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
Corso Italia 40
56125 Pisa, Italy
E-Mail: *boerger@di.unipi.it*
Fax: ++39 50 887226

GIUSEPPE DEL CASTILLO
Universität Paderborn
FB17 Informatik
Warburger Str. 100
33095 Paderborn, Germany
E-Mail: *giusp@uni-paderborn.de*
Fax: ++49 5251 603427

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. The method makes use of Gurevich's concept of evolving algebra. 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.

Introduction

The APE100 massively parallel processor has been built as a dedicated machine for floating point intensive scientific applications and has proved to be rather successful for numerical simulations in Lattice Gauge Theory (see [3, 4]). As preparation for a possible upgrade to a new APE1000 machine (see [5]) 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 are required to be usable for producing executable prototypes and to 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 (a

*in: Yuri Gurevich and Egon Börger, "Evolving Algebras. Mini-Course", Technical Report BRICS-NS-95-4, BRICS, University of Aarhus, July 1995. An abridged version of this paper is in the *Proceedings of the First IEEE International Conference on Engineering of Complex Computer Systems* (ICECCS'95).

characteristic part of the advanced software environment of APE100, see [1]). The ground model APESE has been defined in [7]; 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. In this paper we show how this model can be transformed 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 here on the VLSI superscalar integer processor zCPU 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; see [2] for technical details of the zCPU). In section 1 we provide a formal description of the standard architectural components of zCPU, namely the register file RF (which plays the role of a cache system), the (extended) ALU, the sequencing and addressing unit (which works as dedicated hardware independently and concurrently with the main ALU but can cooperate with the latter to calculate complex addresses for say multidimensional array references), the condition code unit, an instruction decoding unit and other special auxiliary devices and registers. In section 2 we define a method by which given units can be composed in a precise way to complex units. Using well known techniques ¿from the literature (see [12]) the composition can be done in a modular way. This greatly simplifies the task to prove properties of complex units in terms of properties of their simpler constituents. Defining the components as evolving algebras (in the sense of Gurevich [14]) 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 ex. [11, 8, 9]).

In section 3 we explain how for the formal model LEX of the zCPU processor we can make the following informal statement into a precise mathematical assertion and give a mathematical proof for it.

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

In the appendices we provide the full abstract definition of the zCPU processor, obtained by putting together the definitions of its basic units.

The proof of the theorem, which will be given in a sequel to this paper, proceeds by a series of correctness theorems which establish that each level n of the refinement hierarchy leading from APESE to LEX is correctly implemented by the model at level n+1. Note that the "general strategy used throughout in APE100 of moving functions from hardware to software, as long as this does not entail performance penalties" ([6]) results in a sometimes very subtle interplay between the compiler and the hardware, in particular where it comes to deal with VLIW and pipelining parallelism. The freedom of abstraction which comes with the notion of evolving algebras allows us to reflect this strong role of the compiler within the APE100 project without going into its details; since we want to concentrate on the architecture, the compiler properties are explicitly formulated as abstract assumptions for the claims on the behaviour of the processor.

This paper is not the place to compare the new evolving algebra approach to the numerous other formal specification methodologies in the literature. The goal of this paper is to report on a challenging reverse engineering project which has been carried out successfully using the evolving algebra approach. We want to convince the practitioner by an example from real-life that:

- one can use the evolving algebra specification methodology to produce readable but nevertheless precise specifications without previous formal training and without formal overhead;
- the evolving algebra specification method scales to complex systems.

This explains also why we do not give here a formal definition of the underlying semantics of evolving algebras (which however has been rigorously defined in [14]); our specifications can be read and understood as abstract code which can serve as basis for the implementation of executable prototypes.

1 The Datapath Components of the zCPU

The zCPU processor is built out of several main units, namely:

- the register file RF;
- the ALU_MPY_DIV unit (ALU for brevity), an arithmetico-logical unit containing special independent devices for multiplication and division;
- the CC&STATUS unit, which processes the condition codes and the state information for handling jumps and exceptions;
- the Address Generation Unit AGU, for calculating program and data addresses;
- the Input/Output Subsystem IOS, providing the interface between zCPU and its data memory (DATAMEM);
- the INSTR unit, for instruction fetch;
- the units DATAMEM and PROGMEM, data and program memory of the zCPU.

In addition, other components are needed in order to coordinate the operations of the units listed above, namely some internal registers and a DECODE unit, for decoding instructions words read from program memory into the appropriate control codes required by the units.

Each unit is specified as an evolving algebra (in the sense of [14]) 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

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

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

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

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

[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 $f(t_1, \ldots, t_n) := t$ are called *external*: they represent the environment for the evolving algebra.

in each step each rule which can be applied is applied. (For an exact definition of this lock-step semantics of evolving algebras see [14]).

The entries and exits can be viewed as terms, which include 0-ary functions. As an experiment we will use in this paper essentially only 0-ary functions for entries and exits. 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 combinational 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_MPY_DIV and a register: the reader will recognize (and can check through the details provided in the appendices) that the other basic units can be specified in a similar way.

1.1 The register file RF

The zCPU register file defines the interaction between 64 general registers and the rest of the processor. The content $reg(addr)$ of any register $addr \in \{0, \ldots, 63\}$ becomes accessible through one of the five RF-ports $OutPort_i$ $(i = 1, 2, 3, 5)$ and in_port_j $(j = 4, 5)$, where the fifth can be used as input (in_port_5) and as output $(OutPort_5)$ port[2].

The values of the RF-exits $OutPort_i$ are computed from the entries $addr_i$ using reg by the rules

$$OutPort_i := reg(addr_i) \quad \text{for } i = 1, 2, 3. \tag{1}$$

The entries $addr_4 \in \{0, \ldots, 63\}$, $in\_port_4 \in INTEGER$ and $write\_enable_4 \in \{0, 1\}$ are used to update reg on $addr_4$ to in_port_4 if the input port number 4 is enabled for writing: this is formalized by the rule

$$\textbf{if } write\_enable_4 \textbf{ then } reg(addr_4) := in\_port_4. \tag{2}$$

Port number 5 is special because it can be used for either reading or writing: in the latter case the value of $OutPort_5$ becomes undefined. Thus, the behaviour of the RF unit with entries $addr_i, i \in \{1, \ldots, 5\}$, $in\_port_j, write\_enable_j, j \in \{4, 5\}$ and exits $OutPort_k, k \in \{1, 2, 3, 5\}$ is defined by rules (1), (2) and by the following rule for RF-port number 5:

$$\begin{array}{ll} \textbf{if} & write\_enable_5 \\ \textbf{then} & reg(addr_5) := in\_port_5 \\ & OutPort_5 := undef \\ \textbf{else} & OutPort_5 := reg(addr_5) \end{array} \tag{3}$$

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:

$$\begin{array}{rcl} write\_enable_4 & \Rightarrow & addr_4 \notin \{addr_1, addr_2, addr_3, addr_5\} \\ write\_enable_5 & \Rightarrow & addr_5 \notin \{addr_1, addr_2, addr_3\} \\ write\_enable_4 \wedge write\_enable_5 & \Rightarrow & addr_4 \neq addr_5. \end{array}$$

[2]We denote internal registers by capital initial letters and try to adhere to the terminology of [6]. The reader should not confuse the general registers of the register file and the internal registers of the zCPU. The former are named by numbers $0, \ldots, 63$ and their contents accessed using the function reg, the latter are viewed by us as 0-ary functions which can be updated by transition rules (see below 1.3).

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 in 1.3. In fact, the functions in_port_i of RF are internal registers of RF, whose exits are also exits of RF itself.[3]

1.2 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 $carry_{\text{in}}$, $extend_{\text{in}}$, $zero_{\text{in}}$ 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, see appendix A.2).

In case *math_code* indicates an additive, logical or shift operation, $math\_res_{\text{out}}$ is the usual combinational 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 combinational, 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 ALU_MPY_DIV unit is formalized by the rule

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

[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.

where

$$\begin{aligned} \mathrm{start\_mul}(op_1, op_2) &\equiv \mathit{MulOp}_1 := op_1 \\ &\ \mathit{MulOp}_2 := op_2 \\ &\ \mathit{MulStep} := 1 \\ \mathrm{mul\_busy} &\equiv \mathit{MulStep} := \mathit{MulStep} + 1. \end{aligned}$$

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

$$\mathit{mul\_ready}(\mathit{mul\_in}) \quad = \quad \mathit{mul\_in} \neq 0 \wedge \mathit{MulOp}_1 \neq \mathit{undef} \wedge \mathit{MulOp}_2 \neq \mathit{undef} \wedge \mathit{MulStep} \geq 2.$$

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* (see appendix A.2).

1.3 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 $X.out$.

2 Composition of the Datapath Components

In this section we show how to compose the basic units to the zCPU processor in such a way that the behavioural properties we are interested in can be controlled by the "modular" composition technique.

In doing this, we use the natural pictorial representation of evolving algebras with entries and exits as boxes with ingoing and outgoing arrows. Composing units then means to connect outgoing with ingoing arrows. In this paper we can only refer to the literature (see for example [12]), 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 proceed now to show as example how to connect the register file with the ALU_MPY_DIV unit, using some additional small units so as to obtain the kernel for the interpretation of arithmetic

[4]Note that we refer here to *internal registers*, not to the *general purpose 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.

intructions. The connection of outgoing arrows with ingoing arrows is expressed by identifying 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 identifications.

