Architectural Specification, Exploration and Simulation Through Rewriting-Logic*

Mauricio Ayala-Rincón † Reiner W. Hartenstein ‡

Rinaldi Maya Neto § Ricardo P. Jacobi ¶ Carlos H. Llanos ‖

Abstract

In recent years Arvind's Group at MIT has shown the usefulness of term rewriting theory for the specification of processor architectures. In their approach processors specified by term rewriting systems are translated into a standard hardware description language for simulation purposes. In this work we present our current investigation on the use of Rewriting-Logic, which is a more powerful theoretical framework than pure rewriting, for specification, exploration and verification of processor architectures at a higher abstraction level. We adopt the rewriting-logic environment ELAN to specify, explore and verify architectures without the need to resort to the details of hardware description languages for simulation purposes. Our investigation shows that simulation at rewriting-logic level may provide useful insights to guide the architectural design.

Keywords: Rewriting-logic, High Level Specification and Simulation, Design Environment.

1 Introduction

Since the seminal paper of Knuth and Bendix [12], the importance and applicability of Term Rewriting Systems (TRSs) and theory has been made evident in a great variety of fields of Computer Science. A lot of work has been developed which explores the simplicity of the computational formal framework given by rewriting for its application in areas such as automated theorem proving, program synthesis and verification, cryptography and code theory, development and implementation of higher order programming languages and proof assistants.

In recent years some work on applying rewriting techniques to model and verification of digital processors has been developed. In particular, Arvind's group has treated the design of processors over simple architectures [13, 14, 1] and synthesis of digital circuits [8]. Their approach to architectural description was to describe a simple RISC processor using TRS and to translate it to a standard hardware description language for simulation purposes. However, this approach introduces the cost of program translation and detailed hardware simulation, since TRSs are only used to specify, but not to simulate the design.

In this work we address specification and simulation of hardware using a rewriting-logic programming environment called ELAN [5]. Rewriting-logic extends the computational

*Work supported by the Brazilian/German cooperation of the CAPES/DFG foundations.

†Corresponding author. Partially supported by FEMAT Brazilian foundation for mathematical research. Departamento de Matemática, Universidade de Brasília, Brasil ayala@mat.unb.br

‡Fachbereich Informatik, Universität Kaiserslautern, Germany hartenst@rhrk.uni-kl.de

§Departmento de Ciência de Computação, Universidade de Brasília, Brasil rinaldi@cic.unb.br

¶Departmento de Ciência de Computação, Universidade de Brasília, Brasil rjacobi@cic.unb.br

‖IESB Brasília D.F., Brasil llanos@unb.br

power of purely rewriting by allowing logical control of the application of the rewriting rules. We present the description and simulation of simple RISC processors and we illustrate the design exploration of some architectural alternatives, like out-of-order execution and register renaming using ELAN. The processor behavior is defined by a set of rules and logical strategies and different architectural components as memory and registers are discriminated in a natural way taking advantage of the availability of types in the language. Soundness of the processors is shown by proving that they simulate and are simulated by a basic processor. In our ELAN approach the separation between logic and rewriting allows us to define rewrite rules for the instruction set of the processors and to specify strategies describing architectural characteristics such as the size of the reordering buffer (ROB) of speculative processors.

Since each processor instruction is defined by one rewriting rule, it is very easy to modify the proposed architecture either by adding new rules to extend the instruction set architecture (ISA) or by modifying them in order to change the behavior of the processor.

To validate the implementation we have simulated the execution of some sample programs (e.g., generation of the Fibonacci sequence, quick-sort and computation of the Knuth-Morris-Pratt jump function) in the ELAN environment. We can quickly change the description and strategies allowing to estimate the most adequate form to implement the architectural components. Estimations are performed by analyzing the ELAN statistics for the number of times each rewriting rule (i.e. processor instruction) is applied.

It is interesting to notice that rule application in TRSs is, theoretically, performed in a nondeterministic way, which helps modeling concurrent processes. In practice, however, rules are not naturally applied in a nondeterministic manner. In fact, during one step of the whole rewriting process the rule to be applied is usually selected according to the ordering they were defined and the selected rule is applied at the first possible position the rule matches (left-most, inner-most or similarly) [9]. In our approach these problems arise when important architectural aspects such as out-of-order execution is to be simulated. Although our implementations are deterministic we comment how we can overcome these problems in a non purely rewrite based environment like ELAN, where some nondeterministic strategies are available.

2 Theoretical Background

A Term Rewriting System, TRS for short, is defined as a triple $\langle S, R, S_0 \rangle$, where S and R are respectively sets of terms and of rewrite rules of the form

$$s_1 \to s_2 \;\; \textit{if} \;\; p(s_1)$$

being s_1, s_2 terms and p a predicate and where S_0 is the subset of initial terms of S. In the architectural context of [13], terms and rules represent states and state transitions, respectively.

A term s_1 can be rewritten to the term s_2, denoted by $s_1 \to s_2$, whenever there exist a subterm s_1' of s_1 that can be rewritten according to some rewrite rule into the term s_2' such that replacing the occurrence of s_1' in s_1 with s_2' gives s_2. A term that cannot be rewritten is said to be in *normal* or *canonical* form. The relation over S given by the previous rewrite mechanism is called the *rewrite relation* of R and is identified with $\to$. The inverse of this relation is denoted by $\leftarrow$ and its reflexive and transitive closure is denoted by $\to^*$ and its equivalence closure with $\leftrightarrow^*$.

The important notions of terminating property (or Noetherianity) and confluence are defined as usual (for a detailed presentation of rewriting theory see [4]):

a TRS is said to be *terminating* if there are no infinite sequences of the form $s_0 \to s_1 \to \ldots$;

a TRS is said to be *confluent* if for all *divergence* of the form $s \to^* t_1$, $s \to^* t_2$ there exists a term u such that $t_1 \to^* u$ and $t_2 \to^* u$.

