu FR500 Datapath Block Diagram	16
2.5	Fujitsu FR500 Integer Execution Unit	17
2.6	Fujitsu FR500 Floating Execution Unit	18
3.1	A CFG Example	28
3.2	Register Model	34
4.1	Software Pipelining Example	41
4.2	Unpipelined Assembly Code	42
4.3	Software Pipelined Assembly Code	44

Acknowledgments

It has been a great privilege to be supervised by Alan J. Hu, who was and is an inspiration to me in many ways. Without his ideas and encouragement, my research would not have been nearly as successful. Some of the material in this thesis was prepared in collaboration with Alan J. Hu as a paper for the ACM SIGPLAN Joint Conference LCTES/SCOPES'02.

This research was supported in part by a research grant from Fujitsu Laboratories of America.

XIUSHAN FENG

The University of British Columbia

August 2002

To my parents, for their endless patience and support.

Chapter 1

Introduction

Very Long Instruction Word (VLIW) processors are attractive for many embedded applications because of their promise of very high performance without the power- and die-area-consuming instruction-scheduling logic of superscalar processors, and because backward object-code compatibility is typically unnecessary for embedded applications. In such applications, size and speed requirements can often be critical. In order to meet such requirements, compiled code often needs to be hand tuned, and smart compilers are needed to exploit the potential parallelism.

Unfortunately, the exposed parallelism, pipelining, and resource conflicts in an aggressive VLIW design, coupled with the lack of interlocks, makes generating and optimizing code for a VLIW processor extremely challenging. The result is that bugs are easily introduced, or the code is excessively conservative, and therefore, sub-optimal for VLIW processors.

My hypothesis is that formal verification techniques can help. In particular, a symbolic simulation tool that can decide equivalence of VLIW programs can help to find bugs and optimize code.

1.1 Research Overview

I extend previous work on verifying simple assembly code to the more complex assembly code of a VLIW processor. This work builds directly on the work of Currie et al. [7].

1.1.1 Problem

My method and Currie's both analyze programs at the instruction set architecture (ISA) level. The ISA level shows the functionality of individual machine instructions, but hides the hardware implementation details. This is the lowest level that a programmer sees. However, for a modern, aggressive VLIW design, the ISA level actually exposes much of the underlying microarchitectural complexity, making the verification problem much trickier. For example, a VLIW processor has multiple function units allowing parallel execution, but limited resources place constraints on parallel instructions. And some VLIW processors may expose pipelining for certain operations. Such operations will have several cycles of delay slots — the number of cycles required after the source operands are read before the results can be read. In order to correctly verify programs for such a processor, the verification tools have to handle such complexity.

1.1.2 Solution

My thesis develops formal verification tools to handle this complexity. Such verification tools could automatically verify the functional equivalence of two pieces of VLIW assembly code. One piece of code is supposed to be written by an expert or proven to be bug-free, while the other piece of code is generated by a potentially buggy compiler or optimizer, or is the result of hand-optimization. The verification

tool tries to prove that the two programs are equivalent. If they are not equivalent, the verification tool reports debugging information, which is useful for the user to find the bugs.

In order to handle the complexity that the ISA exposes, I introduce several innovations. First, I model the parallel execution of multiple functional units. To accurately model pipeline delays, I introduce a simplified model of the pipeline that allows us to compute the correct result and also verify the presence/absence of resource conflicts. I also model predicated execution precisely, which permits a very detailed and accurate analysis of resource conflicts.

In this thesis, my verification tools use symbolic simulation with uninterpreted functions. I use symbolic simulation due to its flexibility. By using uninterpreted functions, my verification tools can handle much more complex instructions, which can not be handled by traditional verification methods without space explosion. For example, the size of the Binary Decision Diagram (BDD) [4] for multiplication is exponential in the number of bits being considered [5]. I will explain these ideas in more detail in Chapter 2.

1.1.3 Contribution

The primary contribution of my work is a method for the formal verification of VLIW software.

Obviously, the quality of code can be improved by verifying the correctness of optimizations performed by the compiler or by hand.

My approach is implemented in two prototype tools. Though limited in scope, they are able to verify examples of real life code. One prototype verification tool targets the Texas Instruments TMS320C62x family of VLIW digital signal

processors and another prototype verification tool targets the Fujitsu FR500 VLIW processor. Run times on short code segments are negligible, yet the tools are able to verify/debug code optimizations and verify the absence of resource conflicts, including in a situation previously considered unanalyzable. These examples demonstrate the practical usefulness of my ideas.

1.2 Thesis Outline

I will give a brief overview of background in Chapter 2 that helps to show the gap between my problem and previous work. In Chapter 3, I will describe the challenges for my verification methods and detailed solutions to them. To test the effectiveness of my ideas, I apply them to modified industrial and published examples, and demonstrate the capabilities of my tools and the bugs that they are able to find in these examples. Chapter 4 presents examples for the two processors I tested and gives verification results for each tool. Chapter 5 goes into details about some of the limitations of the tools, conclusion, and the things I would like to implement in the future.

Chapter 2

Background

2.1 Previous Work

This work builds directly on the work of Currie et al. [7]. In that work, the authors demonstrated the feasibility of automatic verification of low-level software via symbolic execution and an automated decision procedure. The verification task was to compare two short assembly language routines for equivalence, assuming considerable similarity in control flow, as would occur, for instance, after hand-optimizing a performance-critical kernel. The verification method was demonstrated by building a verification tool targeting a very simple, 16-bit fixed-point digital signal processor. My thesis is the natural extension of Currie et al.'s work to a highly complex, modern VLIW processor.

Hamaguchi et al. [11] employed a similar approach to verify a high-level specification against a low-level implementation. Furthermore, their low-level implementation included a VLIW processor with assembly-level instructions. Subsequent work [10] enhanced performance with better heuristics. Their work addresses the

harder problem of high-level-versus-low-level verification, whereas I consider only the problem of comparing two low-level programs. However, their VLIW processor was a simple, academic design (4-wide, 2-stage pipeline, no unusual architectural features), whereas I tackle a high-end commercial VLIW. The emphasis of my work is on how to deal with the architectural features of VLIW processors, rather than on improving the efficiency of the verification approach, and I believe both directions of research are needed for widespread, practical impact.

Blank et al. [3] have recently presented a broad survey of the general verification paradigm (symbolic simulation/execution with uninterpreted functions) along with techniques for greatly improved efficiency. Unfortunately, I became aware of their work after my project was well underway, so I have not harnessed their ideas yet. In particular, I am building symbolic expressions for results as functions of the original inputs, and these expressions can blow-up exponentially. In contrast, their approach introduces temporary variables for intermediate results, eliminating this expression-size blow-up entirely. To date, I have avoided the expression-size blow-up by some rewriting techniques, but their translation approach is likely the superior solution in general.

In the compiler-research community, Necula [17] has proposed a very similar approach to that of Currie et al., but targeting the verifier for use by the compiler between optimization passes. His work uses hints from the compiler and more sophisticated control-flow analysis, and was demonstrated by verifying the compilation of the Gnu C compiler on itself. I believe that this translation-validation-during-compilation is a promising approach, and that my methods for dealing with the complexity of a high-performance VLIW processor could be integrated with Necula's methods for verifying large programs with complex control flow.

2.2 VLIW Processors

Very Long Instruction Word (VLIW) describes a computer processing architecture in which multiple instructions are put into a very long instruction word. VLIW processors can then take apart the sub-instructions in the very long instruction word without further analysis and run them in parallel. In such architectures, the compiler is in charge of packing sub-instructions into the very long instruction words. Therefore, VLIW architectures execute multiple operations per cycle with more parallelism, simpler hardware and higher performance, but require more compiler support.

Compared to superscalar processors, VLIW architectures move the work of instruction scheduling and maintaining data dependencies to the compiler. Using the compiler to perform this work has lots of advantages. First, the compiler has the ability to look at much larger windows of instructions than hardware can [20]. Because the compiler has less constraints on resources, it can have arbitrarily large software windows to look for parallelism in a big program. Second, the compiler has knowledge of the source code of the program, and source code typically has more information to track program behavior. The compiler can take advantage of this information [20]. Meanwhile, because of smart compilers, VLIW processors can have simpler hardware. Fujitsu researchers estimate that to execute four instructions simultaneously, the VLIW architecture will require a 50% smaller circuit than the superscalar method [21].

Usually, a VLIW processor has the following characteristics [2]:

- Multiple independent instructions per cycle, packed into a single large “instruction word” or “packet”.

- A large complement of independent execution units
- More regular, orthogonal, RISC-like instructions
- Large, uniform register sets
- Wide program and data buses

In this thesis, I have two examples (TI TMS320C6x, and Fujitsu FR500) for VLIW processors. They show the above characteristics.

Currently, VLIW processors are very popular, especially for DSP applications. For such applications, we need cheaper hardware, higher performance, and specific instructions such as those handling media data, and do not need to worry as much about compatibility as do desktop microprocessors. Therefore, in such applications, VLIW processors allow increasing performance far more easily [1].

