Introduction to Formal Processor Verification at Logic Level : A case Study.

Paul Amblard, TIMA-CMP, 46 av. Félix Viallet, 38031 GRENOBLE Cedex, France
Fabienne Lagnier, Vérimag, Equation, 2 avenue de Vignate, 38610 GIERES, France
Michel Levy, LSR-IMAG, B.P. 72, 38042 St MARTIN D'HERES Cedex, France
email : Paul.Amblard, Fabienne.Lagnier, Michel.Levy@imag.fr
Université Joseph Fourier, Grenoble, France.

Abstract—
This paper presents the case study proposed to 3^{rd} year students in our department of computer science. It is a practical activity in the first "Computer Architecture" Unit of the curriculum. This practical activity has several aims : 1) understanding a subtle mechanism in processor architecture, 2) experimenting the relations between logic level and RTL level descriptions and 3) practicing formal methods of verification. The main original point is the use of extraction (and minimization) of the full description of an automaton from the logic schema based on flip-flops and gates. In a certain way, the reverse of classic "automaton synthesis".

I. Introduction

Computer Science is an important component in Grenoble University due to environment : Grenoble region is surrounded by mountains and semiconductor plants. It is sometimes described as a kind of Silicon Valley. 2003 serious estimations[1] give the following data : 17700 people work in computer science, 12300 in electronics, 2300 computer scientists work for electronic components sector and 900 others in the field of electronic CAD.

Our activity is motivated by 2 common sense ideas :

- Not all the computer scientists are processor architects but every computer scientist must know architecture principles. ([12])
- Not all the computer scientists use formal methods in their work, but every computer scientist must know formal methods principles.

One of our tasks is to give all the students a good basic knowledge in processor architecture. Some of them will further specialize in this field. Many others will not and will choose image synthesis, embedded systems, networks engineering or one of many other fields. We want also them to have practiced formal methods.

In this paper we shall give the main ideas founding this activity, then, in section 3 we shall describe the context of education. The techniques used and the example itself are described in further sections 4 and 5.

[1] Reported by "Electronique International Hebdo, 25 Mars 2004, p 38

II. Ideas in processor architecture and verification

The main ideas we want our students to acquire are organized around three topics : 1) processor architecture, 2) abstraction levels in digital circuits description and 3) validation of a digital design.

A. Basic concepts in processor architecture

Let us give a list of basic principles that our students must master at the end of our Unit :

- There is a border between hardware and software. Machine language is the lowest level of software and the highest level of hardware. At the lowest level, interpretation of machine language instructions does not result from execution of a program but from circuits activity.
- Execution of one instruction costs several time steps. One of the reasons is that one instruction could require several memory accesses to fetch the instruction code and the operands.
- One such basic step consits in a state change in a "data-path" automaton. A "data-path" automaton is a Finite State Machine (FSM) but we prefer to describe its *implementation* by registers and ALUs rather than its *function* by a list of states and transition arrows.
- Controlling the flow of steps is done by a controller taking into account Instruction Register and Interruption Requests. The controller behaviour can be described by an FSM with actions associated to states. Microprogramming was the basic technique to implement such controllers.
- Pipelining is a common technique used to implement processors. In this case it is not convenient to describe the controller by a "centralized" function.
- Pipelining introduces a problem for conditional control transfer instructions.

B. Basic concepts in digital circuits description

Before introduction to processor (and computer) architecture, the students already know the basics of digital design. They can synthesize and describe *small* circuits at the logic level. Processor architecture will make obvious the need for Register Transfer Level (RTL) description for *big* circuits.
The 4 kinds of circuits of figure 1 and associated descriptive formalisms are used. Students progress in understanding these categories. Obviously the border between small and big circuits is partially a matter of convention. As a consequence the border between logic level and RT Level is such that some "intermediate complexity" circuits can be described at both levels. The example presented in this paper was designed to be such a circuit. The proof itself relies on logic description.

C. Basic concepts in validation

The main technique used to check correct functionality in computer science is to put the object (hardware or software) at work (truly or by simulation).

Another technique is to list some properties the object must verify and to check them. A formal model must exist allowing to describe these properties and the implementation of the object. A formal checker can be used. This second technique is now commonly used for combinational circuits where the model is Binary Decision Diagram representation of boolean functions. ([1])

A similar approach, based on Model Checking, is used for Finite State Machines. In this case the problem of combinatorial explosion of the number of states cannot be avoided.

