Formal Verification of a DSP Chip Using an Iterative Approach

Ali Habibi[1], Sofiène Tahar[1] and Adel Ghazel[2]
[1]*ECE Dept., Concordia University*
Montreal, Quebec, H3G 1M8 Canada.
[2]*École Superieure Des Communications de Tunis*
2083 Ariana, Tunisia.
[1]{habibi, tahar}@ece.concordia.ca
[2]adel.ghazel@supcom.rnu.tn

Abstract

In this paper we describe a methodology for the formal verification of a DSP chip using the HOL theorem prover. We used an iterative method to specify both the behavioral and structural descriptions of the processor. Our methodology consists of first simplifying the representations of the DSP units. We then prove for each unit that its hardware description implies its behavioral specification. Using the simplified (abstracted) description of the units we have been able to greatly reduce the cost of deducing the behavior of the processor instruction set from the hardware implementation of the processor units. The proposed methodology creates a new representation of the processor at each iteration such that its complexity can be handled by the theorem prover. This allowed us to make a proof of the full instruction set of this processor.

1. Introduction

Hardware and software systems are growing everyday in scale and functionality. This increase in complexity increases the number of subtle errors. Moreover, some of these errors may cause catastrophic loss of money, time, or even in some cases human life. A major goal of software engineering or system design is to enable developers to construct systems that operate reliably despite this complexity. One way of achieving this goal is by using formal methods, which are mathematically-based languages, techniques, and tools for specifying and verifying systems [6]. Although formal methods do not a priori guarantee correctness, they can greatly increase our understanding of a system by revealing inconsistencies, ambiguities, and incompleteness that might otherwise go undetected. Actually, there are many tools performing hardware verification. For instance, the two main stream approaches are model checking and theorem proving.

Model checking is a technique that relies on building a finite model of a system and checking that a desired property holds in that model [6]. Roughly speaking, the check is performed as an exhaustive state space search which is guaranteed to terminate since the model is finite. The technical challenge in model checking is in devising algorithms and data structures that allow us to handle large search spaces. Model checking has been used primarily in hardware and protocol verification; the current trend is to apply this technique to analyze specifications of software systems.

Theorem proving is a technique where both the system and its desired properties are expressed as formulas in some mathematical logic [7]. This logic is given by a formal system, which defines a set of axioms and a set of inference rules. Theorem proving is the process of finding a proof of a property from the axioms of the system. Steps in the proof appeal to the axioms and rules, and possibly derived definitions and intermediate lemmas. While proofs can be constructed by hand, here, we focus only on machine-assisted theorem proving. Theorem provers are increasingly being used today in the mechanical verification of safety-critical properties of hardware and software designs.

In contrast to model checking, theorem proving can deal with infinite state spaces [11]. It relies on techniques like structural induction to prove over infinite domains. Interactive theorem provers, by definition, require interaction with a human, so the theorem proving process is tedious and requires some level of expertise. In the process of finding the proof, however, the human user often gains invaluable insight into the system or the property being proved

2. Related Work

Notable examples about using theorem proving are described in the literature [9]. The most related to our study is the Motorola CAP [5]. During 1992-1996 Brock of Computational Logic, Inc., working in collaboration with Motorola designers, developed an ACL2 [5] specification of the entire Motorola Complex Arithmetic Processor (CAP), a microprocessor for digital signal processing (DSP). The formal specification tracked the evolving design and included a simpler non-pipelined view that was proved equivalent on a certain class of programs.

In this paper, we describe the formal verification using the theorem prover HOL (Higher Order Logic) of the Digital Signal Processor ADSP-2100 of Analog Devices. This processor is a 16 bit fixed-point machine with three computational units, two address generators and one program sequencer. The verification of this processor is very similar in complexity to the Motorola CAP. However, to be able to verify the full instruction set of this processor, we defined an iterative methodology based on the particular characteristics of the architecture of the processor and the way the HOL theorem prover works efficiently.