Although VLIW architectures bring lots of benefits, they introduce some challenges for compilers or optimizers as well. A VLIW compiler should find the parallelism in application programs and convert it into parallel streams of executable instructions. Such challenges together with the greater complexity of VLIW processors will easily create compiler bugs, and make generating and optimizing code for VLIW processors extremely challenging. In Chapter 4, I will demonstrate a bug which appears in a published code optimization example written by a recognized expert.

The following subsections present some details of the two VLIW processors considered in this thesis.

```

    STH .D1 A5,*A8++[2]
|| STH .D2 B5,*B8++[2]
|| SADD .L1 A2,A7,A2
|| SADD .L2 B2,B7,B2
|| SMPYH .M2X B3,A3,B2
|| SMPY .M1X B3,A3,A2
|| [B1] B .S1 LOOP1
|| [B1] SUB .S2 B1,1,B1

```

Figure 2.1: Packet example [22, Example 6-1].

2.2.1 Texas Instruments' TMS320C62x VLIW DSP

Texas Instruments' TMS320C6x family of VLIW digital signal processors [22] is both commercially important and also the epitome of this architectural style, so I have targeted this family for my first research prototype. In particular, I am targeting the C62x family, which are 32-bit fixed-point DSPs that are code-compatible with the more powerful C64x fixed-point family and the C67x floating-point family. In this subsection, I briefly highlight the salient architectural features that make verification difficult; subsequently, I present my solutions to these difficulties in Chapter 3.

The C6x processors are 32-bit VLIW digital signal processors. Instructions are grouped into “execute packets” of up to eight instructions, and these packets can be executed one per clock cycle. Figure 2.1 is a packet example, which is taken from the processor reference guide [22, Example 6-1]. It shows one packet with eight instructions which will be executed in parallel. ‘‘||’ characters signify that an instruction is to execute in parallel with the previous instruction. Figure 2.2 shows the datapath.

Of particular note are both the capability of very high performance, but also the striking non-orthogonality of the design. Functional units have specific

Figure 2.2: TI C62x Datapath Block Diagram. Registers are in two banks, with limited pathways connecting them (1X and 2X). The .L and .S units are integer ALUs with slightly different properties. The .D units are also integer ALUs, but designed for address calculation. The .M units are multipliers. LD and ST are load and store paths, with DA being data address paths. This figure is taken from the processor reference guide [22, Figure 2-1].

capabilities, and there are limited resources for routing data among the functional units and register files. Careful code-tuning is imperative to achieve maximum performance. However, the processor has no interlocks — most potential conflicts in resource utilization are disallowed statically during code generation, but some cases produce undefined results. I will elaborate on this point below.

The other main architectural feature that both enhances performance and complicates code generation and verification is the pipeline. For maximum throughput, the processor is heavily pipelined, with instructions taking up to 11 cycles to complete. Table 2.1 summarizes the pipeline. The pipeline has no interlocks, which simplifies the hardware (more performance at lower cost) but complicates the code. For most instructions, the operand read and result write occur in a single stage, so there are no hazards. However, multiply and load instructions have long latencies and require 1 and 4 delay slots, respectively. Instructions in these delay slots see the old value of the register. Multiple writes to a register in a single clock cycle are illegal, and in most cases, this can be detected during code generation.

Another artifact of the programmer-visible pipeline is that there is a long branch delay. In particular, branches have five delay slots (i.e., the next five execute packets following a branch always execute¹, regardless of whether the branch is taken or not), because the branch doesn't affect the Program Address Generation stage until it reaches the Execute 1 stage. Unlike many processors, branch instructions may appear in the delay slots of other branches. The results in that case may be unintuitive, but they follow naturally from the pipeline definition. Figure 2.3 shows an example of delayed branches.

To ease coding, the architecture provides a multicycle NOP instruction,

¹Unless we're already in the delay slots of a preceding branch.

Stage	Description
Program Address Generate	Determine address of the fetch packet.
Program Address Send	Send address of the fetch packet to memory.
Program Wait	A program memory access is performed.
Program Data Receive	Fetch packet received by CPU.
Dispatch	Determine the next execute packet in the fetch packet and send it to the appropriate functional units.
Decode	Decode instructions in functional units.
Execute 1	Evaluate predicates. Read operands. For load and store instructions, do address generation and write address modifications to register file. For branch instructions, affect the Program Address Generate stage. For single-cycle instructions, write results to a register file.
Execute 2	Load and store instructions send address (and data for store) to memory. Most multiply instructions complete.
Execute 3	Data memory accesses are performed. Store instructions complete.
Execute 4	For load instructions, data received by CPU.
Execute 5	For load instructions, data is written into a register.

Table 2.1: C62x Processor Pipeline. The processor pipeline is 11 stages deep. (TI literature refers to the pipeline stages as “phases”, but I adopt standard terminology here.) Instructions are grouped into “execute packets” of up to eight instructions that proceed through the pipeline in parallel. Note that different instruction types write results with different latencies. This table is summarized from [22, Table 6-1].

```

; ** -----*
Start:                                ;Start point of a program
      B   .S1 label1                  ;A branch waits 5 delay slots to complete
      SUB .D2 B1 , 1, B1 ;Delay slot 1
      SUB .D2 B1 , 1, B1 ;Delay slot 2
      SUB .D2 B1 , 1, B1 ;Delay slot 3
      B   .S1 label2                  ;Delay slot 4, another branch waiting
      SUB .D2 B1 , 1, B1 ;Delay slot 5
                                   ;control flow will now jump to label1
                                   ;Delay slot 1 for the second branch
      SUB .D2 B1 , 1, B1 ;Not executed due to
                                   ;completion of the first branch
      ...
      ...

label1:
      SUB .D2 B1 , 1, B1 ;Delay slot 2 for the second branch
      SUB .D2 B1 , 1, B1 ;Delay slot 3 for the second branch
      SUB .D2 B1 , 1, B1 ;Delay slot 4 for the second branch
      SUB .D2 B1 , 1, B1 ;Delay slot 5 for the second branch
                                   ;control flow will now jump to label2
      SUB .D2 B1 , 1, B1 ;Not executed due to
                                   ;completion of the second branch
      ...
      ...

label2:
      ...
      ...

; ** -----*

```

Figure 2.3: An example of branch instructions which appear in the delay slots of other branches. To make the example simpler, I have made it a serial program, i.e., only one instruction is grouped for each packet, which is either a SUB instruction or a branch instruction. In the above example, both branches will occur due to the existence of delay slots for branch instructions. Of the packets after "label1", only the first 4 are executed, because the branch to "label2" is in the delay slots of the first branch.

which is equivalent to a series of n NOPs executed in sequence.

To complicate matters further, but also to allow very compact and efficient code, all instructions are predicated, i.e., each instruction in each execute packet specifies a register and a condition (equal-zero or not-equal-zero), and only executes if that register is zero or not zero. Predication is known to eliminate many branches, thereby increasing the size of basic blocks and the amount of instruction-level parallelism available [14]. Unfortunately, predication and the absence of pipeline interlocks means that many register-write conflicts may or may not happen, depending on the values of registers at runtime. The processor manual specifically states that these situations cannot be detected, but that the result is undefined when they happen [22, Section 3.7.6]. I will demonstrate that these situations can sometimes be analyzed.

In sum, this architecture follows the VLIW philosophy and is optimized for maximum performance with minimal hardware, with no consideration for easy code generation or verification. The apparent complexity of the programmer's model, however, is not arbitrary, but the logical consequence of the exposed parallelism and pipeline. Accordingly, I claim that although the code is very error-prone for a human to read or write, it is relatively easy to create a simulator for the processor — even a symbolic simulator — that captures the correct semantics. The following chapter describes how I extended the basic verification approach to handle the challenges of this architecture.

2.2.2 Fujitsu's FR500

The FR500 is a general-purpose microprocessor tailored for embedded applications. It integrates system control and media processing with the highest performance

among the embedded processors [16]. This series is specifically designed for media-rich requirements for digital consumer products, such as laser printers, digital still cameras, digital televisions, and other multimedia devices. FR500 is the first product in the FR-V family, which is Fujitsu's generic name for its VLIW architecture microprocessors [21].

The key feature of the FR500 series is a built-in media-execution unit that can simultaneously process eight 16-bit fixed-point MAC calculations. The unit provides a 63-instruction instruction set that can process media data such as MPEG streams, as well as filtering operations [16]. In addition to the media instruction set, the FR500 series also includes a set of integer instructions (including logic operations, load/store instructions, control instructions and compiler support), and a floating-point instruction set. The floating-point execution unit can perform four single-precision floating-point calculations simultaneously, thus accelerating geometry processing (coordinate calculation) of three-dimensional graphics [16]. Figure 2.4 shows the block diagram.

