Using Formal Tools to Study Complex Circuits Behaviour

Paul Amblard
TIMA-CMP, 46 av. Félix Viallet, 38031 GRENOBLE Cedex, France

Fabienne Lagnier
Vérimag, Centre Equation, 2 avenue de Vignate, 38610 GIERES, France

Michel Lévy
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

*We use a formal tool to extract Finite State Machines (*FSM*) based representations (lists of states and transitions) of sequential circuits described by flip-flops and gates. These complete and* **optimized** *representations helps the designer to understand the accurate behaviour of the circuit. This deep understanding is a prerequisite for any verification or test process. An example is fully presented to illustrate our method. This simple pipelined processor comes from our experience in computer architecture and digital design education. ([2])*

1. Introduction

It is now widely accepted that a clean specification of a circuit must be designed in parallel with the design of the circuit itself. This hardware-brainware co-design is necessary to verify, simulate, prove, test, validate the circuit. Our observation is that it is often difficult to express *all* the specifications of a circuit even if we know how to design it.

Our proposal, computer assisted *exploration*, can help to obtain some properties. Particularly it can reveal unexpected, but correct, behaviours. It also help to understand complex behaviours. It will be the case in a pipelined example. An obvious result is that *exploration* can reveal some differences between circuit and specifications. The earlier this discovery can be done in the design process, the better it is.

This work is at the logic level, we deal with flip-flops and gates, so the approach is typically for sequential circuits. At this level of abstraction, the formal model is a Finite State Machine. We are obviously in front of the combinatorial explosion problem : in this approach, if a circuit has N flip-flops, the corresponding automaton has potentially 2^N states. As a consequence, we must try to maintain N small. Our approach is well suited for small size circuits, or mechanisms. Its goal is to analyse a part of a circuit, not a big true device.

Another key point of our approach must be pointed at : underlying principles, and the tool we use, suggest to manipulate the circuit with different formal frames (typically an FSM or flip-flops + gates).

By suggesting to have design and specification in different formalisms and proposing to automatically transform one of them into the other one, we hope to propose a better analysis of the hardware devices.

The first section will explain the principles of our approach. The central section gives detail and analysis of an example extracted from our experience in digital circuit design and computer architecture education. The circuit is inspired by a pipelined processor. A further section compares our results to other approaches in similar situations, and with the main other validation techniques (model-checking, theorem proving, simulation).

2 The principles

All the modern C.A.D. tools contain a FSM synthesis package : given a list of states and transitions (called the *specification*), the tool computes a netlist of gates and flip-flops (called the *implementation*).

The basics of our approach are very simple : given a description of an *implementation* of a Finite State Machine (FSM), the tool computes the expansion of the FSM by states and transitions. Several automata deliver the same output sequence for a same input sequence ; one of them has a

minimal number of states. The tool delivers this minimal equivalent FSM. The designer works with this minimal representation. Obviously this does not allow to deal with *big* automata. We shall give some details on the techniques used to describe the source FSM. The techniques used to compute this expansion are presented in [5] and this present contribution is not about such C. A. D tool but about a new way to use it.

2.1. How do we describe ?

All the descriptions are given in the language LUSTRE ([7] and [8]). LUSTRE looks like Lola, the language used by Wirth in his book. ([15]) Description may be of different types :

- Circuits described as a set of nodes : the nodes contain logic gates and edge-triggered D-type flip-flops. The only data type is boolean.
- Generic circuits of size N, dealing with boolean vectors of size N. In the description registers have N flip-flops. N must be instanciated before effective use, either description of a physical device or description of an automaton. A physical hardware device cannot have N pins. An automaton cannot have N states.
- 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 referred to as a *synchronous* language ([6]).

2.2. An example of describing circuits in Lustre

Here is an example of a n bits adder and a n bus multiplexor. `X:bool^n` defines `X` being a bus of n wires, each of which being a boolean. The least significant wire is `X[0]` and the most significant one is `X[n-1]`.

```
node add1 (a,b,c:bool) returns (r,s:bool);
let
 r = a and b or a and c or b and c;
 s = a xor b xor c;
tel;

node add (const n: int ;
          a,b: bool^n)
      returns   (sum:bool^n);
var carry : bool^(n+1);
let
 carry[0] = false;
 (carry[1..n], sum[0..n-1]) =
 add1(a[0..n-1],b[0..n-1],carry[0..n-1]);
tel;
```

```
node mux1 (i,t,e:bool) returns (s:bool);
let
 s =  i and t or not i and e;
tel;

node mux (const n:int ;
          i:bool;
          t,e:bool^n)
    returns  (s:bool^n);
let
 s[0..n-1] =
         mux1(i^n, t[0..n-1], e[0..n-1]) ;
---boolean i is repeated n times
tel;
```