The use of initial terms, S_0, representing possible initial states in the architectural context (which is not standard in rewriting theory) is simply to define what is a "legal" state according to the set of rewrite rules R; i.e., t is a legal term (or state) whenever there is an initial state $s \in S_0$ such that $s \to^* t$.

Using these notions of rewriting, one can model the operational semantics of the minimalistic RISC architecture $\mathcal{AX}$, where all arithmetic operations are performed on registers and only the *Load* and *Store* instructions can access memory, with respect to a base processor (single-cycle, non-pipelined, in-order execution model) [13].

For example, for the *Load-program-counter* instruction, denoted by $r := Loadpc$, the corresponding rewrite rule is:

$$\begin{array}{rl} Proc(ia, rf, prog) \to & Proc(ia+1, rf[r := ia], prog) \\ & \quad if \quad prog[ia] \;=\; r := Loadpc \end{array}$$

where the processor *Proc(pc,rf,prog)* consists of a program counter *pc*, a register file *rf* and a program *prog*, where r is a register of rf. The program counter ia, holds the address of the instruction to be executed. The register file is a set of registers, where each register has a register name and a value. The program is a set of instructions, in which each instruction is associated with an instruction address *ia*. Thus, the previous rule may be understood as: whenever the current instruction of the program, $prog[ia]$, is a *Load-program-counter* instruction of the form $r := Loadpc$, store the memory word addresed by the current *pc* into the specified register, r, in the register file and proceed to the next step (or equivalently increment the *pc*).

3 RISC processor rewrite based specification

In this section we briefly describe the $\mathcal{AX}$ RISC architecture [13], the specification in ELAN of a basic processor that implements this architecture and the specification of a more elaborated one that allows speculative execution over a reordering buffer (ROB).

Rules for the processors are specified in the ELAN environment and different architectural components as memory, registers, etc. are discriminated in a natural way taking advantage of this typed language.

3.1 $\mathcal{AX}$ RISC architecture

$\mathcal{AX}$ is a set of RISC instructions where all memory access is done by *Load* and *Store* instructions and the arithmetic operations are done over the registers. The instructions are executed in order and after each instruction execution the contents of the program counter (*pc*) is incremented by one except for branch (*Jz*) instructions. The set of instructions is described below.

$$\begin{array}{rl} INST \equiv & r{:=}Loadc(c) \parallel r{:=}Loadpc \parallel Jz(r_1, r_2) \parallel \\ & r{:=}Op(r_1, r_2) \parallel r{:=}Load(r_1) \parallel Store(r_1, r_2) \end{array}$$

The *load-constant* instruction, $r := Loadc(v)$, puts constant c into the register r. The *Load-program-counter* instruction, $r := Loadpc$, puts the content of the program counter into the register r. The *arithmetic-operation* instruction, $r := Op(r_1, r_2)$, performs the arithmetic operation specified by *Op* on the operands specified by the registers r_1 and r_2 and puts the result into the register r. The *branch* instruction $Jz(r_1, r_2)$, sets the program counter to the target instruction address specified by register r_2 when the contents of the register r_1 is zero and increment the program counter by one otherwise. The *load* instruction, $r := Load(r_1)$, loads the memory cell specified by register r_1 into register r. The *store* instruction, $Store(r_1, r_2)$, stores the contents of the register r_2 into the memory cell specified by the register r_1.

3.2 Specification of the basic processor

The operational semantics of the $\mathcal{AX}$ instruction set is defined by a single-cycle, non pipelined, in-order execution processor that we call the basic processor. Figure 1 shows the architecture at register transfer level - RTL.

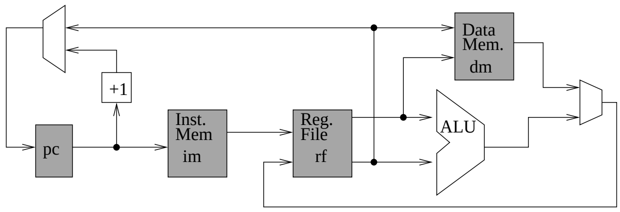

Figure 1: Description of the basic processor

The set of the rewriting rules for the basic processor specified in ELAN is given in the Table 1. The system `Sys` is described by its memory `m` and processor `Proc` which consists of a program counter `ia`, a register file `rf` and a program `prog`: `Sys(m,Proc(ia,rf,prog))`. Here *ia* is used for distingushing between the program counter *pc* and its contents, that is the current instruction address. For understanding these rules compare initially the rewriting rule for the *Load-program-counter* instruction explained at the end of the Section 2 with the **Loadpc** rule in the table specified in ELAN. The role of the "`where _ :=() _`" commands is to set auxiliary variables. Subsequently, we explain the rule **Jz** specified for the branch instruction *Jz*. Whenever the current instruction given by the instruction of the program `prog` at the position given by the program counter `ia` is a branch instruction (this is detected by the predicate `isinstJz(selectinst(prog,ia))`) of the form `Jz(r1,r2)`, the program counter should be changed by the contents of the register `r2` in the case the contents of register `r1` equals to zero (detected by the predicate `valueofReg(r1,rf) == 0`) and by `ia+1` otherwise. All other instructions are specified in a similar way.