The FR500's 4-slot VLIW architecture enables concurrent processing of four 32-bit instructions, i.e. four 32-bit instructions are packed into one 128-bit very long instruction word and will be executed simultaneously. These four instructions are allocated into two integer instruction slots and two media processing or floating-point instruction slots. The integer instruction slots (see Figure 2.5) can be used to load data for the floating-point or media processing operations. By this means, we can reduce the possibility of bottlenecks in the data supply to the floating-point or media execution units. The floating-point execution unit has a Single Instruction Multiple Data (SIMD) architecture, permitting two operations using one instruction, while the media execution unit has a double-SIMD architecture

Figure 2.4: Fujitsu FR500 Datapath Block Diagram. This figure is taken from [21, Figure 2].

Figure 2.5: Fujitsu FR500 Integer Execution Unit. This figure is taken from [21, Figure 3].

permitting simultaneous execution of four operations. Figure 2.6 shows the media execution unit.

In addition, the execution units have two register files, each with 64 registers, that enable concurrent processing of operations in a very long instruction word. Such a design allows the FR500 to attain peak performance at 266MHZ of 532MIPS integer, 1064MFLOPS floating-point, and 4256 MOPS for media processing. The performance is comparable to a high-end microprocessor and handles media-rich data as well as a DSP unit [16].

Figure 2.6: Fujitsu FR500 Floating Execution Unit. This figure is taken from [21, Figure 3].

Chapter 3

Verification Method

The basic goal for my verification method is to verify that two short code segments compute the same result — for example, whether an optimized implementation matches a known-good one. My verification problem is analogous to equivalence checking of combinational circuits, which has been very successful in an industrial setting. In this thesis, what I am attempting to prove is the equivalence of two assembly-language routines.

The formal verification tool, which I will refer to as an “equivalence checker” in this thesis, should first read the specifications of what inputs are initially equal and what outputs should be equal when the routines terminate. The equivalence checker has a simple model of the processor at the ISA level, and then uses this model to symbolically simulate and check equivalence for the two routines. Due to the difficulty of software verification, I have to introduce some simplifying assumptions:

- The two routines have very similar control flow. Such an assumption is based on my verification target — to verify code tuning and optimizations.
- I ignore rounding, precision, and word length. My verification methods (sym-

bolic simulation, uninterpreted functions) cannot model actual bit precisions.

- The control flow must be analyzable — no computed branches, no recursion, loops must be unrollable. I also ignore interrupts.
- I do not model memory size limits or protection. The tool assumes all address are accessible.

Given these assumptions and limitations, the equivalence checker might declare inequivalent two routines that really have the same outputs, but it will not claim equivalence for two routines that do not. I will show some examples in Chapter 5 where the tool will incorrectly declare equivalence when the above assumptions do not hold.

3.1 Basic Symbolic Simulation Approach

This section introduces symbolic simulation, a formal verification method. My equivalence checkers are built upon such a basic symbolic simulation approach and borrow the Stanford Validity Checker (SVC) as the decision procedure. The user initially specifies what registers and memory location are equivalent before execution, defines any axioms, and declares what value should be equivalent after execution. The equivalence checker performs symbolic execution of the routines, and then passes the symbolic expressions of memory or registers to SVC for validity checking.

Among the research on equivalence checking of combinational circuits, Binary Decision Diagrams (BDD) [4] are a very successful choice to represent the outputs. BDDs can be constructed, manipulated, and compared very efficiently. The drawback of BDDs is that, for most functions the size of it can be exponential in the number of variables being considered. This motivates the development of

tools which can operate at a higher level of abstraction. In this thesis, I use symbolic simulation with uninterpreted functions to represent registers and memory at the word-level.

3.1.1 Symbolic Simulation

In symbolic simulation, variables are left as symbols that can represent any arbitrary value instead of being a specific instance of an integer or floating-point number. This allows the simulation tool to represent an entire class of values in a single simulation run. For example, using the method of symbolic simulation to execute the following code, we can get the results for each step, as shown in the comments of the code.

```

; A1 = A1_Init, A2 = A2_Init,
; A3 = A3_Init, A4 = A4_Init
; B2 = B2_Init,
; The above are the initial values.
ADD      A1, A2, B1 ; B1 = (+ A1_Init A2_Init)
MPY .M1   B1, B2, B3 ; 1 delay slot for MPY, delay write.
NOP
; B3 = (* (+ A1_Init A2_Init) B2_Init)
ADD      A3, A4, B2 ; B2 = (+ A3_Init A4_Init)
ADD      B3, B2, B3 ; B3 = (+ (* (+ A1_Init A2_Init) B2_Init)
                        (+ A3_Init A4_Init) )

```

In the above example, all of the values for registers are symbolic expressions. Therefore, one simulation run contains the result of the code for all possible values. To check equivalence of two pieces of code, a decision procedure takes the symbolic results and compares them.

Unfortunately, comparing some complicated computations, such as expression including multiplication, is undecidable in general. In order to solve this problem, the equivalence checker needs to borrow the idea of uninterpreted functions.

3.1.2 Uninterpreted Functions

Uninterpreted functions are a powerful abstraction mechanism for formal verification. In this approach, meaningless function symbols replace actual computation. The function symbol is “uninterpreted” and can represent any function, similar to how a variable symbol represents any value. Therefore, verification can avoid the complex details of the actual computations.

My equivalence checker divides the functions used in a program into three different kinds:

The first is constant propagation for traditional arithmetic operations. For example:

```
add gr1, gr2, gr3 ; Current values of register gr1 and gr2 are
                  ; integers(e.g. 2, 3 respectively),
```

In this example, the operands are all constant — integers. Therefore, the addition operation will add up the constants and update target register “gr3” to “5”.

The second is interpreted functions. Interpreted functions are functions where the equivalence checker knows their meaning. For example, by using an interpreted function, e.g., “+”, the equivalence checker knows that the symbol “+” denotes addition, so it can interpret that “(+ 2 4)” = “(+ 1 5)”, “(+ a b)” = “(+ b a)”, and more complicated “(+ (+ a 2) 4) = (+ (+ a 4) 2)” (i.e., the equivalence checker knows that “+” is a commutative and associative operator).

The third is uninterpreted functions. Uninterpreted functions use meaningless function symbols to hide the actual operations of data. The equivalence checker doesn't interpret such functions. Instead, it takes them as symbolic expressions, and decides equality if the arguments are equal. For example, if $a = a'$ and $b = b'$, then we know that $(\text{FUN } a \ b) = (\text{FUN } a' \ b')$ for any function FUN. In general, the theory of uninterpreted functions with equality is decidable.

For example, here is an operation for the TI C6x: "SHL src2, src1, dst". It is an operation which shifts the src2 operand to the left by the src1 operand and puts the result in dst. The word-level equivalence checker can not handle such kinds of bit operations. However, an uninterpreted function "SHL" can be used to abstract the bit computations. The equivalence checker doesn't need to know the meaning of SHL and what happens for this instruction. It takes SHL as a function symbol. When the checker wants to verify whether $(\text{SHL } \text{src2 } \text{src1}) = (\text{SHL } \text{src2}' \ \text{src1}')$, it claims true if $\text{src2} = \text{src2}'$ and $\text{src1} = \text{src1}'$, else false.

Applying the idea to the example in the previous subsection, the verification tool replaces the operator of multiplication by a meaningless function symbol, for example, "MPY". After that, B3 will be updated to $(\text{MPY } (+ \ A1_Init \ A2_Init) \ B2_Init)$ instead of $(* \ (+ \ A1_Init \ A2_Init) \ B2_Init)$. However, after such replacements, multiplication loses the basic properties such as commutivity. In order to handle commutivity, the equivalence checker needs to have the axiom that $(\text{MPY } A \ B) = (\text{MPY } B \ A)$.

The initial values for registers and memory also use symbolic values. For example, in my equivalence checker, I use "init_mem" to represent the initial values in the memory.

It is obvious that by using uninterpreted functions the verification tool can re-

duce some complexity of the program. Certain mathematical computations needn't happen. Therefore, we can abstract datapath complexity.

3.1.3 Memory

Almost all computations need to get data from memory, and then save results to memory. Efficiently handling memory is important because symbolic expressions for memory tend to become too large.

In symbolic simulation, memory is also introduced as a symbol. It is modeled by using the interpreted functions `read` and `write`. The expression

```
(read mem addressfoo)
```

returns the last value written to location `addressfoo`, and the expression

```
(write mem addressfoo valuebar)
```

returns the new memory where location `addressfoo` is updated to `valuebar`.

The decision procedure knows that $(\text{read} (\text{write } a \ b \ c) \ b) = c$, so for example, the decision procedure can determine that if $b \neq d$, then

$$\begin{aligned} (\text{read} (\text{write } a \ b \ c) \ b) &= (\text{read} (\text{write} (\text{write } a \ b \ c) \ d \ e) \ b) \\ &= c \end{aligned}$$

I should point out that my equivalence checkers use a single memory (only one memory addressing space) for each program, and all reads and writes are targeted to this memory.

