Combining Symbolic Evaluation and Object Oriented Approach for Verifying Processor–like Architectures at the RT–level

Jacques CHAZARAIN Hélène COLLAVIZZA

{jmch,helen}@mimosa.unice.fr
Laboratoire d'Informatique I3S
Université de Nice Sophia Antipolis
CNRS. URA 1376
250 Av. Albert Einstein
06560 Valbonne, FRANCE

Abstract

We consider in this paper the correctness of the translation of an assembly language instruction into a sequence of micro-instructions described at the RT-level. We develop a new method which combines the extensibility and flexibility of object oriented programming paradigm and the efficiency of a specialized computer algebra system. An object oriented programming is naturally well-adapted to express the behaviour associated to each category of components found in hardware description languages. We see each component of the processor as an object (in Common Lisp Object System: CLOS).

The other important feature in order to get a general (and mainly automatic) proof is to be able to execute micro-instructions with symbolic operands. For that purpose, we have implemented a symbolic evaluator which can deal with the operations used at the RT level. The final step of the proof is reduced to the equality of symbolic expressions. Since the memory addressing introduces expressions which need a calculus with operators on bit vectors, we built a specialized, but extensible, computer algebra system for proving the equality of such expressions.

Given the semantics of each level, we succeeded to automatically prove the correctness of the translation into micro-instructions of each assembly language instruction for a given processor.

1 Introduction

It is now widely recognized that simulation could not guarantee a 0-default design, and that formal verification methods must be integrated in the design process (see [7] for a survey of these methods). The concept of "Design for Verifiability" [19] is gaining acceptance in the CAD research community [9], and efficient automatic formal verification tools, such as tautology checkers are already used by designers [12]. In this framework, many works address the problem of formally verifying processor-like circuits. Such a verification is a large problem, so it is usual to consider several levels of description in order to reduce the problem to the proof of the coherence between each specification at

adjacent levels. The levels usually considered are (from the less to the most abstract level) : silicium level, switch level, logic level, register transfer level, micro-programming level, assembly level.

The most significant works were the verification with the Boyer & Moore theorem prover of the FM8501 processor [15] and of the VIPER processor with the HOL prover [6]. Inspired by the pioneer work of Gordon [14], these works show the clearness of using several abstraction levels, and set the foundation of the formal algebra required. In [16] and [23], abstraction mechanisms were used to make a more generic verification of the processor, independently of the processor size. A functional approach was used for proving the assembly instructions of the MTI processor with respect to its micro-program set [10] and to verify a pipelined processor [3]. The notion of observable state and rewriting techniques were used to verify the microcode of a processor-like circuit [18].

The verification aspect we consider here is the correctness of the translation of an assembly language instruction into a sequence of micro-instructions described at the RT-level. The many interesting results already obtained for this level often used some methods dedicated to a particular circuit, or in the opposite, are based on general logical tools not specially crafted for the proof on such objects. In this paper, we developp a new method which combines the extensibility and flexibility of object oriented programming paradigm and the efficiency of a specialized computer algebra system .

In hardware description languages, each component has a well- defined type that charaterizes its physical and temporal behaviour. An object oriented programming is naturally well-adapted to express the behaviour associated to each category of objects. We see each component of the processor as objects (in Common Lisp Object System CLOS) : register, wire, bus, pins, memory, stack, ... All these objects carry some information : value, type, mode ... For instance a bus can have a type 8 bits, mode write or read , ... Each kind of objects has methods to interact, for instance, the communication protocol between the main memory and the data bus is based on active value attached to some control pins such as VMA (valid memory address).

The other important feature in order to get a general proof is to be able to execute micro-instructions with symbolic operands, because the use of real values for the data means only simulation but not formal proof. For that purpose we have built a symbolic evaluator which can deal with the operations used at the RT level. Finally, the proof is reduced to the equality of symbolic expressions. The mode of addressing and the decoding of memory address introduce expressions which need a calculus with operators on bit vectors. So we built a specialized, but extensible, computer algebra system for proving the equality of such expressions.

Given the semantics of each level, we have automatically performed the correctness proof of the translation into micro-instructions of each assembly language instruction for a given processor.

Plan of the paper

2 Proof methodology
3 A simple micro-processor architecture
4 Object oriented model
5 Symbolic evaluation
6 Semantic specification and proof
7 Conclusion
References

2 Proof methodology

Our aim is to prove that the translation between assembly instruction level and micro-instruction level is coherent with the semantics of each level.