Here we identify the RF-exits (output ports) $OutPort_i$ ($i = 1, 2$) with the entries op_i of the ALU_MPY_DIV unit, the ALU_MPY_DIV exit $math\_res_{\rm out}$ 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)[6]. MAC is connected to the $math\_code$ entry of ALU_MPY_DIV (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 combinational unit DECODE (instruction decoding unit).

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 $write\_enable_4$ going through two delay registers WER and WER_2.

In this way we have obtained the following unit (arithmetic subunit) which suffices to compute the result of simple arithmetic instructions, such as the addition instruction `ADD`, as we show in the next section. Note that this unit formalizes a portion of the zCPU block diagram in [6].

For use in the next section, we reassume the relevant rules and conditions of the thus modified arithmetic subunit in the black box form, where all substitutions shown by the figure are done (namely $[R1/addr_1, R2/addr_2, WER_2/write\_enable_4, RR_3/addr_4, Res/in\_port_4]$ for the RF unit and $[MAC_2/math\_code, OutPort_1/op_1, OutPort_2/op_2]$ for the ALU_MPY_DIV unit):

$OutPort_1 := reg(R1)$
$OutPort_2 := reg(R2)$
$MAC_2 := MAC$
$WER := MAC \in \{\texttt{ADD}, \dots\}$
$RR_2 := RR$

$Res := math\_res(MAC_2, md\_ctrl, OutPort_1, OutPort_2, carry_{\rm in}, extend_{\rm in})$
$WER_2 := WER$
$RR_3 := RR_2$

if WER_2 **then** $reg(RR_3) := Res$

(note the rules above are grouped according to the pipeline stages).

Similar constructions can be made to compose the units for the execution of input/output instructions, jumps etc. (see Appendix C).

[6]For reasons of exposition, instead of just one we introduce here two delay registers for RR which allows us to abstract from some peculiarity in the compilation scheme of APE100. See also the footnote at the end of the next section.

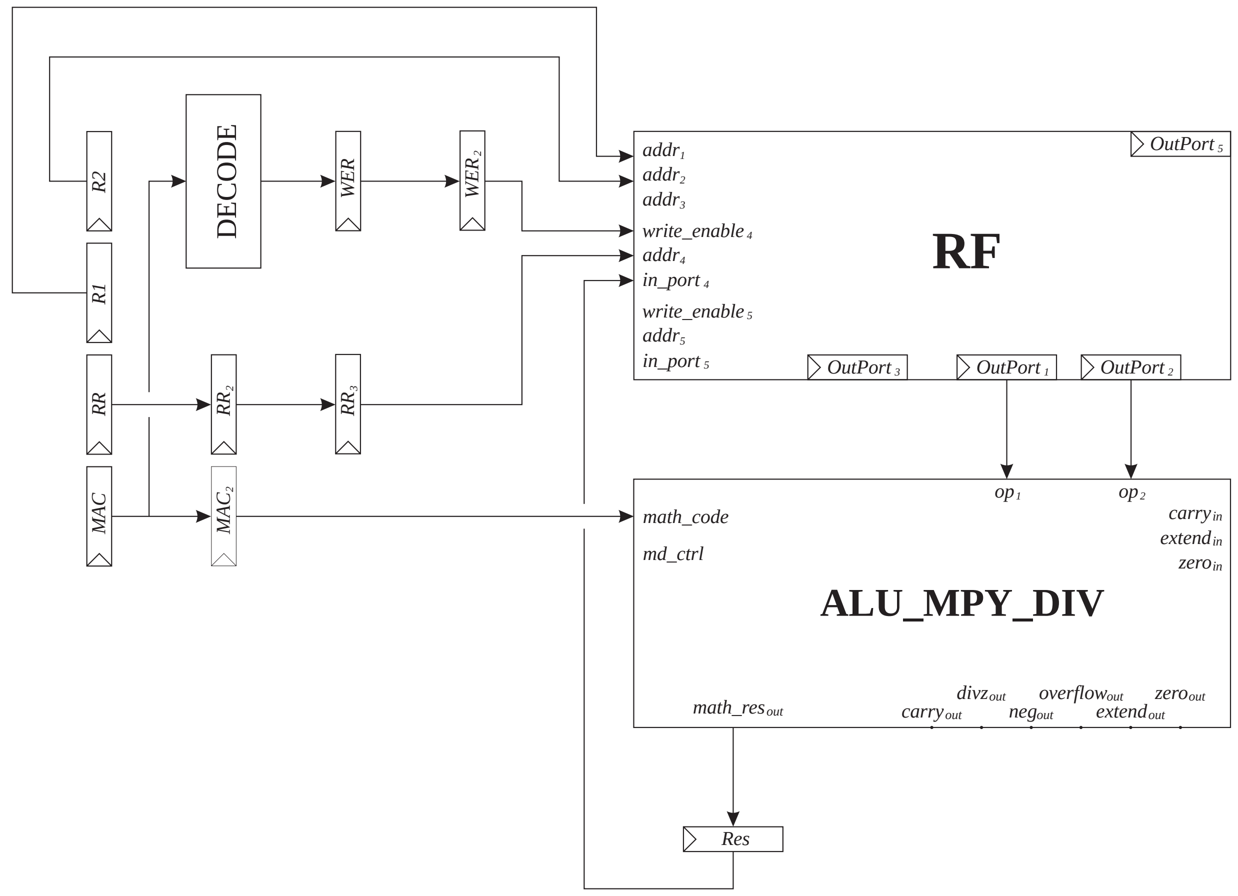


Figure 1: Arithmetic subunit of the zCPU

3 Correct Implementation of the ZIC Instruction Set

The two crucial aspects of the compilation strategy for the zCPU are the following: (i) Assembler instructions are macro-expanded into short sequences of ZIC instructions, whose execution is easily shown to implement the Assembler instructions correctly; (ii) the ZIC instructions are fed into an optimizer to find a schedule which minimizes the time of their execution through pipelined VLIWs and avoids structural and data hazards (i.e. no device overbooking occurs and no data are used before their computation is terminated). We prove in lemmas 1,2 below that this pipelined instruction execution is correct. The reader should keep in mind that we illustrate the technique here only for arithmetical instructions but that a similar construction can be made for input–output and jump instructions in such a way that the correctness of their simultaneous execution together with arithmetical instructions (VLIW scheme) can be proved rigorously. Due to size restrictions, we have to leave the details for a sequel to this paper.

Proposition. The (extended) arithmetic subunit executes additive arithmetic instructions correctly.[7]

[7]By "extended" we want to point to the fact that minor additional components have to be introduced to take care of condition code for zero, overflow etc. A similar proposition can be proved for the extension of the unit for

Proof. Due to space limitations we can only illustrate the proof technique by going through the major steps for an example; we choose the addition instruction ([`ADD` *RR R1 R2*]), whose execution is defined by the rule

$$\textbf{if } MAC = \texttt{ADD} \textbf{ then } reg(RR) := reg(R1) + reg(R2).$$

This rule is part of the evolving algebra ZIC defined in [13], which models APE100 as interpreter of ZCPU Intermediate Code (and which can easily be shown to implement correctly the APESE ground model for APE100 presented in [7]). Its execution presents the three usual phases (we skip the condition codes, which are computed in the second phase):

$$\begin{aligned}
\text{read\_operands} &\equiv OutPort_1 := reg(R1) \\
& OutPort_2 := reg(R2) \\
\text{calc\_result} &\equiv \textbf{if } MAC = \texttt{ADD} \textbf{ then } Res := OutPort_1 + OutPort_2 \\
\text{write\_result} &\equiv \textbf{if } MAC = \texttt{ADD} \textbf{ then } reg(RR) := Res
\end{aligned}$$

In the general case, *Res* will take $math\_res(MAC, md\_ctrl, OutPort_1, OutPort_2, carry_{\text{in}}, extend_{\text{in}})$ (which for $MAC = \texttt{ADD}$ takes the value $OutPort_1 + OutPort_2$, independently of the value of *md_ctrl*, $carry_{\text{in}}$ and $extend_{\text{in}}$).

This leads to an intermediate model ZIC_{PH} where the execution of ZIC-instructions is decomposed into three (or more, for some other instructions) sequential steps, such that the following intuitive statement can be made precise and proved by induction on the length of ZIC-computations.

Lemma 1. The evolving algebra ZIC_{PH} implements the model ZIC correctly.

The second step in the proof of the proposition introduces the parallelism due to pipelining. The rules of ZIC_{PH} are modified in such a way that no conflict arises when they are executed not any more sequentially, but in parallel (using the lock-step semantics of evolving algebras, see [14]), leading to the model LEX ("loadable executable code", the final level in the APE100 compilation chain). A rigorous formulation and a transparent proof can be given for the following statement.

Lemma 2. Under the assumption that the compiler guarantees data and control independence of instructions which enter the pipe, the model LEX implements the model ZIC_{PH} correctly.

Note that the optimizer mentioned above makes sure that the assumption of the lemma is guaranteed. The two preceding lemmas prove the proposition.

Proof. Due to the modular character of our formalization, the proof can be given instructionwise; essentially it consists in a local analysis of the effect of the LEX rules for each type of ZIC instruction. Again we illustrate the proof principle for the case of the `ADD` instruction.