Handling long programs or programs with big loop sizes generates very large expressions involving memory. To reduce the size of these expressions, I introduce some memory simplification methods. These methods are similar to those of Currie's thesis [6]. For example, if the equivalence checker knows that $b \neq d$, it then knows

`(read (write (write a b c) d e) b) = c`

In general, if it can locate the address being read in the memory write history, and if it can prove that no other writes interfered, then it can replace the read expression by the last value written.

Another example is

`(write (write a b c) b e) = (write a b e)`

The equivalence checker scans its write history, and overrides the previous write if the address being written to has already been used. Using similar ideas, the equivalence checker uses some other methods to simplify expressions involving memory. Such simplifications can reduce the size of memory expressions a lot.

3.2 My Extensions

In this part, I will give some details of my approach, which is built upon the basic symbolic simulation methods.

3.2.1 Parallel Execution

The most obvious difference of a VLIW is the multiple instructions executing simultaneously. This is trivially handled in the symbolic simulator. I simply simulate each instruction¹ in an execution packet one-by-one, but I don't update the register file until after all instructions have read their operands. The resulting behavior is identical to parallel execution. For example, here is an execution packet of TI C62x.

¹My current tool implements only a subset of common instructions, since the point of my research is to explore the verification technique rather than build a production tool.

```

        LDW .D2 *B5,B4
|| MPY .M1X B4,A0,A5
|| B     .S1  LOOP1
|| SUB .S2  B1,1,B1

```

The execution results for it are

```

0 cycle : cur_B1 = (- prev_B1 1)
1 cycle : cur_A5 = (MPY prev_B4 prev_A0)
2 cycle : no changes for this packet
3 cycle : no changes for this packet
4 cycle : cur_B4 = (read prev_mem prev_B5)
5 cycle : branch happens

```

where the `cur_` and `prev_` symbols are abbreviations for the current and previous expressions for the values of the registers.

Note: 0 cycle denotes the cycle right after this example packet. “SUB” has no delay slots, therefore, the update of register B1 happens in this cycle. I have already mentioned that “LDW” and “B” have 4 and 5 cycles delay respectively.

3.2.2 Predication

Three issues arise in dealing with predication: the basic functionality, conditional branching, and resource conflicts. The basic functionality of predication is easily expressible in my logic. Indeed, using the “ITE” (if-then-else) operator of SVC logic, the tool simply generates the expression that evaluates to the new value if the predicate is true, or the old value if the predicate is false. For example, the value of the register A3 after this instruction:

```
[B0] ADD .L1 A1, A2, A3
```

is

```
(ITE (/= cur_B0 0) (+ cur_A1 cur_A2) (cur_A3))
```

where `/=` denotes not-equal.

In the next four sections, I will describe how the equivalence checker handles the conditional branching and resource conflicts.

3.2.3 Control Flow Analysis

The control flow analysis in my thesis is standard, using a control flow graph (CFG). I build the CFG similarly to what Currie et al. [7] did. My equivalence checker verifies programs using the control flow graph, but the approach is different from Currie's. I build a single expression for the outputs over all possible traces through the CFG and compare the whole expression for the two programs (see Figure 3.1), whereas Currie et al. transforms the CFG into a tree, performs a depth-first traversal over the tree, symbolically simulating along the way. At each branch, the method of Currie checks the branch condition. There are two possible outcomes which lead to the equivalence for the two routines: the conditions are the same, or the conditions are complementary (the two branch directions of one routine need to be reversed in order to match the traces of the two CFGs). If the check fails (a return other than the above two), Currie's tool reports different control flows. At the end of each trace, his tool checks the equivalence of the expressions for the two routines, and recursively does these checks until an invalidity is found or the simulation terminates. The detailed description of Currie's method can be found in his thesis [6].

The following is a short piece of code to show how my procedure works using C62x code.

Figure 3.1: A CFG Example. This figure shows a control flow graph, which has two branch instructions to test “Condition I” and “Condition II” respectively. “T” or “F” here means “True” or “False”. For this example, the equivalence checker outputs a single expression such as “(ITE (Condition I) (ITE (Condition II) expr1 expr2) expr3)”.

```

        SUB    .L2  B0,1,B0
[B0] B      .S2  L1
        NOP 5
        ADD    .L1  A0,1,A0
        B      .S2  L_end
        NOP 5
L1:
        ADD    .L1  A0,2,A0
L_end:
        ;finish

```

Remember that for branch instructions “B”, we have 5 delay slots. According to my method of control flow handling, the final expression of A0 will be a single expression like this:

$$\begin{aligned}
 A0 = & (ITE \ (/= \ (- \ previous_B0 \ 1) \ 0) \\
 & (+ \ previous_A0 \ 2) \\
 & (+ \ previous_A0 \ 1) \)
 \end{aligned}$$

where Currie will get two expressions, and compare them with another routine based on the branch condition. In this example, if the equivalence checker can prove B0 is a constant integer or can be reduced to a constant integer, there will be no conditional branches. A0 will be a simple expression without introducing “ITE”

It’s obvious that using a single expression for outputs has advantages. For some kinds of routines, my equivalence checker can prove equivalence while Currie’s approach misses it. Consider the following examples (using pseudocode):

<pre> routine 1: if condition_1 = true a := 1; else a := 2; endif if condition_2 = true b := 1; else b := 2; endif </pre>	<pre> routine 2: if condition_2 = true b := 1; else b := 2; endif if condition_1 = true a := 1; else a := 2; endif </pre>
---	---

Currie's approach reports that the two routines have different control flows, although they are equivalent. So do such kinds of routines.

<pre> routine 1: if a < b if c < d x := 1; else x := 2; endif else if c < d x := 3; else x := 4; endif endif endif </pre>	<pre> routine 2: if c < d if a < b x := 1; else x := 3; endif else if a < b x := 2; else x := 4; endif endif endif </pre>
--	--

My approach can find the equivalence for these routines. However, my checker uses a single expression for each register or memory output. This approach generates longer expressions sent to the decision procedure, which raises performance issues with SVC. I didn't do performance comparisons with these two approaches, because run times of the examples available for me have been negligible. What I care about more is to get better control flow graph matching.

As I describe above, predicated branches are handled exactly as conditional branches. The only difference is the lengthy branch delay, which I will describe in the next subsection.

3.2.4 Branch Delay Slots

Because branches have five delay slots, there exists the possibility that other branch instructions will occur in those slots. In fact, this occurs frequently in tight loop kernels. For example, consider the following code fragment:

```
        B .S1 Loop1

        MVK .S2 10, B1
||      MVK .S1 0, A0

Loop2:
        ADD .L1 A0, 1, A0
|| [B1] B    .S1 Loop2
|| [B1] SUB  .S2 B1, 1, B1

        STW .D1 A0. *A1++
        NOP 2

Loop1:
        [B1] ADD .L1 A0, 1, A0
|| [B1] SUB  .S2 B1, 1, B1

        NOP 5
```

In this example, the branch to Loop2 occurs in the delay slots of the branch to Loop1, as well as in its own delay slots on subsequent iterations. The net effect is that the code loops until register B1 reaches 0. Although this example looks contrived, coding in this style is common.

I would like to avoid explicitly modeling the instruction packet fetch mechanism, which is actually somewhat more complex than I have described. Instead, I introduce a small queue for pending branches. Each branch instruction gets queued, along with a counter indicating how many delay slots remain for that branch. On each clock cycle, all counters are decremented. Before each instruction fetch, I check whether a branch at the head of the queue is ready to be taken.

3.2.5 Read-After-Write Delay Slots

As with branch delay slots, delay slots resulting from different instruction latencies are the result of the programmer-visible pipeline, and I handle them in a similar fashion. Conceptually, these delay slots simply delay the assignment of results to the register file. Accordingly, modeling the full pipeline can be avoided by using small queues to delay the writes to the register file.

More precisely, I extend the data structure used for register values. In the basic verification approach, each programmer-visible register is modeled by a variable that can hold the symbolic expression for the value of that register. Instead, I replace this variable with a queue of symbolic expressions. These queues are very short — for the TI C62x DSP, the longest latency is four delay slots for loads, so the queue is only five entries long (current value and four delay slots). For the Fujitsu FR500, there is no latency for any instruction, therefore, there is no problem for read-after-write delay slots. (See Figure 3.2.) Whenever a result is written to a register file, the latency of the operation is checked in a table, and the result is written into the appropriately delayed queue entry. Reads from the register file return the current value. After every clock cycle, all the register-value queues advance one step; if there is no write to a register in that cycle, it keeps its current value.

