

An Architecture Description Language for Massively Parallel Processor Architectures

Alexey Kupriyanov¹, Frank Hannig¹, Dmitrij Kissler¹, Jürgen Teich¹,
Rainer Schaffer², and Renate Merker²

¹ Department of Computer Science 12, Hardware-Software-Co-Design,
University of Erlangen-Nuremberg, Germany.
{kupriyanov, hannig, teich}@cs.fau.de

² Institute of Circuits and Systems,
Department of Electrical Engineering and Information Technology,
Dresden University of Technology, Germany.
{schaffer, merker}@iee1.et.tu-dresden.de

Abstract. In this paper, we introduce an architecture description language for modeling, simulation, and evaluation of massively parallel processor architectures that are designed for special purpose applications from the domain of embedded systems. The architectural description of the processor system is supposed to be done according to two abstraction levels. Architectural parameters of processor elements are characterized on *processor level* and the interaction between processors (i.e., interconnect topology, positioning of the processors, etc.) is described on the *array level*. Key features, semantic, and technical innovations of the proposed architecture description language are demonstrated in this paper.

1 Introduction

Today, the steady technological progress in integration densities and modern microtechnology allows implementation of hundreds of 32-bit microprocessors and more on a single die. Furthermore, the functionality of the microprocessors is increasing continuously, e.g., parallel processing of data with low accuracy (8 bit or 16 bit) within each microprocessor. Due to these advances, massively parallel data processing has become possible in portable and other embedded systems. These devices have to handle increasingly computationally-intensive algorithms like video processing (H.264) or other digital signal processing tasks (3G). Hence, modeling languages will be of great importance during the entire design flow for *processor arrays*. In this paper we propose an *architecture description language* (ADL) to characterize parallel processor architectures.

The rest of the paper is structured as follows: In Section 2, an overview of related work is given. In Section 3, the class of considered target architectures is introduced. The basic modeling concepts for parallel processor architectures are described in Section 4. A case study is presented in Section 5. Finally in Section 6, the paper is concluded with a summary of our achievements and an outlook of future work directions.

2 Related Work

Many architecture description languages have been developed in the field of retargetable compilation. In the following, we list only some of the most significant ADLs. For instance, the hardware description language nML [3] permits concise, hierarchical processor descriptions in a behavioral style. nML is used in the CBC/SIGH/SIM framework [2] and the CHESS system [6]. The machine description language LISA [15] is the basis for a retargetable compiled simulator approach developed at RWTH Aachen, Germany. The project focuses on fast simulator generation for already existing architectures to be modeled in LISA. At the ACES laboratory of the University of California, Irvine, the architecture description language EXPRESSION [8] has been developed. From an EXPRESSION description of an architecture, the retargetable

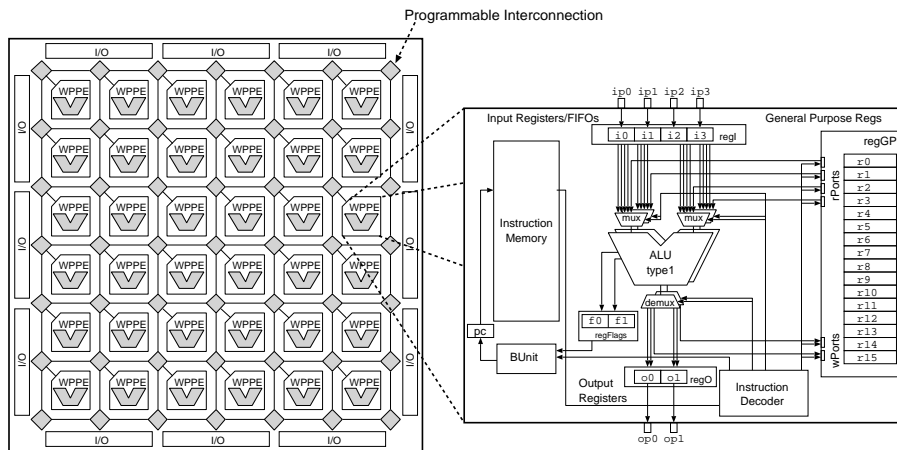


Fig. 1. Example of a WPPA with parameterizable processing elements (WPPEs). A WPPE consists of a processing unit which contains a set of *functional units*. Some functional units allow to compute sub-words in sub-word units in parallel. The processing unit is connected to input and output registers. A small data memory exists to temporary store computational results. An instruction sequencer exists as part of the control path which executes a set of control instructions from a local tiny program memory.