The phases (the sequential steps) of the execution of instructions in ZIC_{PH} become pipe stages in LEX, where at each step all the rules are applied simultaneously. The main problem is to make sure that each pipe stage operates on the correct data.

When for the given instruction (`ADD`) it comes to execute calc_result (in the second pipe stage), the value of *MAC* in the preceding stage is needed; therefore *MAC* is delayed by $MAC_2 := MAC$ and replaced in *math_res* by MAC_2.

multiplications and divisions.

Similarly, in the third pipe stage, for writing the result to the destination register the three values for the RF-entries $write\_enable_4$, $addr_4$ and in_port_4 are needed. The value of $write\_enable_4$ must be 1 if $MAC = \texttt{ADD}$; as this value is computed by the DECODE unit in the first pipe stage, it must be delayed twice (through registers WER and WER_2). Similarly, the address of the destination register RR is read in the first pipe stage and needed in the third one (at the RF-entry $addr_4$); therefore, it is delayed twice through the registers RR_2 and RR_3.[8] Finally, the result to be written has been computed by calc_result (and placed in the Res register) in the previous pipe stage; therefore, no delay registers are needed for it, and the rule write_result of the LEX model is obtained by the write_result rule of the ZIC$_{\mathrm{PH}}$ model, by replacing $MAC = \texttt{ADD}$ by WER_2 and RR by RR_3.

Conclusion

We have shown on an example how to prove that the hardware level model LEX of the APE100 control processor zCPU executes compiled programs correctly if the compiler satisfies the assumptions which are realized by the APE100 compilation chain. Along these lines [13] has developed the specification of the zCPU processor which is the basis for the complete correctness proof, which we have to leave for a sequel to this paper. In the appendix we list the complete formal definition of the zCPU, without further explanation.

Note that from the software point of view zCPU and the MAD chip—the floating point unit of the APE100 processor—can be seen as different elements of just one VLIW system. In terms of our specification method this means that zCPU and MAD can be seen as modules of APE100 whose architectural composition and dynamic interaction can be fully described by our methodology in a precise yet simple way. Note also that the case of zCPU and of APE100 reverse engineering is not restrictive. Our methodology can be applied to other processors as well. Se for ex. [10] where the evolving algebra methodology is used to prove the correctness of general pipelining principles for RISC architectures.

Acknowledgement

We are particularly grateful to Raffaele Tripiccione ¿from the APE group in Pisa for numerous hours spent during the last two years for explaining to us the secrets of the APE100 project.

[8]As has been noted above, the zCPU has not two but only one delay register for RR. Therefore, the value of RR has to be taken during the second pipe stage — this is assured in APE100 by the compiler. In [13] the model is developed with all the details which are needed to describe exactly the behaviour of the zCPU on the basis of the APE100 compilation scheme.

Appendices

In the following four appendices we list with only a few more explanations the full abstract definition of the zCPU processor. Appendix A defines each single unit separately. Appendix B contains the block diagram of the zCPU, giving a rough representation of the units described in Appendix A and their connections. Appendix C defines how the units are composed, by producing formal identifications among exits and entries (corresponding to the connections shown in the block diagram). Appendix D lists the final definition of the zCPU after all the substitution are done.

A Definition of the zCPU units

Before listing the definitions of the units, we define the two main sets of operation codes for the instructions which can be executed by the zCPU, i.e. MAC-type instructions (arithmetic) and IOC-type instructions (input/output, jumps). The MAC-type operation codes are:

$$\mathit{MATH\_OP} = \mathit{ALU\_OP} \cup \mathit{MUL\_OP} \cup \mathit{DIV\_OP} \cup \{\mathtt{NOP}\}$$

where

$$\begin{aligned}
\mathit{ALU\_OP} &= \mathit{LOGIC} \cup \mathit{ADDER} \cup \mathit{SHIFTER} \\
\mathit{LOGIC} &= \{\,\mathtt{OR}, \mathtt{AND}, \mathtt{NOR}, \mathtt{NAND}, \mathtt{XOR}, \mathtt{ZERO}, \mathtt{XNOR}, \mathtt{FF}\,\} \\
\mathit{ADDER} &= \{\,\mathtt{ADD}, \mathtt{ADDC}, \mathtt{SUB}, \mathtt{SUBC}, \mathtt{CMP}\,\} \\
\mathit{SHIFTER} &= \{\,\mathtt{LSH}, \mathtt{ASH}, \mathtt{XSH}, \mathtt{ROT}\,\} \\
\mathit{MUL\_OP} &= \{\,\mathtt{MUL1}, \mathtt{MUL2}, \mathtt{MULA2}, \mathtt{MUL3}\,\} \\
\mathit{DIV\_OP} &= \{\,\mathtt{DIV1}, \mathtt{DIV2}\,\}
\end{aligned}$$

The IOC-type operation codes are:

$$\mathit{IOC\_OP} = \mathit{LOAD} \cup \mathit{STORE} \cup \mathit{BRANCH} \cup \mathit{SPECIAL} \cup \{\mathtt{IONOP}\}$$

where

$$\begin{aligned}
\mathit{LOAD} &= \{\,\mathtt{LD}, \mathtt{LDA}, \mathtt{LDPA}, \mathtt{LDSR}\,\} \\
\mathit{STORE} &= \{\,\mathtt{ST}, \mathtt{STSR}\,\} \\
\mathit{BRANCH} &= \{\,\mathtt{JUMP}, \mathtt{SKIP}\,\} \\
\mathit{SPECIAL} &= \{\,\mathtt{FLD}, \mathtt{FST}, \mathtt{HALT}, \mathtt{BREAK}, \mathtt{SETF}, \mathtt{CLEARF}\,\}
\end{aligned}$$

A.1 Unit RF (Register File)

Entries: $addr_1$; $addr_2$; $addr_3$; $addr_4$, in_port_4, $write\_enable_4$; $addr_5$, in_port_5, $write\_enable_5$

Exits: $OutPort_1$, $OutPort_2$, $OutPort_3$, $OutPort_5$

Integrity constraints:

$$write\_enable_4 \Rightarrow addr_4 \notin \{\, addr_1, addr_2, addr_3, addr_5 \,\}$$
$$write\_enable_5 \Rightarrow addr_5 \notin \{\, addr_1, addr_2, addr_3 \,\}$$
$$write\_enable_4 \wedge write\_enable_5 \Rightarrow addr_4 \neq addr_5$$

Transition rules:

$$
\begin{aligned}
&\text{read\_reg}_{1,2} \;\equiv \\
&\quad \equiv \; \mathit{OutPort}_1 := \mathit{reg}(\mathit{addr}_1) \\
&\qquad\;\; \mathit{OutPort}_2 := \mathit{reg}(\mathit{addr}_2)
\end{aligned}
$$

$$
\begin{aligned}
&\text{read\_reg}_{3} \;\equiv \\
&\quad \equiv \; \mathit{OutPort}_3 := \mathit{reg}(\mathit{addr}_3)
\end{aligned}
$$

$$
\begin{aligned}
&\text{write\_reg}_{4} \;\equiv \\
&\quad \equiv \; \textbf{if } \mathit{write\_enable}_4 \\
&\qquad\;\; \textbf{then } \mathit{reg}(\mathit{addr}_4) := \mathit{in\_port}_4
\end{aligned}
$$

$$
\begin{aligned}
&\text{rw\_reg}_{5} \;\equiv \\
&\quad \equiv \; \textbf{if } \mathit{write\_enable}_5 \\
&\qquad\;\; \textbf{then } \mathit{reg}(\mathit{addr}_5) := \mathit{in\_port}_5 \\
&\qquad\qquad\quad \mathit{OutPort}_5 := \mathit{undef} \\
&\qquad\;\; \textbf{else } \mathit{OutPort}_5 := \mathit{reg}(\mathit{addr}_5)
\end{aligned}
$$

Note that port 5 of the RF can be used for reading and writing.

A.2 Unit ALU (ALU/Multiplier/Divider)

Entries: $\mathit{math\_code}; \mathit{op}_1, \mathit{op}_2; \mathit{carry}_{\mathrm{in}}, \mathit{extend}_{\mathrm{in}}, \mathit{zero}_{\mathrm{in}}; \mathit{md\_ctrl}$

Exits: $\mathit{math\_res}_{\mathrm{out}}, \mathit{carry}_{\mathrm{out}}, \mathit{divz}_{\mathrm{out}}, \mathit{neg}_{\mathrm{out}}, \mathit{overflow}_{\mathrm{out}}, \mathit{extend}_{\mathrm{out}}, \mathit{zero}_{\mathrm{out}}$

Abbreviations:

$$\mathit{md\_ctrl} \;\equiv\; \mathit{mul\_in}, \mathit{start\_div}, \mathit{md\_mux}, \mathit{add\_mul}$$

Note that $\mathit{md\_ctrl}$ stands for "multiplier/divider control".