At each level, we define the semantic of an instruction in the most natural manner: by describing the relation between the processor register and memory states before and after the execution of an instruction of the considered level. We use an object oriented representation for the processor and the memory states in order to have a more extensible approach.

At each time, we compute the new symbolic value of the micro- processor state, using our symbolic evaluator. This differs from the functional point of view advocated in [1] where the functional expressions are kept as long as possible in a not- evaluated form. Since we completely execute symbolically the state modifications, we do not need to consider the intermediate level of micro-sequences as in [11].

We consider the following description levels:

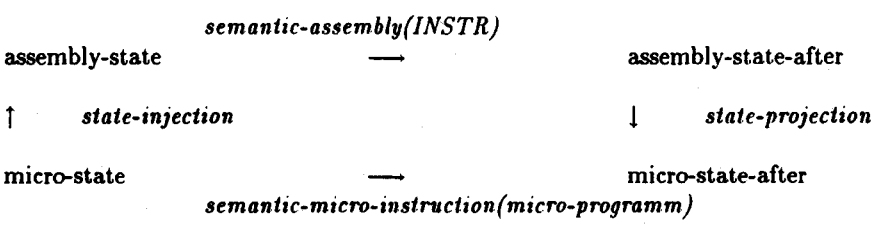
1. top-level (processor cycle):
   Specification of the instruction fetch and the interrupt sequence. The top-state includes the main memory (which contains the code of the instruction to be fetched), the PC and the instruction register IR

2. assembly-level (assembly language):
   Specification of the instruction execution on the assembly- state of visible variables to the assembly language programmer

3. micro-level (micro-instruction level):
   Specification of the transfers involved by a micro-instruction. The micro-state includes the complete processor architecture; in particular, the memory access pins are considered

The processor state at a given level consists of the visible registers and of the main memory. We shall describe it in more detail with an example later. The states are more and more detailed, when the level increases, so there is a natural injection of state-i into state-i+1 and a natural projection in the opposite.

For each instruction INSTR at assembly level we are given its translation into a micro-instruction sequence. The main correctness problem is to prove that the following diagram is commutative:



More explicitly, we must prove that "assembly-state-after" is the same if we execute symbolically the direct arrow Semantic- assembly(INSTR) or if we use the composition of arrows :

state-projection o Semantic-micro-instruction(micro- programm) o state-injection

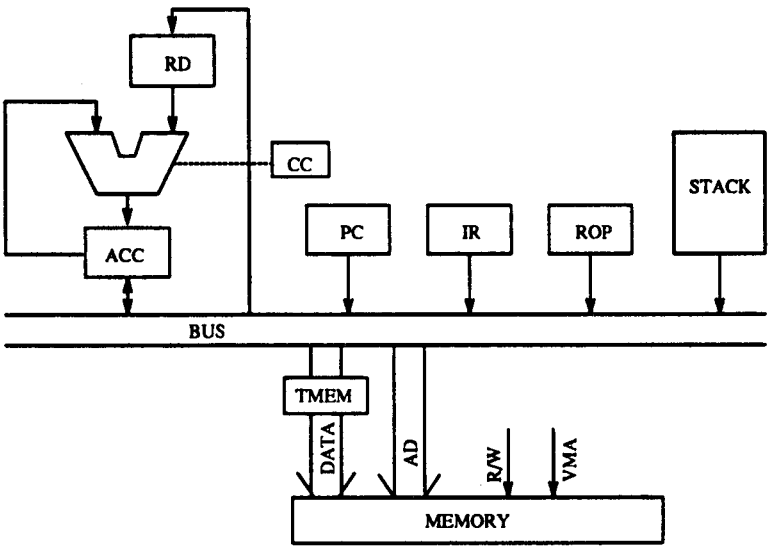
In order to be more specific, we shall explain our method with a simple processor model.

3 A simple micro-processor architecture

We take as example a simple processor, in order to introduce our principles of specification and proof. But the complexity of this processor could be increased with all the qualitative characteristics of an actual processor (see [8]). Its architecture includes:

- a main memory MEMORY
- an internal stack STACK used for subroutine call
- an accumulator ACC
- a program counter PC
- a Condition Code CC, composed of the classical four flags : N, Z, C, O
- the instruction register IR
- an operand register ROP which stores the value of the second instruction word
- a data register RD connected to the ALU input which contains the value of the operand
- a memory buffer called TMEM which holds the value read/written in the memory

The interface between the processor and the main memory uses the bi-directional DATA pins, the AD pins, the Valid Memory Address signal VMA and the R/W signal.