compiler Express and a cycle-accurate simulator can be automatically generated. The Trimaran system [19] has been designed to generate efficient VLIW code. It is based on a fixed basic architecture (HPL-PD) being parameterizable in the number of registers, the number of functional units, and operation latencies. Parameters of the machine are specified in the description language HMDES. MIMOLA [13] is one of RT-level ADLs. It was developed at the University of Kiel, Germany. Originally, it targeted at micro-architecture modeling and design. ISDL is one of Instruction Set Description Languages. It was developed at MIT and is used by the Aviv compiler [10] and the associated assembler. It was also used by the simulator generation system GENSIM [7]. The target architectures for ISDL are VLIW ASIPs. Maril is an ADL used by the retargetable compiler Marion [1]. It contains both instruction set information as well as coarse-grained structural information. The target architectures for Maril are RISC style processors only. There is no distinction between instruction and operation in Maril. TDL stands for target description language. It has been developed at Saarland University in Germany. The language is used in a retargetable post-pass assembly-based code optimization system called PROPAN [11]. Another architecture description language is PRMDL [17]. PRMDL stands for Philips Research Machine Description Language. The target architectures for PRMDL are clustered VLIW architectures. Finally, we refer to the Machine Markup Language (MAML) which has been developed in the BUILDABONG project [4]. MAML is used for the efficient architecture/compiler co-generation of ASIPs and VLIW processor architectures. For a more complete ADL's summary we point out to the surveys in [16], [18], and [14]. All these ADLs have in common that they have been developed for the design of single processor architectures such as ASIPs which might contain VLIW execution. But, to the best of our knowledge, there exists no ADL which covers the architectural aspects of massively parallel processor arrays. Of course, one could use hardware description languages such as Verilog or VHDL but these languages are too low level and offer only insufficient possibilities to describe behavioral aspects.

3 Weakly-Programmable Processor Arrays

In [9], we introduced a new class of massively parallel reconfigurable architectures we call *weakly-programmable arrays* (WPPA). In this paper we only briefly summarize the main concepts. Such architectures consist of an array of processing elements (PE) that may contain sub-

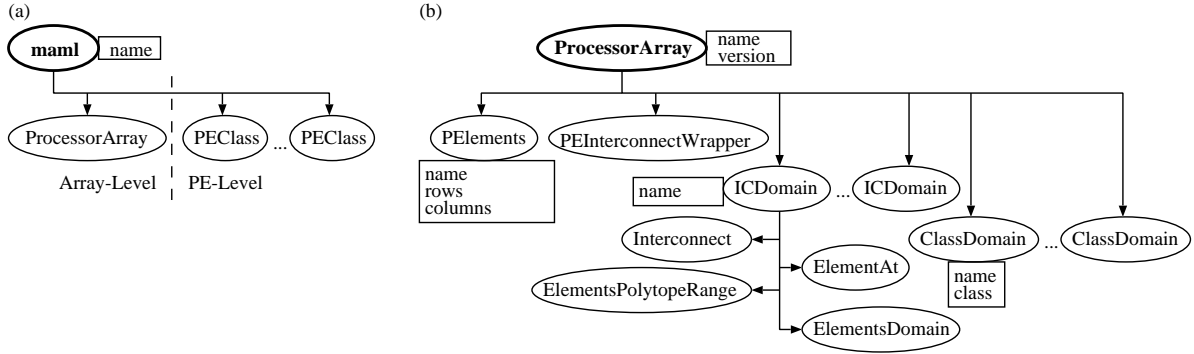


Fig. 2. In (a), root element `<maml>`. In (b), the `<ProcessorArray>` element.

word processing units with only very few memory and a regular interconnect structure. In order to efficiently implement a certain algorithm, each PE may implement only a certain function range. Also, the instruction set is limited and may be configured at compile-time or even dynamically at run-time. The PEs are called weakly-programmable because the control overhead of each PE is optimized and kept small. An example of such an architecture is shown in Fig. 1. The massively parallelism might be expressed by different types of parallelism: (i) several parallel working weakly-programmable processing elements (WPPEs), (ii) multiple functional units within one WPPE, and finally (iii) sub-word parallelism (SWP) within the WPPEs. WPPAs can be seen as a compromise between programmability and specialty by exploiting architectures realizing the full synergy of programmable processor elements and dedicated processing units.

4 WPPA Characterization with Machine Markup Language

Since design time and cost are critical aspects during the design of processor architectures it is important to provide efficient modeling and simulation techniques in order to evaluate architecture prototypes without actually designing them. In the scope of the methodology presented here, we are looking for a flexible reconfigurable architecture in order to find out trade-offs between different architecture peculiarities for a given set of applications. Therefore, a formal description of architectures' parameters is of great importance.