3.3 Specification of a speculative processor

More sophisticated processors can be described using rewriting rules. Here, we describe the implementation of a processor with speculative execution over a ROB. Figure 2 illustrates its architecture in a RTL level. A reordering buffer holds instructions that have been decoded but have no completed their execution. Conceptually, a ROB divides the processor into two asynchronous parts. The first one fetches the instruction and after decoding and renaming registers, dumps it into the next available slot in the ROB. The second one takes any enabled instruction out of the ROB and dispatches it to an appropriate functional unit, including the memory system. We can consider the ROB as a list of instruction templates (each into one ROB slot). Each template contains the instruction address, opcode and operands. Instructions that update a register have an additional field $Wr(r)$, that records the destination register r. In branch instructions the *Sp*(*pia*) field holds the speculated instruction address *pia* that will be used to determine the correctness of the prediction. Memory address instructions contain an extra flag to indicate whether the instruction is waiting to be dispatched (U) or has already been dispatched (D). The index of each ROB

```
[Loadc] Sys(m,Proc(ia,rf,prog)) =>
 Sys(m,Proc(ia+1,insertRF(rf,r,v),prog))
  where instIa :=() selectinst(prog,ia)
  if isinstLoadc(instIa)
  where r :=() nameofLoadc(instIa)
  where v :=() valueofLoadc(instIa)          end
[Loadpc] Sys(m,Proc(ia,rf,prog)) =>
 Sys(m,Proc(ia+1,insertRF(rf,r,ia),prog))
  where instIa :=() selectinst(prog,ia)
  if isinstLoadpc(instIa)
  where r :=() nameofLoadpc(instIa)          end
[Op] Sys(m,Proc(ia,rf,prog)) =>
 Sys(m,Proc(ia+1,insertRF(rf,r,v),prog))
  where instIa :=() selectinst(prog,ia)
  if isinstOp(instIa)
  where r1 :=() reg1ofOp(instIa)
  where r2 :=() reg2ofOp(instIa)
  where r :=() nameofOp(instIa)
  where v:=() valueofOp(r1,r2,rf)            end
[Jz] Sys(m,Proc(ia,rf,prog)) =>
 Sys(m,Proc(nia,rf,prog))
  where instIa :=() selectinst(prog,ia)
  if isinstJz(instIa)
  where r1:=() reg1ofJz(instIa)
  where r2:=() reg2ofJz(instIa)
  choose try where nia:=()ia+1
               if valueofReg(r1,rf)!=0
          try where nia:=()valueofReg(r2,rf)
               if valueofReg(r1,rf)==0
  end                                        end
[Load] Sys(m,Proc(ia,rf,prog)) =>
 Sys(m,Proc(ia+1,insertRF(rf,r0,v0),prog))
  where inst :=() selectinst(prog,ia)
  if isinstLoad(inst)
  where r0 :=() nameofLoad(inst)
  where v0 :=() getMem(inst,rf,m)            end
[Store] Sys(m,Proc(ia,rf,prog)) =>
 Sys(insertMEM(m,valueofReg(rA,rf),
 valueofReg(rB,rf)),Proc(ia+1,rf,prog))
  where inst :=() selectinst(prog,ia)
  if isinstStore(inst)
  where rA :=() nameofStoreR1(inst)
  where rB :=() nameofStoreR2(inst)          end
```

Table 1: Rules for the basic processor

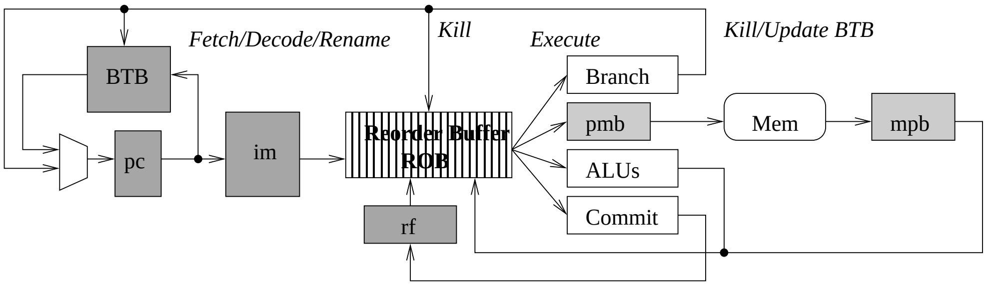

Figure 2: Description of the speculative processor

slot serves as a renaming tag and the templates in the ROB always contain tags or values instead of register names.

Memory access is done through the processor-memory-buffer (pmb) and the memory-processor-buffer (mpb). The address of a speculative instruction is determined by consulting the branch target buffer (BTB). If the prediction is wrong the speculative instruction and all the instructions issued thereafter are abandoned and their future effects on the processor state nullified. Since in this architecture there is no parallelism, when a speculated address is wrong the effect is to eliminate the instruction and the next ones in the ROB. The syntax of the set of instruction templates of the speculative processor is shown below.

$$\begin{aligned}
\text{ROB } \textit{ENTRY} \;\equiv\; & \mathit{Itb}(ia, t := v, Wr(r)) \parallel \\
& \mathit{Itb}(ia, t := \mathit{Op}(tv_1, tv_2), Wr(r)) \parallel \\
& \mathit{Itb}(ia, t := \mathit{Jz}(tv_1, tv_2), \mathit{Sp}(pia)) \parallel \\
& \mathit{Itb}(ia, t := \mathit{Load}(tv_1, mf), Wr(r)) \parallel \\
& \mathit{Itb}(ia, t := \mathit{Store}(tv_1, tv_2, mf))
\end{aligned}$$

Itb stands for instruction template buffer (ITB) and t's and v's for either a tag or a value. mf is the memory flag that can be dispatched (D) or is waiting to be dispatched (U).