A basic node `flipflop` defines an edged-triggered D device :

```
node flipflop (D:bool; clock,reset:bool)
      returns (Q:bool).
```

We use it to implement registers. (Fig 4)

```
node partoffig4 (const n: int ;
         depl: bool^n ;
         clock,reset,cond: bool)
         returns
         (newpc: bool^n);
var pc, spc : bool^n;
let
 pc  = flipflop (newpc, clock^n, reset^n);
 spc = flipflop (pc, clock^n, reset^n);
 newpc = mux (n, cond and br,
                 add (n, spc, depl),
                 plusone (n, pc) );
tel;
```

2.3. What do we obtain ?

A first use is to *compile* the circuit description given by gates and flip-flops. The compiler delivers a description of the given automaton. The description is based on the set of states and the two functions : transition function and output function. If the input description contained several automata, the compiler computes the product automaton. We must be careful and avoid too large machines.
The description of the result automaton is given in an internal textual form or in a graphical form. It could as well be written in VHDL or an other Hardware Description Language. Another tool allows to minimize this automaton.

Different uses can be done with the result of this extraction, this paper concentrates on the third one.

• A first use is to check the circuit obtained by a commercial CAD synthesis tool. Our tool contains, in a certain form, the reverse function. Given a list of states and transitions, one of the tools we use gives the minimal equivalent FSM.

• A second very important task is allowed by this tool, but we shall not enter into the details in the frame of this paper : if we describe two FSM, and if we add a comparator on the outputs, we can check the equivalence of the two FSM (Let us notice that the comparator is virtual in this case). They are equivalent if (and only if) the comparator delivers always "True". In this case the minimal automaton resulting from the composition of the two automata and the comparator has only one state. We used this approach in education [3].

• In this paper we present another use, *exploration*, based on the careful manual analysis of the result of this extraction process.

A second use of the description is simulation. An interactive version of the simulator allows to give inputs to the automaton and to observe outputs. Timing diagrams can de drawn. A batch version allows to put the inputs in a text file. We use this simulator in education.

2.4. What is *exploration* ?

Our technique is based on exploration of the circuit behaviour. Exploration means that a designer is not already completely certain about *What must be done ?* It is too early to give any kind of implementation, formal specification or any similar description. Exploration is what you do on draft paper, with a pencil. You are *entering* into your design and you need to dig into it. You try, in fact, to understand what your circuit will be (or *would be* ?). Exploration is certainly not for a full circuit but for a part of circuit, for a mechanism, for a hardware trick.

The tools used in this phase help you to get all the informations from a rough description. They make a kind of Computer Aided Draft. But your draft paper deals with formal proofs when needed. In the next section an example will be detailed to make clear this approach.

3 A complex behaviour, adapted from SPARC

Our circuit is a pipelined processor. To make the paper as self-consistent as possible, we made drastic simplifications and limited ourselves to a very simple example. Our 3 bits microprocessor could seem ridiculous compared to 50 000 000 transistors ones ! We use this example in digital design education at an introductory level.

Our example is a reduced version of a processor SPARC and is organized around the part computing the next value of the Program Counter (PC). We study the so-called *delayed branch* mechanism.

address	label	instr	
0	`zz`	instr0	
1		instr1	
2		brcond	`ss`
3	`tt`	instr3	
4		instr4	
5		brcond	`zz`
6		brcond	`tt`
7	`ss`	instr7	

Figure 1. A short program in assembly language. All the instr*i* are arithmetic. Label `zz` is at address zero, `tt` at three and `ss` at seven.

3.1. How does progress the Program Counter of a non SPARC processor ?

Let us consider a machine with only two classes of instructions : arithmetic instructions, (their only influence upon the Program Counter is its incrementation), and conditional branches BRCOND. In this simple machine, the Program Counter is coded on three bits. The short program of figure 1 contains 8 instructions, and after address 7, there is address 0.