In order to allow the specification of massively parallel processor architectures we use the *MAchine Markup Language* (MAML) [4] and provide extensions that are needed for modeling WPPAs. MAML is based on the XML notation and is used for describing architecture parameters required by possible mapping methods such as partitioning, scheduling, functional unit and register allocation. Moreover, the parameters extracted from a MAML architectural description can be used for interactive visualization and simulation of the given processor architecture. The four main constraints of well-formed XML documents were followed in order to define MAML due to the XML standard: (i) there is exactly one root element, (ii) every start tag has a matching end tag, (iii) no tag overlaps another tag, and (iv) all elements and attributes must obey the naming constraints. A MAML document has one root element `<maml>` with an attribute `name`, specifying the name of the architecture. The architectural description of an entire WPPA can be subdivided into two main abstraction levels, the *array-level* describing parameters such as the topology of the interconnection, number and location of processor and I/O-ports, etc., and the *PE-level* describing the internal structure of each WPPE's type the WPPA may be composed of. The general structure of a MAML specification is shown in Fig. 2(a).

4.1 Array-Level Architecture Specification

The array-level properties of the WPPA are described in the body of a special MAML tag `<ProcessorArray>`. This tag specifies the parameters of the whole WPPA in general, i.e., the

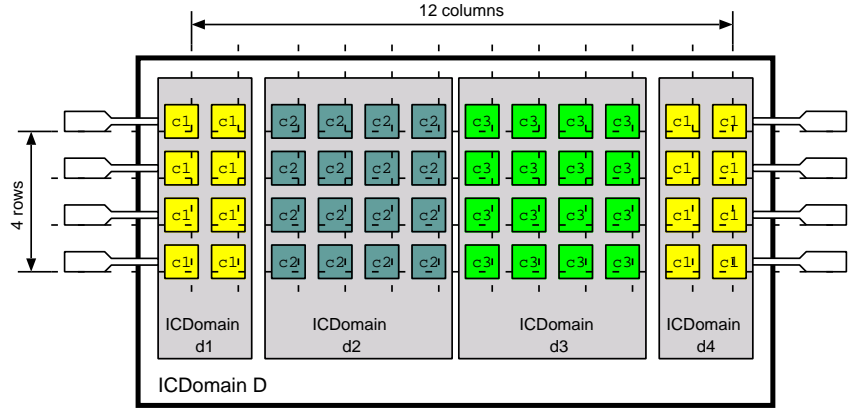


Fig. 3. Interconnect Domains and Class Domains representation.

name of the WPPA, the interconnect topology, the number and types of WPPEs, etc. Parameterizable topologies such as tree, ring, mesh, honeycomb, torus, etc. can be selected as interconnect domains. To the sake of brevity in the rest of the paper we discuss only mesh connected processor arrays. In this case, basis of our "drawing board" is a grid structure where processors can be placed on and parameters are the number of columns and rows.

The interconnect between processor array cells is one of the very important parameters of a WPPA. It is represented by a so-called *interconnect-wrapper* (IW) of PE.

The structure of the `<ProcessorArray>` element is shown in Fig. 2(b). This element contains the attributes `name` and `version` specifying the name of the processor array architecture and its version, respectively. It also contains a following set of subelements: `<PElements>`, `<PEInterconnectWrapper>`, `<ICDomain>`, `<ClassDomain>`. The multiple definition of two last subelements is admissible.

`<PElements>` (stands for *Processor Elements*) defines the number of PEs in the whole processor array, gives the referring name for them. The number of elements is specified as two-dimensional array with fixed number of rows (`rows` attribute) and columns (`cols` attribute). The number of rows multiplied by the number of columns is not necessarily the total number of processors within the array. Since as discussed above, the grid serves only as basis in order to place different types of processors, memories, and I/O-elements. Furthermore, each grid point does not necessarily correspond to one element because the size of the elements could be different. Here, size in terms of physical area is rather subordinate but the logical size in terms of connectors.

`<ICDomain>` (*Interconnect Domain*) specifies the set or domain of the processor elements with the same interconnect topology. The PEs here are either subsets of the set of PEs defined by the `<PElements>` tag or another domain. Recursive definition is not allowed.

`<ClassDomain>` specifies the set or domain of the processor elements with the same architectural structure (PE Class). The PEs here are either subsets of the set of PEs defined by the `<PElements>` tag or another domain. Recursive definition is not allowed.

In Fig. 3, an example of interconnect and class domains representation of a processor array architecture is presented. The memory and I/O-elements are not considered here. Four interconnect and three class domains are shown. For instance, the interconnect domain *d1* contains the PEs of class *c1*. The interconnect topology for the PEs in the interconnect domain *d2* is shown in Fig. 5.