The HOL system is a powerful and widely used computer program for constructing formal specifications and proofs in higher-order logic [3]. It is implemented using the programming language ML (Meta Language) [10]. The strength of HOL comes from two principles proprieties. First backward (goal-directed) proof is supported, and may be freely mixed with forward proof. Second, adherence to definitional extension guarantees that the consistency of the logic is not compromised [4].

This paper describes a methodology to use the HOL theorem prover to formally verify a DSP. The classical way to use HOL is to write specification and implementation descriptions in HOL and to prove that the latter implies the former. What we present here is not just these steps but also the way we construct the representation of our system in each step. The specification task, for example, is iterated several times until we get the optimized specification that we can use in the verification phase. So it is not just a matter of specifying the processor in the classical way but the purpose is to solve the problem of complexity of the processor.

We think that the assets of our methodology are:
1. The ability to treat the full instruction set of a commercial DSP processor despite its design complexity.
2. The application of the proposed methodology to any DSP processor since our approach is based principally on the hierarchical aspect of this class of processors.
3. It opens the first step to define a semi-automated verification approach using the HOL theorem proving system by adapting the proofs that we made while studying the ADSP-2100 family to other processors.

The rest of the paper is structured as following: In Section 2, we describe the ADSP-2100 processor. In Section 3, we define the methodology that we used in the verification process. In Section 4, we present the specification of the processor and we provide the steps of the verification of its instruction set. We conclude the paper in Section 5.

3. The ADSP-2100 Processor

3.1. Architecture of the ADSP-2100 Processor

The ADSP-2100 family is a collection of programmable single-chip microprocessors that share a common base architecture (Figure 1) optimized for digital signal processing (DSP) and other high-speed numeric processing applications [1]. The various family processors differ principally in the type of on-chip peripherals they add to the base architecture. On-chip memory, a timer, serial port(s), and parallel ports are available in different members of the family. In addition, the ADSP-21msp58/59 processors include an on-chip analog interface for voiceband signal conversion [2].

The principle components of the processor are:

Computational Units—Every processor in the ADSP-2100 family contains three independent, full-function computational units: an arithmetic/logic unit (ALU), a multiplier/accumulator (MAC) and a barrel shifter. The computational units process 16-bit data directly and also provide hardware support for multiprecision computations.

Data Address Generators & Program Sequencer—Two dedicated address generators and a program sequencer supply addresses for on-chip or external memory access. The sequencer supports single-cycle conditional branching and executes program loops with zero overhead. Dual data address generators allow the processor to generate simultaneous addresses for dual operand fetches. Together the sequencer and data address generators keep the computational units continuously working, maximizing throughput.

Memory—The ADSP-2100 family uses a modified Harvard architecture in which data memory stores data, and program memory stores both instructions and data. All ADSP-2100 family processors contain on-chip RAM that comprises a portion of the program memory space and data memory space. The speed of the on-chip memory allows the processor to fetch two operands (one from data memory and one from program memory) and an instruction (from program memory) in a single cycle.

