

Formal Verifaction of the ADSP-2100 Processor Using the HOL Theorem Prover

Ali Habibi¹, Sofiène Tahar¹ and Adel Ghazel²

¹Electrical & Computer Engineering Department, Concordia University
Montreal, Quebec, H3G 1M8 Canada
{habibi, tahar}@ece.concordia.ca

²École Supérieure Des Communications de Tunis
2083 Ariana, Tunisia.
adel.ghazel@supcom.rnu.tn

Technical Report

March 2002

***Abstract.** In this technical report, we present the application of formal verification to digital signal processors of the family ADSP-2100 using the HOL (Higher Order Logic) theorem prover. To solve the problem of complexity related to the big number of parameters of the processor, we used a structured method based on our knowledge about this processors family. In this method, we worked on the units of the processor as separate sub-systems in order to simplify their specifications by omitting the internal signals. We show details of the specification and verification strategies used and display experimental results as well the lessons learned.*

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 many 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 such systems [9]. Even 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.

In the past, the use of formal methods in practice seemed hopeless. The notations were too obscure, the techniques did not scale, and the tool support was inadequate or too hard to use. There were only a few non-trivial case studies and together they still were not convincing enough to the practicing software or hardware engineer. Few people had the training to use them effectively on the job [6].

Only recently have we begun to see a more promising picture for the future of formal methods. For software specification, industry is open to trying out notations like Z to doc-

ument a system's properties more rigorously. For hardware verification, industry is adopting techniques like model checking and theorem proving to complement the more traditional one of simulation. In both areas, researchers and practitioners are performing more and more industrial-sized case studies, and thereby gaining the benefits of using formal methods. Actually there are many tools performing hardware verification. Principally the two approaches model checking and theorem proving are getting more interest.

Notable examples about using theorem proving are described in the literature [12]. The most related to our study is the Motorola CAP [6]. During 1992-1996 Brock of Computational Logic, Inc., working in collaboration with Motorola designers, developed an ACL2 specification of the entire Motorola Complex Arithmetic Processor (CAP), a microprocessor for digital signal processing (DSP). The CAP is the most complicated microprocessor yet formalized, with a three stage pipeline, six independent memories, four multiplier-accumulators, over 250 programmer-visible registers, and an instruction set allowing the simultaneous modification of well over 100 registers in a single instruction. The formal specification tracked the evolving design and included a simpler non-pipelined view that was proved equivalent on a certain class of programs. Finally, Brock used ACL2 to verify the binary microcode for several DSP algorithms [5].

The verification of the processor ADSP-2100 is very similar in complexity to the Motorola CAP. However, we use the HOL theorem prover [7] and our object is to verify all the instruction set of the processor. The ADSP-2100 family is a collection of programmable single-chip microprocessors that share a common base architecture optimized for digital signal processing (DSP) and other high-speed numeric processing applications. 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.

The HOL system is an interactive environment for machine-assisted theorem-proving in higher-order logic [10]. The HOL logic is simple but expressive, incorporating higher-order functions and Milner-style polymorphism [4,7]. HOL uses the programming language ML to program proof strategies [11]. Theorems are encoded via an abstract type whose only constructors are the primitive inference rules of the logic. Other proof strategies ultimately resolve into these steps. This combines a high degree of security with great flexibility: the user can safely extend the inference system without being a logician. A large number of derived rules have been programmed for, e.g., proving facts in linear arithmetic, defining recursive datatypes or handling inductive definitions. 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 [1].

The first part of this report presents the specification of the ADSP-2100 signal processor. In the verification section, we highlight the approach we used to deal with the complexity of this digital signal processor. The third and final part of this report outlines the general remarks coming out from this study and the future directions of the formal verification.

2. The ADSP-2100 Family

2.1. Basic Units

The processors of the ADSP-2100 family have the same internal architecture described in Figure 1. This architecture has three separated computational units: Arithmetic and Logic Unit (ALU), Multiplier/Accumulator (MAC) and the Barrel Shifter. These units treat 16 bits fixed point data [3].

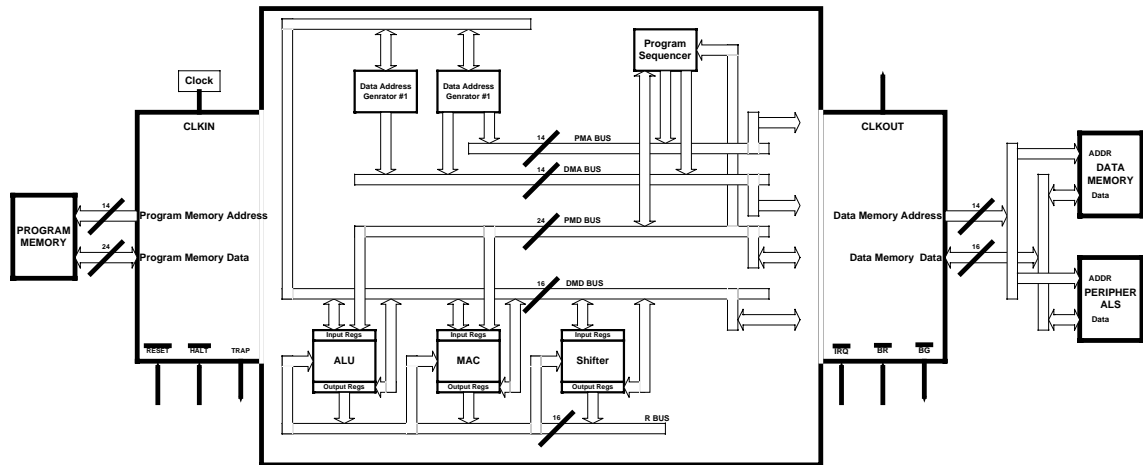


Figure 1. ADSP 2100 Processor Architecture [2]

The two Address Generators and the Program Sequencer are responsible of the addressing management. The use of two address generators allow the execution of two data reading (fetch) from the external memory. In the program sequencer has some embedded units allowing the execution of internal loops which increases the efficiency of the processor [14].

The principal characteristic of this family of processors is that it is based on a modified Harvard architecture [8]. In fact, data is stored in both the data and the program memories. This is very important because it allows the execution of two instructions in the same clock cycle.

2.2. Buses

The processor ADSP-2100 has five internal buses. These interconnect the internal units together and allow them to read (store) data from (to) the external memories.

2.2.1. PMA Bus (Program Memory Address)

According to the actual instruction, the program sequencer generates the new address. This later is written in the PMA bus, 14-bit bus, which enables addressing up to 14KB of external program memory and 14KB of external data memory. The distinction between the two memories is defined according to the pin PMDA.

2.2.2. PMD Bus (Program Memory Data)

This bus serves to load instructions from the external memory to the internal instructions' register. Instructions are loaded during one clock cycle and are executed in the next cycle simultaneously with loading the next instruction. All loaded instructions are also written in the cache memory.

2.2.3. DMA Bus (Data Memory Address)

This is a 14 bit bus allowing the direct access to a 16KB external data memory. The data addresses can be provided by two different sources: an absolute value defined by the instruction (direct addressing) or the output of one of the two address generators (indirect addressing).

2.2.4. DMD Bus

This is a 16 bit bus that allows the access the content of any register and to transfer this content to any other register or any external memory. This transfer take one clock cycle.

2.2.5. R Bus

The units of the processor are organized to work in parallel. To eliminate any possible delay, the internal bus R is used to transfer data between the registers of the three computational units of the processor (ALU, MAC, SHIFTER). This a direct transfer which takes a unique clock cycle.