In the following each subelement is presented in detail:

The `<PElements>` Element. This element contains the following attributes: `name`, `rows`, and `columns`. `<PElements>` defines the set of PEs that are used for constructing the whole

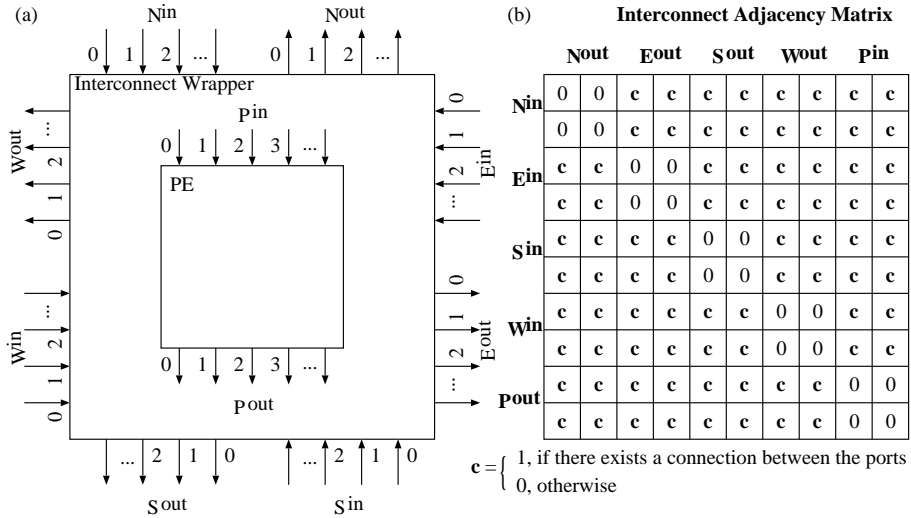


Fig. 4. In (a), Interconnect-Wrapper. In (b), Interconnect Adjacency Matrix.

processor array. The *name* attribute names the set of PEs. The PE classes are specified later on the PE-level of the architecture specification. The *rows* and *columns* attributes define the 2D size of the array of PEs.

For example: `<PElements name="pe" rows="4" columns="12">`. The set *pe* with possibly up to 48 PEs (4×12 array) is defined. Each element can be referred by the name of the set and the 2D indexes. In the example of Fig. 3, each PE can be referred as follows: `pe[1,1] . . . pe[4,12]`.

The Interconnect Domain <ICDomain> Element. The interconnect domain is used to specify the interconnect topology for the set of PEs. The *PE interconnect-wrapper* (IW) concept is used here for that (see Fig. 4(a)). The IW contains each PE. Each interconnect-wrapper has a constant number of inputs and outputs which are connected to the inputs and outputs of neighbor IW instances. In the most complex case, the IW is a configurable full crossbar switched matrix, but in practice, in the most cases, it is less complex since it is a compromise between routing flexibility and cost. Its configuration is specified by the so-called *Interconnect Adjacency Matrix* (IAM) (see Fig. 4(b)). The IW is represented as a rectangle around the PE and consists of input and output ports on the northern, eastern, southern, and western side of it. The number of input ports is represented by the N^{in} , E^{in} , S^{in} , and W^{in} for each side respectively. The same is for the output ports: N^{out} , E^{out} , S^{out} , and W^{out} . The numeration of the ports is done as shown in Fig. 4(a). The PE is placed inside of the interconnect-wrapper. The input ports P^{in} are shown on the top edge and the output ports P^{out} are shown on the bottom edge of the PE.

The IAM is shown in Fig. 4(b). The rows of this matrix depict the input ports of the IW, except the few last rows (dependent on the number of the PEs output ports), which represent the output ports of the PE. The columns represent the output ports of IW, except the few last columns (dependent on the number of the PEs input ports), which represent the input ports of the PE. The matrix contains the values c , which are "1" if there exists a connection between input and output ports, and "0" otherwise.

The <ICDomain> element has an attribute *name* and the set of following subelements: <Interconnect>, <ElementsPolytopeRange>, <ElementAt>, and <ElementsDomain>. It specifies the domain of the processor elements with the same interconnect topology. The PEs here are either the subsets of the set of PEs defined by the <PElements> tag or another domains. Recursive definition is not allowed. The *name* attribute names the domain. The <Interconnect> subelement specifies the interconnect network topology. The attribute *type* here defines the type

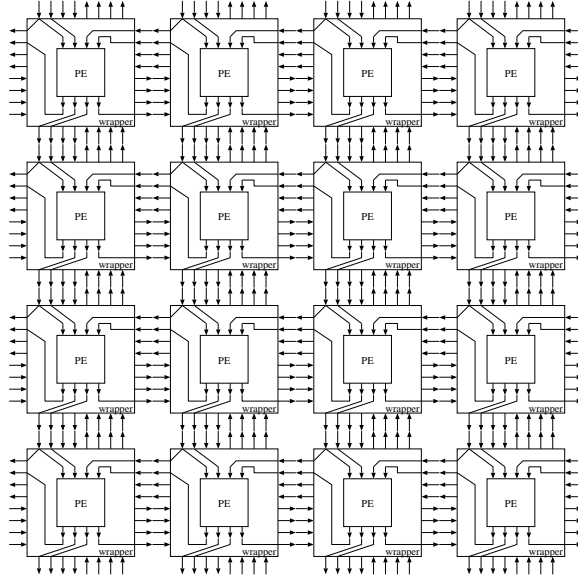


Fig. 5. Exemplary interconnect topology of the PEs from domain $d2$ in Fig. 3.

of the interconnect. The value of the *type* is one of the following: *static* or *dynamic*. They specify whether the defined interconnect is static at synthesis time or can be configured dynamically at run-time (i.e., configurable during processor array run-time). Dependent on interconnect type the different parameters in the body of the $\langle Interconnect \rangle$ subelement should be specified. As said earlier a lot of different interconnect topologies can be modeled using such approach.