This processor includes all the usual instruction classes, except interrupt instructions. Only two addressing modes are available: IM (for IMmediate) and ME (for direct MEmory). The instruction format is one or two 8-bit words; the first word is the instruction code while the second contains the immediate value or the memory address.

Now we are able to introduce our object oriented setting.

4 object oriented model of micro-processor

We associate to each category of processor component a class (in the object programming way). We do not enter here into the syntax detail of the CLOS language we use [22], but we indicate the kind of classes with their main slots. All of the following classes will be considered as son of the class "processor-component".

4.1 The class description

We give the main classes with their slots, generally used to represent processor components:

Class register have slots : name , type , value

Class wire have slots: name, value , connected-to

Class memory have slots : memory-content , DATA , AD ,
control pins such: VMA , R/W

Class bus have slots: name , type , connected-register

Class stack have slots: type , top , stack[1], stack[2], stack[3]

Class CC-register have slots: N, Z, C, O

The slot type of a component means for instance if it is 8-bits or even a symbolic value if we do not need it in the proof.

Some methods are associated with these classes, to take care of the semantic of the component. For the memory, there is a method "memory-content" which takes an address argument and gives the corresponding content. The stack has "push" and "pop" methods which return the new stack and "top" method which returns the top stack value.

The global state of our simple processor is also represented at each level by a class :

Class processor-top-state have slots : MEMORY , PC , IR,

Class processor-assembly-state have slots : MEMORY, PC, CC, ACC, STACK

Class processor-micro-state have slots : MEMORY , PC, CC, ACC, STACK, ROP, BUS, TMEN,

In all these classes, MEMORY is also an object of class memory with slots: memory-content, DATA, AD, VMA, R/W

For any processor, a part of the slots of class memory are used to enable the memory/processor information exchanges. In our simple processor, the communication protocol between the CPU and the memory is governed by the pins VMA and R/W, and it is some specific sequence of values of this pins that activates the memory in read or write mode. The object oriented approach with the daemons, or active value, mechanism gives a nice way to model such protocol. In short, a slot is an active value if a change of value in the slot induces the execution of an action, here reading or writing in the memory. This point will be detailed in 6.3.1.

4.2 Operational Semantic of transfer description

The description of the processor at each abstraction level will be made using a formal model, with a syntax close to processor books description. For example, we indicate that

the effect of an instruction ADD is to change the content of accumulator register ACC into the value of "ACC +8bits RD" by the usual notation :

$$\text{ACC} \rightarrow \text{ACC} + 8\text{bits RD}$$

where RD is the data register and +8bits denotes the add operation on 8-bit vectors.

The general syntax is $X \rightarrow EXP$ where X is a register or wire name and EXP is a symbolic expression. A symbolic expression is a term in the formal sense. A term can be considered as a tree whose nodes are labelled with operation or function names and the leaves with variable or constant names.

The semantic meaning of $X \rightarrow EXP$ is to change the value slot of X into the result of the ***symbolic evaluation*** of EXP. So we have an easy translation of processor specification into CLOS actions driven by the class of the objects. In fact, this is exactly the operational semantic approach [21] where for each instruction we indicate the change in the state by a rule as

$$\frac{state \vdash_{eval\,symbolic} EXP : v}{state \vdash X \rightarrow EXP : state[X \leftarrow v]}$$

where $state[X \leftarrow v]$ means substitution of value v in the slot X of the state. The state is an environment that gives the links between names and values, and this rule written in the natural semantic style [17] means that we replace the value of X by the the symbolic value of EXP.

In general, an instruction induces value changes in several registers or wires, so we need to describe the sequencing of the actions. Implicitly, transfers are made in parallel, the sequencing must be explicitly expressed with ";" .

