A formal method for provably correct composition of a real-life processor out of basic components (The APE100 Reverse Engineering Study)

Egon Börger
Dipartimento di Informatica
Università di Pisa
56125 Pisa, Italy
boerger@di.unipi.it

Giuseppe Del Castillo
FB17 Informatik
Universität Paderborn
33095 Paderborn, Germany
giusp@uni-paderborn.de

Abstract

We present a design approach which allows us to formally specify a real–life processor as composed out of its basic architectural (formally specified) components. The methodology provides means to rely upon hierarchical refinements and modular structuring of the specifications as a discipline to control the behaviour of complex units in terms of the behaviour of their components. In particular this enables us to prove interesting dynamic properties about the processor in terms of properties of its basic architectural components. We have developed the method to accomplish a reverse engineering project for the VLSI implemented microprocessor zCPU, the controller of the successful APE100 massively parallel machine.

1 Introduction

The APE100 massively parallel processor has been built by a group of physicists of the INFN (Istituto Nazionale di Fisica Nucleare) as a dedicated machine for floating point intensive scientific applications and has proved to be rather successful for numerical simulations in Lattice Gauge Theory [1]. As preparation for a possible upgrade to a new APE1000 machine, we have accepted the challenging reverse engineering task to construct formal models for the architecture in such a way that the upgrading process can be guided by these models. The models are intended to provide precise descriptions between the existing block diagrams and verbal explanations on one side and the C-code for the APE100 simulator on the other side; they can be used to produce executable prototypes and offer the possibility to experiment with design decisions at various levels of abstraction.

We have developed a series of formal models, at different levels of abstraction, which correspond to views of the architecture as provided by different languages in the APE100 compilation chain. The ground model APESE has been defined in [2]; it reflects the APE100 model of parallel execution as viewed by the user who approaches the machine as programmer in the high level parallel programming language Apese, a parallel Fortran like user expandable language especially designed for APE100. We have transformed this model by stepwise refinement to a provably correct model LEX (loadable executable code) of APE100 at the hardware level, going through mainly two other intermediate models Assembler and ZIC (zCPU intermediate code) which correspond to languages of the APE100 compilation chain.

We concentrate our attention in this paper on the VLSI superscalar integer processor zCPU [3] which acts as controller for APE100 and represents the most original part of the project (including the pipelining and VLIW parallelism for the execution of compiled ZIC code). In section 1 we explain how the standard architectural components of zCPU can be described formally. In section 2 we show how given units can be composed in a precise way to complex units. Using well known techniques from the literature (see [4]) the composition can be done in a modular way. Defining the components as evolving algebras (in the sense of Gurevich [5]) to which we add entries and exits allows us to adopt also the evolving algebra refinement techniques which have been used successfully to formally specify and prove properties of complex systems (see for example [6, 7, 8]).

In a sequel to this paper we explain how the model LEX of the zCPU can be used to make the following informal statement into a precise mathematical assertion and to prove it.

Theorem. Under precisely stated assumptions on the compiler, the model LEX of the processor zCPU executes compiled Apese programs correctly.

We want to convince the practitioner by an example from real-life that: (i) one can use the evolving algebra specification methodology to produce readable but nevertheless precise specifications without previous formal training and without formal overhead; (ii) the evolving algebra specification method scales to complex systems.

For a full account of this paper we refer the reader to [9]. For a formal definition of the underlying semantics of evolving algebras see [5]. For a tutorial introduction to the evolving algebra specification method see [10].

2 The Datapath Components of the zCPU

The zCPU processor is built out of several main units, namely: the register file RF, the arithmetical-logical unit ALU_MPY_DIV (ALU for brevity), the condition code and status register unit CC&STATUS, the address generation unit AGU, the input/output subsystem IOS, the data memory DATAMEM and the program memory PROGMEM. Additionally, an instruction decoding unit DECODE and some internal registers are needed to coordinate the operation of the above units.

Each unit is specified formally as an evolving algebra (in the sense of [5]) with entries and exits. The latter are vehicles for an explicit description of a desired input/output behaviour. This behaviour is defined by finitely many rules of the evolving algebra and possibly some conditions on the functions which appear in the rules. Each rule is of the form

$$\textbf{if } \mathit{Cond} \textbf{ then } \mathit{Updates}$$

where *Cond* is a first-order expression and *Updates* a finite set of function updates

$$f(t_1, \dots, t_n) := t$$

which are executed *simultaneously* each time *Cond* is true.[1] For the description of the parallelism in APE100 it is convenient to rely upon the lock-step interpretation of evolving algebras under which in each step each rule which can be applied is applied. (For an exact definition of this lock-step semantics of evolving algebras see [5]).