Definitions:

$$
\begin{aligned}
\mathit{math\_res}_{\mathrm{out}} &= \mathit{math\_res}(\mathit{math\_code}, \mathit{md\_ctrl}, \mathit{op}_1, \mathit{op}_2, \mathit{carry}_{\mathrm{in}}, \mathit{extend}_{\mathrm{in}}) \\
\mathit{carry}_{\mathrm{out}} &= \mathit{carry}(\mathit{math\_code}, \mathit{op}_1, \mathit{op}_2, \mathit{carry}_{\mathrm{in}}) \\
\mathit{divz}_{\mathrm{out}} &= \mathit{divz}(\mathit{op}_2) \\
\mathit{neg}_{\mathrm{out}} &= \mathit{neg}(\mathit{math\_code}, \mathit{md\_ctrl}, \mathit{op}_1, \mathit{op}_2, \mathit{carry}_{\mathrm{in}}, \mathit{extend}_{\mathrm{in}}) \\
\mathit{overflow}_{\mathrm{out}} &= \mathit{overflow}(\mathit{math\_code}, \mathit{op}_1, \mathit{op}_2, \mathit{carry}_{\mathrm{in}}) \\
\mathit{extend}_{\mathrm{out}} &= \mathit{extend}(\mathit{op}_1, \mathit{op}_2) \\
\mathit{zero}_{\mathrm{out}} &= \mathit{zero}(\mathit{math\_code}, \mathit{md\_ctrl}, \mathit{op}_1, \mathit{op}_2, \mathit{carry}_{\mathrm{in}}, \mathit{extend}_{\mathrm{in}}, \mathit{zero}_{\mathrm{in}})
\end{aligned}
$$

where

```
$math\_res(math\_code, md\_ctrl, op_1, op_2, carry_{in}, extend_{in})$ =
    if $math\_code \notin MUL\_OP \cup DIV\_OP$
    then $alu\_res(math\_code, op_1, op_2, carry_{in}, extend_{in})$
    else $mul\_div\_res(md\_ctrl)$

$alu\_res(math\_code, op_1, op_2, carry_{in}, extend_{in})$ =
    if $math\_code$ = ADD then $add(op_1, op_2)$
    else if $math\_code$ = ADDC then $addc(op_1, op_2, carry_{in})$
    else ...
```

```
$mul\_div\_res(md\_ctrl)$ =
    if $md\_mux \in \{0,1\}$
    then  if $mul\_ready(mul\_in)$
            then $mul\_res(md\_mux, add\_mul, MulOp1, MulOp2)$
            else $undef$
    else  if $div\_ready(div\_start) \lor mod\_ready(div\_start)$
            then $div\_res(DivOp1, DivOp2)$
            else $undef$
```

$mul\_ready(mul\_in) = (mul\_in \neq 0 \land MulOp1 \neq undef \land MulOp2 \neq undef \land MulStep \geq 2)$
$div\_ready(div\_start) = (start\_div \neq 1 \land DivOp1 \neq undef \land DivOp2 \neq undef \land DivStep = 34)$
$mod\_ready(div\_start) = (start\_div \neq 1 \land DivOp1 \neq undef \land DivOp2 \neq undef \land DivStep = 35)$

Note that, in the zCPU, computing multiplications needs two clock cycles; for the division, the ALU makes the quotient available after 34 clock cycles, the remainder of the division after 35 clock cycles. The definitions above are needed to express this behaviour of the ALU (see also the transition rules below and the related remark).

```
$mul\_res(md\_mux, add\_mul, x, y)$ =
    if $md\_mux = 0 \land add\_mul = 0$ then $mull(x,y)$
    else if $md\_mux = 0 \land add\_mul = 1$ then $mula(x,y)$
    else $mulm(x,y)$
```

```
$div\_res(x,y)$ =
    if $div\_ready(div\_start)$ then $div(x,y)$
    else if $mod\_ready(div\_start)$ then $mod(x,y)$
    else $undef$
```

$divz(op_2) = (op_2 = 0)$

$neg(math\_code, md\_ctrl, op_1, op_2, carry_{\text{in}}, extend_{\text{in}}) = (math\_res_{\text{out}} < 0)$

```
$zero(math\_code, md\_ctrl, op_1, op_2, carry_{\text{in}}, extend_{\text{in}}, zero_{\text{in}})$ =
    if $math\_code \notin \{\texttt{ADDC}, \texttt{SUBC}, \texttt{MUL3}\}$
    then $math\_res_{\text{out}} = 0$
    else $(math\_res_{\text{out}} = 0) \land zero_{\text{in}}$
```

Transition rules (for multiplier and divider subunits):

```
mul_rule ≡
  ≡ **if** $mul\_in = 0$
    **then** $MulOp1 := op_1$
          $MulOp2 := op_2$
          $MulStep := 1$
    **else** $MulStep := MulStep + 1$
```

```
div_rule ≡
  ≡ if $start\_div = 1$
    then $DivOp1 := op_1$
         $DivOp2 := op_2$
         $DivStep := 1$
    else $DivStep := DivStep + 1$
```

Note that the way multiplications and divisions are computed requires two different actions: first, the multiplier/divider must be activated (by *mul_in*/*start_div*) in order to begin computing the operation; second, the result must be stored (when it is ready). In the meantime a new multiplication/division should not be started. As in the zCPU we have different operation codes for starting and ending multiplications/divisions, it is responsibility of the compiler to schedule instructions in such a way that the ALU is always provided with the correct values of *mul_in*/*start_div*. Note that, except for the rules which formalize the interaction between the main ALU and the multiplier and divider subunits, all the ALU functions are specified by pure equations without using any transition rules.

A.3 Unit CC&STATUS (Condition Codes & Status Register)

Entries: $math\_cc$, if_status_{in}, $halt_{\text{in}}$, $parity\_err_{\text{in}}$; $enable\_cc$; $write\_mask$, val

Exits: $zcpu\_ex$, $status\_register$

Abbreviations:

$$math\_cc \equiv carry_{\text{in}}, divz_{\text{in}}, neg_{\text{in}}, overflow_{\text{in}}, extend_{\text{in}}, zero_{\text{in}}$$
$$enable\_cc \equiv en\_carry, en\_divz, en\_neg, en\_overflow, en\_extend, en\_zero$$

Only certain instructions require that the condition code registers are set: therefore writing into those registers is governed by enabling bits (*enable_cc*). See below the "write_cc_math" rule.

$$\begin{aligned}
\text{update\_mask}(val) \equiv\ & Msk\_Carry := val[3] \\
& Msk\_Divz := val[7] \\
& Msk\_iFstatus := val[11] \\
& Msk\_Neg := val[15] \\
& Msk\_oVerflow := val[19] \\
& Msk\_eXtend := val[23] \\
& Msk\_Zero := val[27]
\end{aligned}$$

Definitions:

$$\begin{aligned}
status\_register =\ & Ex\_Parity\,.\,ParityErr\,.\,Ex\_Halt\,.\,Halt \\
& .\,Msk\_Zero\,.\,Ex\_Zero\,.\,Zero\,.\,0 \\
& .\,Msk\_eXtend\,.\,Ex\_eXtend\,.\,eXtend\,.\,0 \\
& .\,Msk\_oVerflow\,.\,Ex\_oVerflow\,.\,oVerflow\,.\,0 \\
& .\,Msk\_Neg\,.\,Ex\_Neg\,.\,Neg\,.\,0 \\
& .\,Msk\_iFstatus\,.\,Ex\_iFstatus\,.\,iFstatus\,.\,0 \\
& .\,Msk\_Divz\,.\,Ex\_Divz\,.\,Divz\,.\,0 \\
& .\,Msk\_Carry\,.\,Ex\_Carry\,.\,Carry\,.\,ExcpReg
\end{aligned}$$

The status register contains in one machine word the relevant information about the condition codes and exceptions ("status information"): it is accessible through the instructions `LDSR` and `STSR`.

$$\begin{aligned} zcpu\_ex \;=\; & Halt \vee ParityErr \vee (iFstatus \wedge Msk\_iFstatus) \vee \\ & \vee (Carry \wedge Msk\_Carry) \vee (Divz \wedge Msk\_Divz) \vee \\ & \vee (Neg \wedge Msk\_Neg) \vee (oVerflow \wedge Msk\_oVerflow) \vee \\ & \vee (eXtend \wedge Msk\_eXtend) \vee (Zero \wedge Msk\_Zero) \end{aligned}$$

Note that we summarize the status information in the definition of the exit *status_register*: its dynamic change is reflected by the local updates of its components through the transition rules given below. Similarly, the exit *zcpu_ex* — used for detecting conditions leading to exceptions — depends on the value of registers contained in the CC&Status unit.