The $\langle AdjacencyMatrix \rangle$ subelement defines the IAM of the interconnect-wrapper for the PEs which belong to the current domain. The matrix is described row by row using the tags $\langle NInput \rangle$, $\langle EInput \rangle$, $\langle SInput \rangle$, $\langle WInput \rangle$, and $\langle WOutput \rangle$ with the attributes *idx* and *row*, specifying the rows of the matrix (the rows with all zeros can be skipped). The value of *idx* defines the *index* of the matrix row or the port for the appropriate side of the IW. An example of the interconnect topology of PEs is shown in Fig. 5. The interconnect-wrapper has four input and four output ports on each of its edges. It contains a PE with four input and four output ports.

The $\langle ElementsPolytopeRange \rangle$ Subelement. This subelement is used to define a subset of PEs that are grouped together in order to organize one domain. The set of PEs is defined by the points of the integer lattice defined as follows:

$$\begin{pmatrix} i \\ j \end{pmatrix} = \left\{ \begin{pmatrix} i \\ j \end{pmatrix} \in \mathbb{Z}^2 \mid \begin{pmatrix} i \\ j \end{pmatrix} = L \cdot \begin{pmatrix} x \\ y \end{pmatrix} + m \wedge A \cdot \begin{pmatrix} x \\ y \end{pmatrix} \leq b \right\}$$

$A \cdot \begin{pmatrix} x \\ y \end{pmatrix} \leq b$ describes a polytope which is affinely transformed by $L \cdot \begin{pmatrix} x \\ y \end{pmatrix} + m$. An example of this concept is shown in Fig. 6. The processor array contains two domains of processor elements: domain $D1$ has a triangular shape and domain $D2$ is a set of PEs placed in shape of square.

The $\langle ElementAt \rangle$ Subelement. The $\langle ElementAt \rangle$ subelement is used to select one PE by the index of the row and the column.

The $\langle ElementsDomain \rangle$ Subelement. The $\langle ElementsDomain \rangle$ subelement contains an attribute *name*. The $\langle ElementsDomain \rangle$ subelement defines the subset of PEs specified in another domain. The *name* attribute specifies the name of the another domain. The recursive definition is not allowed here.

The PE Class Domain $\langle ClassDomain \rangle$ Element. Attributes of this element are: *name* and *class*. It also contains the following subelements: $\langle ElementsPolytopeRange \rangle$, $\langle ElementAt \rangle$, $\langle ElementsDomain \rangle$.

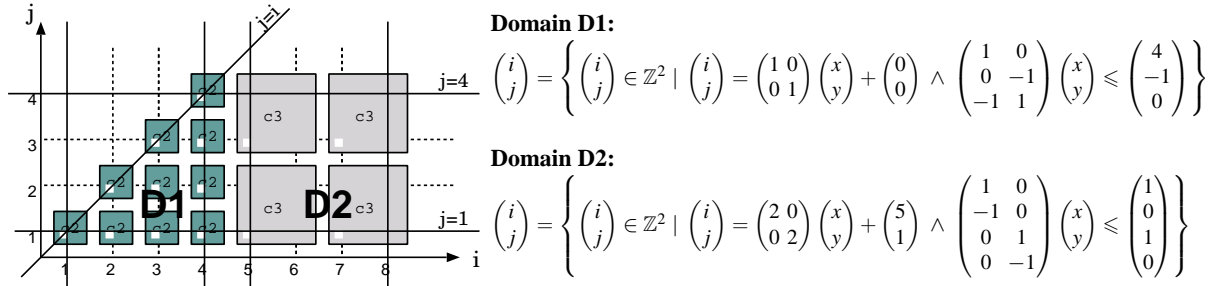


Fig. 6. Polytope domains representation.

$\langle \text{ClassDomain} \rangle$ specifies the set or domain of the processor elements with the same architectural structure (PE Class). The PEs here are either the subsets of the set of PEs defined by the $\langle \text{PElements} \rangle$ tag or another domain. Recursive definition is not allowed. The *name* attribute names the domain. The *class* attribute specifies the class of the PE architecture for all PEs in this domain. The classes of PE architecture are described in the PE-level section of the MAML description. The subelements $\langle \text{ElementsPolytopeRange} \rangle$ and $\langle \text{ElementsDomain} \rangle$ are the same as in the $\langle \text{ICDomain} \rangle$ element.