For example, when we give the semantic of the micro-instruction of a first cycle of reading by:

BUS → PC
VMA → 1
R/W →1 ;
AD → BUS
PC → inc(PC)

that means the parallel actions on BUS, VMA, R/W and after the parallel actions on AD and PC (this sequencing corresponds to the two phases of the cycle. We will use in the transfer descriptions all the usual bit-vector operators, denoted with the CASCADE syntax [2]:

- concatenation : V1@ V2 ,
- truncate to the component from m to n : V[m:n]
- conversion from bit vector to integer : $.

In the functional semantic description as developped in [20, 11], the advantage is to factor as much as possible common actions. But if the factorization is a gain in some cases (processors with regular sequences of micro-instructions) it introduces a set of names and functional expressions which are not easy to automatically generate. Furthermore, the simplification process is separed from the symbolic evaluation of the functional model, and so the symbolic expressions are more difficult to simplify. In this object oriented method, the evaluation is distributed among the objects, and we do not have to introduce new function names for each kind of component. This is also an important point for the extensibility of the description of a processor architecture. In a future work we expect to be able to automatically generate the model from an abstract syntax description given by the designer (assembly level), and from a VHDL description (RT-level).

5 Symbolic evaluation

The actions associated to the operational semantic of instructions are computed by symbolic evaluations [5]. Of course, the value of a symbolic expression is not the result of the standard Lisp evaluator, but it is the result of a specialized symbolic evaluator built on the model of evaluator used in computer algebra systems (say Maple, Mathematica). That means essentially that :

- a variable which denotes a symbolic expression evaluates to this expression or to itself if it is unbound
- for composed expression $(op\ exp_1, \ldots, exp_N)$ the method of evaluation is attached with the head operator *op*. For example there is a special method for the if or for the + operators. For each operator we can also declare general property as : commutativity, or associativity, or neutral element,or distributivity,or... If there is no special method for the operator *op*, we recursively evaluate the sub-expressions $exp_1, \ldots, exp_N$ and return finally the symbolic value $(op\ \overline{exp_1}, \ldots, \overline{exp_N})$. where overline means values.

The symbolic evaluator takes care of the addressing modes in the operands of instructions at the assembly-level. All the addressing computation is done by a specialized operator Operand-Value which takes at least two arguments: an addressing mode and an operand. The symbolic evaluation methods attached with Operand-Value is the immediate translation of the addressing rules of the assembly language. Let consider the simple example of a LOAD instruction with two modes, immediate or direct, and one operand. We do not care with the concrete syntax of the assembly langage, we use an abstract syntax representation : "mnemonic mode operand" where mode is optional. The semantic of

LOAD mode operand

is given in every case by

ACC → Operand-Value (mode,operand)

It is the symbolic evaluator that will give the value of Operand- Value (mode,operand):

- if mode = immediate : it evaluates to operand
- if mode = direct : it evaluates to the memory content at the address associated with operand: (memory-content ($ operand))

The type of the bit vector (8 bits , 16bits ,...) is found in the slot type of the operand. But as advocated by J Joyce [16], we can often do the whole proof without knowing such kind of information. In our approach, it is possible to put a symbolic value in this slot as long as we do not need a special value.

Now we have the tools to give - a portion - of the semantic description at each level of our simple processor .

6 Specification and proof of our simple processor

For each level we give the semantic of the most typical instructions. It is written for a humane reader, the connection between the real specification is only a question of parsing.

6.1 Top-level

At this level, we describe the processor cycle. These instructions are not available to the assembly programmer but are important for the understanding of the architecture of the machine.

We recall that the top-state is defined with the slots : PC, IR, MEMORY.

The semantic of "execute-inst" is given at assembly-level for each instruction; fetch-inst is defined by:

```
fetch-inst: IR → memory-content($ (PC)) ;
            PC → inc(PC)
```

we have denoted "inc" the unit increment operator on bit vectors.

The fetch sequence is realized at level 3 by the following micro- instruction sequence:

```
fetch-inst :                          (* read at the address PC; load of IR *)
        mi-ad-pc; mi-read; mi-load-ir
```

6.2 Assembly-Level

We recall the slots state of the assembly-level : MEMORY, PC, CC, ACC, STACK

The addressing modes are IM (for immediate) , ME (for direct memory mode).