Theorem provers are often used in processor verification at a research level. (See for example conferences such as CAV, DATE, DAC, FMCAD, CHARME). M. Velev has very well described the introduction of such a technique in advanced education ([10]).

Our case study uses a variant of Model Checking and simulation.

III. Context of education

For the reader to understand WHERE our case study occurs in the curriculum, let us briefly present it. Our Unit is in a curriculum of computer science where formal methods and techniques are important. This Unit is the first one in computer architecture. The companion book (in french) is [2]. Archives of exams are available at [17].

In this section, we describe the place of architecture and formal methods in the curriculum of our department.

Before introduction to computer architecture, basic digital design techniques are already known (Karnaugh maps, FSM synthesis). Similarly, basic programming in C and in assembly language are already known. Assembly language is based on Sparc[2]. Concurrently with Computer Architecture, introduction to compilation and advanced programming in assembly language occur : How to program functions, environment and frame pointer management in stacks ? How to read code produced by a standard C compiler ?

This Unit of Computer Architecture is the first encounter between students and the Hardware/Software interface. A home-made simulator allows to discover the principles of instructions execution. It is based on a microprogrammed organization. After that introduction, simulation of a pipelined machine is made. The students must introduce the "by-pass" mechanism in the RTL description of the given processor. In the Unit, memory hierarchy is "independant" from instructions interpretation. We do not deal with the interactions between caching and executing. In this Unit, nothing is said about advanced techniques [9].

The practical activity presented hereafter illustrates (a part of) the pipeline organization of a SPARC.

For many reasons, not described in this paper, our department strongly focusses on formal methods. For instance in digital circuits design introduction [3], some circuits proofs are done. Students simulate the circuits at the logic level, then they are also invited to check equivalence of two combinational circuits, then they also verify equivalence between two synchronous implementations of Finite States Machines. Similarly programming techniques education are taught in relation with formal descriptions and verification of programs ([7]).

IV. Environment and Proof techniques

This section presents the tools and techniques used for this activity. The main originality is to use a kind of "reverse" synthesis of Finite State Machines.

A. How do we describe ?

All the descriptions are given in the language LUSTRE ([4] and [5]). In circuits description LUSTRE is close to Lola, the language used by N. Wirth in his book. ([11])

Description may be of different types :

- Circuits described as a set of nodes : the nodes contain logic gates and edge-triggered D-type flip-

[2]For technical reasons, we are currently moving from Sparc to another processor

Type	of circuits	implementation objects	description formalisms	level
small	combinational circuits	gates (pass transistors)	boolean algebra	logic
big	combinational circuits	adders, coders, MUXes	arithmetic and composition	RTL
small	sequential (synchronous)	gates, flip-flops	FSM bubbles and arrows	logic
big	sequential circuits	registers, ALUs, busses,..	FSM with actions	RTL

Fig. 1. The different kinds of circuits

```
node mux1bit (i, t, e: bool) returns (s:bool);
let
  s = (i and t) or (not i and e);
tel;

node add1bit (a,b, ret_in :bool) returns (som, ret_out: bool);
let
  som     = a xor b xor ret_in ;
  ret_out = a and b or a and ret_in or b and ret_in;
tel;

node addNbits (const N: int; a,b: bool^N) returns (r: bool^N);
var carry : bool^(N+1) ;
let
  carry[0] = false ;
  (r[0..N-1],carry[1..N]) = add1bit(a[0..N-1], b[0..N-1], carry[0..N-1]);
tel;
```

Fig. 2. A flavour of Lustre descriptions

flops. The only data type is boolean. (Cf node `mux1bit` or `add1bit`) hereafter.

- Generic circuits of size N, dealing with boolean vectors of size N. Registers have N flip-flops. Adder can be N bits wide. (Cf node `addNbits` hereafter). Notice the "implicit" repetition of add1bit in addNbits. N must be instanciated before effective use. This allows us to have a same description for any N-bits circuits, we only need to change one constant. The same feature exists in VHDL.
- Circuits described as a hierarchical or compositional set of nodes. The nodes can be different (cooperating) automata. The language is such that, basically, all the automata share the same clock. Due to this feature, LUSTRE is often refered to as a *synchronous* language. ([6])

This Lustre example in figure 2 is given for illustration.

B. What do we obtain ?

From the logic description, it is possible to simulate the circuit. The gate or flip-flops delays are not taken into account. Timing diagrams can be drawn. This step is done by the students.

Another use, more orignal can be made : We *compile* the circuit description. Let us examine the meaning of this compilation.