4.2 PE-Level Architecture Specification

A schematic view of a possible processor element model is depicted in Fig. 1. The processor element consists of three register files: input registers regI , output registers regO , and general purpose registers regGP . As a special register it has a *program counter* pc and a set of registers-flags from the register bank regFlags . Consequently, the registers from regI are given the names i plus the index from 0 to 3, as we have four input registers. In the same manner, the registers which belong to regO can be referred to with the names $o0$, $o1$. The registers of regGP are referred to by $r0$. . $r15$. The flag-registers are called $f0$, $f1$. Input and output registers are connected to the input and output ports: $\{ip0, ip1, ip2, ip3\}$ and $\{op0, op1\}$, respectively. Reading from and writing to the registers of any register bank is established through the read- and write ports (rPort , wPort). A WPPE can have one or several functional units (FUs) of the same or different ALU types which can increase the performance of a WPPE by enabling parallel computing by the execution of VLIW instructions. FUs with a wide word width might be also configured to provide sub-word parallelism (SWP). An instruction decoder decodes the processor instructions stored in the instruction memory and drives the multiplexers and demultiplexers which select registers and register banks for the source and target (result) operands of FUs.

The architectural properties of the PEs are defined by *PE-classes*. The properties of one class can be instantiated as well on one PE as on a set of PEs. PE-classes can extend or implement other already earlier defined PE-classes, thus providing larger MAML-code efficiency. PE-classes are defined by the $\langle \text{PEClass} \rangle$ tags.

The $\langle \text{PEClass} \rangle$ Element. The $\langle \text{PEClass} \rangle$ element contains the following attributes: *name*, *implements* and the following subelements: $\langle \text{Resources} \rangle$, $\langle \text{Register} \rangle$, $\langle \text{Resmap} \rangle$, $\langle \text{Opnames} \rangle$, $\langle \text{Operations} \rangle$, $\langle \text{Units} \rangle$. The *name* attribute names the PE-class. The *implements* attribute provides the name of another PE-class which inherits all subelements and parameters of the current PE-class. The further description of any subelement in the body of this class will overwrite the appropriate subelement. The *implements* attribute can contain no value which would mean that the PE-class is supposed to be constructed from the scratch.

The $\langle \text{Resources} \rangle$ Element. The attributes of this element are *name* and *num*. The description of communication components (i.e., read/write ports, busses, etc.) of the internal architec-

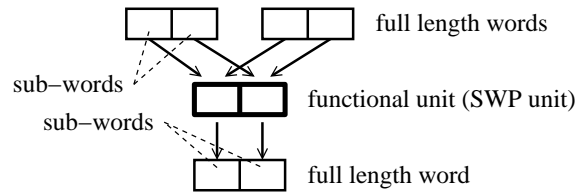


Fig. 7. Functional unit with two sub-words.

ture of PE is provided by the `<Resources>` element. The *name* attribute names the communication resource, and the *num* attribute sets the quantity of it. For example:

```
<Resource name = "wPort1" num = "4">.
```

The `<Register>` Element. In order to distinguish between ordinary general purpose register banks and I/O register banks, different tags are defined. Each of them has two attributes specifying the name of the register bank and the number of registers in it. Additionally, there might not be only simple I/O registers but also FIFOs or small FSMs to control the communication. The instruction memory is separated from the general memory by introduction of tag *InstructionMemory* with the memory size in bytes as attribute. The MAML element `<Register>` is used for specifying register banks of PE and contains the following subelements: `<Registerbank>`, `<InputRegisterbank>`, `<OutputRegisterbank>`, `<InstructionMemory>`, `<Memory>`, `<SpRegister>`. It defines such storage components as register banks, registers, memory, instruction memory, and FIFOs of the internal PE architecture.

`<Registerbank>` specifies the register bank by the definition of the storage elements attributes *bankname* and *num* (number of the registers in the register bank). `<SpRegister>` provides the specification of special registers in a certain register bank.

The `<Resmap>` Element. The `<Resmap>` element provides the specification of the so-called *resource-mapping-function* which assigns the read/write ports to the register banks, connects the functional units to the register banks through the buses or dedicated direct links, and defines the pipeline stages. The assignment of the ports is performed using the attribute *case_bank_of_get_rport* or *case_bank_of_get_wport*. There is an attribute *bank* which defines the name of a register bank, and *rport* or *wport* attributes selecting read or write ports. The dependencies specified here are extracted by the entry of a keyword *get_rport* or *get_wport* in the appropriate subelement of the `<Operations>` element.