2.2.6. Memory

The architecture of the family of processors ADSP-2100 allows the storage of data in the data memory and data mixed with instructions in the program memory. Every processor contains one RAM and/or one ROM.

2.3. Instruction Set of the ADSP-2100 Family

The instruction set of the ADSP-2100 family has two principle proprieties: the classification of the instructions and the multiplicity of instructions per command [2]. The instructions are classified by the concerned unit. In other terms, the instructions of the ALU have the same format that is different form that of the program sequencer for example. On the other hand, the majority of the instructions are composed in two parts: the first is a memory access (for reading or writing) and the second is an operand execution (ADD operand for example).

The general format of the instructions is described in Figure 2. It is composed from two principal parts: the identifier of the instruction and the parameters of the instructions. In the Figure 4.3, the example of the general instruction executing of an operation related to the ALU with two data reading operations the first from the program memory and the second from the data memory.

General Format:

Opcode Identifier	Opcode Parameters
-------------------	-------------------

Example: Opcode type 1

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	PD	DD	AMF				Yop	Xop	PM	PM	DM	DM											
																	I	M			I	M		

Figure 2. General Format of the Instructions of the ADSP-2100 Family [2]

The instructions of the ADSP-2100 family can be classified into the following classes :

1. MAC/ALU instructions with memory access: they concern the operations done by the ALU and MAC units with memory access for reading.
2. Memory access instructions: they allow the allow the access to the data and program memories in read/write modes.

3. MAC/ALU instructions with inter-registers data transfer: they allow the execution of ALU/MAC operations with an inter-register data transfer.
4. Registers direct access instructions: they allow reading the content of one of the registers of the three computational units (ALU, MAC and Shifter).
5. Conditional MAC/ALU instructions: they allow the execution of one ALU/MAC instruction when a condition, based on one of the output signals of one of the computational units, is satisfied.
6. Shift instructions: they concern the operations relative to the Shifter unit.
7. Mode control instructions: they allow the change of the processor working modes. This is the case for example if we want to change the active set of registers.
8. Program sequence instructions: these are the JUMP, the conditional JUMP and the DO UNTIL loop instructions.
9. Division instructions: they concern the operations of division done by the two sub-units DIVS and DIVQ.
10. Control instructions: these are the NOP (No Operation) and Idle operations.

3. Formal Verification with Theorem Proving

3.1. HOL Logic

The HOL language is a variant of Church's simple theory of types. The syntax and semantics of the particular logical system supported by HOL [1] is described in detail in the Table 1. The meta-language for this description will be English, enhanced with various mathematical notations and conventions. The object language of this description is the HOL logic.

3.2. HOL System

The HOL system [4] is a powerful and widely used computer program for constructing formal specifications and proofs in higher order logic. The system is used in both industry and academia to support formal reasoning in many different areas, including hardware design and verification, reasoning about security, proofs about real-time systems, semantics of hardware description languages, compiler verification, program correctness, modeling concurrency, and program refinement. HOL is also used as an open platform for general theorem-proving research.

HOL uses the programming language ML [10] to program proof strategies. Theorems are encoded via an abstract type whose only constructors are the primitive inference rules of the logic. Other proof strategies ultimately resolve into these steps. This combines a high degree of security with great flexibility: the user can safely extend the inference system without being a logician. A large number of derived rules have been programmed for, e.g., proving facts in linear arithmetic, defining recursive datatypes or handling inductive definitions. (The large user base contributes to the stock of derived rules.) Backward (goal-directed) proof is supported, and may be freely mixed with forward proof.

HOL as a theorem prover can deal directly with infinite state spaces [14]. 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 slow

and often error-prone. In the process of finding the proof, however, the human user often gains invaluable insight into the system or the property being proved.

Table 1. Terms of the HOL Logic [7]

Kind of term	HOL notation	Standard notation	Description
Truth	T	T	true
Falsity	F	\wedge	false
Negation	$\sim t$	$\neg t$	<i>not t</i>
Disjunction	$t_1 \setminus / t_2$	$t_1 \vee t_2$	<i>t₁ or t₂</i>
Conjunction	$t_1 / \setminus t_2$	$t_1 \wedge t_2$	<i>t₁ and t₂</i>
Implication	$t_1 ==> t_2$	$t_1 \Rightarrow t_2$	<i>t₁ implies t₂</i>
Equality	$t_1 = t_2$	$t_1 = t_2$	<i>t₁ equals t₂</i>
\forall -quantification	$!x.t$	$\forall x.t$	<i>for all x: t</i>
\exists -quantification	$?x.t$	$\exists x.t$	<i>for some x: t</i>
ϵ -term	$@x.t$	$\epsilon x.t$	<i>an x such that: t</i>
Conditional	if t then t_1 else t_2	$(t \rightarrow t_1, t_2)$	<i>if t then t₁ else t₂</i>

3.3. ML Language

ML is a general purpose functional language [11] which has been used extensively at Cambridge University, Edinburgh University, INRIA and elsewhere for implementing theorem provers. The HOL system is written and embedded in ML. HOL proofs are ML programs and so must conform to the syntax and semantics of ML.

There are several reasons why ML is a suitable language in which to write theorem provers. First it has a strong yet flexible type system. Typing is used at compile time to check that things are used in a manner consistent with their definition. For example, in the implementation of HOL we define a type thm (theorem) and the ML compiler uses its strong typing to ensure that if either an argument to a function or the result returned by a function is supposed to be of type thm , then it is.

Second, ML has abstract datatypes which are used to group together a datatype and operations over it. Abstract datatypes are written in such a way that direct access to the datatype constructors is not possible. Instances of the data type can be created and accessed only through specially tailored functions defined within the abstract definition. HOL implements theorems as an abstract datatype and supplies only one mechanism for generating new theorems, namely the inference rule.

Finally, ML supports and encourages the use of functions as first-class values. Patterns of ML commands in HOL proofs can be readily combined into more powerful commands. Common higher-level commands may be named and saved for later use, thus raising the level of abstraction and understanding.

4. Specification and Verification Methodology

4.1. Problem Description and Proposed Solution

The classical way of using HOL in the verification of a processor is to perform the following steps [10]:

1. Write the specification of the behavior of the processor (according to the HOL syntax).
2. Write the specification 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. In fact, the complexity of 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 methodology for the verification of the DSP processor. This methodology will satisfy certain characteristics to guarantee its capability to deal with the processor features, to take advantage 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 3 including four principle steps. First, we simplify the processor units’ specifications. Next, we construct the specification of the implementation of the processor. Then, we write the specification of the behavior of the processor. Finally, we make the proof of the goal “*Processor Implementation* \Rightarrow *Processor Specification*” for every instruction.

The unit simplification task is an obligatory task if we want to succeed in the verification of the processor. In fact, to our best knowledge there is no work that did the verification of the instruction set of a DSP commercial processor. Even [6] has just made the specification of the processor. The cause of the difficulty of that task comes from the size of the specifications. In fact, the specification of the implementation of any commercial DSP processor (in ML for our case for example) takes more than thousand pages. So, just writing it will be very complex. Then what about manipulating such a long specification using a theorem prover while constructing complex proofs! The more reductions we are able to perform, the more the chances to achieve our verification goal of the full instruction set of the processor are higher.

4.2. Simplification of the Processor Units

Before writing the full specification 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 specification of the processor. This is done by eliminating the internal signals in each unit.

We use a progressive method in simplifying the units specifications. We start by the basic component in the unit. Then we add the components of the unit one by one and 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.