Transition rules:

```
write_cc_math ≡
  ≡ if $en\_carry$ then $Carry := carry_{in}$
    if $en\_divz$ then $Divz := divz_{in}$
    if $en\_neg$ then $Neg := neg_{in}$
    if $en\_overflow$ then $oVerflow := overflow_{in}$
    if $en\_extend$ then $eXtend := extend_{in}$
    if $en\_zero$ then $Zero := zero_{in}$
write_cc_ifstatus ≡
  ≡ $iFstatus := if\_status_{in}$
```

Note that, in APE100, the value of if_status_{in} bit depends on the state of the floating point units of APE100.

```
write_cc_halt ≡
  ≡ $Halt := halt_{in}$
write_cc_parity ≡
  ≡ $ParityErr := parity\_err_{in}$
write_exception_mask($write\_mask, val$) ≡
  ≡ if $write\_mask$
    then update_mask($val$)
exception_rule ≡
  ≡ if $\neg ExcpReg$
    then $ExcpReg := zcpu\_ex$
         $Ex\_Halt := Halt$
         $Ex\_Parity := ParityErr$
         $Ex\_iFstatus := iFstatus$
         $Ex\_Carry := Carry$
         $Ex\_Divz := Divz$
         $Ex\_Neg := Neg$
         $Ex\_oVerflow := oVerflow$
         $Ex\_eXtend := eXtend$
         $Ex\_Zero := Zero$
```

A.4 Unit AGU (Address Generation Unit)

Entries: $abs_{in}, pma_{in}, base, disp, pma\_mux_{in}$

Exits: $dma_{\mathrm{out}}, pma_{\mathrm{out}}$

Definitions:

$$\begin{aligned} dma_{\mathrm{out}} &= gen\_addr(abs_{\mathrm{in}}, pma_{\mathrm{in}}, base, disp) \\ pma_{\mathrm{out}} &= next\_pma(abs_{\mathrm{in}}, pma_{\mathrm{in}}, base, disp, pma\_mux_{\mathrm{in}}) \end{aligned}$$

where

$$\begin{aligned} & gen\_addr(abs_{\mathrm{in}}, pma_{\mathrm{in}}, base, disp) = \\ & \quad (1 - abs_{\mathrm{in}})pma_{\mathrm{in}} + base + disp \end{aligned}$$

$$\begin{aligned} & next\_pma(abs_{\mathrm{in}}, pma_{\mathrm{in}}, base, disp, pma\_mux_{\mathrm{in}}) = \\ & \quad \text{if } pma\_mux_{\mathrm{in}} = 0 \text{ then } 0 \\ & \quad \text{else if } pma\_mux_{\mathrm{in}} = 1 \text{ then } pma_{\mathrm{in}} + 1 \\ & \quad \text{else } gen\_addr(abs_{\mathrm{in}}, pma_{\mathrm{in}}, base, disp) \end{aligned}$$

Note that upon insertion of the AGU unit into the zCPU processor the exits dma_{out} and pma_{out} will be written into the registers *DMA* and *PMA* resp. (see A.11 and C.11 for further explanation.)

A.5 Unit PROGMEM (Program Memory)

Entries: *addr*

Exits: pmd_{out}

Definitions:

$$pmd_{\mathrm{out}} = instr(addr)$$

Note that *addr* will be provided by the Address Generation Unit (AGU) and pmd_{out} will be read by the Instruction Fetch unit (INSTR). See C.5 and C.7 below.

A.6 Unit DATAMEM (Data Memory)

Entries: $addr, data_{\mathrm{in}}, mem\_write\_enable$

Exits: $data_{\mathrm{out}}$

Definitions:

$$data_{\mathrm{out}} = mem(addr)$$

Transition rules:

```
write_mem ≡
  ≡ **if** $mem\_write\_enable$
    **then** $mem(addr) := data_{\mathrm{in}}$
```

Note that $data_{\mathrm{in}}$ will be read from the Input/Output Subsystem (IOS), namely from its *val_to_store* exit (see C.6).

A.7 Unit INSTR (Instruction Fetch)

Entries: pmd_{in}

Exits: *MAC*, *RR*, *R1*, *R2*, *IOC*, *RD*, *RA*, *Disp*

Definitions:

$$\begin{array}{lcl} MAC & = & MAC(PMD) \\ RR & = & RR(PMD) \\ R1 & = & R1(PMD) \\ R2 & = & R2(PMD) \\ IOC & = & IOC(PMD) \\ RD & = & RD(PMD) \\ RA & = & RA(PMD) \\ Disp & = & Disp(PMD) \end{array}$$

Transition rules:

instruction_fetch $\equiv$
$\equiv \ PMD := pmd_{\text{in}}$

A.8 Unit IOS (Input/Output Subsystem)

Entries: *st_data*, *val_from_RF*, *status_register*;
ld_addr, dma_{in}, *val_from_mem*

Exits: *Dat*, *val_to_load*, *val_to_store*

Note that *st_data*, *ld_addr*, dma_{in} and *val_from_mem* take their values from the registers *StData*, $LdAddr_2$, *DMA* and from the exit $data_{\text{out}}$ of unit DATAMEM respectively (see C.8). Note also that for notational simplicity we declare the register *Dat* to be also an IOS exit.

Definitions:

val_to_store = if *st_data* then *val_from_RF* else *status_register*
val_to_load = if *ld_addr* then DMA_2 else *Dat*

Transition rules:

delay_DMA $\equiv$
$\equiv \ DMA_2 := dma_{\text{in}}$

Note that this update of DMA_2 is in accordance with the view of DMA_2 as delay register for *DMA*; namely in C.8 dma_{in} will be replaced by *DMA*.

read_mem $\equiv$
$\equiv$ *Dat* := *val_from_mem*

A.9 Unit EVALCOND (Evaluation of Branch Conditions)

Entries: *which_cond*; $carry_{\text{in}}$, $\mathit{if\_status}_{\text{in}}$, neg_{in}, $overflow_{\text{in}}$, $zero_{\text{in}}$

Exits: *yes_no*

Definitions:

$$\mathit{yes\_no} = \mathit{eval\_cond}(\mathit{which\_cond}, carry_{\text{in}}, \mathit{if\_status}_{\text{in}}, neg_{\text{in}}, overflow_{\text{in}}, zero_{\text{in}})$$

Note that *yes_no* will be connected to the corresponding DECODE entry (see C.10).

A.10 Unit DECODE (Instruction Decoding)

Entries: $math\_code$, io_code, yes_no

Exits: pma_mux_{out}, abs_{out}, $halt_{\text{out}}$; MAC_2, WER, $EnableCC$, $MDCtrl$;
$MemWE$, $StData$, WED, $LdAddr$, $WrMask$, $Cancel$

Abbreviations:

$$EnableCC \equiv En\_Carry, En\_Divz, En\_Neg, En\_oVerflow, En\_eXtend, En\_Zero$$
$$MDCtrl \equiv MulIn, StartDiv, MDMux, AddMul$$

Definitions:

$$abs_{\text{out}} = IOC \notin \{\texttt{LDPA}, \texttt{SKIP}\}$$
$$pma\_mux_{\text{out}} = \text{if } (io\_code \in \{\texttt{JUMP}, \texttt{SKIP}\} \wedge yes\_no \wedge \neg Cancel) \text{ then } 2 \text{ else } 1$$
$$halt_{\text{out}} = (io\_code = \texttt{HALT}) \wedge \neg Cancel$$

Transition rules:

```
MAC_decode ≡
  ≡ $MAC_2 := math\_code$
    $WER := (math\_code \in MATH\_OP \setminus \{\texttt{CMP}, \texttt{MUL1}, \texttt{DIV1}\}) \wedge \neg Cancel$
```

Note that the update of MAC_2 is in accordance with the view of MAC_2 as delay of MAC; indeed $math\_code$ is a DECODE entry which in C.10 will be replaced by MAC.

```
decode_EnableCC ≡
  ≡ $En\_Carry := (math\_code \in LOGIC \cup ADDER) \wedge \neg Cancel$
    $En\_Divz := (math\_code = \texttt{DIV1}) \wedge \neg Cancel$
    $En\_Neg := (math\_code \in MATH\_OP \setminus \{\texttt{MUL1}, \texttt{DIV1}\}) \wedge \neg Cancel$
    $En\_oVerflow := (math\_code \in LOGIC \cup ADDER \cup \{\texttt{ASH}\}) \wedge \neg Cancel$
    $En\_eXtend := (math\_code \in SHIFTER) \wedge \neg Cancel$
    $En\_Zero := (math\_code \in MATH\_OP \setminus \{\texttt{MUL1}, \texttt{DIV1}\}) \wedge \neg Cancel$

decode_MDCtrl ≡
  ≡ $MulIn := (math\_code \neq \texttt{MUL1}) \vee Cancel$
    $StartDiv := (math\_code = \texttt{DIV1}) \wedge \neg Cancel$
    $MDMux := ($ if $(math\_code = \texttt{DIV2})$ then 2
                 else if $(math\_code = \texttt{MUL3})$ then 1
                 else 0 $)(1 - Cancel)$
    $AddMul := (math\_code = \texttt{MULA2}) \wedge \neg Cancel$

IOC_decode ≡
  ≡ $MemWE := (io\_code \in \{\texttt{ST}, \texttt{STSR}\}) \wedge \neg Cancel$
    $StData := (io\_code = \texttt{ST}) \wedge \neg Cancel$
    $WED := (io\_code \in \{\texttt{LD}, \texttt{LDA}, \texttt{LDPA}\}) \wedge \neg Cancel$
    $LdAddr := (io\_code \in \{\texttt{LDA}, \texttt{LDPA}\}) \wedge \neg Cancel$
    $WrMask := (io\_code = \texttt{LDSR}) \wedge \neg Cancel$
    $Cancel := (io\_code \in \{\texttt{JUMP}, \texttt{SKIP}\} \wedge yes\_no \wedge \neg Cancel)$
```

A.11 Registers

In order to implement pipelining, we need also to introduce some delay registers. As all the registers behave in the same way, we can define the behaviour of a "register unit" by means of the following schema (describing a generic register X)[9]:

Entries: $X.in$

Exits: $X.out$

Definitions:

$X.out = X$

Transition rules:

write_register_X $\equiv$
$\equiv \; X := X.in$

(Recall that, for brevity, we write simply X in place of $X.out$ within terms and transition rules).