The addressing-methods used to compute the operand value are :

immediate Operand-Value (IM , operand) = operand

direct Operand-Value (ME , operand) = (memory- content ($ operand))

6.2.1 Semantic of ASSEMBLY-level

We only give a three typical instructions

```
ADD mode operand      (* addition *)
        ACC → ACC +8 Operand-Value ( mode , operand)
        CC → cc-fct (+8 , ACC , Operand-Value ( mode , operand))
        PC → inc(PC)
```

where the operator "cc-fct" gives the new formal flags; their values depend on the operands and on the operation.

```
STO mode operand      (* store instruction *)
        Operand-Value ( mode , operand) → ACC
        PC → inc(PC)

JSR operand           (* subroutine call *)
        STACK → push('0000 @ CC)
        PC → inc(PC);
        STACK → push(PC);
        PC → oper
```

6.2.2 Translation from assembly instructions to micro- instructions :

```
ADD IM :                    (* read at the address PC; loading of ROP *)
                            (* ALU computation*)
        mi-ad-pc; mi-read; mi-load-rop ; mi-load-rd-im ; mi- add

ADD ME :                    (* read at the address PC; loading of ROP *)
                            (* read of memory operand; loading of RD*)
                            (* ALU computation*)
        mi-ad-pc; mi-read; mi-load-rop ; mi-ad-rop; mi-read; mi-load-rd ; mi-add

STO :                       (* read at the address PC; loading of ROP *)
                            (* write of ACC at the address ROP*)
        mi-ad-pc; mi-read; mi-load-rop ; mi-load-acc-tmem; mi-ad-rop-ec; mi-ec
```

6.3 Micro-Level

The semantic description at the RT-level details the memory behaviour and the transfers on the data path.

The global system is decomposed in two parts: the processor and the main memory. The memory communication pins are visible and we model the memory/processor exchanges. Since this processor has synchronous memory accesses, we consider that the memory is seen as a slave by the processor.

The micro-state slots are :

PC, CC, ACC, PILE, IR, BUS, ROP, RD, TMEM, MEMORY

The MEMORY has slots

MEMORY-CONTENT, VMA, R/W, DATA,AD

The memory behaviour is defined by the modifications on its output pins DATA and on its internal state, that are involved by the stimuli sent by the processor on its input pins VMA, R/W, DATA. At this level, we do not consider the internal memory addressing circuit that is supposed to be correct (this may be proven at the logic level).

For this processor, the memory accesses take two processor cycles: one to set the address (VMA takes the value '1') the other to transfer the data from/to the memory. For a write access, the DATA pins must stay stable during the two cycles. This temporal behaviour implies that a read or write access is enabled according to the previous value of the memory input pins. Since these values are memorized in the memory addressing circuit that is not considered at this level, we introduce a new class of component, which keeps memory of their values at the preceding clock (the time scale is the processor clock). We call this class "processor- component-remember" and the new slot is called "past-value", the components VMA, R/W and DATA are objects of this class. In our case the read acces is commanded by the following condition

VMA = '0' and past-value(VMA) = '1' and R/W = '1' and past-value(R/W) = '1'

which triggers the action

DATA ← memory-content ($(past-value(AD)))

and the write acces is commanded by

VMA = '0' and past-value(VMA) = '1' and R/W = '0' and past-value(R/W,1) = '0'
and DATA = past-value(DATA)

which triggers the action

memory-content ($(past-value(AD))) → DATA

This is implemented in CLOS using an after method.
We give below the semantic of micro-instructions that implement ADD ME instructions:

```
mi-ad-pc :                          (* first read cycle at the address PC *)
        BUS → PC;
        VMA → '1'
        R/W → '1'
        AD → BUS;
        PC → inc(PC)

mi-read :                           (* second read cycle the read value is *)
                                    (* available on DATA; TMEM is loaded*)
        VMA → '0'
        R/W → '1 ;
        TMEM → DATA

mi-load-ir :                        (* loading of IR *)
        BUS → TMEM;
        IR → BUS

mi-load-rop :                       (* loading of ROP *)
        BUS → TMEM;
        ROP → BUS

mi-ad-rop :                         (* first read cycle at the address ROP *)
        BUS → ROP;
        VMA → '1'
        R/W → '1'
        AD → BUS

mi-load-rd :                        (* loading of RD for mode ME *)
        BUS → TMEM;
        RD → BUS

mi-load-rd-im :                     (* loading of RD for mode IM *)
        BUS → ROP;
        RD → BUS

mi-add :                            (* ALU computation for ADD *)
        ACC → ACC +8 RD
        CC → fct-cc(+8,ACC,RD)
```

6.4 Correctness proof

After loading the semantics and the translation we are ready to prove the commutativity of the diagram for each instruction with symbolic operands

It is important to notice that we use symbolic arguments, otherwise it will not be a proof but simply a simulation on a particular value. Let us explain more this point : what is the connection between proof and symbolic evaluation? This connection is based on the well known logical rule of universal quantifier introduction, which can be written in the natural deduction style [13]:

$$\frac{H \vdash p(x \leftarrow a)}{H \vdash \forall x\, p(x)}$$

where p is a formula, " x" is a free variable of p, "H" is a set of hypothesis formulas and "a" will play the role of a symbolic value.