This program could exhibit different behaviours, depending upon the values of the condition at the instants where it is tested. We can represent them by significant sequences of instructions :

- The sequence of instructions `[instr0, instr1, brcond ss, instr7]` occurs if the condition is Yes when evaluated at the instruction at address 2.
- `[instr0, instr1, brcond ss, instr3]` occurs if this same condition is No.
- `[instr3, instr4, brcond zz, instr0]` occurs if Yes occurs at address 5.
- `[instr3, instr4, brcond zz, brcond tt ,instr3]` if No occurs at address 5 and Yes at address 6.

3.2. How does progress a SPARC Program Counter ?

The system of SPARC is different from the standard one and is well known ([18], [16]) :

There are Control Transfer Instructions (CTI). Different CTI exist : Jump and Link, Conditional Branch and Call. We shall simplify here by considering only conditional branch instructions.

	code
	Inst1
I2	Brcond `label`
	Inst3
	Inst4
	...
`label`	Inst5
	...

line	value of cond	Inst Sequence
I2	true	[Inst1 Brcond Inst3 Inst5,..]
I2	false	[Inst1 Brcond Inst3 Inst4,..]

Figure 2. Delayed branch mechanism. The first table is a small SPARC program. The second table gives possible behaviours assuming that Inst1, Inst3, Inst4 are not Control Transfer Instructions,

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*. There is a mechanism of *annul bit*. We do not introduce it in the frame of this paper.

In the small program of figure 2 two sequences of instructions may occur (assuming that Inst1, Inst3, Inst4, Inst5 are not CTI) :
- if the condition is true when it is examined in instruction I2 the sequence of instructions is Inst1 Brcond Inst3 Inst5
- if the condition is false when it is examined in instruction I2 the sequence of instructions is Inst1 Brcond Inst3 Inst4

This behaviour is made possible by the existence of a (classical) register Program Counter (PC) and of another information named Next Program Counter (nPC). 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 [13].

The complete documentation ([16]) explains the different possible behaviours in this case. We take here a simplified version.

We shall present such a situation in figure 3 : the program contains two consecutive conditional branches. They appear in lines 5 and 6. The program is similar to the one of figure 1.

3.3. Our exploration experiment with this Very Reduced Computer

Our experiment was this one :

We got a VHDL description of a SPARC architecture from the European Space Agency site (Leon version [17]) and we

address	label	instr
0	`zz`	instr0
1		instr1
2		brcond `ss`
3	`tt`	instr3
4		instr4
5		brcond `zz`
6		brcond `tt`
7	`ss`	instr7

line	value of cond	Prog Counter values
2	true	[0, 1, 2, 3, 7]
2	false	[0, 1, 2, 3, 4]
5 then 6	false then false	[3, 4, 5, 6, 7, 0]
5 then 6	true then false	[3, 4, 5, 6, 0, 1]
5 then 6	false then true	[3, 4, 5, 6, 7, 3]
5 then 6	true then true	[3, 4, 5, 6, 0, 3]

Figure 3. A SPARC program with intricated branches and the possible behaviours.

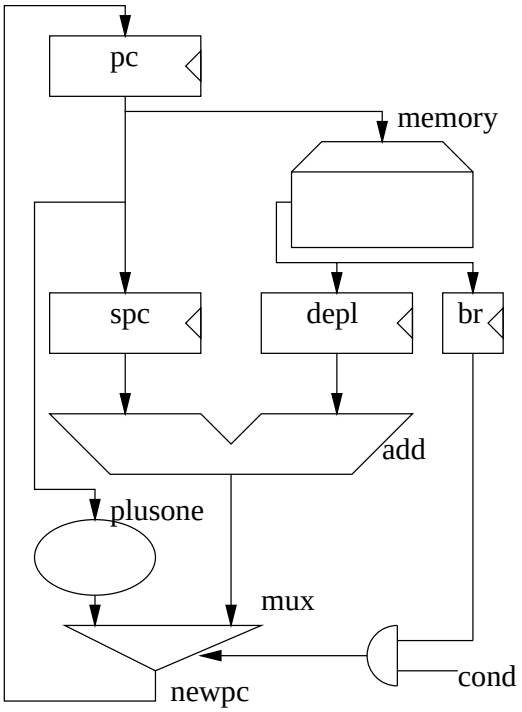

Figure 4. Organization of the Program Counter updating in reduced SPARC processor. Instead of a condition code register we use cond as an external input.

simplified it.