4.3. Constructing the Specification of the Processor Architecture

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 [15]. Each class is defined by the used units. For example, instruction 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.

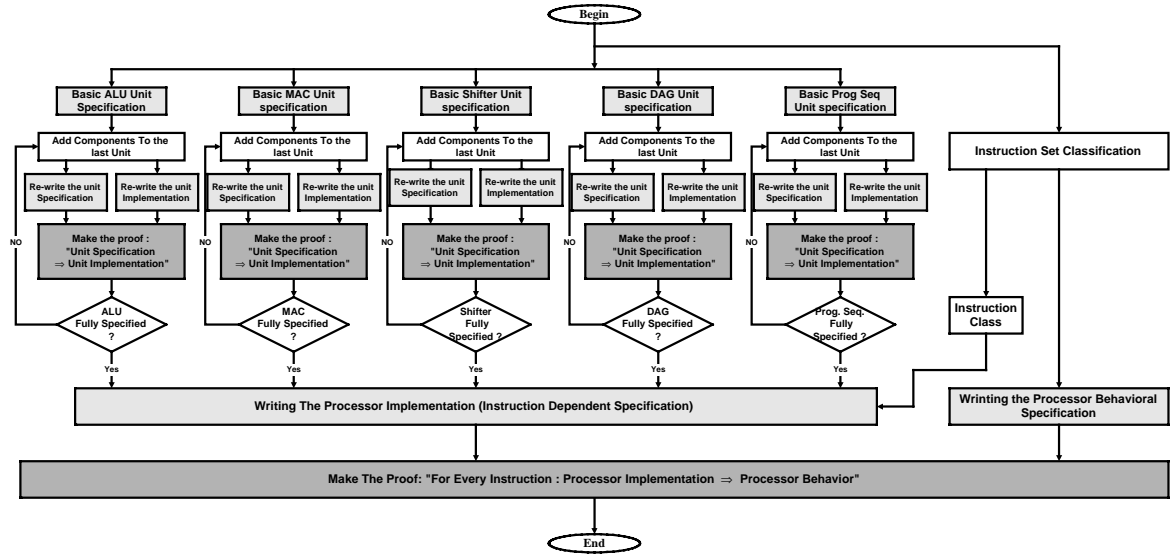


Figure 3. Progressive Verification Methodology

5. Formal Specification

In our case the formal specification concerns two parts: the specification of the hardware and the specification of the behavior. Our approach is to simplify the proofs as possible. For this reason, we started by specifying the units of the processor. Then we wrote a functional specification of these units. Our first task was then to make the proof of the following implication:

hardware representation \Rightarrow functional representation for every units.

The first task will allow us to manipulate simpler representations of the units during the verification phase. In fact, this will give us a high level representation of the units omitting the internal signals and therefore minimizing the number of parameters.

5.1. Hypothesis and Approximations

In our study of the family of ADSP-2100 processors, we considered some hypothesis. These later have as objective to simplify our verification task without affecting the validity of the results of our work.

5.1.1. Memory Access Time

We considered that the time of the execution of one instruction is one clock cycle. This may appear contradictory to the real functioning of the processor because we did not take into account the time of accessing the external memories. But, since for the case of ADSP-2100 all instructions are loaded in the cache memory, we can suppose that the effective time of the execution of one instructions is a single clock cycle.

5.1.2. Transfer of Data in the Buses

We considered that the data is transferred from/to buses is instantaneous. This is not contradictory with reality since this action is very short compared to the clock cycle duration. Therefore, the fact of setting its value to zero will not affect of the global behavior of the real system.

5.1.3. Registers Access

Registers are accessed at the beginning of the clock cycle for writing and at the end of the clock cycle for reading. In our representation this will be seen as the value presented in the output of a register at t is that was in its input at $t-1$ (i.e. at the end of the last cycle). This representation keep the content of the register correct at any time t and make it easy to manipulate the registers while making the proofs.

5.2. Components Specification

In our manipulation of the ADSP-2100 processors family we manipulated many data types. These are summarized in Table 2.

5.2.1. Multiplexers

In each unit of the processors there are many multiplexers allowing the selection between the possible inputs and their direction to the corresponding outputs. An enumeration of all the multiplexers used in the ADSP-2100 family has given us seven classes that are used. For this reason, we wrote the specification of these classes only. We considered in our specification that the transfer of data between the input and output is instantaneous. This is not contradictory with the reality because the transfer time is very short.

Table 3 illustrates the classes of multiplexers used in the specification of the units of the processors ADSP-2100.

The following specification, written in ML, presents the case of one multiplexer of the class 4. In this case, the multiplexer has two inputs the first of type Word8 and the second of type Word16, and one output of type Word8. The selection between the two inputs is done according to the value of the boolean parameter *input_select* which, when set to true (T) gives the first input to the output and when set to false (F) gives the second input to the output:

```
MUX_8_16_8 = Define(  
  MUX_8_16_8  
  (FromWord16ToWord8 :word16 -> word8)  
  (input1 :num -> word8, input2 :num -> word16, output :num -> word8, input_select :num -> bool)  
  =  
  (!t :num.  
    (  
      (input_select t = T) ∧ (output t = input1 t)  
      ∨ ((input_select t = F) ∧ (output t = (FromWord16ToWord8 (input2 t))))  
    )  
  )  
);
```

This defines for every time t , the output of the multiplexer *output* is given by:

1. The input one *input1* if the selection parameter *input_select* is at the value T .
2. The input two *input2* if the selection parameter *input_select* is at the value F .

The HOL system reply to the last specification by saying that the last definition is well formed. It stores it then as theorem called *MUX_8_16_def* which is of the reserved type *Thm.thm*.

Table 2. The Data Types used in the Specification of the Processor ADSP-2100

Type	Meaning	Comments
Bool	Type Boolean : True or False.	This type concerns the control bits and the output flags of the computational units of the processor.
Word5	5 bits word.	This type concerns the exponent parameter of the Shifter unit.
Word8	8 bits word	This type concerns the shift parameter of the Shifter unit and the third register of the MAC unit.
Word14	14 bits word	This is the type of the PMA and DMA buses.
Word16	16 bits word	This type concerns the majority of the input/output registers of the units of the processor. It concerns also the PMD and R buses.
Word24	24 bits word	This type concerns the bus PMD which allows the access to the program memory.
Word32	32 bits word	This type concerns the output of the bloc ShifterArray of the unit Shifter.
Word40	40 bits word	This type concerns the first input of the bloc ADD/SUB-TRACT of the unit MAC.
Num	A natural value.	This type is used to represent the time (t).
OpCode	Vector of 24 boolean	The representation of the instructions codes.

Table 3. The Classes of Multiplexers

Class	Inputs		Output
	Number	Type	Type
1	2	(Word16,Word16)	Word16
2	2	(Word16,Word8)	Word16
3	2	(Word16,Word8)	Word8
4	2	(Word8,Word16)	Word8
5	2	(Word8,Word16)	Word5
6	2	(Word14,Word16)	Word14
7	3	(Word8,Word8,Word8)	Word8

5.2.2. Registers

The units of the processor contain a large number of registers that are distinguished by their sizes and by their structures. In fact, the sizes of the registers vary depending on the task to which they are affected. On the other hand, many registers are grouped together to form the same entity. Another fact, is that all the registers are duplicated, so in an interruption the processor uses the second bank of registers without affecting the contents of the first bank of registers.

Table 4 summarises the classes of the registers that we defined for the hardware specification of the units of the processor.