Given a circuit description of the FSM logic implementation by gates and flip-flops, the Lustre compiler [4], [16] can *compute* the automaton as a set of states and a full description of the two functions : transition function and output function. Obviously this is the reverse task compared to the very common synthesis tools available in all standard CAD packages.

If the input description contained several automata, the compiler computes the product automaton. The description of the result automaton is given either in an internal textual form, or in C language, ready for compilation, or in a graphical form. It could as well be given in VHDL or an other Hardware Description Language. The execution of the C version gives the same results than a simulator. We use the textual form with the students.

A complementary tool gives the minimal automaton equivalent to the proposed one.

This essential facility is used in the introduction to digital design to formally check equivalence between 2 circuits. ([3])

V. THE CASE STUDY

Our circuit is a pipelined processor. We got a VHDL description of a SPARC architecture from the European Space Agency site (Leon version [14]). To make the example as simple as possible, we made drastic simplifications and limited ourselves to a simple mechanism : delayed control transfer instructions. The example is organized around the part

	code	possible behaviours
	Instr1	sequence if `cond` is true at I2
I2	Brcond `label`	Instr1, Brcond label, Instr3, Instr5,..
	Instr3	
	Instr4	
	...	
`label`	Instr5	sequence if `cond` is false at I2
	...	Instr1, Brcond label, Instr3, Instr4, ...

Fig. 3. Delayed branch mechanism : a small SPARC program and the two possible behaviours given by the sequences of instructions

address	label	instr		sequences (values of PC)
0	`zz`	instr0		0 1 2 3 4 (not at 2)
1		instr1		0 1 2 3 7 (yes at 2)
2		brcond	`ss`	
3	`tt`	instr3		3 4 5 6 7 . (not at 5) (not at 6)
4		instr4		3 4 5 6 0 1 (yes at 5) (not at 6)
5		brcond	`zz`	3 4 5 6 7 3 (not at 5) (yes at 6)
6		brcond	`tt`	3 4 5 6 0 3 (yes at 5) (yes at 6)
7	`ss`	instr7		

Fig. 4. A complex short SPARC program and the possible behaviours. If the condition tested in line 2 is true, the sequence of values of the program counter is 0, 1, 2, 3, 7 ; if this condition is false the sequence is 0, 1, 2, 3, 4. Similarly if the condition tested in line 5 is false and condition tested in line 6 is false, the sequence of instructions is 3, 4, 5, 6, 7.

computing the next value of the Program Counter (PC). We study the so-called *delayed branch* mechanism.

A. How does progress a SPARC *Program Counter ?*

The system of SPARC is different from the standard one and is well known ([15], [13]). To make this paper self-contained, we recall it. There are Control Transfer Instructions (CTI). Different CTI exist : Jump and Link, Conditional Branch and Call. The instruction written immediately after a CTI is executed first, then the transfer of control occurs. This mechanism is known as *Delayed Branch*. The instruction inserted is said to be in the *Delay Slot*. We shall simplify here by considering only conditional branch instructions. We do not use the mechanism of *annul bit* in the frame of this paper.

In the small program of figure 3 two sequences of instructions may occur (assuming that Instr1, Instr3, Instr4, Instr5 are not CTI) :

- if the condition is **true** when it is examined in instruction I2 the sequence of instructions is
`[Instr1, Brcond label, Instr3, Instr5]`

- if the condition is **false** when it is examined in instruction I2 the sequence of instructions is
`[Instr1, Brcond label, Instr3, Instr4]`

This behaviour is made possible by the existence of a (classical) register Program Counter (PC) and of another information named Next Program Counter (nPC) in the documentation. The immediate question is obviously : *What occurs when two CTI are written consecutively ?* (However the standard practice of a programmer is not to write programs with such features [8].) The complete documentation ([13], [15]) explains the different possible behaviours in this case. We take here a simplified version.

We shall present such a situation in figure 4 : the program contains two consecutive conditional branches. They appear in lines 5 and 6.

Let us examine this small program. The expected behaviours depends upon the values of the condition during execution of instruction 5 and 6. For instance if the condition tested in instruction at address 5 and the condition tested in instruction at address 6 are both true, the sequence of values of the Program Counter is 3, 4, 5, 6, 0, 3. The others sequences are given on the figure itself.