The entries and exits can be viewed as 0-ary functions. Each function can be constrained by conditions, which can serve various purposes. For example, exits are often defined by equations; in the special case of a combinatorial unit all of them are defined as functions of only entries. Another use of conditions on functions are (integrity) contraints, which are assumed (or guaranteed) for a correct behaviour of the unit under consideration.

We are going to define now three characteristic units, namely RF, ALU and (internal) registers: the reader will recognize that the other basic units can be specified in a similar way.

The register file RF. The zCPU register file defines the interaction between 64 general registers and the rest of the processor. The content $\mathit{reg}(\mathit{addr})$ of any register $\mathit{addr} \in \{0, \dots, 63\}$ becomes accessible through one of the five RF-ports $\mathit{OutPort}_i$ ($i = 1,2,3,5$) and $\mathit{in\_port}_j$ ($j = 4,5$), where the fifth can be used as input ($\mathit{in\_port}_5$) and as output ($\mathit{OutPort}_5$) port.[2]

The values of the RF-exits $\mathit{OutPort}_i$ are computed from the entries addr_i using *reg* by the rules

$$\mathit{OutPort}_i := \mathit{reg}(\mathit{addr}_i) \quad \text{for } i = 1,2,3. \tag{1}$$

The entries $\mathit{addr}_4 \in \{0, \dots, 63\}$, $\mathit{in\_port}_4 \in \mathit{INTEGER}$ and $\mathit{write\_enable}_4 \in \{0, 1\}$ are used to update *reg* on addr_4 to $\mathit{in\_port}_4$ if the input port number 4 is enabled for writing: this is formalized by the rule

$$\textbf{if } \mathit{write\_enable}_4 \textbf{ then } \mathit{reg}(\mathit{addr}_4) := \mathit{in\_port}_4. \tag{2}$$

Port number 5 is special because it can be used for either reading or writing: in the latter case the value of $\mathit{OutPort}_5$ becomes undefined. Thus, the behaviour of the RF unit with entries $\mathit{addr}_i, i \in \{1, \dots, 5\}$, $\mathit{in\_port}_j, \mathit{write\_enable}_j, j \in \{4, 5\}$ and exits $\mathit{OutPort}_k, k \in \{1,2,3,5\}$ is defined by rules (1), (2) and by the following rule for RF-port number 5:

$$\begin{array}{ll} \textbf{if} & \mathit{write\_enable}_5 \\ \textbf{then} & \mathit{reg}(\mathit{addr}_5) := \mathit{in\_port}_5 \\ & \mathit{OutPort}_5 := \mathit{undef} \\ \textbf{else} & \mathit{OutPort}_5 := \mathit{reg}(\mathit{addr}_5) \end{array} \tag{3}$$

The RF unit works under the additional assumption that it is not allowed to read and to write a register at the same time, as well as to write to the same register through the two input ports 4 and 5. These conditions are formalized by the following integrity contraints on the RF entries:

$$\begin{array}{l} \mathit{write\_enable}_4 \Rightarrow \mathit{addr}_4 \notin \{\mathit{addr}_1, \mathit{addr}_2, \mathit{addr}_3, \mathit{addr}_5\} \\ \mathit{write\_enable}_5 \Rightarrow \mathit{addr}_5 \notin \{\mathit{addr}_1, \mathit{addr}_2, \mathit{addr}_3\} \\ \mathit{write\_enable}_4 \wedge \mathit{write\_enable}_5 \Rightarrow \mathit{addr}_4 \neq \mathit{addr}_5. \end{array}$$

A peculiarity of the RF units consists in its exits: usually exits are defined by equations and possibly depend on the internal state of the unit, while updates are used to modify the internal state of the unit. In RF the exits are written through updates: this is just a notational shorthand similar to that used for internal registers. In fact, the functions $\mathit{OutPort}_i$ of RF are internal registers of RF, whose exits are also exits of RF itself.[3]