Figure 3.2: Register Model. This figure shows the data structures used to model each register. In the basic verification algorithm, only the “Current Value” variable would exist, and reads and writes would access and update this variable. To handle delay slots, I add a queue of pending writes. Reads still come from the Current Value, but values to be written are queued depending on their latency. The queue advances one position to the left at each clock cycle, updating the Current Value as appropriate. The attempt to write to a queue slot that is already full indicates a resource conflict. To handle predication with delay slots, I store *write-histories* in the pending writes queue, rather than values. Each write-history is a list of predicate-value pairs. It is easy to verify that at most one predicate in a write history can be true, otherwise there is a conflict. Since the simulator is symbolic, the “values” are actually symbolic expressions denoting the value as a function of the initial values.

3.2.6 Resource Conflicts

There are four pages of resource constraints described in the TI C62x processor reference guide [22, 3-17..3-20]. A complete verification tool needs to check all the constraints. However, the checks for some statically detectable resource conflicts are uninteresting from a research perspective, because they are straightforward checks of execution packets against various simple rules, and existing tools handle these checks already. Therefore, my equivalence checker only handles a subset of the resource constraints. The resource constraints I am handling are impossible to be handled statically. Such constraints may introduce undefined results for registers.

The interesting case is when multiple instructions attempt to write to the same register. This could happen because of multiple instructions within one execute packet, or because of instructions with different latencies trying to write in the same cycle. For example,

```
MPY .M1 A0, A1, A2
ADD .L1 A4, A5, A2
```

will result in both instructions trying to write register A2 simultaneously, because the multiply instruction has one delay slot.

These register-write conflicts could still be checked statically during code generation, or in my tool by disallowing multiple assignments to the same queue position. The situation becomes more complex with predication. Indeed, the processor reference manual explicitly states that examples like:

```
[!B1] ADD .L2 B5, B6, B7
|| [B0] SUB .S2 B8, B9, B7
```

are legal if zero or one predicates are true, but will produce undefined results if more

than one predicate is true, and that the presence or absence of a conflict cannot be detected [22, Section 3.7.6].

My work contradicts this claim; in most instances, these conflicts can be checked. To handle this case, I extend the verifier further. Instead of a queue of symbolic expressions for each register, I have a queue of *write-histories* for each register. Each write-history is a list of all the (symbolic expression) values that are scheduled to be assigned to that register in that cycle, along with the predicate condition required for the assignment to take place. In the simplest cases, the write-history is either empty (no write to the register) or has a single, unpredicated expression. Whenever a new expression is scheduled to be assigned in a given cycle, it is first checked against all entries already in the write-history. The decision procedure determines if the predicate is mutually exclusive with all the other writes. If so, the tool has **proven** that this assignment is conflict-free; if not, the tool flags the potential conflict to the user.

For example, consider the following code segment:

```
ADD .L2 B1, B0, B2
ADD .L2 B2, B0, B3

[!B1] ADD .L1 A0, A1, A0
|| [!B2] ADD .S1 A0, A2, A0
|| [!B3] ADD .D1 A0, A3, A0
```

The first two ADD instructions initialize B1 through B3 to be equally spaced B0 apart. The next three ADD instructions are in a single execute packet and might conflict in writing to A0 in the same cycle. My tool flags this as a potential conflict, because B0 might be equal to zero.

If I prepend an instruction guaranteeing that B0 is not zero:

```
MVK .S2  1, B0

ADD .L2  B1, B0, B2

ADD .L2  B2, B0, B3


[!B1]  ADD .L1  A0, A1, A0
|| [!B2]  ADD .S1  A0, A2, A0
|| [!B3]  ADD .D1  A0, A3, A0
```

the tool verifies that a conflict cannot happen. This class of resource conflict was previously considered to be unanalyzable, but runtime on my examples was a tiny fraction of a second. Integrating the analysis I am proposing into a compiler should enable generating more efficient code with greater confidence in its correctness.

3.2.7 Condition Code

Unlike the TI C6x, the FR500 has condition codes. There are two types of condition codes for the FR500. One is the Integer Condition Code, (ICC) used for integer instructions; the other is Floating-point/media Condition Code, (FCC) used for floating-point/media instructions. For each type of condition code, there are four banks. They are ICC0 through ICC3 and FCC0 through FCC3. Each bank has 4 flags. ICC_n ($n = 0..3$) has N(Negative), Z(Zero), V(Overflow), C(Carry). FCC_n ($n = 0..3$) has EQ(equal), LT(less than), GT(greater than), UO(disable to compare). There are more detailed descriptions in the FR500 instruction set manual [9].

Conditioned instructions are those instructions which are used together with condition codes. For such instructions, they have to explicitly specify which condition codes are used. For example,

```

subicc  gr14,#1,gr14,icc0
cmpi    gr14,#256,icc1
bge     icc1,#0, Label1

```

In this example, “subicc”(subtract), “cmpi”(compare) and “bge”(Branch greater or equal) are instructions using condition codes. They use icc0 and icc1 explicitly.²

When I implement the equivalence check, I treat each bank of condition codes as a register file, and each flag as a register.

Therefore, for the above example, because the instruction “subicc” changes all the flags(page 3, FR500 [9]) , the equivalence checker will update flags N, Z, V and C, e.g., the Z flag of icc0 will be updated to “(= (- gr14 1) 0)”. For the compare instruction “cmpi”, it modifies icc1 based on the result of “(cmpi gr14 256)”. “bge” is a branch instruction, according to the FR500’s instruction set manual [9], it will check “Not(N xor V)” of icc1. If it is true, a branch will taken. If the equivalence checker cannot prove it is either true or false, both branch directions will be analyzed (see the section of control flow analysis in this chapter for details).

²I should mention that in the branch instruction “bge”, there is a “#0”. It is a hint bit. When there is a high probability of branching, the programmer or compiler should specify 2 to the hint bit. When there is not much probability of branching, the programmer or compiler specifies 0 to the hint bit. The hint bit can be used for branch prediction.

Chapter 4

Examples

I have implemented my verification methods into two tools. In this section, I will give some examples of my tools and show verification results.

4.1 TI TMS320C6x

4.1.1 Software Pipelining Example

My first example demonstrates the verification of a non-trivial code optimization. The example is taken from an article, written by an expert on DSP code optimization, explaining how to optimize code for high-performance DSPs [18]. This article appeared in a trade magazine, and also on the magazine's website.

The most difficult example in the article demonstrates software pipelining a short loop, ostensibly targeting the C62x. Software pipelining is a powerful instruction scheduling technique that exposes additional parallelism in loops, thereby improving performance [15]. The basic idea is to rearrange the computation such that portions of different loop iterations execute at once, similarly to hardware pipelin-

ing. A prologue is required to start the pipelined computation, and an epilogue is required to “flush the pipeline” at the end of the computation.

When I found these examples, I was very happy, because it is not easy to find publically available equivalent code segments. The article stated that the code was for the TI C62x family, which was perfect, because my tool was also targeted for the TI C62x. The C62x family are fixed-point DSPs. I did not notice that the code was actually floating-point code for the TI C67x family. In particular, the MPYSP instruction was not listed in the C62x instruction set, but I assumed that the MPYSP instruction was simply the fixed-point MPY instruction. I will return to this fixed-point/floating-point confusion later.

Figure 4.1 gives the desired functionality in C. Figure 4.2 gives partially optimized, but unpipelined assembly code, which is reasonably readable. Figure 4.3 gives the software pipelined code taken from the article. According to the article, the code was generated by the compiler if the input vectors are declared to be constants. This code is quite hard to read, but the article clearly explains the principles involved.

Intuitively, the @-signs in the comments indicate which iteration of the original loop is being processed by which instructions. For example, the first iteration of the unpipelined loop corresponds to the LDW instructions on lines 17 and 18 of Figure 4.3, followed by the SUB instruction on line 25, the branch on line 27, the multiply on line 31, and so forth. It is also crucial to remember the five branch delay slots. Another way to understand the code is to note that the loop kernel on lines 44-50 of Figure 4.3 performs all the same operations as the original, unpipelined loop kernel, but can do far more operations in parallel because different loop iterations are being processed at the same time. The performance improvement is that the

```

void example1(float *out,
              float *input1,
              float *input2)
{
    int i;

    for(i = 0; i < 100; i++)
    {
        out[i] = input1[i] * input2[i];
    }
}

```

Figure 4.1: Software Pipelining Example. This C code specifies the desired functionality. (Listing taken from [18].)

loop kernel now runs in 2 cycles instead of 10.

I ran my tool on this example, comparing the pipelined and unpipelined programs. The tool discovered that the pipelined code has a bug. In particular, the result of the first iteration is never written, and there is an extra result written at the end. The fix is to add an additional STW instruction to the packet at lines 38–39, and to delete the STW instruction at line 67. Runtime for my tool was less than a second both for finding the bug as well as for verifying my fix.

I was surprised to find a bug in the article. To confirm my results, I asked a friend to run the code on real hardware. Unluckily, he only had a C6201 system available, so he made the same modification (MPYSP to MPY) as I did. His results match mine exactly, supporting my equivalence checker [12].