For this experiment, we saved only :

- the Program Counter (pc) and its ghost copy (spc),
- the Next Program Counter value (newpc),
- the combinational incrementer (plusone) associated with these registers,
- the adder used to add a displacement to obtain the branch target address
- the Instruction Register containing the current instruction. It contains two fields : br is operation code, depl is a displacement of branch instructions.

The Register Transfer Level description of the system is given graphically in figure 4.

Our circuit is composed of this restricted SPARC and of an 8 words memory containing the aforementioned program. This memory can be a ROM because we do not use any STORE instructions.

Let us examine the small program used as a test-bench :(figure 3) The expected behaviour 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.

3.4. Boolean level description

Let us recall that the automata computing facility of the Lustre compiler [7], [19] can compute the minimal automaton from one of its descriptions. For instance a description given in logic gates and flip-flops. To use this facility, we restricted the data path to 3 address bits and to 4 data bits. The ROM contains 8 4-bits words as in figure 3. The Op-Code has only one bit (true for a BRCOND false for a NOP) and the displacement is coded on 3 bits. It was enough for our experiment as will be shown.

To put focus on the role of the condition, we considered it to be an external input. The logic description is simple : 3 bits adder, 3 bits incrementer,... We compiled the Lustre description of this logic description, obtained an automaton, and minimized it. We obtained the automaton described by figure 5.

3.5. How do we understand this automaton ?

Figure 5 gives the states obtained from the compiler. We named them 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

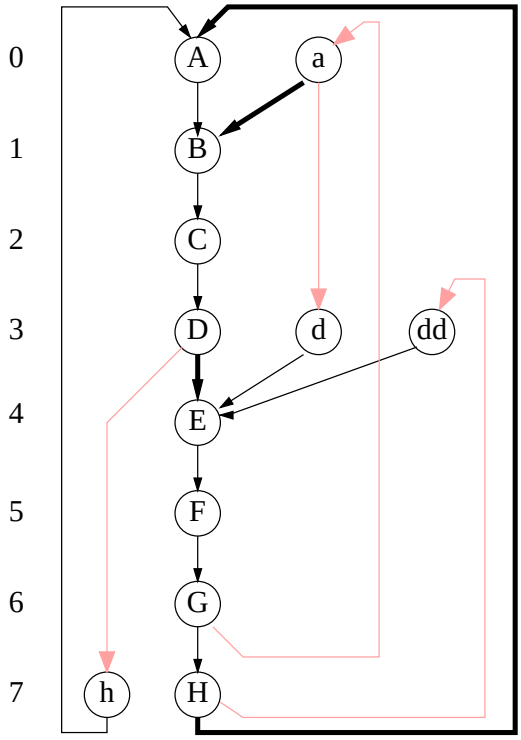

Figure 5. All the possible states of program from figure 2 The left column gives the Program Counter values. The picture has three kinds of arrows : black thin arrows correspond to standard PC incrementation, (example : from PC=1 to PC=2), black bold arrows correspond to rejected control transfers when the condition is false (example : from PC=3 to PC=4), grey arrows correspond to control transfers when the condition is true (example : from PC=3 to PC=7).

transition in the automaton :
- Arrow D $\rightarrow$ h (PC = 3 $\rightarrow$ PC = 7) corresponds to the instruction Brcond at address 2 and a condition True.

All the possible behaviours given in figure 3 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.

Exploration gave us confidence that our PC computation mechanism is correct with respect to the specification of the processor with delayed branch. We could also observe that our simplified model introduces a simulation artifact : *cond* seems to be tested one clock cycle too late.

4 Comparison of *exploration* with other approaches

In this section we compare the principles of this technique to other approaches.

• Obviously our *exploration* must not be confused with the *state exploration* used in certain model-checkers.

• A first characteristic of our approach is its relation to simulation. Exploration can give some complete informations while simulation cannot. Let us explain :
- In a first step, simulation allows the designer to check consistency between the implementation and the intention. This part is known to be difficult and unsafe. The behaviour of the implementation is seen by timing diagrams and the reliability depends upon the testbench prepared. The problem of elaborating a good testbench is highly difficult. In exploration, we do not need to give a testbench. And we have, in a certain way, ALL the possible testbenches. Obviously it can be too much... But the behaviour obtained by this technique is complete.
- In a second step, simulation can take into account some informations extracted from layout steps. Nothing can replace these computations managing the wires and gates delay. Exploration is not useful at this level.