Table 4. The Classes of Registers

Class	Number of registers	Size(number of bits)	Examples
1	1	16	Registers AF and AR in ALU.
2	2	2x16	The register AX which is composed of the two sub-registers AX0 and AX1.
3	1	8	Register MR2 of the unit MAC.
4	1	5	Register SB of the unit Shifter.
5	4	4x14	Register I of the address generator which is composed of 4 sub-registers each of them is a 14 bits register.

The following code illustrates the specification of the register AF of the ALU which is a generic representation of the members of the registers of the class 1. We considered the input of the register *AF_input*, the output of the register *AF_output* and the parameter *banc_select*. This last is a Boolean that controls the used banks of registers.

```

ALU_AF_reg = (^ALU_AF_reg(AF_input, AF_output, bank_select) =
!t .
(? AF_output_bank0 AF_output_bank1 .
(
(bank_select t = T) => ( if (t = 0) then F else (AF_output_bank0 t = AF_input (t-1)) )\
| ( if (t = 0) then F else (AF_output_bank1 t = AF_input (t-1)) )
)
^
(
(bank_select t = T) => (AF_output t = AF_output_bank0 t)
| (AF_output t = AF_output_bank1 t)
)
)
);

```

This defines for every time t , the output of the register AF_output corresponds to its input at the time $(t-1)$. We used also the parameter $banc_select$ to make the selection between the two possible banks.

5.2.3. Data Buses

In our representation of data buses we considered that the output of the bus corresponds to its input at the same instant. This approximation did not alter the real functioning mode of the processor. In fact, the data is read at the beginning of the cycle and is written at its end. We wrote a simple representation of all the buses to have a clear and simple representation of the processor. The case of the specification of the bus PMA is illustrated in the following:

```

PMABusImp = (^PMABusImp(PMABusInput :num -> word14,
PMABusOutput :num -> word14, t:num) =
(
(PMABusOutput t) = (PMABusInput t)
)
);

```

5.2.4. Memory Access

The family of processors ADSP-2100 contains three types of memory: BOOT memory, program memory and data memory. The program and data memories can be read and written. But the BOOT memory can be accessed only for reading. To represent the modes of accessing the memory, we considered 4 functions that, according to their input parameters, access one of three memories. Table 5 illustrates these four functions.

The memory access functions take in their consideration the parameters RD_LOW (READ) and WR_LOW (WRITE) that controls reading and writing operations. This is illustrated in the specification of the operation *storedata*:

```

!program_mem_data data_mem_data Index_register Modify_register PMS_low DMS_low RD_low
WR_low data_to_write t.store_data program_mem_data data_mem_data
(Index_register,Modify_register,PMS_low,DMS_low,RD_low,WR_low, data_to_write,t) =
((RD_low t,WR_low t) = (T,F))
^ (((PMS_low t,DMS_low t) = (F,T))
^ (data_to_write t = program_mem_data (Index_register (t - 1)) (Modify_register (t - 1)) (t - 1))
^ ((PMS_low t,DMS_low t) = (T,F)))

```

Table 5. The Memory Access Functions

Function	Parameters	Access
fetch_data	(BMS_low, PMS_low, DMS_low) = (F,T,T)	Read data from the boot memory.
	(BMS_low, PMS_low, DMS_low) = (T,F,T)	Read data from the program memory.
	(BMS_low, PMS_low, DMS_low) = (T,F,T)	Read data from the data memory.
fetch_address	(BMS_low, PMS_low, DMS_low) = (F,T,T)	Read address from the boot memory.
	(BMS_low, PMS_low, DMS_low) = (T,F,T)	Read address from the program memory.
	(BMS_low, PMS_low, DMS_low) = (T,T,F)	Read address from the data memory.
Store_data	(PMS_low, DMS_low) = (F,T)	Store data in the program memory.
	(PMS_low, DMS_low) = (T,F)	Store data in the data memory.
Store_address	(PMS_low, DMS_low) = (F,T)	Store addresses in the program memory.
	(PMS_low, DMS_low) = (T,F)	Store addresses in the data memory.

5.3. Units Specification

For every unit of the processor, we defined a behavioral specification and a hardware specification of the unit. This is done to simplify the instruction verification task by manipulating a simple representation of the processor.

5.3.1. Hardware Specification of the ALU

The ALU, see Figure 4, is the unit responsible for the arithmetic and logic operations. It is composed of a group of input registers, a group of output registers, many multiplexers and one base unit making the arithmetic and logic operations.

To specify the ALU we considered the specifications of the registers and the multiplexers previously presented. The basic unit of the ALU is presented by many functions that get as inputs two parameters X and Y and give as output one result R plus the control flags. The parameters of input/outputs that we used are defined in the Table 6.

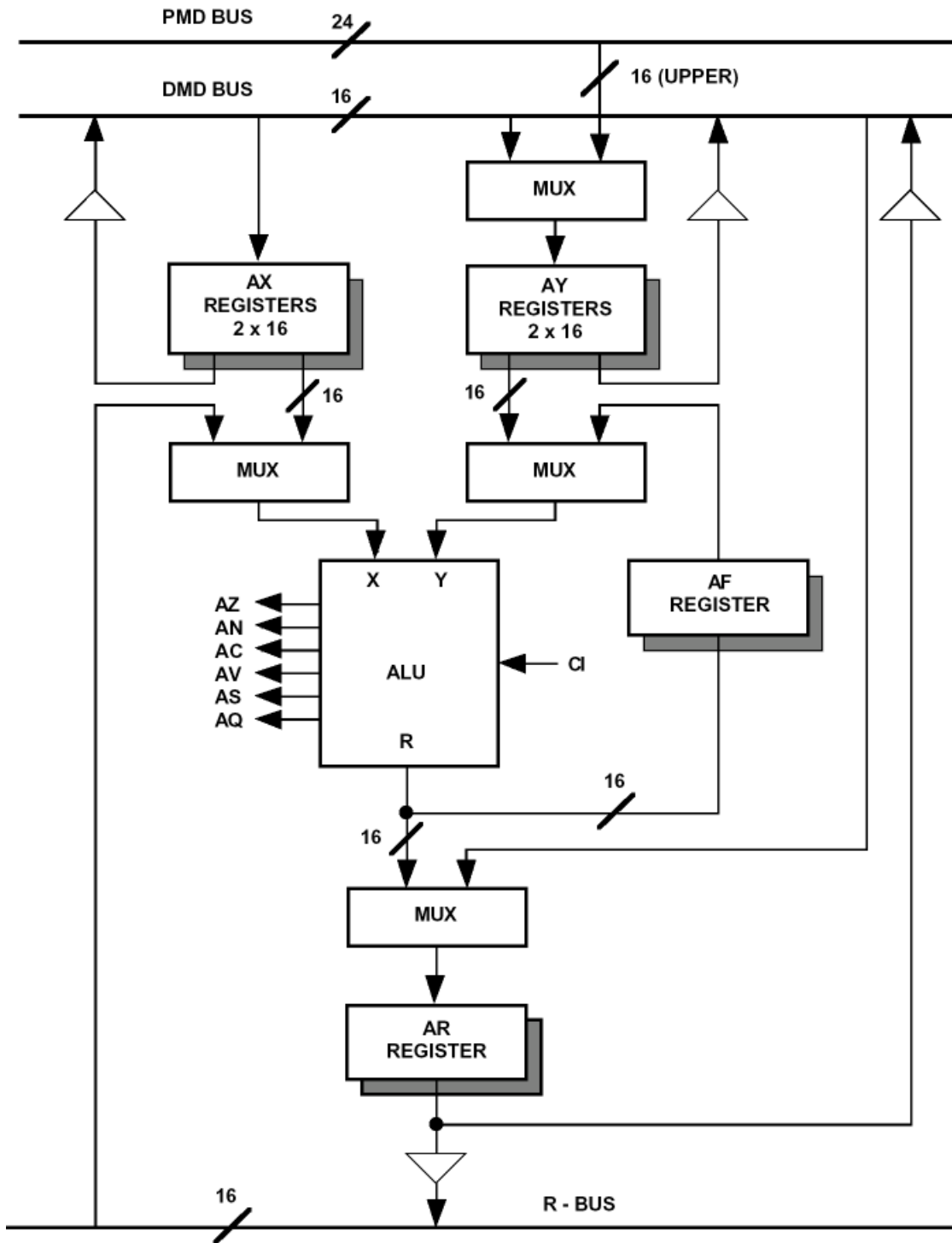


Figure 4. ALU Unit Architectue [2]