The complete set of rewriting rules for the speculative processor implemented in ELAN is given in [3]. Here we explain the operational semantics of three of these rules: `PsOp`, `PsJzIssue` and `PsJumpCorrectSpec`, whose specification is given in the Table 2.

```
[PsOp] Sys(m,Proc(ia,rf,ITB(ia1,k,t(k)|-Op(v,v1),wf,sf).itbs2, btb, prog)) =>
 Sys(m,Proc(ia,rf,ITB(ia1,k,t(k)|-execOponval(v,v1),wf,sf).itbs2, btb,prog))          end
[PsJzIssue] Sys(m,Proc(ia,rf,itbs,btb,prog)) =>
  Sys(m,Proc(pia,rf, insEndITBs(ITB(ia,k,Jz(k0,k1),NoWreg,Spec(pia)),itbs),btb,prog))
       where instIa :=() selectinst(prog,ia) if isinstJz(instIa)
       where r1 :=() reg1ofJz(instIa) where r2 :=() reg2ofJz(instIa)
       where k :=() lengthof(itbs)+1
       where k0 :=() searchforLastTag(r1,rf,itbs)
       where k1:=()searchforLastTag(r2,rf,itbs)
       where pia:=()getbtb(ia,btb)                                                      end
[PsJumpCorrectSpec] Sys(m,Proc(ia,rf,ITB(ia1,k,Jz(0,nia),wf,Spec(pia)).itbs,btb,prog)) =>
  Sys(m,Proc(ia,rf,itbs,btb,prog)) if pia==nia                                          end
```

Table 2: Examples of implemented rules for the speculative processor

Arithmetic operation and value propagation rules deal with the computation of arithmetic operations (`PsOp`), the propagation of its results through the ITB and the exclusion of an instruction template from the ITB when the result had already been solved and committed to the register file (this means that the renaming tag it addresses does not occur in the ROB anymore). A value is only committed to the register file when the instruction referencing it is on the head of the ITB. This approach is conservative since it avoids the need to reconstruct the state of the register file in the event of wrong speculations.

Rules `PsJzIssue` and `PsJumpCorrectSpec` (Table 2), belong respectively to the set of issue rules and to the set of branch completion rules. Issue rules are those used for the issuing of the instructions which generate templates stored in the ITB and branch completion rules are those which deal with the resolution of speculations. When a branch instruction is issued the processor has to know which will be the next instruction to be fetched. The next predicted instruction is indicated by the BTB that is an indexed list. Let us suppose the program instruction in the position `ia` is being issued, the next value of the program counter, here we will called it `pia`, is looked up in the BTB using as index the current program counter (`pia :=() getbtb(ia,pia)`) and then the execution resumes at the `pia` value. When the ITB element containing the branch instruction reaches the head of the ITB it is the time to check if the speculation was done correctly or if the processor needs to fix the mistake and restart the execution at the correct program counter value, simply by ignoring the remaining instructions already in the ITB. The rules `PsJumpCorrectSpec`, `PsJumpWrongSpec`, `PsNoJumpCorrectSpec` and `PsNoJumpWrongSpec` deal with this issue (see [3]). Exemplifying, lets say the head of the ITB is of the form `ITB(ia,k,Jz(v,nia),wf,Spec(pia))`, the rule has to check whether the value `v` is zero or not and then, respectively, check whether either the speculated address `pia` coincides with `nia` or with `ia+1`. In this event the prediction has been proved correct and the execution resumes. Otherwise the program counter must be set, respectively, either to the value of `ia+1` or to the correct value of the branch represented by `nia`, depending on whether the wrong speculation was a branch or not, and the ITB must be completely emptied because the remaining instructions should not be executed. These rules also control the updating of the BTB for dynamic speculation (through the `changebtb` rule).

As previously mentioned, one useful feature of rewrite based design of processors is the possibility to prove the correctness of the implementation of the specified instruction set. The main idea, according to [13, 14, 1], is to design a function that can extract all the programmer visible states, i.e., the program counter, the register file and the memory from the system. In particular, it is easy to show that the speculative processor simulates the basic one. In fact, a basic processor term can be "upgraded" to one of the speculative processor simply by adding an empty ITB and an arbitrary BTB to the processor. Contrariwise, the key observation is that during the execution over an speculative processor, if no instruction is issued then the ITB will soon become empty. Only instruction issue rules can further expand the ITB. Thus, we can define another rewriting system which uses the same grammar as the speculative processor and include all its rules except the instruction issue ones [3].

3.4 Specification of a pipelined processor

To give a flavor about the way pipelined processors can be specified by rewriting rules, we illustrate how the phases of execution of an instruction of the processor can be specified by rewriting rules in ELAN. The main objective of presenting this simple description here is to convince the reader that the designer may refine the processor specification in the level of detail he wishes.

In pipeline processors the execution of an instruction is divided into a set of successive subtasks. For instance, the overall execution of one $\mathcal{AX}$ instruction can be divided into three subtasks: *fetch*, *decode* and *execute*. To each subtask is associated one pipeline stage. Each stage operates in parallel and synchronously to the others. All the pipeline stages operate like an assembly line, that is, receiving their input typically from the previous stage and delivering their outputs to the next stage. In the Table 3 we present the ELAN rules for the pipeline phases of the instructions `Loadc, Loadpc` and `Op`. The other instructions may be similarly implemented. Here we only present the processor `Proc` containing arguments which discriminate between the instruction and data memory: `Im` and `dm`. The former is described by the address of the current instruction `a`, the instruction itself `inst` and the contents of the instruction memory `cm`: `Im(a,inst,cm)`.

```
[Fetch] Proc(ia,Im(a,inst,cm),rf,alu,dm) => Proc(ia+1,Im(ia,FetchInst(ia,cm),cm),rf,alu,dm)
end
[Decode] Proc(ia,Im(a,inst,cm),
            Rf(oldvalue1,oldvalue2,oldvalue3,oldvalue4,oldvalue5,oldvalue6,reg),alu,dm) =>
 Proc(ia,Im(a,inst,cm),Rf(firstop,secondop,regdest,firstvalue,secondvalue,data,reg),alu,dm)
       where firstop:=()DecodeOp(1,inst,a) where secondop:=()DecodeOp(2,inst,a)
       where regdest:=()DecodeOp(3,inst,a) where firstvalue:=()ValueOfReg(firstop,reg)
       where secondvalue:=()ValueOfReg(secondop,reg) where data:=()DecodeOp(4,inst,a)    end
[Execute] Proc(ia,Im(a,inst,cm),Rf(firstop,secondop,regdest,firstvalue,secondvalue,data,reg),
             Alu(oper,op1,op2,opout),dm) =>
 Proc(ia,Im(a,inst,cm),
      Rf(firstop,secondop,regdest,firstvalue,secondvalue,data,Ins(reg,regdest,opresult)),
      Alu(1,firstvalue,secondvalue,opresult),dm)
       where opresult:=()op(1,firstvalue,secondvalue) if isinstAdd(inst)            end
[Execute] Proc(ia,Im(a,inst,cm),Rf(firstop,secondop,regdest,firstvalue,secondvalue,data,reg),
alu,dm) =>
 Proc(ia,Im(a,inst,cm),
      Rf(firstop,secondop,regdest,firstvalue,secondvalue,data,Ins(reg,regdest,data)),alu,dm)
       if isinstLoadpc(inst)                                                          end
[Execute] Proc(ia,Im(a,inst,cm),Rf(firstop,secondop,regdest,firstvalue,secondvalue,data,reg),
alu,dm) =>
 Proc(ia,Im(a,inst,cm),
      Rf(firstop,secondop,regdest,firstvalue,secondvalue,data,Ins(reg,regdest,data)),alu,dm)
       if isinstLoadc(inst)                                                           end
```