The `<Opnames>` Element. In order to provide a complete design flow, starting from the architecture specification and finishing with the compiler generation, the results of compilation must be represented in binary code. This binary code is put as a stimuli entry data for WPPE architecture simulation. In order to handle that, the MAML description uses an instruction image binary coding. All operations and binary image coding for them are listed by the `<Opnames>` element. The operation name is specified by the attribute *name* and the operation image binary coding is set by the *code* attribute. The attribute *function* describes the functionality of the operation, which is given in C-code and it is directly put into the code of automatically generated simulator.

The `<Opnames>` section of the MAML also enables a description of sub-word parallel operations. Sub-word parallelism allows the parallel execution of equal instructions with low data word width (e.g., two additions with 16 bit data) on functional units with high data word width (e.g., a 32 bit adder). Fig. 7 shows a functional unit, also called SWP unit, with two sub-words in the input and output data words. The data words with a set of sub-words are called full length words (FLWs). Also the execution of complex instructions (e.g., multiply and add, $y = \sum_i a_i \cdot b_i$) with multiply data input (2 data pairs with 16 bit word width) and single data output (1 data with 32 bit word width) is possible. Consequently, the type of SWP (operation and number of sub-words) have to be given in the description for each FU. Mostly the sub-words, which are packed

together in full length words, have to be rearranged for the next calculation. Therefore, additional instructions are needed, which are called packing instructions. The packing instructions can be added to given FUs or to extra FUs, where they have to be described. As the sub-words are stored in the full length words and the rearranging will be done in the FUs, no additional parameters are needed for the registers description.

The <Operations> Element. This element has one attribute *exelength* and the following set of subelements: <Opname>, <Opdirection>, <Input>, <Execution>, <Output>. This element describes the resource usage for each operation, i.e. how many cycles it occupies the functional unit, the operands direction, and which resource in which cycle will be occupied. The operations with the same parameters are grouped in operation sets under the subelement <Operationset>. The subelement <Opdirection> specifies the direction of the operands which are given by the *direction* attribute. The available values for this attribute are in, out, or inout. The subelement <Input> describes the operation fetch phase. The attribute *cycle* specifies in which cycle the fetch process is currently standing, *source* describes which source operands are active, and the *name* attribute shows which resource is occupied. The value of *name* can be the direct name of the resource, the string nores (means *no resource*), or one of the keywords get_rport|get_bus. <Execution> describes the execution phase of the operations. The attribute *cycle* specifies what resources are assumed in each cycle (starting with 0) of execution, *source* describes which source operands are active, and the *name* attribute shows which resource is occupied. The value of *name* can be the direct name of the resource, the executional unit, the execution stage, or the string nores. The subelement <Output> describes the result storage in the registers. It is equivalent to the <Input> element. Only the keywords in this case are get_wport and get_bus.

5 Case-Study

We have modeled one processor element with a configurable interconnect-wrapper using MAML as proposed in this paper. Moreover, a generically parameterized synthesizable VHDL-model was developed for it. Such parameters as number of functional units, bitwidth/size of register file, bitwidth/size of instruction memory, the depth of input/output FIFOs, interconnect-wrapper configuration in form of an adjacency matrix which is given as a generic variable can be fed into the VHDL-model and after compilation an appropriate processor element can be synthesized onto an FPGA. To allow dynamic reconfiguration of the interconnect structure the adjacency matrix is analyzed and the number of drivers for each output signal is extracted. In the case of a single driver for the given output signal a hard-wired connection is generated. If there are multiple drivers a multiplexer with the appropriate number of inputs is generated for this output. These generated multiplexers can be controlled at run-time by a configuration bus or even by the processor itself. The modeled processor element has a register file with 2 input, 2 output and 28 general purpose 32-bit registers, instruction memory for the storage of sixteen 76-bit VLIW instructions, 3 functional units performing multiplication, addition and shift operations. The given processor element with fully occupied adjacency matrix as the configuration for the interconnect-wrapper has been synthesized for the *Xilinx Virtex II Pro* FPGA with 13696 slices. The resulting processor could be clocked with more than 105 MHz and occupies one fifth of the FPGA. But remind, that the interconnect-wrapper was synthesized as full cross-bar. I.e., restricting the interconnect-wrapper will result in a considerably more efficient area of the design. The processor element with the configurable interconnect-wrapper has been modeled using MAML. It contains roughly without considering comments 550 lines of XML-code. In order to validate a MAML-model a special parser was developed and integrated into our framework. The modeled regular processor architecture can be graphically represented in the scope of this framework.

6 Conclusions and Future Work

In this paper, we proposed an architecture description language for the systematic characterization, modeling, simulation and evaluation of massively parallel reconfigurable processor architectures that are designed for special purpose applications from the domain of embedded systems. Key features, semantic, and technical innovations of ADL MAML for regular processor architectures were presented. The usability of proposed modeling approach was shown in a case study, where one processor element with a configurable interconnect-wrapper was developed. As future work we define further integration of the regular processor architecture modeling approach into our *ArchitectureComposer* [5] framework, further extension of this framework with a VHDL-backend generator for parallel processor array architectures, and an efficient simulation environment and visualization (extension of RASIM simulator [12]). This will allow rapid-prototyping and validation of our mapping and compilation techniques we are currently developing in parallel for such array architectures.