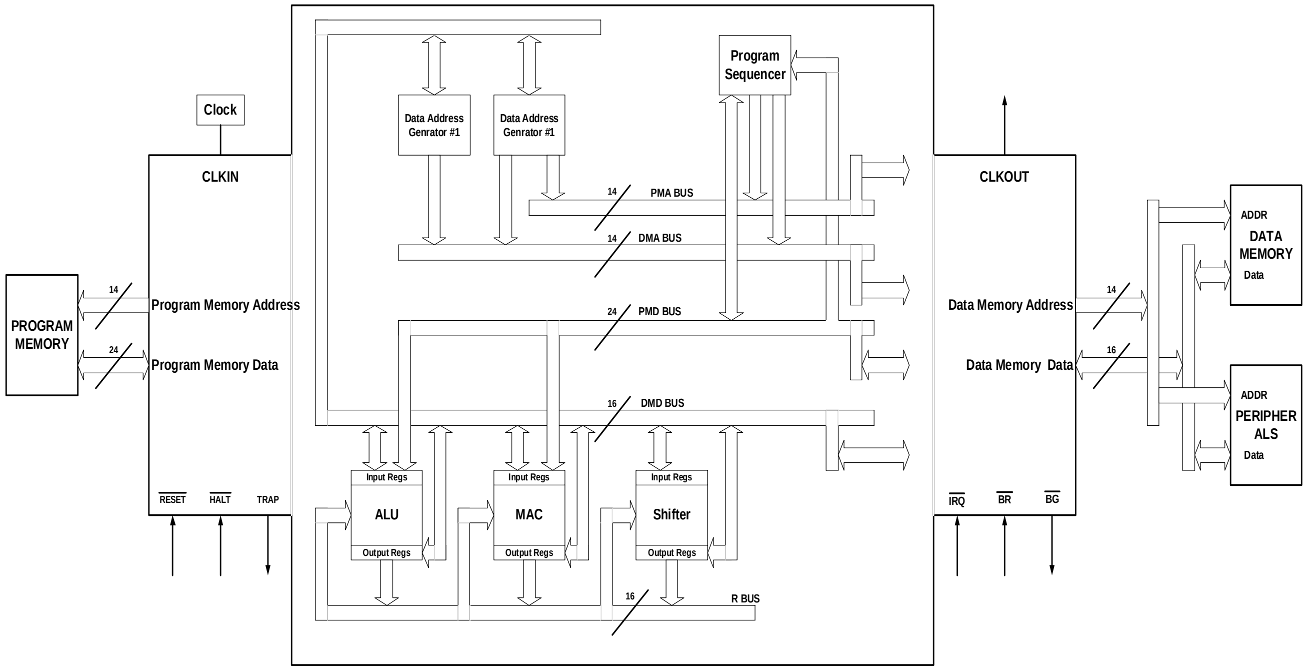

Figure 1. ADSP 2100 Processor Architecture [2].

The processors have five internal buses. The program memory address (PMA) and data memory address (DMA) buses are used internally for the addresses associated with program and data memory. The program memory data (PMD) and data memory data (DMD) buses are used for the data associated with the memory spaces. The buses are multiplexed into a single external address bus and a single external data bus; the BMS, DMS and PMS signals select the different address spaces. The R bus transfers intermediate results directly between the various computational units.

3.2. Instruction Set of the ADSP-2100 Processor

The ADSP-2100 family shares a single unified instruction set designed for upward compatibility with higher-integration devices. For instance, the ADSP-2171, ADSP-2181, and ADSP-21msp58/59 processors have a number of additional and enhanced instructions [11].

The ADSP-2100 family instruction set provides flexible data moves. Multifunction instructions combine one or more data moves with a computation. Every instruction can be executed in a single processor cycle. The assembly language uses an algebraic syntax for readability and ease of coding. A comprehensive set of software and hardware tools supports program development [8].

4. Iterative Verification Methodology

The classical way of using HOL in the verification of a processor is to perform the following steps [9]:

1. Write a description of the behavioral of the processor (according to the HOL syntax).
2. Write a description of the implementation of the processor (according to the HOL syntax).
3. Make the proof (using HOL) of the implication:

"*Processor Implementation*
$\Rightarrow$ *Processor Specification*".

This classical way of verifying processors cannot be used to verify a DSP processor like the ADSP-2100. In fact, the complexity of such a DSP processor in terms of number of variables, variable types and the variety of instructions makes it impossible the use of this direct way [6].

We propose to define a verification methodology which will satisfy certain characteristics to take advantages from the processor architecture and to prepare the proof goal in the simplest way to the HOL system.

The methodology that we defined is described in Figure 2 including four principle steps. First, we simplify the processor units' descriptions. Next, we construct the description of the implementation of the processor. Then, we write the processor specification. Finally, we make the proof of the goal "Processor Implementation $\Rightarrow$ Processor Specification" for every instruction.