Table 3: Examples of implemented rules for pipelined processors

In the rule **Fetch**, the function `FetchInst(ia,cm)` fetches the code of the instruction at address `ia` from the instruction memory `cm`. In this way instructions can be given more realistically by binary codes and not symbolically as in the previous specifications. The instruction is divided in fields from which operands are selected.

In the rule **Decode**, the function `DecodeOp` decodes the register addresses of the first and second operand and the destination register. In this function each register is identified by a different index. The function `ValueOfReg` selects the contents of the first and second operands given by the corresponding registers from the list of registers `reg` of the register file `Rf`. The function `DecodeOp` is also used to obtain the constant value of the current instruction (used in the `Loadc` and `Loadpc` instructions), this value is put in the `data` register. Observe that in this specification the register file consists of six fields for these register addresses and values and the list of registers itself: `Rf(_,_,_,_,_,_,reg)`.

In contrast to the fetch and decode phases, which are identical for the three instructions, the execute phase should discriminate the instructions of the processor. In fact, there are three **Execute** rules each corresponding to one of the three illustrated instructions: `Loadc, Loadpc` and `Op`. The **Execute** rules are conditional rewriting rules and the premises `isinstLoadc, isinstLoadpc` and `isinstAdd` are satisfied according to the instruction `inst` being processed. Observe that except for the conditions, the **Execute** rules for `Loadc, Loadpc` are identical and consequently they can be merged into a sole rule with the premise `isinstLoadc(inst)` or `isinstLoadpc`. The **Execute** rule for the `Op` instruction uses the arithmetic logic unit `Alu` to produce the result of the abstract operation between the `firstvalue` and `secondvalue` operands. The result is computed via the `op` function and set in the `opresult` variable. The `Alu` combined with the `op` function enable the execution of different arithmetic operands. The case here presented is the one of the addition that is discriminated by the "1" as first operand of the `Alu` and of the `op` function.

As here presented rewriting rules can be applied in any ordering. In the next section we will explain how logical strategies are used to give the correct ordering to the simulation of the phases of the execution of instructions of pipelined processors.

Well-known problems such as pipeline stalls caused by RAW dependencies and their typical solutions such as bypassing used to solve define-use and load-use conflicts [15] could be specified and simulated in our rewriting-logic approach.

4 Results

4.1 Rewriting-logic based simulation

The natural separation available in ELAN between rewriting and logic strategies makes it possible to control in an adequate way the application of rules and for many aspects of hardware the controlled simulation of them. For instance, one of the basic hardware aspects in our implementation of the speculative processor that can be controlled by the logic and strategies is the size of the buffer. It is enough to define adequate strategies that control the application of a limited (by the size of the desired buffer to be simulated) number of the issue rules. Suppose you want to simulate a ROB of size n, whose control is done by filling and emptying it completely alternately. Then the following strategy can be used:

$$\mathit{repeat} * \left(\begin{array}{l} \mathit{first\ one}(\mathit{issue\_rules}); \\ \quad \left.\begin{array}{c} \mathit{first\ one}(\mathit{issue\_rules} \cup \mathit{id}); \\ \vdots \\ \mathit{first\ one}(\mathit{issue\_rules} \cup \mathit{id}); \end{array}\right\} \quad n\text{-}1 \\ \mathit{normalise}(\mathit{first\ one}(\mathit{non\_issue\_rules})) \end{array} \right)$$

where $firstone$ is a strategy that takes, among several candidates, the first rule that apply. In a similar way one can implement other strategies for controlling the ROB in different forms [3]. For example, for maintaining a ROB of size n filled during the whole execution, one can start as before and between the normalization via non issue rules all these rules are treated individually according to being rules that either maintain or decrease the number of instruction templates in the ROB. For instance, after a wrong branch speculation the ROB is emptied and immediately it should be filled. For giving the correct ordering to the simulation of the execution of the phases of the instructions of pipelined processors we use the following obvious strategy:

$$\mathit{repeat} * (\texttt{Fetch; Decode; Execute})$$

In contrast to the control of the ROBs other interesting aspects of speculative processors like the method of branch prediction may be controlled directly by the rewriting rules. The advantages of having ROBs is that instruction templates may be charged and these templates partially executed by the pipeline control, until the branch is resolved. When a branch instruction template `Jz(r1,r2)` is charged into the ROB, the next instruction template to be loaded is unknown since at this point of the execution the values of the tags associated with the registers `r1` and `r2` are not necessarily resolved. In speculative processors a decision about which is the next instruction to be executed must be taken according to the contents in a table known as the *branch template buffer* (`BTB`). Some well-known dynamic branch prediction schemes are easy to simulate by including simple rewrite rules. We mention two of these that are called *1-bit* and *2-bit dynamic prediction* [15]. An initial prediction is given in the `BTB`. You can explicitly give, for instance, pairs $(1, 2), ..., (j, j+1), ..., (n, n+1)$ meaning that after execution of each rule the initial prediction is to jump to the following instruction (actually, this is only necessary for branch instructions). Predictions are based on the execution history. In 1-bit dynamic prediction, the predictions for the n^{th} instruction