I reached the conclusion that there is a bug in the original code and wrote this in my paper [8]. It turns out that the code in the article is actually correct, but for the floating-point C67x, not the fixed-point C62x, as stated in the article.

Brett L Huber of Texas Instruments discovered my mistake:

```

1 _example1:
2 ;** -----*
3     MVK     .S2    0x64,B0
4
5     MVC     .S2    CSR,B6
6 ||     MV     .L1X  B4,A3
7 ||     MV     .L2X  A6,B5
8     AND     .L1X  -2,B6,A0
9     MVC     .S2X   A0,CSR
10 ;** -----*
11 L11:      ; PIPED LOOP PROLOG
12 ;** -----*
13 L12:      ; PIPED LOOP KERNEL
14     LDW     .D2    *B5++,B4
15 ||     LDW     .D1    *A3++,A0
16     NOP     2
17 [ B0]SUB    .L2    B0,1,B0
18 [ B0]B      .S2    L12
19     MPYSP   .M1X   B4,A0,A0    ; I assume C62x MPY here.
20     NOP     3
21     STW     .D1    A0,*A4++
22 ;** -----*
23     MVC     .S2    B6,CSR
24     B       .S2    B3
25     NOP     5
26 ; BRANCH OCCURS

```

Figure 4.2: Unpipelined Assembly Code. According to the article [18], this code was compiler-generated, and the compiler was unwilling to pipeline the loop because of possible aliasing between the inputs and the output. The correspondence with the C code is fairly clear.

```

1  _example2:
2  ;** -----*
3      MVK      .S2    0x64,B0
4
5      MVC      .S2    CSR,B6
6  ||      MV      .L1X  B4,A3
7  ||      MV      .L2X  A6,B5
8
9      AND      .L1X  -2,B6,A0
10
11     MVC      .S2X   A0,CSR
12  ||      SUB      .L2    B0,4,B0
13
14  ;** -----*
15 L8:      ; PIPED LOOP PROLOG
16
17     LDW      .D2    *B5++,B4  ;
18  ||      LDW      .D1    *A3++,A0  ;
19
20     NOP      1
21
22     LDW      .D2    *B5++,B4  ;@
23  ||      LDW      .D1    *A3++,A0  ;@
24
25     [B0] SUB      .L2    B0,1,B0  ;
26
27     [B0] B      .S2    L9      ;
28  ||      LDW      .D2    *B5++,B4  ;@@
29  ||      LDW      .D1    *A3++,A0  ;@@
30
31     MPYSP    .M1X   B4,A0,A5  ; I assume C62x MPY here
32  || [B0] SUB      .L2    B0,1,B0  ;@
33
34     [B0] B      .S2    L9      ;@
35  ||      LDW      .D2    *B5++,B4  ;@@@
36  ||      LDW      .D1    *A3++,A0  ;@@@
37
38     MPYSP    .M1X   B4,A0,A5  ;@ I assume C62x MPY here
39  || [B0] SUB      .L2    B0,1,B0  ;@@
40

```

(Figure 4.3, continued on next page)

(continuation of Figure 4.3)

```

41 ;** -----*
42 L9:      ; PIPED LOOP KERNEL
43
44      [B0] B      .S2   L9          ;@@
45 ||      LDW      .D2   *B5++,B4    ;@@@@
46 ||      LDW      .D1   *A3++,A0    ;@@@@
47
48          STW      .D1   A5,*A4++    ;
49 ||      MPYSP     .M1X  B4,A0,A5    ;@@ I assume C62x MPY here
50 || [B0] SUB      .L2   B0,1,B0      ;@@@
51
52 ;** -----*
53 L10:     ; PIPED LOOP EPILOG
54          NOP      1
55
56          STW      .D1   A5,*A4++    ;@
57 ||      MPYSP     .M1X  B4,A0,A5    ;@@@ I assume C62x MPY here
58
59          NOP      1
60
61          STW      .D1   A5,*A4++    ;@@
62 ||      MPYSP     .M1X  B4,A0,A5    ;@@@@ I assume C62x MPY here
64          NOP      1
65          STW      .D1   A5,*A4++    ;@@@
66          NOP      1
67          STW      .D1   A5,*A4++    ;@@@@
68 ;** -----*
69          MVC      .S2   B6,CSR
70          B        .S2   B3
71          NOP      5
72 ; BRANCH OCCURS

```

Figure 4.3: Software Pipelined Assembly Code. If the inputs are declared to be const, the compiler performs software pipelining, resulting in much more efficient code. But, does this do the same thing as Figure 4.2? (Listing taken from [18].)

You claim to have found a bug in a published code optimization example. After examining the code in Figure 4.3, I have determined that the software-pipelined loop is in fact correct. I believe the problem is that you are assuming the MPYSP instruction takes two cycles; actually, it is a floating-point instruction taking four cycles. This instruction is only legal on the C67x, which is a floating-point variant of the C62x; this may explain the confusion [13].

I added the instruction MPYSP as an uninterpreted function with 3 delay slots, and ran it again. It proved that Huber's diagnosis is right, I made a wrong claim in my paper.

However, my equivalence checker still proved valuable. There **was** a bug in my modified C62x code, and the tool found it. When I added the MPYSP instruction to the checker (My checker targets the C62x, but for this example, introducing an uninterpreted function MPYSP doesn't cause any trouble.), the checker proved the equivalence of the two code segments. This example clearly demonstrates how easily a bug crept into hand-tuned code, and how difficult it is to understand and verify VLIW code.

4.2 Fujitsu FR500

4.2.1 Two pieces of equivalent FR500 code

In this section, I will give two assembly programs for the FR500. I received these examples, courtesy of Fujitsu Laboratories of America. The code comes from research there on optimizing compilers [19], Two compilers, the fcc911s and fcc935s, used different combinations of optimization options (object code size minimization,

speed, etc.) to generate assembly code from C programs. I used my tool to verify the equivalence of the generated code.

In the example presented here, there are two pieces of FR500 code, `convolv.asm` and `convol.three.asm` (Code listings are given in Appendix A, The compiler generated `convolv.asm` without any optimization option and `convol.three.asm` with Level 3 speed optimization.). The C program[A.1], which implements a convolution algorithm for the FR500, gives the functionality of the example program.

Obviously, such kinds of examples are the targets of my equivalence checker. We do not need to know the details of the examples, we just want to check whether the optimizations performed by the compiler are correct.

The two programs, `convolv.asm` and `convol.three.asm`, contain 79 and 225 instructions (including labels), respectively. The runtime for my tool to verify equivalence was 2.6s, running on a INTEL Pentium III 736Mhz with 128M RAM. Memory usage was about 12M bytes.

My equivalence checker confirms the equivalence of the two pieces of code. This example demonstrates such an equivalence checker can work with compilers to verify optimization.

Chapter 5

Limitations, Conclusions and Future Work

This thesis has extended previous work on verification of low-level code to handle the complexity of modern, high-performance VLIW processors. My proof-of-concept tools demonstrate that my approach can easily find bugs or confirm correctness in situations that are extremely challenging without automated assistance.

In the rest of this chapter, I will discuss limitations and future work. I will cover both limitations of the techniques I used and limitations of the decision procedure.

5.1 Limitations

In the previous chapters, I've mentioned some of the limitations. In this section, I try to summarize them and give some examples.

5.1.1 Bit Reasoning

As I've already mentioned, my method, which extends a basic symbolic simulation approach, is at the word-level. That abstracts some datapath complexity. However, the equivalence checker will miss some inequivalences and equivalences. For example, the equivalence checker for the TI C6x doesn't handle overflow at all. And the checker will miss the equivalence of, for example, (OR 0101 1010) and (OR 1111 0000), which are both equal to 1111. For another example, consider the SHL instruction.

SHL (.unit) src2, src1, dst

Description: *The src2 operand is shifted to the left by the src1 operand. The result is placed in dst. When a register is used, the six LSBs specify the shift amount and valid values are 0 – 40. When an immediate is used, valid shift amounts are 0 – 31. If $39 < src1 < 64$, src2 is shifted to the left by 40. Only the six LSBs of src1 are used by the shifter, so any bits set above bit 5 do not affect execution.*¹

Because I didn't handle registers at the bit-level, my equivalence checker will report inequivalence for two instructions such as

```
SHL .S1 A0, 8, A2 ; where A0 has a value in binary is
                    ; xxD0 00F0

SHL .S1 A0, 4, A2 ; where A0 has a value in binary is
                    ; xD00 0F00
```

Note: 'x' here denotes any hexadecimal value from 0 to F.

But actually these two instructions are equivalent.

¹Taken from [22, 3-113]. Where LSB is the Least Significant Bit.

Conversely, due to overflow, my equivalence checker can report equivalence for two routines which are not equivalent. For example, consider two computations to compute $(A + B) - C$ and $(A - C) + B$. The operators “+” and “-” are interpreted. Therefore, the equivalence checker will report equivalence for these two computations. However, if $A + B$ overflows, whereas $A - C + B$ does not, the two computations will have different results.