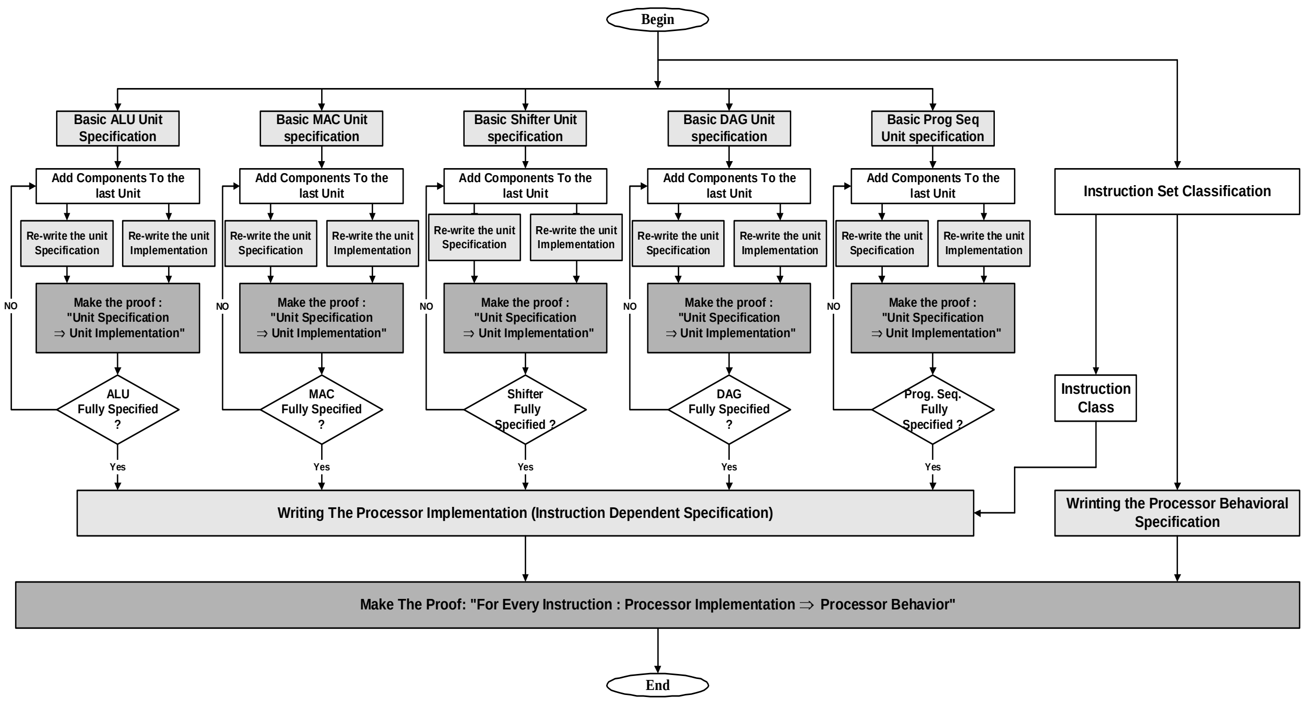
Begin
Basic ALU Unit Specification
Basic MAC Unit specification
Basic Shifter Unit specification
Basic DAG Unit specification
Basic Prog Seq Unit specification
Add Components To the last Unit
Re-write the unit Specification
Re-write the unit Implementation
Make the proof : "Unit Specification ⇒ Unit Implementation"
ALU Fully Specified ?
MAC Fully Specified ?
Shifter Fully Specified ?
DAG Fully Specified ?
Prog. Seq. Fully Specified ?
NO
Yes
Instruction Set Classification
Instruction Class
Writing The Processor Implementation (Instruction Dependent Specification)
Wrinting the Processor Behavioral Specification
Make The Proof: "For Every Instruction : Processor Implementation ⇒ Processor Behavior"
End

Figure 2. Progressive verification methodology.

4.1. Simplification of the Processor Units

Before writing the full description of the implementation of the processor, we have to simplify the units implementations. This step allows the minimization of the number of variables and parameters used in the description of the processor. This is done by eliminating the internal signals in each unit.