The arithmetical unit ALU_MPY_DIV. The arithmetical unit of zCPU consists of three parts which can work in parallel, one for the additive, logical and shift operations, one for (3 types of) multiplication and one for division. The entries are *math_code* (indicating the operation code), op_i (for the two operands), three condition code entries $\mathit{carry}_{\mathrm{in}}$, $\mathit{extend}_{\mathrm{in}}$,

[1] Note that such a rule transforms a structure ("state") — i.e. a set of functions $\mathcal{S}$ over given domains — into another structure $\mathcal{S}'$ which differs from $\mathcal{S}$ by some of the functions being changed for some arguments. Functions which appear in an evolving algebra but never as outer function f of a function update $f(t_1, \dots, t_n) := t$ are called *external*: they represent the environment for the evolving algebra.

[2] We denote internal registers by capital initial letters and try to adhere to the terminology of [3]. The reader should not confuse the general registers of the register file and the internal registers of the zCPU. The former are accessed using the function *reg*, the latter are viewed by us as 0-ary functions which can be updated by transition rules (see the definition of internal registers below).

[3] This notation is used in other units as well (e.g. DECODE): when a register name (denoted by upper case initial) appears in the list of exits of a unit, it should be interpreted as explained here.

$zero_{\text{in}}$ and four entries *md_ctrl* for multiplier and divider control. The exits are $math\_res_{\text{out}}$ for the computed result, and one for each condition code (the above plus negative value, overflow, division by zero). These exits are characterized in a purely functional way. Technically speaking this means that we abstract from the time needed by the device to compute the values at the exits which correspond to the values appearing at the entries. In particular, $math\_res_{\text{out}}$ is defined as a function

$$math\_res_{\text{out}} = math\_res(\,math\_code, md\_ctrl, \\ op_1, op_2, carry_{\text{in}}, extend_{\text{in}}\,).$$

(The exits for condition codes are defined in a similar way).

In case *math_code* indicates an additive, logical or shift operation, $math\_res_{\text{out}}$ is the usual combinatorial function of op_1, op_2, $carry_{\text{in}}$ and $extend_{\text{in}}$ (and the functions corresponding to the condition codes are similarly defined).

In case *math_code* indicates an operation for MPY or DIV, the entry *md_mux* (in *md_ctrl*) distinguishes between multiplicative operations and division. In case of a multiplication the entry *md_mux* and an additional entry *add_mul* will determine which function will be used to compute the value of the operation in question on the arguments op_1, op_2.

This function is however not combinatorial, because more than one clock cycle is needed for its computation by the unit. The two dedicated hardware devices which execute multiplications and divisions interfere with the main ALU pipeline only when the multiplication or division instructions are issued or when the result is ready for write–back. Therefore the ALU can execute other operations while multiplications or divisions are in progress. As a consequence — we consider now the case of multiplications — at the beginning (when the entry *mul_in* in *md_ctrl* satisfies $mul\_in = 0$) the operands op_i must be stored in internal registers $MulOp_i$ of the multiplier and a counter (*MulStep*) must be set to determine when the multiplication result is ready, namely after 2 further clock cycles. The compiler is assumed to guarantee that the distance between two consecutive multiplicative instructions is at least 3, i.e. that *mul_in* changes from 0 to 1 and will not assume 0 again before 2 clock cycles.

Thus, the behaviour of the MPY part of the unit ALU_MPY_DIV is formalized by the rule

$$\textbf{if } mul\_in = 0 \textbf{ then } \text{start\_mul}(op_1, op_2) \textbf{ else } \text{mul\_busy}$$

where

$$\begin{aligned} \text{start\_mul}(op_1, op_2) \quad &\equiv \quad MulOp_1 := op_1 \\ & \quad MulOp_2 := op_2 \\ & \quad MulStep := 1 \\ \text{mul\_busy} \quad &\equiv \quad MulStep := MulStep + 1. \end{aligned}$$

The function *mul_ready*, indicating when the result of the multiplication is ready to be written back into the destination register, is defined by:

$$mul\_ready(mul\_in) = mul\_in \neq 0 \wedge MulOp_1 \neq undef \\ \wedge MulOp_2 \neq undef \wedge MulStep \geq 2.$$

A similar formalization is done for the behaviour of the DIV–subdevice of ALU_MPY_DIV, making use of the fourth entry *start_div* in *md_ctrl*.