In our case, for each slots s_j of the processor state, we have to prove equationnal formulas of the form

$$H \vdash \forall x_1 \cdots x_k \; (E_j(x_1, \cdots, x_k) = E'_j(x_1, \cdots, x_k))$$

where E_j and E_j are the expressions of sj resulting from the direct arrow or the composition of arrows, and H represents the properties of all the operators we use: arithmetical, bit vector... So it reduces to the proof of a simple equality

$$H \vdash (E_j(a_1, \cdots, a_k) = E'_j(a_1, \cdots, a_k))$$

where the a_i are new - or symbolic - variables.

Such a proof is based on the main property of the symbolic evaluator : the conservation of the semantic value between an expression and its symbolic value, taking account of the arithmetical hypothesis H on the operators . For instance, let us take a trivial example, the expression $(+\ 5\ a_2\ 10\ a_1)$ has symbolic value $(+\ 15\ a_1\ a_2)$ but these two expressions keep the same semantic value, given the associativity, commutativity and arithmetical properties of + .

When the proof gives a check (and it was the case for us before we found all the right translations) the reason can be in any of the four arrows of the diagram. But generally, we suppose that the semantic arrows are correct and we want to know if the translation into micro-instructions is correct. In that case, it will be interesting to add some tools which help to discover the origin of the error.

For instance, consider the proof of the diagram for the translation : ADD ME address (here adress is a symbolic variable) into its sequence of micro-instructions "mi-ad-pc ; mi-read ; mi- load-ir ; mi-ad-rop ; mi-read ; mi-load-rd ; mi-add". Our system will execute the semantics associated with each micro-instruction and compare the state obtained with the state resulting from the assembly semantics of ADD ME address. This execution, will automatically enable the dialiog between memory and CPU, so DATA takes the value read in the memory. In our implementation in Common Lisp this takes 0.8 s on a SUN IPC, and it is the same magnitude for the others instructions.

In order to use the incrementality of our object oriented approach, we have considered a more realistic processor that have all the mains characteristics of a real processor [8]. We associate to this more complex architecture, a class named: realistic-processor which inherits of all what has already been done in the simple-processor class, so we need only to indicate the new slots and the new methods and the supplementary classes. The adressing is more complicated and for the equality proofs, our symbolic evaluator uses often the following kind of commutation formula between $ and @:

$ (V1@V2) = 2** (length V2) * $ V2 + $ V1.

7 Conclusion

We have shown how an object oriented approach can be used in the semantic description of micro-processor architectures at several levels : assembly programming, micro-sequence and micro-instruction. This Object model of micro-processor give also a (dynamic) type checking . Furthermore, the use of active values to model memory control signals shown the adequacy of CLOS to automatically implement the memory / processor communication protocol.

The proof of the translation of assembly instruction into micro- instructions is based on a specialized computer algebra system in order to deal with symbolic operands. The magnitude of proof time seems very satisfactory. In some complicated cases, our automatic proof could be enable to conclude, so we plan to add an interactive proof system to deal with these special remaining cases.

There remains a lot of work to do : firstly our implementation is a prototype, we have to add a nice user interface with menus, mouse, dialog ,...,. More complicated will be to add an automatic generation of the semantic description taking input from a well adapted specification tool (interactive tool for the specification of assembly-level in abstract syntax, hardware description language for the micro- level).

References

[1] D. BORRIONE, P. CAMURATI, P. PRINETTO, J.L PAILLET: "A functional approach to formal hardware verification: The MTI experience", Proc. IEEE International Conference on Computer Design ICCD'88, New York, Oct. 88.