We use an iterative method in simplifying the units descriptions. We start by the basic component in the unit. Then we add the components of the unit one by one and we simplify the new representation by eliminating the new internal signals. This step is repeated until the full unit is constructed. This can be seen as the re-assembling of the unit from its components.

The unit simplification task is an obligatory task if we want to succeed in the verification of the processor. In fact, in the literature, there is no work performing the of the instruction set of a DSP commercial processor. Even the work in [5] has just made the specification of the processor. The cause of the difficulty of that task comes from the size of the formal specifications. In fact, the formal description of the implementation of any commercial DSP processor (in ML for our case) takes more than thousand pages. So, just writing it will be very complex. What to say then about manipulating such a long formal description using a theorem prover while constructing complex proofs! The more reductions we are able to perform, the higher chances to achieve our verification goal of the full instruction set of the processor.

4.2. Stepwise Processor Architecture Specification

To construct the instruction dependent specification of the implementation of the processor, we use both the specification of the processor units, defined in the previous step, and a classification of the instruction set.

The instruction set is classified into different classes [12]. Each class is defined by the used units. For example, instructions related to the ALU form a class. Using this classification, we will represent for each class a specific representation of the processor such that only concerned units are integrated. In fact, eliminating units that are not participating in the instruction will simplify the complexity of the processor representation without modifying its behavior.

An important point to take care of while construction the units specification is the order in which components of a particular unit are added one by one. The purpose, in the unit specification procedure, is to eliminate all the internal signals. However, a random order will give a more complex representation of the system making it more difficult the verification task.

4.3. Verification of the Instruction Set

The first step in the instruction set verification is to write the specification of the behavior (the programmer level) of the processor for each instruction according to the instruction set classification discussed in the previous paragraph. Then, it will be possible to construct the goal "Processor Implementation ⇒ Processor Behavior". This time the proof of this goal will be relatively simple since the two descriptions are close in terms of level of abstraction. But, even in this step, we think that performing a progressive proof is better than working directly in the goal as it is.

The instruction dependent representation of the processor can be seen as a re-writing of both descriptions of the Implementation and the Behavior of the processor according to the verified instruction. This task can be automated since it is just a matter of connecting units concerned by the instruction. This will require the definition of a table representing the mapping between the instructions and their related units. In this work we made this task manually. However, in a future work, we plan to define a rigorous mathematical context for the automation of that procedure.

Dealing with Complexity

For the users of HOL the proposed method can be considered as a natural way to solve the problem of complexity of the system. Nevertheless, we can say that it is very simple to guess that the correct way to verify a complex system is to divide it into a collection of small subsystems, but is it always possible to do that division? This method requires a very good knowledge about the system to be verified and about the theorem prover. In fact, we did not need just to reduce the system complexity but also to reduce it in the way that the theorem prover system can handle it

5. ADSP-2100 Specification and Verification

5.1. Specification of the Units

As defined in the methodology, the unit specification is performed in an iterative way. Figure 3 presents the construction of the specification of the ALU unit. As described in this figure, we start with the basic unit of the ALU, and then we progressively add the components of this unit (registers, multiplexers, bus connections,) until we construct the whole ALU.

In each step, we perform two actions:

1. Write the description of the implementation of the constructed unit and the specification of the same constructed unit.
2. Make the proof of the goal:
   "Unit Implementation ⇒ Unit Specification".

The purpose of all these steps is to write a representation of the unit which is close to its behavior.

Experimental Results

The number of iterations and the CPU time per iteration in the construction of the descriptions of the units are summarized in Table 1. The number of iterations refers to the number of steps performed until constructing the specification of the full unit. The average CPU time is the time taken by the proof of the goal: "Processor Implementation Unit ⇒ Processor Specification Unit". The experiments were done on a Pentium II PC with 64 MB memory.

The number of iterations depends on the complexity of the unit. The average CPU time illustrates the effect of our methodology in constructing the specification of the units. It can be noticed from this table that all the proofs took a short time (few seconds) which highlights the effect of our methodology in simplifying the unit specifications.