Size	10 ran	10 ord	20 ran	20 ord	30 ran	30 ord
1-bit correct	51	60	128	225	218	490
wrong	29	34	66	74	106	114
2-bit correct	50	73	134	258	216	543
wrong	30	21	60	41	108	61
Size	40 ran	40 ord	50 ran	50 ord		
1-bit correct	324	855	398	1320		
wrong	148	154	194	194		
2-bit correct	323	928	404	1413		
wrong	149	81	188	101		

Table 4: ELAN statistics for quick-sort executed with 1-bit and 2-bits dynamic predictions

are based on the last branch, indicating if it was taken or not. Once one detects that a prediction fails the corresponding value in the `BTB` is changed to the correct address of the next instruction to be executed. The 2-bits dynamic prediction can be modeled as a finite state machine with four different states for a prediction: *strongly taken, weakly taken, weakly not taken, strongly not taken.* If the branch prediction is either *strongly taken* (*strongly not taken*) or *weakly taken* (*weakly not taken*) and the prediction is correct, then the state is changed to *strongly taken* (*strongly not taken*). If the branch prediction is *strongly taken* (*strongly not taken*) and fails, then the state is changed to *weakly taken* (*weakly not taken*) . If the branch prediction is *weakly taken* (*weakly not taken*) and fails, then the `BTB` is changed to the next instruction (to the address of the jump) and the state of the prediction is changed to be *weakly not taken* (*weakly taken*). Of course, the rewrite based manipulation of these strategies controls only the own strategy, but not the way in that the `BTB` has to be updated once a prediction fails.

4.2 Analysis of performance of processors

We can estimate and compare the performance of different implementations of $\mathcal{AX}$ architecture by codifying programs in $\mathcal{AX}$ assembly language and executing them with the ELAN description of the processors. Some algorithms like quick-sort, generation of the Fibonacci sequence and the computation of the Knuth-Morris-Pratt jump function were used for this purpose.

The performance of proposed processors or of different ways to implement them may be determined by analyzing the ELAN statistics. For instance, one can estimate whether 1-bit performs better than 2-bits prediction in the execution of an assembly description of quick-sort over the speculative processor implemented with the strategy of filling and emptying alternatively the ROB. The total number of wrong and correct predictions with the two methods for ordered and random lists are given in the Table 4. Observation of the differences between the wrong number of predictions for both methods gives an important insight about the advantages of 2-bits over 1-bit prediction, since in the worst case a wrong prediction flushes ROB which has been filled with instruction templates over which previous operations have been executed. One can check on the table that the difference in the number of wrong predictions is much more significant with ordered lists than with random lists.

Another advantage of rewrite based descriptions is that, according to the strategy to be adopted, the rules may be selected in a non deterministic way. This is specially useful when modeling the natural concurrency of hardware modules. For example, out-of-order execution of instruction templates on ITB may be simulated by allowing true nondeterministic application of rewrite rules over ITB during any time of the execution. For that, instead of the usual CONS operator "`.`" of instruction templates and ITBs (which appears as

[PsOp]`Sys(m,Proc(ia,rf,itbs1#ITB(ia1,k, t(k)|-Op(v,v1),wf,sf)#itbs2,btb,prog)) =>` `Sys(m,Proc(ia,rf,itbs1#ITB(ia1,k,t(k)|-execOponval(v, v1),wf,sf)#itbs2,btb, prog)) end`

Table 5: Non deterministic rewrite rule for `[PsOp]`

`inst_temp.itbs` in our implementation) one can define a new operator "`#`" for concatenating ITBs and/or instruction templates. Then ITBs are represented as `itbs1#inst_temp#itbs2` being `itbs1` and `itbs2` lists of instruction templates and `inst_temp` a sole instruction template. Rules of our rewriting system are modified by replacing all their ITBs with this new representation as we illustrate by showing the new rule for `[PsOp]` in the Table 5. The new `[PsOp]` rule may be applied by matching `ITB(ia1,k,t(k)|-Op(v,v1),wf,sf)`, the instruction template, not only at the first but at any position of the current ITB: `itbs1#ITB(ia1,k, t(k)|-Op(v,v1),wf,sf)#itbs2`.

The rewriting system obtained by changing all rules as suggested above solves the problem of having out-of-order execution, since in the theory rules are applied nondeterministically. But in the practice, in purely rewrite based systems, this solution does not work since the application of a rule is decided by searching for either left-most or right-most (innermost) redices over the ITBs (according to the way the constructor "`#`" is defined) [9]. To make effective use of the natural concurrency of rewriting-logic descriptions, availability of true nondeterministic strategies are necessary to decide which rule to apply and at which position. With some additional effort, ELAN strategy constructors like *don't know choose* (that gives all possible reductions) can be adapted to simulate the needed nondeterminism of the ROBs [16, 10, 11].

5 Future work

5.1 Circuit design

Circuit design may be addressed by rewriting. In the sequel we illustrate how rewriting may be used as an assistant tool for deducing appropriate algebraic terms with many regularities that result adequate for circuit design. We use an example of multiplication presented in [2]. The desired multiplication may be established as $\sum_{i=0}^{3} 2^i \cdot (y_i \cdot X)$ that is equal to $2^0 \cdot (y_0 \cdot X) + 2^1 \cdot (y_1 \cdot X) + 2^2 \cdot (y_2 \cdot X) + 2^3 \cdot (y_3 \cdot X)$.