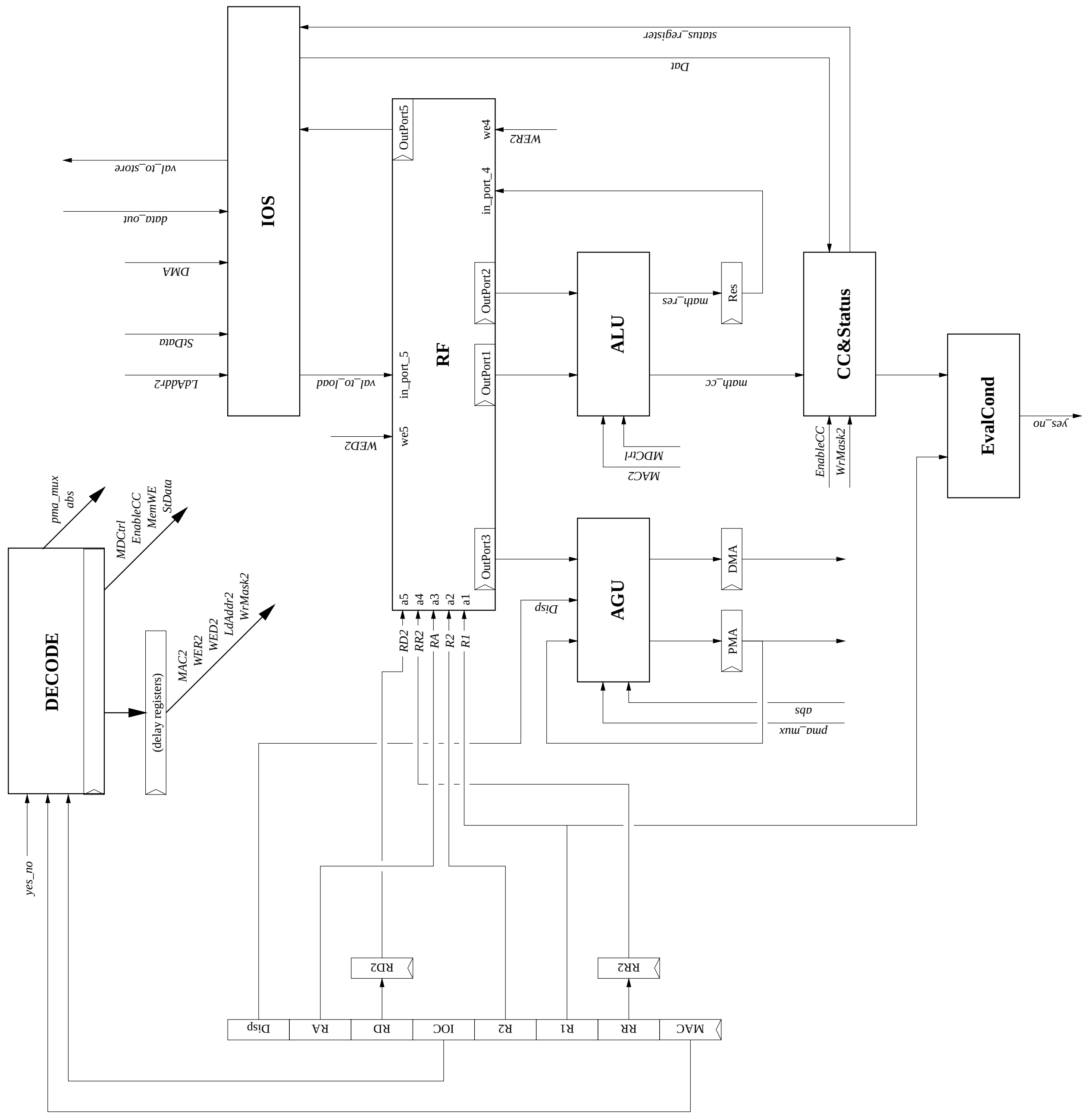
In the zCPU we have the following delay registers: $WER_2, RR_2, WED_2, RD_2, LdAddr_2, WrMask_2$.
We have also three other registers in the zCPU:

- *Res*, used as temporary storage for the ALU results;
- *PMA* (*program memory address*), the program counter;
- *DMA* (*data memory address*), containing the computed address needed for load/store operations.

[9]See also 1.3 for more details on internal registers.

B ZCPU Block Diagram

The following block diagram represents pictorially the zCPU at the level of its main building blocks. It is intended as an help to get an intuitive idea of what is formally defined in Appendices A, C and D, rather than as a complete reference (for example, the units DATAMEM and PROGMEM are not represented, as well as some other items which are instead included in the textual description).



C Composition of the zCPU units

This appendix contains the identifications of exits and entries, which are needed to compose the zCPU units described above, according to the block diagram shown in Appendix B. We proceed in the same order as in Appendix A, defining a substitution for the entries of each unit presented there.

In order to resolve possible ambiguities between names used within definitions of different units, as well as to ease the task of following the connection paths between the units, we adopt the notational convention of prefixing names of exits by the unit name (for example, the exit $carry_{\mathrm{out}}$ of the ALU unit is globally identified as ALU.$carry_{\mathrm{out}}$).

For more clarity and consistency of notation, we also use similar prefixes before the name of registers defined in A.11: the convention here is that the prefix of the register name is the name of the unit which provides the value to be written in the register (see C.11).

However, each register has a unique name, which allows us to remove all the prefixes from register names in the final description of the zCPU (Appendix D).

C.1 Unit RF (Register File)

$$
\begin{array}{lcl}
addr_1 & = & \mathrm{INSTR.}\mathit{R1} \\
addr_2 & = & \mathrm{INSTR.}\mathit{R2} \\
addr_3 & = & \mathrm{INSTR.}\mathit{RA} \\
addr_4 & = & \mathrm{INSTR.}\mathit{RR}_2 \\
in\_port_4 & = & \mathrm{ALU.}\mathit{Res} \\
write\_enable_4 & = & \mathrm{DECODE.}\mathit{WER}_2 \\
addr_5 & = & \mathrm{INSTR.}\mathit{RD}_2 \\
in\_port_5 & = & \mathrm{IOS.}\mathit{val\_to\_load} \\
write\_enable_5 & = & \mathrm{DECODE.}\mathit{WED}_2
\end{array}
$$

C.2 Unit ALU (ALU/Multiply/Divide)

$$
\begin{array}{lcl}
math\_code & = & \mathrm{DECODE.}\mathit{MAC}_2 \\
op_1 & = & \mathrm{RF.}\mathit{OutPort1} \\
op_2 & = & \mathrm{RF.}\mathit{OutPort2} \\
carry_{\mathrm{in}} & = & \mathrm{CC\&STATUS.}\mathit{carry}_{\mathrm{out}} \\
extend_{\mathrm{in}} & = & \mathrm{CC\&STATUS.}\mathit{extend}_{\mathrm{out}} \\
zero_{\mathrm{in}} & = & \mathrm{CC\&STATUS.}\mathit{zero}_{\mathrm{out}} \\
mul\_in & = & \mathrm{DECODE.}\mathit{MulIn} \\
start\_div & = & \mathrm{DECODE.}\mathit{StartDiv} \\
md\_mux & = & \mathrm{DECODE.}\mathit{MDMux} \\
add\_mul & = & \mathrm{DECODE.}\mathit{AddMul}
\end{array}
$$

where

$$
\begin{array}{lcl}
\mathrm{CC\&STATUS.}\mathit{carry}_{\mathrm{out}} & \equiv & \mathrm{CC\&STATUS.}\mathit{status\_register}[1] \\
\mathrm{CC\&STATUS.}\mathit{extend}_{\mathrm{out}} & \equiv & \mathrm{CC\&STATUS.}\mathit{status\_register}[21] \\
\mathrm{CC\&STATUS.}\mathit{zero}_{\mathrm{out}} & \equiv & \mathrm{CC\&STATUS.}\mathit{status\_register}[25]
\end{array}
$$

C.3 Unit CC&STATUS (Condition Codes & Status Register)

$\mathit{carry}_{\text{in}}$	=	$\text{ALU}.\mathit{carry}_{\text{out}}$
$\mathit{divz}_{\text{in}}$	=	$\text{ALU}.\mathit{divz}_{\text{out}}$
neg_{in}	=	$\text{ALU}.\mathit{neg}_{\text{out}}$
$\mathit{overflow}_{\text{in}}$	=	$\text{ALU}.\mathit{overflow}_{\text{out}}$
$\mathit{extend}_{\text{in}}$	=	$\text{ALU}.\mathit{extend}_{\text{out}}$
$\mathit{zero}_{\text{in}}$	=	$\text{ALU}.\mathit{zero}_{\text{out}}$
$\mathit{if\_status}_{\text{in}}$		[external function]
$\mathit{halt}_{\text{in}}$	=	$\text{DECODE}.\mathit{halt}_{\text{out}}$
$\mathit{parity\_err}_{\text{in}}$	=	$\mathit{parity\_check}(\mathit{PMD})$
$\mathit{en\_carry}$	=	$\text{DECODE}.\mathit{En\_Carry}$
$\mathit{en\_divz}$	=	$\text{DECODE}.\mathit{En\_Divz}$
$\mathit{en\_neg}$	=	$\text{DECODE}.\mathit{En\_Neg}$
$\mathit{en\_overflow}$	=	$\text{DECODE}.\mathit{En\_oVerflow}$
$\mathit{en\_extend}$	=	$\text{DECODE}.\mathit{En\_eXtend}$
$\mathit{en\_zero}$	=	$\text{DECODE}.\mathit{En\_Zero}$
$\mathit{write\_mask}$	=	$\text{DECODE}.\mathit{WrMask}_2$
val	=	$\text{IOS}.\mathit{Dat}$