Table 6. Input/Output Parameters of the ALU

Parameter	Type	Meaning
<i>DMDBUSoutput</i>	Num-> Word16	The output of the bus DMD.
<i>AX0input</i>	Num-> Word16	The input of the register AX0.
<i>AX1input</i>	Num-> Word16	The input of the register AX1.
<i>AY0input</i>	Num-> Word16	The input of the register AY0.
<i>AY1input</i>	Num-> Word16	The input of the register AY1.
<i>ARoutput</i>	Num-> Word16	The output of the register AR.
<i>AFoutput</i>	Num-> Word16	The output of the register AF.
<i>AZ</i>	Num-> bool	True if the result of the ALU is null.
<i>AN</i>	Num-> bool	True if the result of the ALU is negative.
<i>AC</i>	Num-> bool	Carry bit of the operations of the ALU.
<i>ALUMUXXinput</i>	Num-> bool	The parameter of selection between the two possible inputs of the multiplexer that gives the input of the register AX.
<i>ALUMUXYinput</i>	Num-> bool	The parameter of selection between the two possible inputs of the multiplexer that gives the input of the register AY.
<i>ALUMUXARinput</i>	Num-> bool	The parameter of selection between the two possible inputs of the multiplexer that gives the input of the register AR.
<i>Axinputselect</i>	Num-> bool	The parameter of selection between the two sub-registers AX0 and AX1.
<i>Ayinputselect</i>	Num-> bool	The parameter of selection between the two sub-registers AY0 and AY1.
<i>FunctionCode</i>	Num-> Num	The function to be executed by the ALU.

An important remark is that all the types used within the ALU are the same (Word16). Therefore, it is possible to use a single abstract type to represent all parameters of this unit. This is very important because it allows us to do a generic proofs that are independent from the sizes of the buses or the registers. This way our proofs are true even for systems considering data types either than Word16.

5.3.2. Functional Specification of the ALU

In the functional specification of the ALU we tried to represent this unit in a simple and optimized way. Our representation is composed of three parts: the selection of the inputs, the selection of the appropriate function and the storage of the result.

Inputs Determination

The inputs of the base unit of the ALU depend on the multiplexers at the input of the registers AX and AY and on the two inputs X and Y. In our representation we selected the value of input of this unit directly from the values of control of the multiplexers. These commands are in our case: *ALUMUXXinput*, *ALUMUXYinput*, *AXinputselect* and *AYinputselect*.

Selection of the Appropriate Function

The function that will be executed by the ALU comes from the instruction by means of the Program Sequencer. The parameter *FunctionCode* identify the actual operation. The possible operations are defined in the Table 7.

Data Storage

At the end of the operation of the ALU, the result has to be stored either in the register AF only or in both registers AF and AR. This result can be either written in the program or data memories by means of the R bus.

The functional specification of the ALU as written in the HOL environment is:

```
ALU_spec = (^ALU_spec(DMD_BUS_output, AX0_input, AX1_input, AY0_input, AY1_input,
AR_output, AF_input, AF_output, AZ, AN, AC, AV, AS, ALU_MUX_Xinput, ALU_MUX_Yinput,
ALU_MUX_ARinput, AX_input_select, AY_input_select) =
(
! t.
  (? X AY_output.
    (
      (ALU_MUX_Xinput = T) => ( X t = AR_output t)
      | ((AX_input_select = T) => (X t = AX0_input (t-1) )
      | (X t = AX1_input (t-1) )
    )
  )
  ^ (
    (AY_input_select = T) => (AY_output t = AY0_input (t-1))
    | (AY_output t = AY1_input (t-1))
  )
  ^ (? Y R.
    ( (ALU_MUX_Yinput = T) => ( Y t = AY_output t) | (Y t = AF_output t) )
    ^ ALU_unit(X, Y, R, AZ, AN, AC, AV, AS)
    ^ ((ALU_MUX_ARinput = T) => (AR_output t = R (t-1))
    | (AR_output t = DMD_BUS_output (t-1))
  )
  ^ ( AF_output t = R (t-1) )
));
```

5.3.3. Experimental Results

To write the specification of the implementation of the ALU we made several iterations. We started by the basic ALU unit that we added components one by one until we construct the full unit. The number of iterations and the CPU time per iteration in the construction of the specifications of the units are summarized in Table 8. 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* \Rightarrow *Processor Specification Unit*”. The experiments were done on a Pentium II PC with 64 MB memory.

Table 7. The ALU Functions

Function	Type	Operation
AddOneToWord16	Word16 -> Word16	Increment a Word16 element.
AddWord16WithCI	Word16 -> Word16 -> Word16 -> Word16	Addition of two elements of type Word16 with carry consideration.
AddWord16	Word16 -> Word16 -> Word16	Addition of two elements of type Word16
NotWord16	Word16 -> Word16	The boolean negation of Word16.
MinusWord16	Word16 -> Word16	The negative value Word16.
SubWord16WithCI	Word16 -> Word16 -> Word16 -> Word16	Subtract two elements of type Word16 with carry consideration.
SubWord16	Word16 -> Word16 -> Word16	Subtract two elements of type Word16.
SubOneFromWord16	Word16 -> Word16	Decrement one Word16 element.
AndWord16	Word16 -> Word16 -> Word16	Logic AND between two Word16 elements.
OrWord16	Word16 -> Word16 -> Word16	Logic OR between two Word16 elements.
XorWord16	Word16 -> Word16 -> Word16	Logic XOR between two Word16 elements.
AbsWord16	Word16 -> Word16	Absolute value of a Word16 elements.

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.

Table 8. Iterative Specification Results

Unit	Number of Iterations	CPU Time
<i>ALU</i>	7	0.5 s/iteration
<i>MAC</i>	9	2.2 s/iteration
<i>Shifter</i>	11	3.7 s/iteration
<i>DAG</i>	8	3.1 s/iteration
<i>Program Sequencer</i>	18	4.8 s/iteration

6. Formal Verification

6.1. Verification Plan

In our study we started by verifying the following implication:

$$\text{Hardware specification of the unit} \Rightarrow \text{Behavioral specification of the unit}$$

The above specifications were described in the previous chapter. Our object is to manipulate a simplified representations of the processors' units. This way the verification of the instructions will be simplified.

Our approach in the simplification of the representation of the units is structured. In other terms, we did not attack directly the global proof but we started by simplifying it. The following scheme is an explanation of this idea:

Principal Proof

$$\text{Hardware specification of the unit} \Rightarrow \text{Behavioral specification of the unit}$$

Sub-proofs

$$\text{Step 1:} \quad \text{Hardware specification of the unit} \Rightarrow 1^{\text{st}} \text{ intermediate specification of the unit}$$

$$\text{Step 2:} \quad 1^{\text{st}} \text{ intermediate specification of the unit} \Rightarrow 2^{\text{nd}} \text{ intermediate specification of the unit}$$

.