Properties of shift-left (*shl*) and shift-right (*shr*) binary operators used in algebraic circuits synthesis can be naturally described in a rewriting-based language. The set of rewriting rules below describes some properties of these operators over binary number variables w, u and v, that will be further showed useful in this context.

$$\begin{array}{ll} 2 \cdot w \rightarrow \mathit{shl}(w) & \mathit{shr}(u) + \mathit{shr}(v) \rightarrow \mathit{shr}(u+v) \\ \mathit{shl}(\mathit{shr}(w)) \rightarrow w & \mathit{shl}(u) + \mathit{shl}(v) \rightarrow \mathit{shl}(u+v) \\ \mathit{shr}(\mathit{shl}(w)) \rightarrow w & \end{array}$$

In addition to the rules for *shl* and *shr*, algebraic rules for the operation of selection of the i^{th} less significant bit of a binary number Y, $\Pi(Y, i)$, can be stated as

$$\Pi(Y, 0) \rightarrow \mathit{lsb}(Y) \qquad \Pi(Y, \mathit{succ}(n)) \rightarrow \Pi(\mathit{shr}(Y), n)$$

where *lsb* denotes the least significant bit operator and *succ* and 0 are the usual constructors for the natural numbers. Then y_0 corresponds to $\mathit{lsb}(Y)$; y_1 to $\mathit{lsb}(\mathit{shr}(Y))$, y_2 to $\mathit{lsb}(\mathit{shr}^2(Y))$ and y_3 to $\mathit{lsb}(\mathit{shr}^3(Y))$.

By applying the rewrite rules for *shl* and *shr*; the multiplication can be formulated as

$$\begin{array}{l}\sum_{i=0}^{3} 2^i \cdot (y_i \cdot X) \to^* \\ y_0 \cdot X + shl(y_1 \cdot X) + shl^2(y_2 \cdot X) + shl^3(y_3 \cdot X) \to^* \\ y_0 \cdot X + shl(y_1 \cdot X + shl(y_2 \cdot X + shl(y_3 \cdot X)))\end{array}$$

were by shl^n and shr^m we denote n and m compositions of the operators *shl* and *shr*, respectively.

Replacing adequately the previous patterns at the expression $y_0 \cdot X + shl(y_1 \cdot X + shl(y_2 \cdot X + shl(y_3 \cdot X)))$, we obtain

$$\begin{array}{l}\sum_{i=0}^{3} 2^i \cdot (y_i \cdot X) \to^* \\ y_0 \cdot X + shl(y_1 \cdot X) + shl^2(y_2 \cdot X) + shl^3(y_3 \cdot X) \to^* \\ y_0 \cdot X + shl(y_1 \cdot X + shl(y_2 \cdot X + shl(y_3 \cdot X)))\end{array}$$

from which we can notice the regularities that will be useful for the construction of the desired circuit schema:

$$\begin{array}{l}lsb(Y) \cdot X + \underbrace{shl(lsb(shr(Y)) \cdot X +}_{CondAddShift} \\ \underbrace{shl(lsb(shr^2(Y)) \cdot X +}_{CondAddShift} \ \underbrace{shl(lsb(shr^3(Y))) \cdot X + \vec{0}}_{CondAddShift})))\end{array}$$

In order to obtain more regularity than in the previous expression we can precede the whole expression with "*shr*(*shl*(". But the main problem of the resulting expression is that the first *CondAddShift* port has as input y_3, that is the last of the four bits of Y that can be computed with sequences of the form $lsb(shr(shr(\ldots(Y)\ldots)))$. To avoid this problem from the pure beginning observe that

$$\begin{array}{l}\sum_{i=0}^{3} 2^i \cdot (y_i \cdot X) = 2^4 \cdot \sum_{i=0}^{3} shr^{4-i}(y_i \cdot X) \to^* \\ shl^4(shr(y_3 \cdot X) + shr^2(y_2 \cdot X) + \\ \qquad\qquad shr^3(y_1 \cdot X) + shr^4(y_0 \cdot X)) \to^* \\ shl^4(shr(y_3 \cdot X + shr(y_2 \cdot X + shr(y_1 \cdot X + shr(y_0 \cdot X))))) \\ = shl^4(shr(lsb(shr^3(Y)) \cdot X + shr(lsb(shr^2(Y)) \cdot X + \\ \qquad\qquad shr(lsb(shr(Y)) \cdot X + shr(lsb(Y) \cdot X)))))\end{array}$$

Regularities of the internal expression (without the external shr^4) can be described as

$$\begin{array}{l}\underbrace{shr(lsb(shr^3(Y)) \cdot X +}_{CondAddShift} \ \underbrace{shr(lsb(shr^2(Y)) \cdot X +}_{CondAddShift} \\ \underbrace{shr(lsb(shr(Y)) \cdot X +}_{CondAddShift} \ \underbrace{shr(lsb(Y) \cdot X + \vec{0})}_{CondAddShift} \ )))\end{array}$$

And from that expression one can straightforwardly build and schema for the desired circuit schema presented in the Figure 3.

One of the interesting particularities of rewriting that emerges in this context is that rewriting should be directed to normal forms that are not *simplified* as usual. Namely, rewriting should be guided here in such a way that the obtained canonical forms are *simple* from the point of view of hardware implementation. That is not the standard in what we could call *classical algebraic rewriting*, where terms are simplified to *shorter* and simpler forms.