[2] D. BORRIONE, C. LE FAOU: "Overwview of the CASCADE Multi-level Hardware Description Language and its Mixed-Mode Simulation Mechanism", Proc. IFIP WG 10.2 Int. Conf. CHDL 85, Tokyo, Japan, Aug 85

[3] M BICKFORD, M.SRIVAS: "Verification of a Pipelined Microprocessor using CLIO", Proc Work. on "Hardware Specification, Verification and Synthesis: Mathematical Aspects", Ithaca, Jul 89, LNCS 408.

[4] P. CAMURATI, P. PRINETTO: "Formal verification of hardware correctness: an introduction", Proc. 8th IFIP int. Conf CHDL 87, Armsterdam, April 1987 (also in computer 88)

[5] W.J. CARTER, W.H.JOYNER, D. BRAND: "Symbolic Simulation for Correct Machine Design", ACM IEEE 16th Design Automation Conference, June 1979, pp 280-286.

[6] A. COHN: "A Proof of Correctness of the Viper Microprocessor", in proc. workshop "VLSI Specification, Specification and Synthesis", ed. G Birtwistle and P.A Subrahmanyam, Kalgary, KWA 1987.

[7] P. CAMURATI, P. PRINETTO: "Formal verification of hardware correctness: an introduction", Proc. 8th IFIP int. Conf CHDL 87, Armsterdam, April 1987 (also in computer 88)

[8] J. CHAZARAIN, H. COLLAVIZZA : "An Object Oriented Approach for Specifying and Proving Processor Like Architectures at the RT-Level" Research Rapport RR-92-69 CNRS URA 1376 Sophia Antipolis , 1992.

[9] L. CLAESEN, D. BORRIONE, H. EVEKING, G.J. MILNE, J.L. PAILLET, P. PRINETTO: "CHARME: towards formal design and verification for provably correct VLSI hardware", in proc. of the advanced work. on Correct Hardware Design Methodology, Turin,12–14 June 1991.

[10] H.COLLAVIZZA: "Functional Semantics of Microprocessors at the Microprogram Level and Correspondence with the Machine Instruction Level", Proc. EDAC, March 1990.

[11] H.COLLAVIZZA: "Semantique Fonctionnelle des Microprocesseurs: L'environnement de Spécication et de preuve "μSPEED" Thesis University of Aix Marseille I 1991.

[12] D. VERKEST, L.CLAESEN: "Special Benchmark Session on Tautology Checking", Proc. IFIP WG 10.2/10.5 Work. on "Applied Formal Methods for Correct VLSI Design", Houthalen, Belgium, 13-16 Nov. 1989 (Ed. by L. CLAESEN, North Holland)

[13] G. GENTZEN: "Investigation into Logical Deduction" Thesis 1935, reprinted in "The collected papers of Gerhard Gentzen" E. Szabo, North Holland, Amsterdam 1969

[14] M. GORDON: "LCF-LSM", TR 41, Computer Laboratory, University if Cambridge, England, 1983

[15] W.A. HUNT: "FM8501: A Verified Microprocessor", Technical Report 47, Institute for Computing Science, University of Texas at Austin, Feb 1986.

[16] J. JOYCE: "Generic Specification of Digital Hardware", Proc. Work. on Designing Correct Circuits, 26-28 Sept 1990, Glasgow (Springer Verlag)

[17] G. KAHN: "Natural Semantics", Proc of Symp on Theoretical Aspects of Computer Science, Passau, Germany , LNCS 247,1987.

[18] M. LANGEVIN, E. CERNY: "Verification of processor- like circuits", in proc. of the advanced work. on Correct Hardware Design Methodology, Turin,12-14 June 1991.

[19] G.J. MILNE: "Design for Verifiability", Proc. Work on "Hardware Specification, Verification and Synthesis: Mathematical Aspects", Ithaca, Jul 89, LNCS 408.

[20] J.L.PAILLET: "Functional Semantics of Microprocessors at the Machine Instruction Level", Proc 9th IFIP WG 10.2 Conf CHDL,Washington, June 89.

[21] G. PLOTKIN : "A structural approach to operational semantics" Report DAIMI FN-19, Computer Science DPt Aarhus Univ, Danemark, 1981.

[22] G. L. STEELE Jr.: "Common LISP The Language" Second Edition Digital Press , 1990.

[23] WINDLEY: "The formal Verification of Generic Interpreters", Phd Thesis, University of California, Division of Computer Science, 1990.