Unit	Number of Iterations	CPU time
ALU	7	0.5 s/iteration
MAC	9	2.2 s/iteration
Shifter	11	3.7 s/iteration
DAG	8	3.1 s/iteration
Program Seq.	18	4.8 s/iteration

Table 1. Iterative Unit Specification.

5.2. Instruction Set Classification

The second step in simplifying the verification is to classify the instructions of the ADSP-2100 processor. In fact, these instructions can be classified into subsets corresponding to the concerned unit or to the performed operation. We made the following classification:

Class1: *Memory access instructions*: This class contains the instructions that have access to external memory without doing any other instruction.

Class2: *Registers manipulation instructions*: These instructions concern the manipulation of the data registers (registers of ALU, MAC and Shifter units) and the non-data registers (state registers, addresses, etc.).

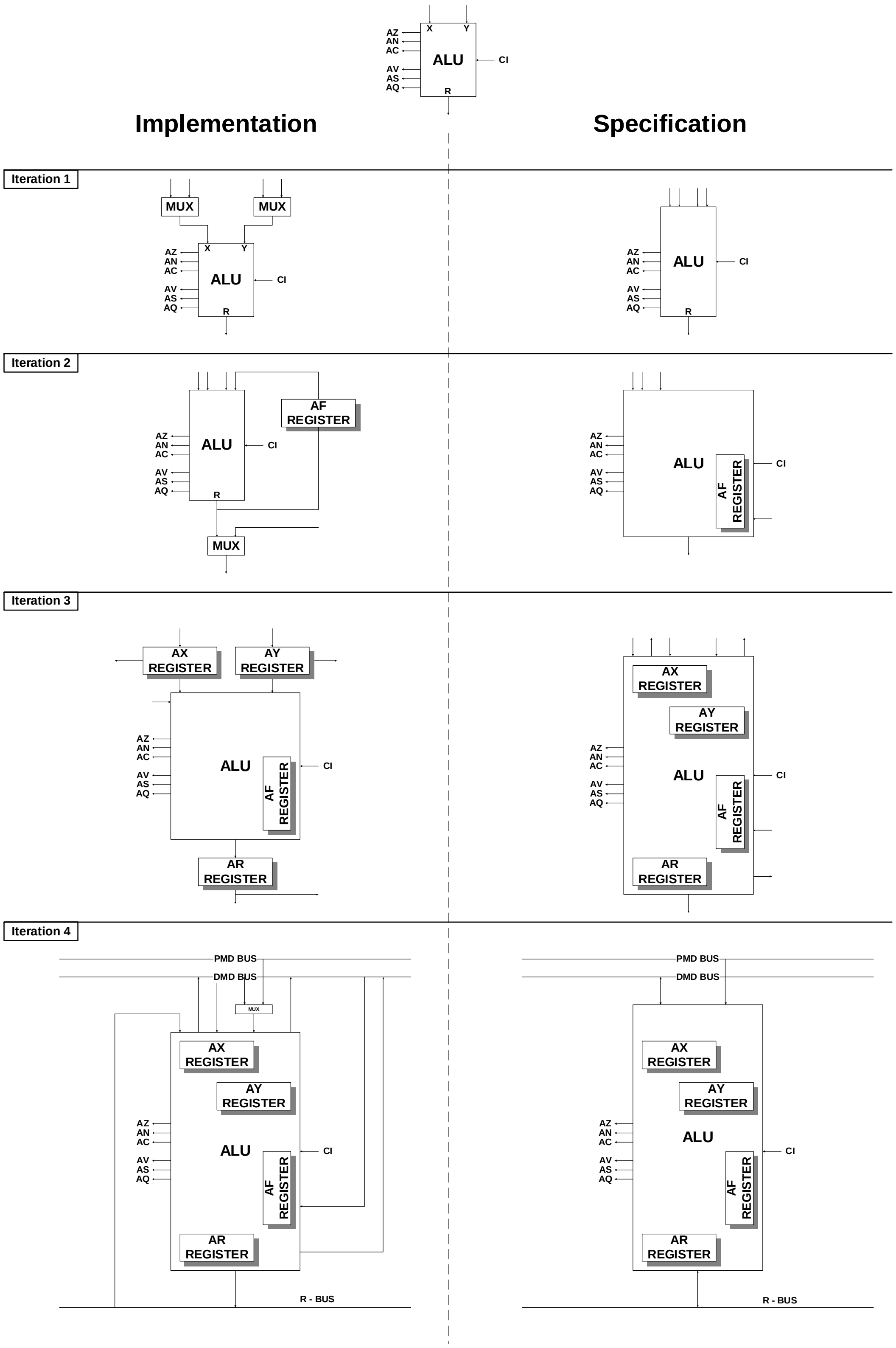

Figure 3. Iterative construction of the ALU unit specification.

Class3: *ALU instructions*: This set of instructions concerns the operations of the ALU with access to the program and data memories or the inter-registers transfer.
Class4: *MAC instructions*: This set of instructions concerns the operations of the MAC with access to the program and data memories or the inter-registers transfer.
Class5: *Shifter instructions*: This set of instructions concerns the operations of the Shifter with access to the program and data memories or the inter-registers transfer.

5.3. Verification of the Complete Processor

For every class of instruction we wrote a description of the implementation of the processor. In this description we integrated only the units or components (buses, memories,) related to the instruction. We defined an instruction dependent description of the processor to simplify its representation as much as possible. In fact, integrating the units that are not related to the instruction increases the complexity of the problem without changing the behavior of the processor.