B. Our experiment with this Very Reduced Computer

For this experiment, we use only :

- the Program Counter (PC)
- the Next Program Counter value (nPC),
- the combinational incrementer associated with these registers,
- the Instruction Register (RInst) containing the current instruction. It has two fields : opcode and displacement.
- the adder used to add a displacement (depl) to ob-

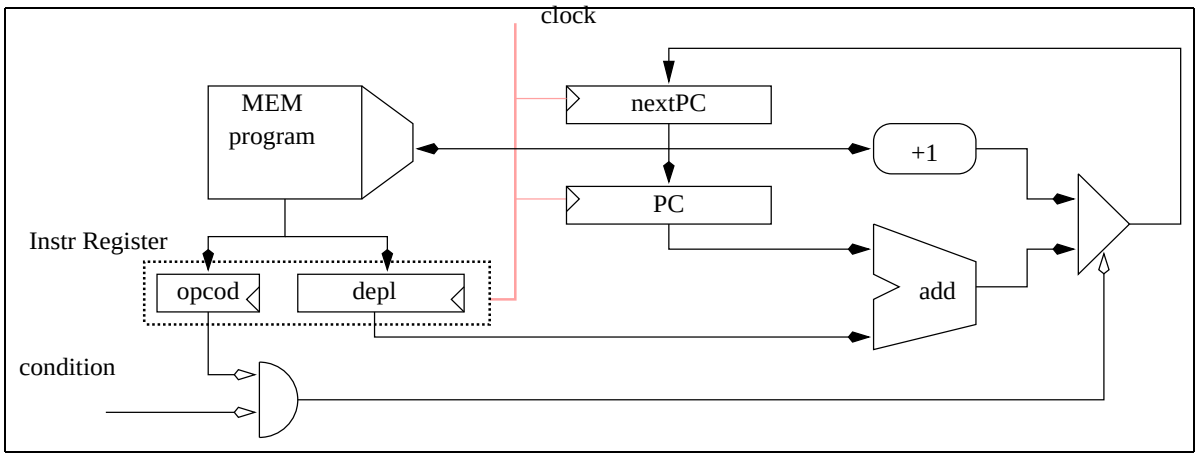

Fig. 5. Organization of the Program Counter updating in reduced SPARC p rocessor

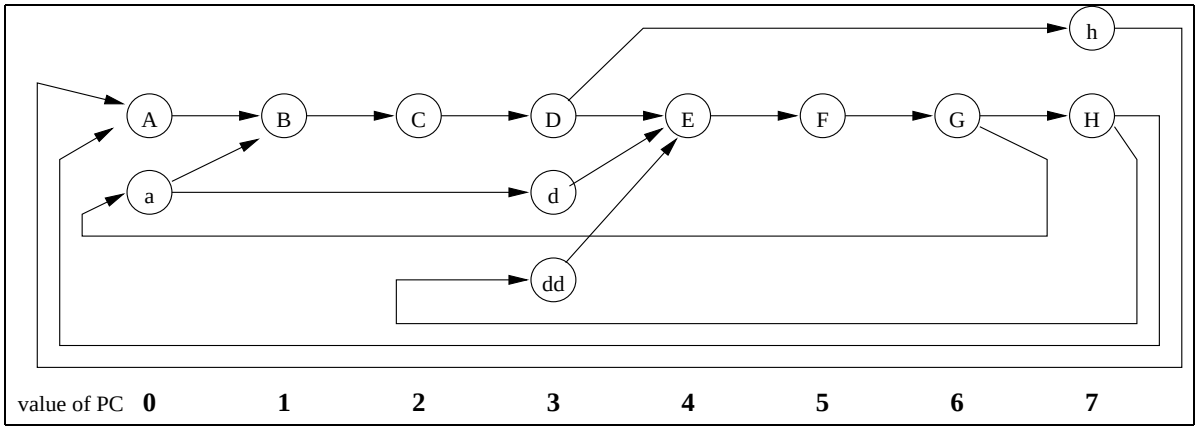

Fig. 6. The automaton obtained by the Lustre compiler. The corresponding values of the Program Counter are indicated. When a state has two "next states", the first one correspond to cond = true, the other one to cond = false.

tain the branch target address

The Register Transfer Level description of the system is given graphically in figure 5.

Our circuit is composed of this restricted SPARC and of a memory containing the aforementionned program. This memory can be a ROM because we do not use any STORE instructions.

Our experiment is based on a description of this processor+memory at a logic level. In this example we restricted the data path to 3 address bits and to 4 data bits. The ROM contains 8 4-bits words as in figure 4. The Op-Code has only one bit (true for a BRCOND, false for a NOP) and the displacement is coded on 3 bits. It is enough for our experiment as will be shown.