.

.

$$\text{Step n:} \quad (n-1)^{\text{th}} \text{ intermediate specification of the unit} \Rightarrow n^{\text{th}} \text{ intermediate specification of the unit}$$

$$\text{Final step:} \quad n^{\text{th}} \text{ intermediate specification of the unit} \Rightarrow \text{Behavioral specification of the unit}$$

The intermediate specifications are hybrid representations of the units. In other words, some of the hardware components of the unit are replaced by their behavioral specifications. This way the new specification contains a hardware part and a functional part. This way allow us to simplify the verification task. The idea behind that is that the units of the processor are formed by sub-units that can be specified each one alone.

6.2. Simplification of the Units

In place of using the hardware specifications of the units in writing the hardware specification of the processor, we used the behavioral specifications of these units. Therefore, the first step for us was to make the following proof for each unit of the processor:

$$\text{Hardware specification of the unit} \Rightarrow \text{Behavioral specification of the unit}$$

For both the behavioral and the hardware specifications of the ALU were presented in the previous example. Our goal in this case is to make the proof that:

$$\text{Hardware specification of the ALU} \Rightarrow \text{Behavioral specification of the ALU.}$$

To solve this problem, we decomposed the principle proof into many sub-proofs. This is done by considering sub-systems of the ALU and by adding components to that sub-system until getting the full representation of the ALU (see Table 9 and Figure 5).

Table 9. Steps of the Proof of the ALU

Step	Proof	
	<i>Hardware Specification</i>	\Rightarrow <i>Behavioral Specification</i>
1	The multiplexer at the output of the register AY. The base unit of the ALU.	\Rightarrow Sub-system 1 of the ALU.
2	Sub-system 1 of the ALU. The multiplexer at the input of the register AY. The register AY. The register AF.	\Rightarrow Sub-system 2 of the ALU.
3	Sub-system 2 of the ALU. The multiplexer at the input of the register AX. The register AX.	\Rightarrow Sub-system 3 of the ALU.
4	Sub-system 3 of the ALU. The multiplexer at the input of the register AR. The register AR.	\Rightarrow Functional Specification of the ALU.

Table 9 describes the steps that we used to make the proof of the implication between the hardware specification of the ALU and its behavioral specification. In the first step, we considered a sub-system of the ALU containing the base unit of the arithmetic and logic operations and the selection of the first input Y. Then in the second step, we considered the registers AY and AR. By adding to each iteration a new set of components we ended by representing all ALU unit.

The ALU specification construction is presented in Figure 5. We start by specifying the basic ALU unit only. Then progressively we add the other components (registers, multiplexers...) one by one. In each iteration we eliminate some internal signals from the ALU specification. For example, in “iteration 2” the new specification of the ALU will not contain any connection between the register **AF** and the basic ALU unit. This register will be represented as a parameter of the new specification and not as a component. In the final iteration, we will get a specification of the unit describing this unit as single box and not as a connection of many separate boxes.

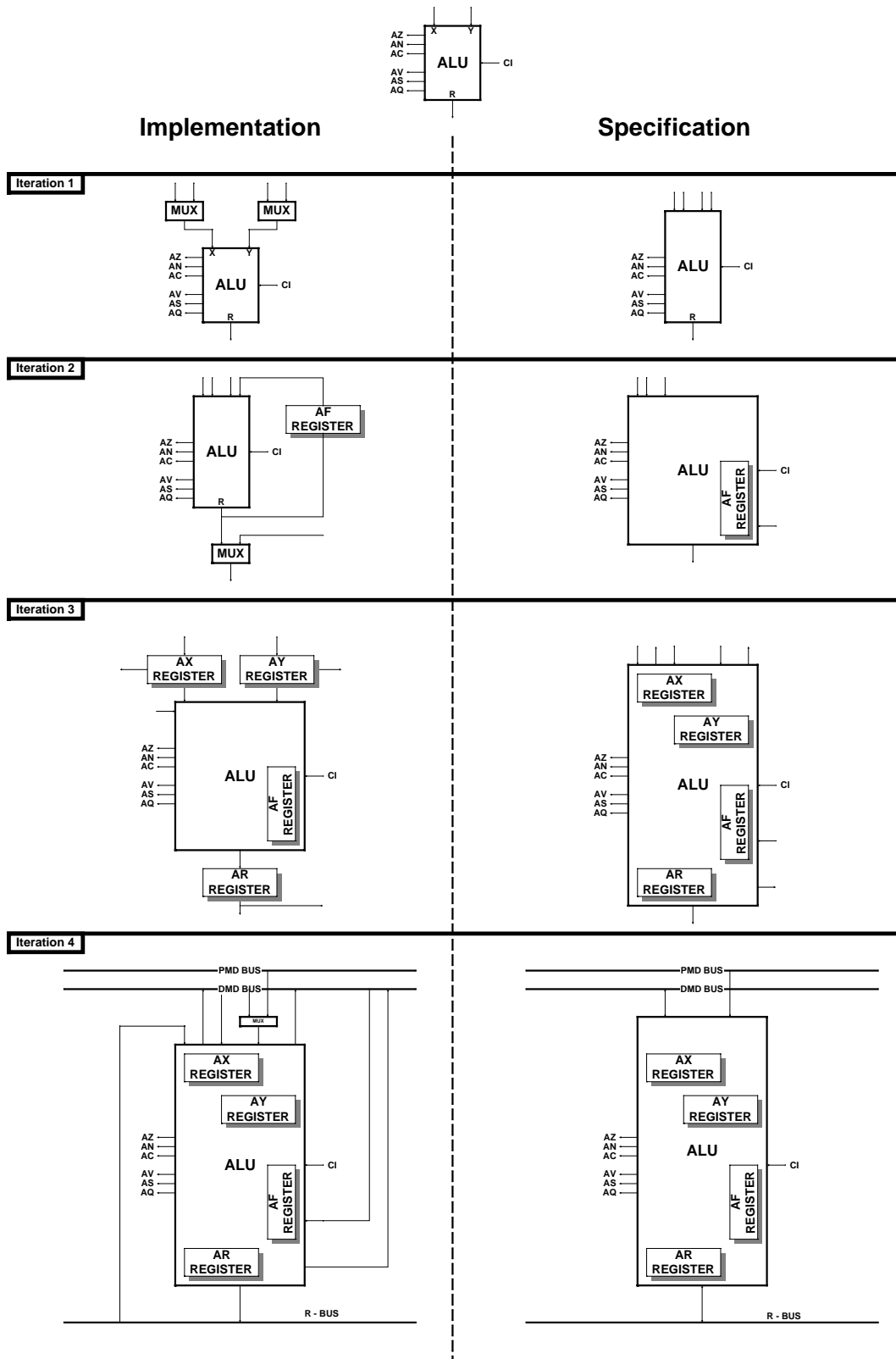


Figure 5. ALU Unit Specification

Our comprehension of the way the ALU is working allowed us to decompose the implication between the hardware of this unit and its behavior. This has given us a simple proofs to do. The solution for these proofs was relatively simple to find and their duration of them did not exceed few seconds. A sample of proof that we made is the following:

```

g`
! DMD_BUS_output AX0_input AX1_input AY0_input AY1_input
  AR_output AF_output
  AZ AN AC AV AS
  ALU_MUX_Xinput ALU_MUX_Yinput ALU_MUX_ARinput
  AX_input_select AY_input_select .
ALU_second_sub_imp(
  DMD_BUS_output,
  AX0_input, AX1_input, AY0_input, AY1_input,
  AR_output, AF_output,
  AZ, AN, AC, AV, AS,
  ALU_MUX_Xinput, ALU_MUX_Yinput, ALU_MUX_ARinput,
  AX_input_select, AY_input_select
)
==>
  ALU_second_sub_spec(
    DMD_BUS_output,
    AX0_input, AX1_input, AY0_input, AY1_input,
    AR_output, AF_input, AF_output,
    AZ, AN, AC, AV, AS,
    ALU_MUX_Xinput, ALU_MUX_Yinput, ALU_MUX_ARinput,
    AX_input_select, AY_input_select
  )
;

```

6.2.1. Goal Setup

Our goal in this case is to make the proof of the following:

$$ALU_second_sub_imp \Rightarrow ALU_second_sub_spec$$

where:

ALU_second_sub_imp: the implementation of the second sub-system de of the ALU.