Goal Division

Despite the simplification of the units specifications, mentioned in Section 4.1, the processor description is still complex because of the big number of parameters. To deal with this problem, we performed the instruction set verification in two steps:
1. Defining an instruction dependant representation of the processor: This is a simplified representation of the processor containing only the components involved in the instruction.
2. Making the global proof: Using the proofs of the last step, it is relatively simple to consider all the full processor description in a global proof.

The Idea of goal division is summarized in Figure 4.

5.4. Experimental Results

Table 2 gives the average CPU times of the proofs for each class. These results prove the capability of our methodology to perform the verification of a DSP chip using the HOL theorem prover. In fact, despite the complexity of the studied DSP processor, the total time taken by all the proofs did not exceed few minutes (if we consider the full instruction set).

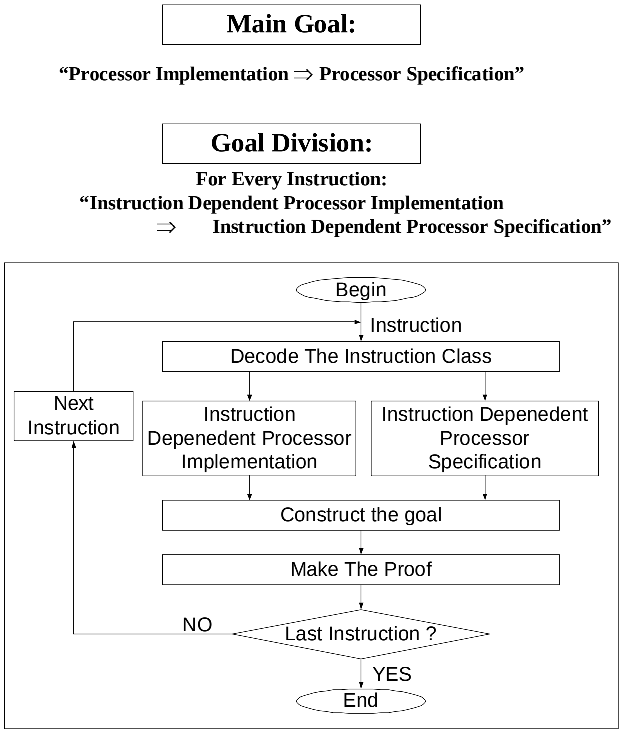
Main Goal:
"Processor Implementation ⇒ Processor Specification"
Goal Division:
For Every Instruction:
"Instruction Dependent Processor Implementation
⇒ Instruction Dependent Processor Specification"
Begin
Instruction
Decode The Instruction Class
Next Instruction
Instruction Depenedent Processor Implementation
Instruction Depenedent Processor Specification
Construct the goal
Make The Proof
NO
Last Instruction ?
YES
End

Figure 4. Goal division procedure.