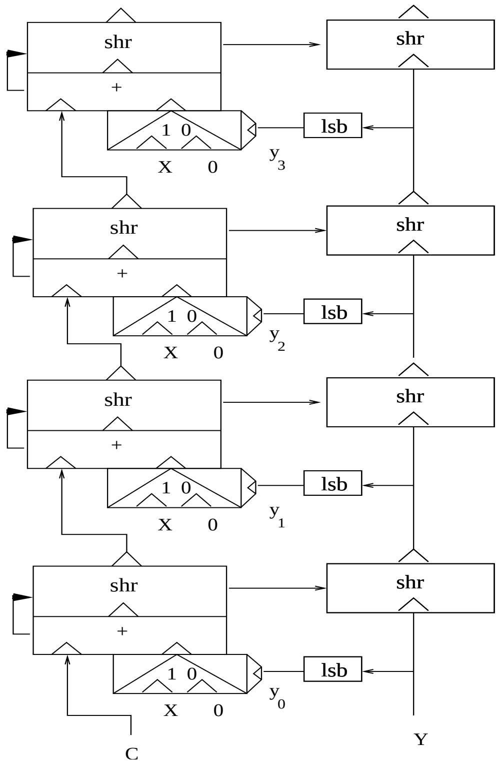

Figure 3: Circuit schema for 4-bit numbers multiplication

5.2 Reconfigurable architectures

We are currently investigating the modeling of more complex processor organizations, and future research will address in particular modeling and simulation of reconfigurable architectures [6], which are non-standard models of computing where two layers of programming are needed (the one for the run time operation and the other for the processor reconfiguration). Two kinds of machine paradigms have to be explored: reconfigurable instruction stream processors as well as data stream processors based on very powerful reconfigurable data path arrays (rDPUs) [7]. Working on data stream processors also includes distribuited memory architectural exploration. This all is of great interest, since no simulation is possible over standard hardware description languages such as Verilog and VHDL. For discriminating these two layers higher-order rewriting-logic based simulation appears to be an adequate theoretical framework.

6 Conclusions

We showed how processors may be specified using rewriting-logic systems and illustrated why the rewriting part as well as the logical part of ELAN result adequate for the simulation of hardware components, using as an example the simulation of branch prediction in speculative processors (that was done in our case by pure rewriting) and the control of the size of ROBs (which was done in our case by logic strategies). After having specified the rules for the instructions of a processor, the intrinsic separation between logic and rewriting in ELAN results versatile enough for dealing with different ROBs designs without additional effort over these rewrite specification. Additionally, we illustrated how statistics of application of rewrite rules may be used for estimating and comparing performance of different processors.

Through rewriting-logic one can describe an architecture as precisely as one wants. For example, rules of the speculative processor may be atomized in order to reflect the behavior of lower-level hardware components as pipelines and functional units of processors like fetch, decode and execution units as shown in the Section 3.4.

References

[1] Arvind and X. Shen. Using Term Rewriting Systems to Design and Verify Processors. Technical Report 419, Laboratory for Computer Science - MIT, 1999. in IEEE Micro Special Issue on "Modeling and Validation of Microprocessors", 1999.

[2] M. Ayala-Rincón, R. W. Hartenstein, R. Jacobi, and C. Llanos. Designing Arithmetic Digital Circuits via Rewriting-Logic. Available at `www.mat.unb.br/~ayala/publications.html`, 2002.

[3] M. Ayala-Rincón, R. M. Neto, R. Jacobi, C. Llanos, and R. W. Hartenstein. Applying ELAN Strategies in Simulating Processors over Simple Architectures. In B. Gramlich and S. Lucas, editors, Proc. 2^{nd} *Workshop on Reduction Strategies in Rewriting and Programming*, *ENTCS* volume 70(6):16 pages. Elsevier 2002.

[4] F. Baader and T. Nipkow. *Term Rewriting and* All That. Cambridge University Press, 1998.

[5] P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, and C. Ringeissen. An overview of ELAN. In C. Kirchner and H. Kirchner, editors, *ENTCS*, volume 15. Elsevier, 1998.

[6] R. W. Hartenstein. A decade of reconfigurable computing: a visionary retrospective. In *Proceedings of the DATE 2001 on Design, automation and test in Europe*, pages 642–649. IEEE Press, 2001.

[7] R. W. Hartenstein. (Invited paper): Trends in Reconfigurable Logic and Reconfigurable Computing. In *Proceedings of the Ninth IEEE Int. Conf. on Electronics, Circuits and Systems - ICECS 2002*, Dubrovnik (Croatia), 2002.

[8] J. C. Hoe and Arvind. Hardware Synthesis from Term Rewriting Systems. Technical Report 421 A, Laboratory for Computer Science - MIT, 1999.

[9] H. Hussmann. *Nondeterminism in Algebraic Specifications and Algebraic Programs.* Birkhäuser, 1993.

[10] H. Kirchner and P.-E. Moreau. Non-deterministic computations in ELAN. In J. Fiadeiro, editor, *Recent Developments in Algebraic Specification Techniques, Proc. 13th WADT'98*, volume 1589 of *LNCS*, pages 168–182. Springer, 1998.

[11] H. Kirchner and P.-E. Moreau. Promoting Rewriting to a Programming Language: A Compiler for Non-Deterministic Rewrite Programs in Associative-Commutative Theories. *Journal of Functional Programming*, 11(2):207–251, 2001.

[12] D. E. Knuth and P. B. Bendix. *Computational Problems in Abstract Algebra*, chapter Simple Words Problems in Universal Algebras, pages 263–297. J. Leech, ed. Pergamon Press, Oxford, U. K., 1970.

[13] X. Shen and Arvind. Design and Verification of Speculative Processors. Technical Report 400 A, Laboratory for Computer Science - MIT, 1998.

[14] X. Shen and Arvind. Modeling and Verification of ISA Implementations. Technical Report 400 B, Laboratory for Computer Science - MIT, 1998.

[15] D. Sima, T. Fountain, and P. Kacsuck. *Advanced Computer Architectures: a Design Space Approach.* Addison-Wesley, 1997.

[16] M. Vittek. A Compiler for Nondeterministic Term Rewriting Systems. In H. Ganzinger, editor, *Proc. Seventh Int. Conf. on Rewriting Techniques and Applications RTA-96, New Brunswick, NJ, USA*, volume 1103 of *LNCS*, pages 154–168. Springer, July 1996.