• A second characteristic is the relation to model checking. We use a formal approach, and associated tools. The LUSTRE compiler is in fact a model checker. It contains the functionality to build and minimize an automaton. But in this paper we present how exploration informations can be obtained from these tools. In the examples presented hereafter, the model checker verifier is used with a trick to obtain ALL the states of an automaton. This is generally frightening for people involved in model checking. They use equivalence relations on the states to avoid combinatorial explosion. In our examples these complete Finite State Machine structures *are the* useful information.

• It is difficult to compare exploration to Theorem Prover based techniques. Exploration can only help the designer to establish the expected properties or to discover certain counter-examples. By this experiment, we have not *demonstrated* that our SPARC is correct with regard to a given specification. We have done a kind of *symbolic execution* of the machine language program of figure 3 and we have seen that we obtain all the expected behaviours. Giving any kind of proof would have needed a specification of the delayed branch mechanism. Such a specification is difficult to establish. A proof of the implementation of a delayed branch mechanism appears in [10] and is based on the use of the theorem prover P.V.S. ([14])

We can certainly not give general conclusions from this study : in particular it is unrealistic to try to generalize such a study to a full processor. We simply made possible the accurate study of a subtle behaviour and we get confidence in our implementation of this behaviour. Our technique should be compared to J. Levitt and K. Olukotun's *unpipelining*.

" Our technique, which we call unpipelining, removes pipeline stages from an implementation while preserving the implementation's behaviour, collapsing it into a single stage through a series of transformations. The complexity due to the pipelining is completely eliminated and the deconstructed pipeline can be compared directly to the ISA specification. [11] "

We also break the complexity of pipeline implementations by considering the developed form. But we establish properties, at a logic level, by a tool similar to a model-checker while those authors use a theorem demonstrator to manipulate formulas representing the behaviour at Register Transfer Level.

5. Conclusion

This kind of exploration, based on human understanding of computer generated automata is very fruitful. We are aware that it is also difficult.

Our other experiments, not presented here because they were too big, show that 100 states is a maximal complexity. It means that we must simplify drastically a mechanism to study it. In a previous study ([4]) we obtained a 6500 state automaton and it was impossible to manage it by hand.

It remains of course tempting to explore the behaviour of other tricks in computer architecture. Branch Target Buffer or Register Renaming as in the processor AX ([1]) are good candidates. At least we can fully make ours those authors' comment : *" Experience in teaching computer architectures partially motivated this work."*

References

[1] Arvind and X. Shen, Using Term Rewriting Systems to Design and Verify Processors, IEEE Micro, May-June 1999, pp 36-46.

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

[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] P. Amblard, A Finite State Description of the Earliest Logical Computer : the Jevons' Machine, Mixed Design of Integrated Circuits and Systems, (Ed A. Napieralski, Z. Ciota, A. Martinez, G. De Mey, J . Cabestany), Kluwer, 1998, pp 195-202.

[5] A. Bouajjani, J.-C. Fernandez, N. Halbwachs, P. Raymond, C. Ratel, Minimal state graph generation, Science of Computer Programming, Vol. 18, 1992, pp 247–269.

[6] N. Halbwachs, Synchronous programming of reactive system, Kluwer Academic Pub., 1993

[7] 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.

[8] 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.

[9] S. Huang, K. Cheng, K. Chen, C. Huang, F. Brewer : AQUILA: An Equivalence Checking System for Large Sequential Designs, IEEE Transactions on Computers, May 2000, pp 443-464.

[10] D. Kroening, W. Paul and S. Mueller, Proving the Correctness of Pipelined Micro-Architectures, Proc. of the ITG/GI/GMM Workshop, (Ed K. Waldschmidt and C. Grimm), VDE Verlag, 2000, pp 89-98. `http://www-wjp.cs.uni-sb.de/projects/comparch/papers/pipe.pdf`

[11] J. Levitt and K. Olukotun, A Scalable Formal Verification Methodology for Pipelined Microprocessors, DAC 1996, Las Vegas, June 1996, pp 558-563.

[12] K. McMillan, Verification of Infinite State Systems by Compositional Model Checking, CHARME 1999, Bad Herrenhalb, september 1999, pp 219-233. (LNCS 1703)

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

[14] M. Srivas, H. Rueβ and D. Cyrluk, Hardware Verification using PVS, Formal Hardware Verification, Ed T. Kropf, 1997, pp 156-205. (LNCS 1287)

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

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

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

[18] `http://www.sparc.org/standards/V8.pdf`

[19] `http://www-verimag/SYNCHRONE`