Instruction Class	CPU Times
Class1	2.5 s
Class2	2.8 s
Class3	2.1 s
Class4	4.1 s
Class5	3.8 s

Table 2. Instruction Set Verification.

6. Conclusions

In this paper, we investigated the formal verification using the HOL theorem prover of the DSP ADSP-2100 processor. The main contributions of this work are: (1) successful use of an iterative proof by construction approach in the HOL theorem prover environment; (2) this is the first time the full instruction set of a DSP commercial chip is verified using a theorem prover; finally, (3) the verification of any other DSP processor will be a matter of adapting and reusing the proofs that we made according to the methodology that we defined in this paper.

We did not present an automatic method but we think that our methodology can be automated. While using any higher-order theorem prover we are under the obligation to guide the proving system to complete the proof. Nevertheless, when dealing with simple goals and for a class of systems (DSP processors for example), we think it is possible to define some algorithms that try to make the proofs automatically. This is will be one of our principal future work in continuation to the one presented in this paper.

By using HOL to verify a complex DSP processor we confirmed the strength of this tool to support the verification of complex systems. However, to verify large circuits it is a must to define a methodology which is strongly dependent on the circuit architecture. Hence, we think it would be better to integrate the verification within the design steps.

Integrating verification in the design flow requires that system developers would all be trained sufficiently well to use formal methods or tools. Ideally, they would routinely use the mathematics underlying the notation of a formal specification language as simply as means of communicating ideas to others on their team or of documenting their own design decisions. They would also routinely use tools like model and proof checkers with as much ease as they use compilers. To make this possible, tools have to integrate the design flow with verification using a problem-dependant methodology and accessible notations to non-experts.

7. References

[1] Applications Engineering Staff of Analog Devices, DSP Division. Digital Signal Processing Applications Using the ADSP-2100 Family. Prentice Hall, Englewood Cliffs, NJ 07632, 1996.

[2] Applications Engineering Staff of Analog Devices, DSP Division. Data sheet of the ADSP-2100 Family DSP - Microcomputers - ADSP-21xx, 1996.

[3] P. Andrews. An Introduction to Higher Order Logic: To Truth through Proof. Academic Press, New York, 1986.

[4] G. Birtwistle, B. Graham, and S.- K. Chin: new_ theory ‘HOL‘; An Introduction to Hardware Verification in Higher Order Logic, August 1994.

[5] B. Brock, M. Kauffmann and J.S. Moore, ACL2 Theorems about Commercial Microprocessors, Formal Methods in Computer-Aided Design, Lecture Notes in Computer Science 1166, Springer-Verlag, November 1996, pp. 275-293.

[6] E. M. Clarke and J. M. Wing. Formal Methods: State of the Art and Future Directions. ACM Computing Surveys, December 1996.

[7] M. Gordon, and T. Melham, Introduction to HOL: A theorem Proving Environment for Higher-Order Logic, Cambridge University Press: Cambridge, UK, 1993.

[8] R. J. Higgins, Digital Signal Processing in VLSI. Englewood Cliffs, NJ: Prentice-Hall, 1990.

[9] T.F. Melham, Higher Order Logic and Hardware Verification, Cambridge Tracts in Theoretical Computer Science 31, Cambridge University Press, 1993.

[10] R. Milner, M. Tofte, R. Harper and D. MacQueen. The definition of Standard ML. The MIT Press, Cambridge, Massachusetts and London, England, 1997.

[11] W. S. Steven, The Scientist and Engineer's Guide to Digital Signal Processing, Second Edition. California Technical Publishing, 1999.

[12] Tahar and R. Kumar: A Practical Methodology for the Formal Verification of RISC Processors; Formal Methods in Systems Design, Kluwer Academic Publishers, 13(2): 159-225, September 1998