5.1.2 Uninterpreted Functions and Axiom Handling

In the equivalence checker, uninterpreted functions greatly reduce datapath complexity, but due to their limitations, they also introduced errors for some cases. For example, $(\text{MPY } a \text{ } bn)$ should be equal to $(\text{MPY } an \text{ } b)$ for all n . However, it is impossible to introduce such axioms to the decision procedure, because it is impossible to give infinite axioms. The equivalence checker will report inequivalence for such cases, although they are equivalent.

Fortunately, the equivalence checker is conservative, except for situations such as overflow. Based on the definition of uninterpreted functions with equality, two computations of an uninterpreted function are equivalent if and only if all the parameters are equal.

5.1.3 Memory Size

I assume that there are no memory size limits or protection, and that all addresses are accessible. Therefore, there are some possible cases in which memory access is invalid, but the verification tool won't notice it. For example, in order to save a register's data to a temporary address, two programs might use different addresses: one is valid, but the other is invalid. After some computations, both programs

read the data back to the register. In an actual system, the program which visits an invalid address will report an error and exit. But my equivalence checker can't detect such kinds of invalidity and reports that the values read into the registers are equivalent for both programs.

5.2 Future Work

The most obvious direction for future work is to continue development on our tool to fully support the C6x and FR500 architectures. This entails better control-flow analysis and support, as well as implementing all instructions and checks for static resource conflicts. Eventually, such a tool should be integrated into compilers or integrated development environments.

The direction of future development depends largely on the techniques available for dealing with control flow. If I am able to find effective means for analyzing programs, then it is possible to develop a highly useful stand-alone verification tool. If the control flow analysis remains a difficult challenge, then the more likely path for future impact is to integrate the verification into the compiler, which has more information available to it on the control flow of the program.

In the long term, the broad trend in compiler technology is towards ever more sophisticated program analysis to enable better and better compilation. The verification style I am proposing is, in some sense, just a more detailed, careful analysis of the code. My work shows that such analysis is becoming feasible and useful. In the future, such techniques may become standard components of optimizing compilers.

Bibliography

- [1] VLIW Questions by Peter Wayner, November 1994.
<http://www.byte.com/art/9411/sec12/art1.htm>.
- [2] Inc Berkeley Design Technology. VLIW Architecture for DSP: A Two-Part Lecture, 1999.
- [3] Claudia Blank, Hans Eveking, Jens Levihn, and Gerd Ritter. Symbolic simulation techniques — state-of-the-art and applications. In *International Workshop on High-Level Design, Validation, and Test*, pages 45–50. IEEE, 2001.
- [4] R.E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):667–691, August 1986.
- [5] 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.
- [6] David W. Currie. A tool for formal verification of DSP assembly language program, 1999. Msc Thesis of University of British Columbia.

- [7] David W. Currie, Alan J. Hu, Sreeranga Rajan, and Masahiro Fujita. Automatic formal verification of DSP software. In *37th Design Automation Conference*, pages 130–135. ACM/IEEE, 2000.
- [8] X. Feng and A.J.Hu. Automatic Formal Verification for Scheduled VLIW Code. In *Joint Conference on Languages, Compilers and Tools for Embedded Systems & Software and Compilers for Embedded Systems*, pages 84–92. ACM, 2002.
- [9] FUJITSU Ltd. *FR-V Family, FR500 Series Instruction Set Manual*, May 2001. Ver. 1.9 2001.5.
- [10] Kiyoharu Hamaguchi. Symbolic simulation heuristics for high-level design descriptions with uninterpreted functions. In *International Workshop on High-Level Design, Validation, and Test*, pages 25–30. IEEE, 2001.
- [11] Kiyoharu Hamaguchi, Hidekazu Urushihara, and Toshinobu Kashiwabara. Symbolic checking of signal-transition consistency for verifying high-level designs. In *Formal Methods in Computer-Aided Design: Third International Conference*, pages 455–469. Springer, 2000. Lecture Notes in Computer Science Vol. 1954.
- [12] Jizhong Han. Private Communication, January 4, 2002.
- [13] Brett L. Huber. Private Communication, August 27, 2002.
- [14] Wen-Mei W. Hwu, Richard E. Hank, David M. Gallagher, Scott A. Mahlke, Daniel M. Lavery, Grant E. Haab, John C. Gyllenhaal, and David I. August. Compiler technology for future microprocessors. *Proceedings of the IEEE*, 83(12):1625–1640, December 1995.

- [15] Monica S. Lam. Software pipelining: An effective scheduling technique for VLIW machines. In *Conference on Programming Language Design and Implementation*, pages 318–328. ACM SIGPLAN, 1988.
- [16] FUJITSU Ltd. FR500 Series VLIW Embedded Microprocessors: Merging the art of multimedia with the advanced computing technology. *FR500 Series*, 2001. We accessed the on-line article at <http://edevic.fujitsu.com/fj/CATALOG/AD07/07-00029/index.e.html>.
- [17] George C. Necula. Translation validation for an optimizing compiler. In *Conference on Programming Language Design and Implementation*, pages 83–94. ACM SIGPLAN, 2000.
- [18] Rob Oshana. Optimization techniques for high-performance DSPs. *Embedded Systems Programming*, March 1999. We accessed the on-line article at <http://www.embedded.com/1999/9903/9903osha.htm>.
- [19] Sreeranga P. Rajan. Private Communication, April 7, 2002.
- [20] Philips Semiconductors. An Introduction To Very-Long Instruction Word(VLIW) Computer Architecture. www.semiconductors.philips.com/acrobat/other/vliw-wp.pdf.
- [21] Takao Sukemura. FR500 VLIW-architecture High-performance Embedded Microprocessor. *FUJITSU SCIENTIFIC & TECHNICAL JOURNAL*, 36(1):31–38, June 2000. <http://magazine.fujitsu.com/us/vol36-1/paper06.pdf>.
- [22] Texas Instruments. *TMS320C6000 CPU and Instruction Set Reference Guide*, October 2000. Literature Number SPRU189F.

Appendix A

Two equivalent FR500 programs

A.1 C Functionality

```

/*****
 *
 *  A convolution algorithm.
 *
 *****/

#define NTAPS  16
#define SIZE   256
#define WIDTH  16
#define ANS    16128

#ifdef TIME_SOFTWARE
#include "../..../include/time.h"
#endif
```



```

int main()
{
    int A[SIZE];
    int B[SIZE];
    int taps[NTAPS];
    int checksum;
    int i;

#ifdef TIME_SOFTWARE
#include "../../include/time.begin"
#endif

    for(i = 0; i < NTAPS; i++)
        taps[i] = (1 << WIDTH) - 1;

    for(i = 0; i < SIZE; i++)
        A[i] = (1 << WIDTH) - 1;

    for(i = 0; i < (SIZE - NTAPS); i++) {
        B[i] =
            A[i] * taps[0] +
            A[i+1] * taps[1] +
            A[i+2] * taps[2] +
            A[i+3] * taps[3] +
            A[i+4] * taps[4] +
            A[i+5] * taps[5] +
            A[i+6] * taps[6] +
            A[i+7] * taps[7] +

```

```

        A[i+8] * taps[8] +
        A[i+9] * taps[9] +
        A[i+10] * taps[10] +
        A[i+11] * taps[11] +
        A[i+12] * taps[12] +
        A[i+13] * taps[13] +
        A[i+14] * taps[14] +
        A[i+15] * taps[15];
    }

    checksum = 0;
    for (i=0;i<SIZE;i++)
        checksum += A[i] & 0x3f;
#ifdef TIME_SOFTWARE
#include "../../include/time.end"
#endif

#ifdef PRINTOK
    printf("RESULT: Checksum %d\n", checksum);
#endif

    if(checksum==ANS) {
#ifdef PRINTOK
        printf("RESULT: Success!\n");
        printf("RESULT: Return-value %d (%x) \n", 0xbadbeef, 0xbadbeef);
#endif
        return 0xbadbeef;
    }

```

```

    else {
#ifdef PRINTOK
        printf("RESULT: Failure!\n");
        printf("RESULT: Return-value %d (%x) \n", 0xbad, 0x0bad);
#endif
        return 0xbad;
    }
}

```

A.2 convolv.asm (with no optimization)

```

        .program      convol
        .library      "lib935.lib"

_main:
L_47:
    subi.p sp,#16,sp
    sethi #hi(65535),gr11
    setlo.p #lo(65535),gr11
    setlos #63,gr12
    mov sp,gr10
L_55:
    sti.p gr11,@(gr10,-1024)
    subicc gr12,#1,gr12,icc0
    sti gr11,@(gr10,-1020)
    sti gr11,@(gr10,-1016)
    sti gr11,@(gr10,-1012)
    addi.p gr10,#16,gr10