References

1. D. Bradlee, R. Henry, and S. Eggers. The Marion System for Retargetable Instruction Scheduling. In *Proc. ACM SIG-PLAN91 Conf. on Programming Language Design and Implementation*, pages 229–240, Toronto, Canada, June 1991.
2. A. Fauth. Beyond Tool-Specific Machine Descriptions. In P. Marwedel and G. Goossens, editors, *Code Generation for Embedded Processors*, pages 138–152. Kluwer Academic Publishers, 1995.
3. A. Fauth, J. Van Praet, and M. Freericks. Describing Instruction Set Processors using nML. In *Proceedings on the European Design and Test Conference, Paris, France*, pages 503–507, March 1995.
4. D. Fischer, J. Teich, M. Thies, and R. Weper. Design Space Characterization for Architecture/Compiler Co-Exploration. In *ACM SIG Proceedings International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES 2001)*, pages 108–115, Atlanta, GA, U.S.A., November 2001.
5. D. Fischer, J. Teich, M. Thies, and R. Weper. BUILDABONG: A Framework for Architecture/Compiler Co-Exploration for ASIPs. *Journal for Circuits, Systems, and Computers, Special Issue: Application Specific Hardware Design*, pages 353–375, 2003.
6. G. Goossens, J. Van Praet, D. Lanneer, W. Geurts, and F. Thoen. Programmable Chips in Consumer Electronics and Telecommunications. In G. de Micheli and M. Sami, editors, *Hardware/Software Co-Design*, volume 310 of *NATO ASI Series E: Applied Sciences*, pages 135–164. Kluwer Academic Publishers, 1996.
7. G. Hadjiyiannis, P. Russo, and S. Devadas. A Methodology for Accurate Performance Evaluation in Architecture Exploration. In *Proc. 36th Design Automation Conference (DAC99)*, pages 927–932, New Orleans, LA, June 1999.
8. A. Halambi, P. Grun, A. Khare, V. Ganesh, N. Dutt, and A. Nicolau. EXPRESSION: A Language for Architecture Exploration through Compiler/Simulator Retargetability. In *Proceedings Design Automation and Test in Europe (DATE'1999)*, 1999.
9. F. Hannig, H. Dutta, A. Kupriyanov, J. Teich, R. Schaffer, S. Siegel, R. Merker, R. Keryell, B. Pottier, D. Chillet, D. Ménard, and O. Sentieys. Co-Design of Massively Parallel Embedded Processor Architectures. In *Proceedings of the first ReCoSoC workshop*, Montpellier, France, June 2005.
10. S. Hanono. *Aviv: A Retargetable Code Generator for Embedded Processors*. PhD thesis, Massachusetts Inst. of Tech., June 1999.
11. D. Kästner. *Retargetable Postpass Optimization by Integer Linear Programming*. PhD thesis, Saarland University, Germany, 2000.
12. A. Kupriyanov, F. Hannig, and J. Teich. Automatic and Optimized Generation of Compiled High-Speed RTL Simulators. In *Proceedings of Workshop on Compilers and Tools for Constrained Embedded Systems (CTCES 2004)*, Washington, DC, U.S.A., Sept. 2004.
13. R. Leupers and P. Marwedel. Retargetable Code Generation based on Structural Processor Descriptions. In *Proceedings on Design Automation for Embedded Systems*, volume 3, pages 1–36, March 1998.
14. P. Mishra and N. Dutt. Architecture Description Languages for Programmable Embedded Systems. In *IEE Proceedings on Computers and Digital Techniques*, Toronto, Canada, 2005.
15. S. Pees, A. Hoffmann, and H. Meyr. Retargeting of Compiled Simulators for Digital Signal Processors Using a Machine Description Language. In *Proceedings Design Automation and Test in Europe (DATE'2000)*, Paris, March 2000.
16. W. Qin and S. Malik. Architecture Description Languages for Retargetable Compilation. In *The compiler design handbook: optimizations machine code generation*. CRC Press, 2002.
17. A. Terechko, E. Pol, and J. Eijndhoven. PRMDL: A Machine Description Language for Clustered VLIW Architectures. In *Proceedings Design Automation and Test in Europe (DATE'2001)*, page 821, Munich, Germany, Mar. 2001.
18. H. Tomiyama, A. Halambi, P. Grun, N. Dutt, and A. Nicolau. Architecture Description Languages for System-on-Chip Design. In *Proc. Proc. APCHDL*, Fukuoka, Japan, 1999.
19. Trimaran. <http://www.trimaran.org>.