C.4 Unit AGU (Address Generation Unit)

abs_{in}	=	$\text{DECODE}.\mathit{abs}_{\text{out}}$
pma_{in}	=	$\text{AGU}.\mathit{PMA}$
base	=	$\text{RF}.\mathit{OutPort3}$
disp	=	$\text{INSTR}.\mathit{Disp}$
$\mathit{pma\_mux}_{\text{in}}$	=	$\text{DECODE}.\mathit{pma\_mux}_{\text{out}}$

C.5 Unit PROGMEM (Program Memory)

addr	=	$\text{AGU}.\mathit{PMA}$

C.6 Unit DATAMEM (Data Memory)

addr	=	$\text{AGU}.\mathit{DMA}$
$\mathit{data}_{\text{in}}$	=	$\text{IOS}.\mathit{val\_to\_store}$
$\mathit{mem\_write\_enable}$	=	$\text{DECODE}.\mathit{MemWE}$

C.7 Unit INSTR (Instruction Fetch)

pmd_{in}	=	$\text{PROGMEM}.\mathit{pmd}_{\text{out}}$

C.8 Unit IOS (Input/Output Subsystem)

$$\begin{aligned}
\mathit{st\_data} &= \text{DECODE}.\mathit{StData}\\
\mathit{val\_from\_RF} &= \text{RF}.\mathit{OutPort5}\\
\mathit{status\_register} &= \text{CC\&STATUS}.\mathit{status\_register}\\
\mathit{ld\_addr} &= \text{DECODE}.\mathit{LdAddr}_2\\
\mathit{dma}_{\text{in}} &= \text{AGU}.\mathit{DMA}\\
\mathit{val\_from\_mem} &= \text{DATAMEM}.\mathit{data}_{\text{out}}
\end{aligned}$$

C.9 Unit EVALCOND (Evaluation of Branch Conditions)

$$\begin{aligned}
\mathit{which\_cond} &= \text{INSTR}.\mathit{R1}\\
\mathit{carry}_{\text{in}} &= \text{CC\&STATUS}.\mathit{carry}_{\text{out}}\\
\mathit{if\_status}_{\text{in}} &= \text{CC\&STATUS}.\mathit{if\_status}_{\text{out}}\\
\mathit{neg}_{\text{in}} &= \text{CC\&STATUS}.\mathit{neg}_{\text{out}}\\
\mathit{overflow}_{\text{in}} &= \text{CC\&STATUS}.\mathit{overflow}_{\text{out}}\\
\mathit{zero}_{\text{in}} &= \text{CC\&STATUS}.\mathit{zero}_{\text{out}}
\end{aligned}$$

where

$$\begin{aligned}
\text{CC\&STATUS}.\mathit{carry}_{\text{out}} &\equiv \text{CC\&STATUS}.\mathit{status\_register}[1]\\
\text{CC\&STATUS}.\mathit{if\_status}_{\text{out}} &\equiv \text{CC\&STATUS}.\mathit{status\_register}[9]\\
\text{CC\&STATUS}.\mathit{neg}_{\text{out}} &\equiv \text{CC\&STATUS}.\mathit{status\_register}[13]\\
\text{CC\&STATUS}.\mathit{overflow}_{\text{out}} &\equiv \text{CC\&STATUS}.\mathit{status\_register}[17]\\
\text{CC\&STATUS}.\mathit{zero}_{\text{out}} &\equiv \text{CC\&STATUS}.\mathit{status\_register}[25]
\end{aligned}$$

C.10 Unit DECODE (Instruction Decoding)

$$\begin{aligned}
\mathit{math\_code} &= \text{INSTR}.\mathit{MAC}\\
\mathit{io\_code} &= \text{INSTR}.\mathit{IOC}\\
\mathit{yes\_no} &= \text{EVALCOND}.\mathit{yes\_no}
\end{aligned}$$

C.11 Registers

$$\begin{aligned}
\text{DECODE}.\mathit{WER}_2.\mathit{in} &= \text{DECODE}.\mathit{WER}\\
\text{INSTR}.\mathit{RR}_2.\mathit{in} &= \text{INSTR}.\mathit{RR}\\
\text{DECODE}.\mathit{WED}_2.\mathit{in} &= \text{DECODE}.\mathit{WED}\\
\text{INSTR}.\mathit{RD}_2.\mathit{in} &= \text{INSTR}.\mathit{RD}\\
\text{DECODE}.\mathit{LdAddr}_2.\mathit{in} &= \text{DECODE}.\mathit{LdAddr}\\
\text{DECODE}.\mathit{WrMask}_2.\mathit{in} &= \text{DECODE}.\mathit{WrMask}\\
\text{ALU}.\mathit{Res}.\mathit{in} &= \text{ALU}.\mathit{math\_res}_{\text{out}}\\
\text{AGU}.\mathit{PMA}.\mathit{in} &= \text{AGU}.\mathit{pma}_{\text{out}}\\
\text{AGU}.\mathit{DMA}.\mathit{in} &= \text{AGU}.\mathit{dma}_{\text{out}}
\end{aligned}$$

D Full definition of the zCPU

This appendix contains the full definition of the zCPU processor. It is obtained by combining the units of Appendix A, performing the substitutions corresponding to the equations listed in Appendix C and to the equations defining the exits of the different units. For further explanations see [13]. Note that the zCPU itself can be seen as a unit, with its own entries and exits.

Unit zCPU

Entries: $\mathit{if\_status}_{\mathrm{in}}$

Exits: $\mathit{zcpu\_ex}$

Abbreviations:

$$\begin{array}{rcl}
\mathit{absolute} & \equiv & \mathit{IOC} \notin \{\,\mathtt{LDPA}, \mathtt{SKIP}\,\} \\
\mathit{yes\_no} & \equiv & \mathit{eval\_cond}(\mathit{R1}, \mathit{Carry}, \mathit{iFstatus}, \mathit{Neg}, \mathit{oVerflow}, \mathit{Zero}) \\
\mathit{pma\_mux} & \equiv & \text{if } (\mathit{IOC} \in \{\,\mathtt{JUMP}, \mathtt{SKIP}\,\} \wedge \mathit{yes\_no} \ \wedge \ \neg \mathit{Cancel}) \\
 & & \text{then } 2 \\
 & & \text{else } 1 \\
\mathit{val\_to\_store} & \equiv & \text{if } \mathit{StData} \text{ then } \mathit{OutPort}_5 \text{ else } \mathit{status\_register} \\
\mathit{val\_to\_load} & \equiv & \text{if } \mathit{LdAddr}_2 \text{ then } \mathit{DMA}_2 \text{ else } \mathit{Dat} \\
\mathit{status\_register} & \equiv & \mathit{Ex\_Parity} \,.\, \mathit{ParityErr} \,.\, \mathit{Ex\_Halt} \,.\, \mathit{Halt} \\
 & & .\, \mathit{Msk\_Zero} \,.\, \mathit{Ex\_Zero} \,.\, \mathit{Zero} \,.\, 0 \\
 & & .\, \mathit{Msk\_eXtend} \,.\, \mathit{Ex\_eXtend} \,.\, \mathit{eXtend} \,.\, 0 \\
 & & .\, \mathit{Msk\_oVerflow} \,.\, \mathit{Ex\_oVerflow} \,.\, \mathit{oVerflow} \,.\, 0 \\
 & & .\, \mathit{Msk\_Neg} \,.\, \mathit{Ex\_Neg} \,.\, \mathit{Neg} \,.\, 0 \\
 & & .\, \mathit{Msk\_iFstatus} \,.\, \mathit{Ex\_iFstatus} \,.\, \mathit{iFstatus} \,.\, 0 \\
 & & .\, \mathit{Msk\_Divz} \,.\, \mathit{Ex\_Divz} \,.\, \mathit{Divz} \,.\, 0 \\
 & & .\, \mathit{Msk\_Carry} \,.\, \mathit{Ex\_Carry} \,.\, \mathit{Carry} \,.\, \mathit{ExcpReg}
\end{array}$$

Integrity constraints:

$$\begin{array}{l}
\mathit{WER}_2 \Rightarrow \mathit{RR}_2 \notin \{\, \mathit{R1}, \mathit{R2}, \mathit{RA}, \mathit{RD}_2 \,\} \\
\mathit{WED}_2 \Rightarrow \mathit{RD}_2 \notin \{\, \mathit{R1}, \mathit{R2}, \mathit{RA} \,\} \\
\mathit{WER}_2 \wedge \mathit{WED}_2 \Rightarrow \mathit{RR}_2 \neq \mathit{RD}_2
\end{array}$$

Note that it is possible to ensure that these contraints are satisfied by imposing appropriate contraints on the compiler.