To put focus on the role of the condition, we consider it to be an external input. The logic description is simple : 3 bits adder, 3 bits incrementer,... The only input of this circuit is `cond`.

The students are invited to compile the Lustre description of this logic description, obtain an automaton, and minimize it.

They obtain the automaton described by figure 6.

C. Results and comments

Figure 6 gives the states obtained from the compiler. How do we understand this automaton ? We name the states A, B, C, D, E, F, H and a, d, dd and h. A is the initial state. In regards to the states we added the corresponding values of the Program Counter. For instance in states D, d and dd, the PC value is 3. Let us comment a transition in the automaton :
- Arrow D $\rightarrow$ h (PC = 3 $\rightarrow$ PC = 7) correspond to the instruction Brcond at address 2 and a condition True. As we have already seen, in this case the sequence of values of PC contains 2, 3, 7.

All the possible behaviours given in figure 4 correspond to a path in this automaton. The sequence of values of the PC 3, 4, 5, 6, 0, 1 (condition true in

instruction line 5 and condition false in instruction at line 6) correspond to the sequence of sates D, E, F, G, a, B.

Detailed exploration of this automaton gave us confidence that our PC computation mechanism is correct with respect to the specification of the processor with delayed branch. All the PC sequences of figure 4 are present on the automaton. We could also observe that our simplified model has introduced an artefact : *cond* seems to be tested one clock cycle too late.

The main activity of students is this task of understanding. They have to enter in the SPARC documentation and they must relate the results of this lab sketch to the "true world"

VI. Conclusion

We have presented an introductory activity for three fields : computer architecture, digital logic design of processors and formal verification. Obviously no generalisation of this technique can be made for a true size processor. We make the students aware of this point.

This case study also shows the relations between RTL and logic levels. It seems to us necessary to explore some logic implementations of RTlevel tricks such as pipeline, branch delayed instructions, ..

Introduction to read papers about formal proofs of processors would be a further step, but it needs complementary knowledge for the students. Thanks to M. Velev ([10]) we have a rich list of references as a starting point.

References

[1] **S. B. Akers :** Binary Decision Diagrams, IEEE Transactions on Computers, Vol C-27, June 1978.

[2] **P. Amblard, J.C. Fernandez, F. Lagnier, F. Maraninchi, P. Sicard et P. Waille :** *Architectures Logicielles et Matérielles*, Dunod, 2000 (in french).

[3] **P. Amblard, F. Lagnier and M. Levy :** Introducing Digital Circuits Design and Verification Concurrently, Proceedings of the 3^{rd} European Workshop on Microelectronics Education, Aix en Provence, 18-19 May 2000, Kluwer, pp 261-264.

[4] **N. Halbwachs, P. Caspi, P. Raymond and D. Pilaud :** The Synchronous Data-flow Programming Language Lustre, Proceedings of the IEEE, pp 1305-1320, September 1991.

[5] **N. Halbwachs, F. Lagnier and C. Ratel :** Programming and Verifying Real-time Systems by Means of the Synchronous Data-flow Programming Language Lustre, IEEE Transactions on Software Engineering, Special Issue on the Specification and Analysis of Real-Time Systems, September 1992, pp 785-793.

[6] **N. Halbwachs :** *Synchronous programming of reactive systems*, Kluwer Academic Pub., 1993

[7] **M. Levy, L. Trilling :** A PVS-Based Approach for Teaching Constructing Correct Iterations FM'99 (Formal Methods), LNCS 1709, pp 1859-1860

[8] **R. Paul :** *SPARC Architecture Assembly Language Programming, and C*, Prentice-Hall, Inc., 1994

[9] **Sima D., Fountain and Kacsuk** *Advanced Computer Architectures*, 1997, Addison-Wesley

[10] **M. Velev :** Integrating Formal Verification into an Advanced Computer Architecture Course, ASEE 2003 Annual Conference, `http://www.ece.cmu.edu/~mvelev/ASEE03.pdf`

[11] **N. Wirth :** *Digital Circuit Design*, Springer-Verlag, 1995.

[12] I.E.E.E Micro, may-june 2000, Computer Architecture Education.

[13] *The SPARC Architecture Manual*, version 8, Prentice-Hall, Inc., 1992.

[14] `http://www.estec.esa.nl/wsmwww/leon/leon.html`

[15] `http://www.sparc.com/standards/V8.pdf`

[16] `http://www-verimag.imag.fr/SYNCHRONE`

[17] `http://tima-cmp.imag.fr/~amblard/EXAMS/exams.html`