A register unit. A *register*[4] X can be viewed as a very simple unit, represented by an evolving algebra with one entry $X.in$ and one exit $X.out$, as well as a 0-ary dynamic function X holding its contents. The unit contains the transition rule

$$X := X.in$$

which formalizes writing the given value into the register, and a definition

$$X.out = X$$

which defines the output of the register unit simply as the content of the register.[5] Once this has been said, we clearly identify $X.out$ and X: in the following we shall write only X, without distinguishing it notationally from $X.out$.

3 Composition of the Datapath Components

Composing units means to connect exits with entries. As is well known, a global specification of the composed unit can be obtained by substituting in the appropriate places of the components the entries with the exits, according to those connections. In this abstract we can only refer to the literature [4], where it is shown how the resulting notion of computation of a "composed" unit can be defined rigorously in terms of the notion of computation of the components. It is also shown there that all the combinations we need can be obtained in a modular way from the basic units by applying parallel or sequential composition and feedback.

We show as example how to connect the register file with the ALU, using some additional small units so as to obtain the kernel for the interpretation of arithmetic intructions.

Here we identify the RF-exits $OutPort_i$ ($i = 1, 2$) with the entries op_i of the ALU, the ALU exit $math\_res_{\text{out}}$ with the *Res* register entry *in* and the *Res* register exit *out* with the RF-entry in_port_4.

The fields of the current (arithmetic) instruction are contained in 4 additional registers, namely *MAC* for the mathematical operation code, *Ri* for the address of the register which contains the i-th operand ($i = 1, 2$), *RR* for the address of the destination register. We connect *Ri* with the entry $addr_i$ of RF and *RR* with $addr_4$ (passing through two additional registers RR_2 and RR_3, which delay the value for two steps). *MAC* is connected to the *math_code* entry of the ALU (again passing through a delay register MAC_2). Since the value of *MAC* is also needed for computing a certain portion of the control code, we connect it to the combinatorial unit DECODE (instruction decoding unit).

[4] Note that we refer here to *internal registers*, not to the *general registers* of the register file, represented in our model by the function *reg*.

[5] Note that the crucial effect of a register is that the value of the entry is made available for the next step at the exit.

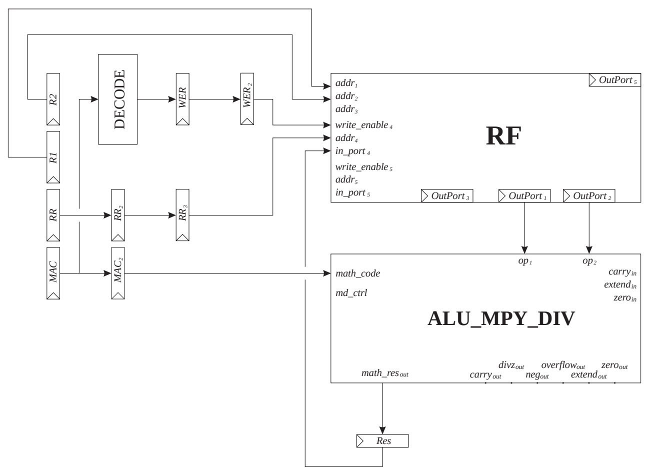

Figure 1: Arithmetic subunit of the zCPU

In particular, DECODE provides the information for enabling writing through the RF-port number 4. Since this value is needed after 2 steps — namely the time needed to compute the result of the arithmetic operation — it is passed from a DECODE-exit to the RF-entry $\mathit{write\_enable}_4$ going through two delay registers *WER* and WER_2.

In this way we obtain the following arithmetic subunit, which suffices to compute the result of simple arithmetic instructions, such as ADD. Note that this unit formalizes a portion of the zCPU block diagram in [3] and is obtained by carrying out all the substitutions corresponding to connections between units (shown in figure 1), such as [$\mathit{R1}/\mathit{addr}_1$, $\mathit{R2}/\mathit{addr}_2$, $\mathit{WER}_2/\mathit{write\_enable}_4$, $\mathit{RR}_3/\mathit{addr}_4$, $\mathit{Res}/\mathit{in\_port}_4$] for the RF unit and [$\mathit{MAC}_2/\mathit{math\_code}$, $\mathit{OutPort}_i/\mathit{op}_i$] for the ALU:

$$\begin{array}{l}
\mathit{OutPort}_1 := \mathit{reg}(\mathit{R1}) \\
\mathit{OutPort}_2 := \mathit{reg}(\mathit{R2}) \\
\mathit{MAC}_2 := \mathit{MAC} \\
\mathit{WER} := \mathit{MAC} \in \{\mathtt{ADD}, \ldots\} \\
\mathit{RR}_2 := \mathit{RR} \\
\mathit{Res} := \mathit{math\_res}(\mathit{MAC}_2, \mathit{md\_ctrl}, \mathit{OutPort}_1, \\
\qquad\qquad\qquad \mathit{OutPort}_2, \mathit{carry}_{\mathrm{in}}, \mathit{extend}_{\mathrm{in}}) \\
\mathit{WER}_2 := \mathit{WER} \\
\mathit{RR}_3 := \mathit{RR}_2 \\
\textbf{if } \mathit{WER}_2 \textbf{ then } \mathit{reg}(\mathit{RR}_3) := \mathit{Res}.
\end{array}$$

(The rules above are grouped according to pipeline stages).

Similar constructions can be made to compose the units for the execution of the other instructions, such as input/output and jumps.

References

[1] A. Bartoloni et al., *A Hardware Implementation of the APE100 Architecture*, in: Int. Journal of Modern Physics, C 4 (1993), pp. 969 sqq. (see also pp. 955 sqq.).

[2] E. Börger, G. Del Castillo, P. Glavan, D. Rosenzweig, *Towards a mathematical specification of the APE100 architecture: the APESE model*, in: B. Pehrson and I. Simon (Eds.), *IFIP 13th World Computer Congress 1994*, Volume I: *Technology/Foundations*, Elsevier, Amsterdam, 396–401.

[3] G. Bastianello et al., *A high performance single chip processing unit for parallel processing and data acquisition systems*, in: Nuclear Instruments and Methods in Physics Research, A324 (1993), pp. 543 sqq.

[4] A. Brüggemann, L. Priese, D. Rödding, R. Schätz, *Modular decomposition of automata*, in: Springer LNCS 171, 1984, 198-236.

[5] Y. Gurevich, *Evolving Algebras 1993: Lipari Guide*, in: Specification and Validation Methods, Ed. E. Börger, Oxford University Press, 1995.

[6] E. Börger, D. Rosenzweig, *The WAM - Definition and Compiler Correctness*, in: *Logic Programming: Formal Methods and Practical Applications* (C. Beierle, L. Plümer, Eds.), Elsevier Science B.V./North-Holland, Series in Computer Science and Artificial Intelligence, 1995, pp. 20–90 (chapter 2).

[7] E. Börger, I. Durdanovic, *Correctness of Compiling Occam to Transputer Code*, in: Y. Gurevich and E. Börger, *Evolving Algebras. Mini-Course*, Technical Report BRICS-NS-95-4, pp. 153–194, BRICS, University of Aarhus, July 1995.

[8] E. Börger, U. Glässer, W. Müller, *Formal Definition of an Abstract VHDL'93 Simulator by EA-Machines*, in: Carlos Delgado Kloos and Peter T. Breuer (Eds.), *Formal Semantics for VHDL*, pp. 107–139, Kluwer Academic Publishers, 1995.

[9] E. Börger, G. Del Castillo, *A formal method for provably correct composition of a real-life processor out of basic components (The APE100 Reverse Engineering Study)*, in: Y. Gurevich and E. Börger, *Evolving Algebras. Mini-Course*, Technical Report BRICS-NS-95-4, pp. 195–222, BRICS, University of Aarhus, July 1995.

[10] E. Börger, U. Glässer, *Modelling and Analysis of Distributed and Reactive Systems using Evolving Algebras*, in: Y. Gurevich and E. Börger, *Evolving Algebras. Mini-Course*, Technical Report BRICS-NS-95-4, pp. 128–152, BRICS, University of Aarhus, July 1995.

Note: [6], [7], [8], [9] and [10] are available through anonymous ftp in the directory *pub/Papers/boerger* of the machine *apollo.di.unipi.it* (131.114.4.36) and through WWW at the address *http://www.uni-paderborn.de/Personal/eas.html*.