Definitions:

$$\begin{array}{rcl}
\mathit{zcpu\_ex} & = & \mathit{Halt} \vee \mathit{ParityErr} \vee (\mathit{iFstatus} \wedge \mathit{Msk\_iFstatus}) \vee \\
 & & \vee\, (\mathit{Carry} \wedge \mathit{Msk\_Carry}) \vee (\mathit{Divz} \wedge \mathit{Msk\_Divz}) \vee \\
 & & \vee\, (\mathit{Neg} \wedge \mathit{Msk\_Neg}) \vee (\mathit{oVerflow} \wedge \mathit{Msk\_oVerflow}) \vee \\
 & & \vee\, (\mathit{eXtend} \wedge \mathit{Msk\_eXtend}) \vee (\mathit{Zero} \wedge \mathit{Msk\_Zero})
\end{array}$$

Transition rules:

instruction_fetch $\equiv$
$\equiv$ $PMD := instr(PMA)$

MAC_decode $\equiv$
$\equiv$ $MAC_2 := MAC$
$WER := (MAC \in MATH\_OP \setminus \{\texttt{CMP}, \texttt{MUL1}, \texttt{DIV1}\}) \wedge \neg Cancel$

decode_EnableCC $\equiv$
$\equiv$ $En\_Carry := (MAC \in LOGIC \cup ADDER) \wedge \neg Cancel$
$En\_Divz := (MAC = \texttt{DIV1}) \wedge \neg Cancel$
$En\_Neg := (MAC \in MATH\_OP \setminus \{\texttt{MUL1}, \texttt{DIV1}\}) \wedge \neg Cancel$
$En\_oVerflow := (MAC \in LOGIC \cup ADDER \cup \{\texttt{ASH}\}) \wedge \neg Cancel$
$En\_eXtend := (MAC \in SHIFTER) \wedge \neg Cancel$
$En\_Zero := (MAC \in MATH\_OP \setminus \{\texttt{MUL1}, \texttt{DIV1}\}) \wedge \neg Cancel$

decode_MDCtrl $\equiv$
$\equiv$ $MulIn := (MAC \neq \texttt{MUL1}) \vee Cancel$
$StartDiv := (MAC = \texttt{DIV1}) \wedge \neg Cancel$
$MDMux := (\text{if } (MAC = \texttt{DIV2}) \text{ then } 2 \text{ else if } (MAC = \texttt{MUL3}) \text{ then } 1 \text{ else } 0)(1 - Cancel)$
$AddMul := (MAC = \texttt{MULA2}) \wedge \neg Cancel$

IOC_decode $\equiv$
$\equiv$ $MemWE := (IOC \in \{\texttt{ST}, \texttt{STSR}\}) \wedge \neg Cancel$
$StData := (IOC = \texttt{ST}) \wedge \neg Cancel$
$WED := (IOC \in \{\texttt{LD}, \texttt{LDA}, \texttt{LDPA}\}) \wedge \neg Cancel$
$LdAddr := (IOC \in \{\texttt{LDA}, \texttt{LDPA}\}) \wedge \neg Cancel$
$WrMask := (IOC = \texttt{LDSR}) \wedge \neg Cancel$
$Cancel := (IOC \in \{\texttt{JUMP}, \texttt{SKIP}\} \wedge yes\_no \;\wedge\; \neg Cancel)$

Note the condition on the right side of the *Cancel* update expresses that a jump is going to take place, which implies that the instruction word following the jump, which has already been fetched, must be canceled (i.e. decoded as a "no operation"). This works under the compiler condition that there are never two immediately successive jumps.

delay_WER_RR $\equiv$
$\equiv$ $WER_2 := WER$
$RR_2 := RR$

IOC_delay $\equiv$
$\equiv$ $WED_2 := WED$
$LdAddr_2 := LdAddr$
$WrMask_2 := WrMask$

delay_RD $\equiv$
$\equiv$ $RD_2 := RD$

read_operands $\equiv$
$\equiv$ $OutPort_1 := reg(R1)$
$OutPort_2 := reg(R2)$

```
read_addr_reg  $\equiv$
  $\equiv$  $OutPort_3 := reg(RA)$

read_data_reg  $\equiv$
  $\equiv$  $OutPort_5 := reg(RD_2)$

write_result  $\equiv$
  $\equiv$  **if**  $WER_2$
        **then**  $reg(RR_2) := Res$

write_dest_reg  $\equiv$
  $\equiv$  **if**  $WED_2$
        **then**  $reg(RD_2) := val\_to\_load$

calc_result  $\equiv$
  $\equiv$  $Res := math\_res(\ MAC_2, MulIn, StartDiv, MDMux, AddMul,$
                          $OutPort_1, OutPort_2, Carry, eXtend\ )$

mul_rule  $\equiv$
  $\equiv$  **if**  $MulIn = 0$
        **then**  $MulOp1 := OutPort_1$
                  $MulOp2 := OutPort_2$
                  $MulStep := 1$
        **else**  $MulStep := MulStep + 1$

div_rule  $\equiv$
  $\equiv$  **if**  $StartDiv = 1$
        **then**  $DivOp1 := OutPort_1$
                  $DivOp2 := OutPort_2$
                  $DivStep := 1$
        **else**  $DivStep := DivStep + 1$

write_cc_math  $\equiv$
  $\equiv$  **if**  $En\_Carry$
        **then**  $Carry\ := carry(MAC_2, OutPort_1, OutPort_2, Carry)$

        **if**  $En\_Divz$
        **then**  $Divz\ := divz(OutPort_2)$

        **if**  $En\_Neg$
        **then**  $Neg\ := neg(\ MAC_2, MulIn, StartDiv, MDMux, AddMul,$
                            $OutPort_1, OutPort_2, Carry, eXtend\ )$

        **if**  $En\_oVerflow$
        **then**  $oVerflow\ := overflow(MAC_2, OutPort_1, OutPort_2, Carry)$

        **if**  $En\_eXtend$
        **then**  $eXtend\ := extend(OutPort_1, OutPort_2)$

        **if**  $En\_Zero$
        **then**  $Zero\ := zero(\ MAC_2, MulIn, StartDiv, MDMux, AddMul,$
                             $OutPort_1, OutPort_2, Carry, eXtend, Zero\ )$
```

```
write_cc_ifstatus $\equiv$
  $\equiv$ $iFstatus := if\_status_{\mathrm{in}}$
write_cc_halt $\equiv$
  $\equiv$ $Halt := (IOC = \texttt{HALT}) \wedge \neg Cancel$
write_cc_parity $\equiv$
  $\equiv$ $ParityErr := parity\_check(PMD)$
exception_rule $\equiv$
  $\equiv$ if $\neg ExcpReg$
      then $ExcpReg := zcpu\_ex$
           $Ex\_Halt := Halt$
           $Ex\_Parity := ParityErr$
           $Ex\_iFstatus := iFstatus$
           $Ex\_Carry := Carry$
           $Ex\_Divz := Divz$
           $Ex\_Neg := Neg$
           $Ex\_oVerflow := oVerflow$
           $Ex\_eXtend := eXtend$
           $Ex\_Zero := Zero$
calc_PMA $\equiv$
  $\equiv$ $PMA :=$ if $pma\_mux = 0$ then $0$
              else if $pma\_mux = 1$ then $PMA + 1$
              else $(1 - absolute)PMA + OutPort_3 + Disp$
calc_DMA $\equiv$
  $\equiv$ $DMA := (1 - absolute)PMA + OutPort_3 + Disp$
delay_DMA $\equiv$
  $\equiv$ $DMA_2 := DMA$
write_mem $\equiv$
  $\equiv$ if $MemWE$
      then $mem(DMA) := val\_to\_store$
read_mem $\equiv$
  $\equiv$ $Dat := mem(DMA)$
write_mask $\equiv$
  $\equiv$ if $WrMask_2$
      then update_mask($val\_to\_load$)
      where update_mask($val$) $\equiv$
              $\equiv$ $Msk\_Carry := val[3]$
                 $Msk\_Divz := val[7]$
                 $Msk\_iFstatus := val[11]$
                 $Msk\_Neg := val[15]$
                 $Msk\_oVerflow := val[19]$
                 $Msk\_eXtend := val[23]$
                 $Msk\_Zero := val[27]$
```

References

[1] The APE100 Collaboration, *APE100 Primer*, INFN, A100/PRIM/G02.

[2] The APE100 Collaboration, *zCPU User Guide*, INFN, A100/ZCPU/G01.

[3] A. Bartoloni et al., *A Hardware Implementation of the APE100 Architecture*, in: International Journal of Modern Physics, C 4 (1993), p. 969.

[4] A. Bartoloni et al., *The Software of the APE100 Processor*, in: International Journal of Modern Physics, C 4 (1993), p. 955.

[5] A. Bartoloni et al., *APEmille: A Parallel Processor in the Teraflops Range*, manuscript, 1995.

[6] 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), p. 543.

[7] 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.

[8] E. Börger, I. Durdanovic, *Correctness of Compiling Occam to Transputer Code*, March 1995.

[9] 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.

[10] E. Börger, S. Mazzanti, *A correctness proof for pipelining in RISC architectures* (manuscript).

[11] 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).

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

[13] G. Del Castillo, *Descrizione matematica dell'architettura parallela APE100*, Tesi di Laurea (in Italian), Università di Pisa, Pisa, 1995.

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