ALU_second_sub_spec: the specification of the second sub-system de of the ALU.

6.2.2. The Proof

The proof is done in four steps:

1. Rewriting the specifications of the components of the two subsystems *ALU_second_sub_imp* and *ALU_second_sub_spec*.

```

REWRITE_TAC[ ALU_second_sub_imp, ALU_second_sub_spec,
  ALU_first_sub_spec, ALU_AX_reg, ALU_AY_reg, ALU_MUX]

```

2. Consider all possible Boolean values of the parameters: *ALU_MUX_Xinput*, *ALU_MUX_ARinput*, *ALU_MUX_Xinput*, *AX_input_select*, *AY_input_select* and *banc_select*.

```

MAP EVERY BOOL_CASES_TAC [ `ALU_MUX_Xinput : bool`,
  `ALU_MUX_Yinput : bool`,
  `ALU_MUX_ARinput : bool`,

```

```

    ``AX_input_select : bool``,
    ``AY_input_select : bool``,
    ``banc_select : bool``]

```

3. Decompose the principal goal into small sub-goals:

```
STRIP_TAC
```

4. Use algorithms of reductions to solve small sub-goals:

```
bossLib.PROVE_TAC []
```

The complete proof is:

```

e (
  REWRITE_TAC[ALU_second_sub_imp, ALU_second_sub_spec,
    ALU_first_sub_spec, ALU_AX_reg, ALU_AY_reg, ALU_MUX]
  THEN
    REPEAT GEN_TAC
  THEN
    MAP EVERY BOOL_CASES_TAC [``ALU_MUX_Xinput : bool``,
      ``ALU_MUX_Yinput : bool``,
      ``ALU_MUX_ARinput : bool``,
      ``AX_input_select : bool``,
      ``AY_input_select : bool``,
      ``banc_select : bool``]
  THEN
    REPEAT GEN_TAC
  THEN
    STRIP_TAC
  THEN
    ASM_REWRITE_TAC[ALU_second_sub_imp]
  THEN
    STRIP_TAC
  THEN
    bossLib.RW_TAC bossLib.arith_ss []
  THEN
    bossLib.PROVE_TAC []
);

```

The HOL System Response:

The response of the HOL system to the last proof is:

```

Initial goal proved.
|- !DMD_BUS_output AX0_input AX1_input AY0_input AY1_input AR_output
  AF_output AZ AN AC AV AS ALU_MUX_Xinput ALU_MUX_Yinput
  ALU_MUX_ARinput AX_input_select AY_input_select.

```

```

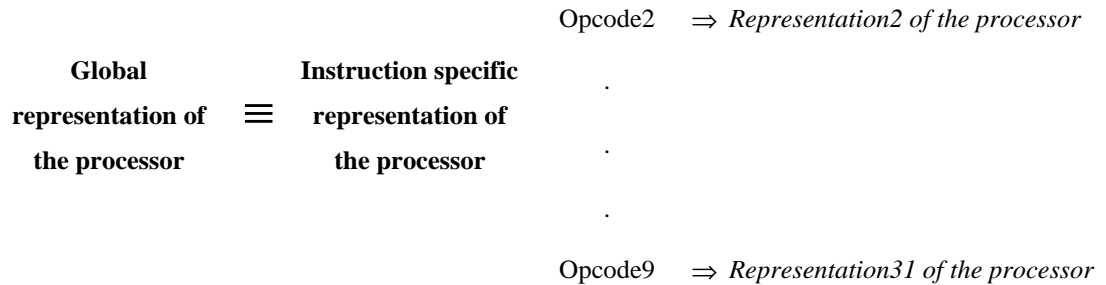
ALU_second_sub_imp
(DMD_BUS_output,AX0_input,AX1_input,AY0_input,AY1_input,AR_output,
AF_output,AZ,AN,AC,AV,AS,ALU_MUX_Xinput,ALU_MUX_Yinput,
ALU_MUX_ARinput,AX_input_select,AY_input_select) ==>
ALU_second_sub_spec
(DMD_BUS_output,AX0_input,AX1_input,AY0_input,AY1_input,AR_output,
AF_input,AF_output,AZ,AN,AC,AV,AS,ALU_MUX_Xinput,
ALU_MUX_Yinput, ALU_MUX_ARinput,AX_input_select,AY_input_select)
: Goalstackpure.goalstack

```

Therefore, the goal was proved and the system declare it as a new theorem!

6.3. Simplification of the Processor

In our manipulation of the proofs we used a hierarchical approach to deal with the complexity of the system. In other terms, we did not considered a unique representation of the processor but a set of representations. Each of these representations is related to one of the opcodes. In fact, for each opcode only a subset of units of the processor are concerned by the instruction.



Global Representation of the Processor

The representation contains a full description of the processor. It contains all the inputs and outputs.

Instruction Specific Representation of the Processor

The representation takes into account the knowledge of the way the processor is working. In fact, we only take in our account the units related the instruction being executed. For example, if an instruction concerns only the ALU, there is no need to add the specification of the MAC. This way we are looking to the system differently depending on each instruction.

6.4. Verification of the Instructions

To deal with the large number of parameters of the specification of the processor we made the instruction set verification in two steps:

Defining an instruction dependent representation of the processor. This is a simplified representation of the processor containing only the components involved in the instruction.

Making the global proof. Using the proofs of the last step, it is relatively simple to consider the full processor specification in a global proof.

This instruction allows the immediate writing of data in the memory. The selection between the two address generators is the boolean parameter **G**. The address where data is

to be written is given by the content of the registers **I** and **M**, which specify the registers *Index* and the register *Modify* of the selected address generator.

The representation of the processor in this case is described in Figure 6. The first states concern reading the instruction and decoding the operation to be done. The steps related to the instruction are summarized in Table 11. Our representation of the processor in the case of this instruction consider only the concerned components and registers.

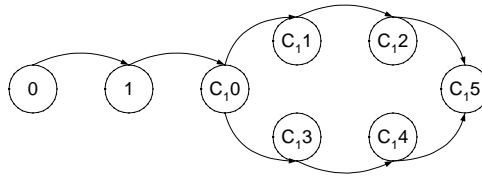


Figure 6. Behavioral Representation of the Processor

6.4.1. Hardware Specification

Program Sequencer (which corresponds to the states: 1, C₁₀ and C₁₃ of Figure 6) where:

Input parameters: the control of selection of the appropriate address generator **G**, controls of the registers **I** and **M** which specify the memory address.

Output parameters: the memory control parameters.