```

```

bge icc0,#3,L_55
L_57:
setlos.p #127,gr11
mov gr0,gr10
cmpi.p gr11,#4,icc0
mov sp,gr12
blt icc0,#0,L_95
L_91:
ldi.p @(gr12,-1024),gr5
addi gr12,#8,gr13
ldi @(gr12,-1020),gr12
andi gr5,#63,gr33
L_65:
ldi.p @(gr13,-1024),gr6
add gr10,gr33,gr7
andi.p gr12,#63,gr10
addi gr13,#8,gr8
ldi.p @(gr13,-1020),gr9
add gr7,gr10,gr14
andi.p gr6,#63,gr10
subi gr11,#1,gr15
ldi.p @(gr8,-1024),gr34
add gr14,gr10,gr35
andi.p gr9,#63,gr10
addi gr8,#8,gr13
ldi.p @(gr8,-1020),gr12
add gr35,gr10,gr10
subi.p gr15,#1,gr11

```

```

andi gr34,#63,gr33
cmpi gr11,#4,icc0
bge icc0,#3,L_65
L_99:
add.p gr10,gr33,gr36
andi gr12,#63,gr5
add.p gr36,gr5,gr10
subi gr11,#1,gr11
mov gr13,gr12
L_95:
L_97:
ldi.p @(gr12,-1024),gr37
ldi @(gr12,-1020),gr39
subicc.p gr11,#1,gr11,icc1
addi gr12,#8,gr12
andi.p gr37,#63,gr38
andi gr39,#63,gr40
add gr10,gr38,gr10
add.p gr10,gr40,gr10
bge icc1,#3,L_97
L_93:
setlos #16128,gr41
cmp gr10,gr41,icc2
bne icc2,#3,L_69
L_68:
sethi.p #hi(195935983),gr10
setlo.p #lo(195935983),gr10
bra L_70

```

```

L_69:
setlos #2989,gr10
L_70:
mov.p gr10,gr8
addi sp,#16,sp
ret
.end

```

A.3 convol_three.asm (-03)

```

.program convol
.library "lib935.lib"
_main:
L_47:
._FUNCTION "fp", "fp", 1040
._CONFIG S, B
._SYMBOL "A", V, 0x80000005, A, -1024
._ARRAY 4, 1, 255, 0
._SYMBOL "B", V, 0x80000005, A, -1953789045
._ARRAY 4, 1, 255, 0
._SYMBOL "taps", V, 0x80000005, A, -1953789045
._ARRAY 4, 1, 15, 0
._SYMBOL "checksum", V, 0x80000005, R, "gr11"
._SYMBOL "i", V, 0x80000005, A, 0
subi.p sp,#16,sp
._LINE 0, 33
sethi #hi(65535),gr11

```

```

._LINE 0, 33
setlo.p #lo(65535),gr11
._LINE 0, 33
setlos #63,gr12
._LINE 0, 33
mov sp,gr10
L_55:
._LINE 0, 34
sti.p gr11,@(gr10,-1024)
._LINE 0, 33
subicc gr12,#1,gr12,icc0
._LINE 0, 34
sti gr11,@(gr10,-1020)
._LINE 0, 34
sti gr11,@(gr10,-1016)
._LINE 0, 34
sti gr11,@(gr10,-1012)
._LINE 0, 33
addi.p gr10,#16,gr10
._LINE 0, 33
bge icc0,#3,L_55
L_57:
._LINE 0, 57
setlos.p #63,gr13
._LINE 0, 56
mov gr0,gr11
._LINE 0, 58
cmpi.p gr13,#5,icc0

```

```

._LINE 0, 57
mov sp,gr10
._LINE 0, 58
blt icc0,#0,L_99
L_95:
._LINE 0, 58
addi.p gr10,#16,gr12
._LINE 0, 58
ldi @(gr10,-1024),gr33
._LINE 0, 58
ldi.p @(gr10,-1020),gr34
._LINE 0, 58
ldi @(gr12,-1024),gr35
L_65:
._LINE 0, 58
andi.p gr33,#63,gr33
._LINE 0, 58
ldi @(gr10,-1016),gr5
._LINE 0, 58
add.p gr11,gr33,gr6
._LINE 0, 58
ldi @(gr10,-1012),gr7
._LINE 0, 58
andi.p gr34,#63,gr11
._LINE 0, 58
addi gr12,#16,gr10
._LINE 0, 58
add.p gr6,gr11,gr8

```



```

._LINE 0, 58
andi gr5,#63,gr11
._LINE 0, 58
add.p gr8,gr11,gr9
._LINE 0, 58
andi gr7,#63,gr11
._LINE 0, 58
ldi.p @(gr12,-1020),gr14
._LINE 0, 58
add gr9,gr11,gr15
._LINE 0, 58
subi.p gr13,#1,gr36
._LINE 0, 58
ldi @(gr10,-1024),gr33
._LINE 0, 58
andi.p gr35,#63,gr11
._LINE 0, 58
ldi @(gr12,-1016),gr37
._LINE 0, 58
add.p gr15,gr11,gr38
._LINE 0, 58
ldi @(gr12,-1012),gr39
._LINE 0, 58
andi.p gr14,#63,gr11
._LINE 0, 58
addi gr10,#16,gr12
._LINE 0, 58
add.p gr38,gr11,gr40

```

```

._LINE 0, 58
andi gr37,#63,gr11
._LINE 0, 58
add.p gr40,gr11,gr41
._LINE 0, 58
andi gr39,#63,gr11
._LINE 0, 58
ldi.p @(gr10,-1020),gr34
._LINE 0, 58
add gr41,gr11,gr11
._LINE 0, 58
subi.p gr36,#1,gr13
._LINE 0, 58
ldi @(gr12,-1024),gr35
._LINE 0, 58
cmpi gr13,#5,icc0
._LINE 0, 58
bge icc0,#3,L_65
L_103:
._LINE 0, 58
ldi.p @(gr10,-1016),gr42
._LINE 0, 58
andi gr33,#63,gr33
._LINE 0, 58
ldi.p @(gr10,-1012),gr44
._LINE 0, 58
add gr11,gr33,gr43
._LINE 0, 58

```

```

andi.p gr34,#63,gr3
._LINE 0, 58
ldi @(gr12,-1020),gr47
._LINE 0, 58
add.p gr43,gr3,gr45
._LINE 0, 58
andi gr42,#63,gr32
._LINE 0, 58
add.p gr45,gr32,gr46
._LINE 0, 58
andi gr44,#63,gr4
._LINE 0, 58
ldi.p @(gr12,-1016),gr7
._LINE 0, 58
add gr46,gr4,gr5
._LINE 0, 58
andi.p gr35,#63,gr36
._LINE 0, 58
ldi @(gr12,-1012),gr9
._LINE 0, 58
add.p gr5,gr36,gr8
._LINE 0, 58
andi gr47,#63,gr3
._LINE 0, 58
add.p gr8,gr3,gr14
._LINE 0, 58
andi gr7,#63,gr4
._LINE 0, 58

```

```

subi.p gr13,#1,gr6
._LINE 0, 58
add gr14,gr4,gr15
._LINE 0, 58
andi.p gr9,#63,gr37
._LINE 0, 58
addi gr12,#16,gr10
._LINE 0, 58
add.p gr15,gr37,gr11
._LINE 0, 58
subi gr6,#1,gr13
L_99:
L_101:
._LINE 0, 58
ldi.p @(gr10,-1024),gr33
._LINE 0, 58
ldi @(gr10,-1020),gr35
._LINE 0, 58
ldi.p @(gr10,-1016),gr37
._LINE 0, 58
ldi @(gr10,-1012),gr39
._LINE 0, 57
subicc.p gr13,#1,gr13,iccl
._LINE 0, 57
addi gr10,#16,gr10
._LINE 0, 58
andi.p gr33,#63,gr34
._LINE 0, 58

```

```

andi gr35,#63,gr36
._LINE 0, 58
add.p gr11,gr34,gr11
._LINE 0, 58
andi gr37,#63,gr38
._LINE 0, 58
add.p gr11,gr36,gr11
._LINE 0, 58
andi gr39,#63,gr40
._LINE 0, 58
add gr11,gr38,gr11
._LINE 0, 58
add.p gr11,gr40,gr11
._LINE 0, 57
bge icc1,#3,L_101
L_97:
._LINE 0, 68
setlos #16128,gr41
._LINE 0, 68
cmp gr11,gr41,icc2
._LINE 0, 68
bne icc2,#3,L_69
L_68:
._LINE 0, 73
sethi.p #hi(195935983),gr10
._LINE 0, 73
setlo.p #lo(195935983),gr10
._LINE 0, 73

```

```
bra L_70  
L_69:  
  _LINE 0, 80  
  setlos #2989,gr10  
L_70:  
  mov.p gr10,gr8  
  addi sp,#16,sp  
  _CONFIG E  
  _LINE 0, 82  
  ret  
  _CONFIG E  
  .end
```