Address generator (which corresponds to the states: C₁₂ and C₁₄ depending on the used address generator) where:

Input parameters: the controls of the registers **I** and **M** which specify the memory address.

Output parameters: the content of the registers **I** and **M** that will be used to access the external memory.

Memory access (which corresponds to the state C₁₂) where:

Input parameters: memory access control parameters (specified by the program sequencer), data to write in the memory.

Table 10. Steps of the Instruction Execution

State	Description
0	Loading the instruction.
1	Decoding the operation to be done. The value of the operation is stored in parameter FC (FunctionCode).
C ₁ 0	Decoding the boolean parameter G which defines the selection between the two possible address generators.
C ₁ 1	Generate the controls for reading form the first generator.
C ₁ 2	Reading the content of the registers I and M of the first generator.
C ₁ 3	Generate the controls for reading from the second generator.
C ₁ 4	Reading the content of the registers I and M of the second generator.
C ₁ 5	Storing data according to the values of the registers I and M .

6.4.2. Behavioral Specification

This instruction will store the data specified by the instruction to the memory location defined by the content of the registers **I** and **M** of the selected address generator (selected by the parameter **G**).

6.4.3. Verification

In this phase we made the proof of the implication between the implementation and the behavioral specifications of the processor. We started by dividing the principle proof. In fact, at first we considered the decoding of the appropriate address generator that will be used to read the address from, then we added the module of reading the data from that address generator and finally the storage of the data in the memory.

6.4.4. Experimental Results

Table 11 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 all the instruction set).

Table 11. Instruction Set Verification Results

Class of Instruction	Average CPU Times of the Proof
<i>Class1</i>	2.5 s
<i>Class2</i>	2.8 s
<i>Class3</i>	2.1 s
<i>Class4</i>	4.1 s
<i>Class5</i>	3.8 s

7. Results and Concluding Remarks

Without making a simplification of the units specifications, it was impossible to make the verification of the ADSP-2100 processor using HOL. In fact, the specification of such a complex system took more than 100 pages. The simple task of reading that specification is difficult.

We tried to divide the proofs as much as we can. This way we got simpler proofs that took just few seconds to be solved by the HOL system. On the other hand, finding the best proof was also simple since the sub-proofs were short and relatively at the same level of abstraction.

We detected some weakness in the HOL system. Some of them are:

- Some features of other type systems such as subtypes and dependent types are not supported.
- The insistence on resolving proofs into simple primitive inferences can make HOL very slow.
- Many common proof strategies have not been automated by derived rules, necessitating low-level guidance by the user.

No one method or tool can serve all purposes [6]. We need to support all different kinds. From past experience, we have learned what kinds can have the most impact. To be attractive to practitioners, methods and tools should satisfy the following criteria. We realize that some of these criteria are ideals, but they are still good to strive for.

- Methods and tools should provide significant benefits almost as soon as people begin to use them. It should be possible to amortize the cost of a method or tool over many uses. For example, it should be possible to derive benefits from a single specification at several points in a program's life cycle: in design analysis, code optimization, test case generation, and regression testing.
- Methods and tools should work in conjunction with each other and with common programming languages and techniques. Developers should not have to learn a new methodology completely to begin receiving benefits. The use of tools for formal methods should be integrated with that of tools for traditional software development, e.g., compilers and simulators.
- Notations and tools should provide a starting point for writing formal specifications for developers who would not otherwise write them. The knowledge of formal specifications needed to start realizing benefits should be minimal.
- Methods and tools should support evolutionary system development by allowing partial specification and analysis of selected aspects of a system.

Hybrid Methods

Given that no one formal method is likely to be suitable for describing and analyzing every aspect of a complex system, a practical approach is to use different methods in combination. When combining methods it is important to consider both:

1. Finding a suitable style for using different methods together; and
2. Finding a suitable meaning for using different methods together.

It is possible to combine the theorem proving and the simulation methods as shown in Figure 7.

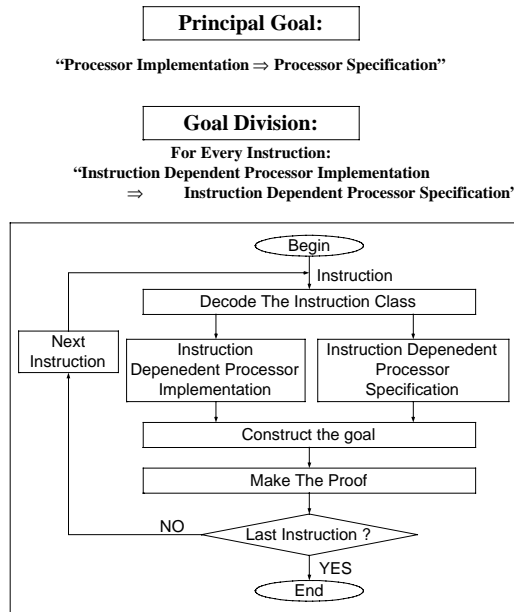


Figure 7. Hybrid Method : “Theorem Proving & Simulation”

Commercial pressure to produce higher-quality software is always increasing. Formal methods have already demonstrated success in specifying commercial and safety-critical software; and in verifying protocol standards and hardware designs. In the future, we expect that the role of formal methods in the entire system development process will increase, especially as the tools and methods successful in one domain can be carried over to others domains. Progress, however, will strongly depend on continued support for basic research on new specification languages and new verification techniques.

Ideally, system developers would all be trained sufficiently well that they would not even think that they are using a formal method or tool. They would routinely use the mathematics underlying the notation of a formal specification language as simply a means of communicating ideas to others on their team or of documenting their own design decisions. They would routinely use tools like model and proof checkers with as much ease as they use compilers. Therefore, notations and tools have to be accessible to non-experts.

References

- [1] P. Andrews. An Introduction to Higher Order Logic: To Truth through Proof. *Academic Press*, New York, 1986.
- [2] Applications Engineering Staff of Analog Devices, DSP Division. *Digital Signal Processing Applications Using the ADSP-2100 Family*. Prentice Hall, Englewood Cliffs, NJ 07632, 1996.
- [3] Applications Engineering Staff of Analog Devices, DSP Division. *Data sheet of the ADSP-2100 Family DSP - Microcomputers - ADSP-21xx*, 1996.
- [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. Kaufman, and J.S. Moore, Heavy inference : Theorems about commercial microprocessors, *Formal Methods in Computer-Aided Design*, Springer-Verlag, November 1996.
- [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] C. Kern and M. Greenstreet, "Formal Verification in Hardware Design: A Survey", *ACM Transactions on Design Automation of E. Systems*, Vol. 4, April 1999, pp. 123-193.
- [10] T.F. Melham, *Higher Order Logic and Hardware Verification*, Cambridge Tracts in Theoretical Computer Science 31, Cambridge University Press, 1993.
- [11] R. Milner and. M. Tofte. The definition of Standard ML. *The MIT Press, Cambridge, Massachusetts and London, England*, 1991.
- [12] C.H. Pygott. *Verification of VIPERS's ALU*. Technical report, Divisional Memo (Draft), the Royal Signals and Radar Establishment, 1991.
- [13] Stavridou, T. F. Melham and R. T. Boute. : *Theorem Provers in Circuit Design*. Proceedings of the IFIP WG10.2 International Conference on Theorem Provers in Circuit Design: Theory, Practice, and Experience, Nijmegen, The Netherlands, 22-24 June 1992.
- [14] W. S. Steven, The Scientist and Engineer's Guide to Digital Signal Processing, Second Edition. *California Technical Publishing*, 1999.
- [15] S. 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.