PD-XML: Extensible Markup Language for Processor Description

S. P. Seng[1], K. V. Palem[2], R. M. Rabbah[2], W.F. Wong[3], W. Luk[1], P.Y.K. Cheung[1]
[1] Imperial College of Science, Technology and Medicine, England, {sps,wl}@doc.ic.ac.uk,{p.cheung}@ic.ac.uk
[2] School of Electrical and Computer Engineering, Georgia Institute of Technology, USA, {palem, rabbah}@ece.gatech.edu
[3] Department of Computer Science, National University Singapore, Singapore, wongwf@comp.nus.edu.sg

ABSTRACT

This paper introduces PD-XML, a meta-language for describing instruction processors in general and with an emphasis on embedded processors, with the specific aim of enabling their rapid prototyping, evaluation and eventual design and implementation. The proposed methodology is based on the extensible markup language XML widely used structured information exchange and collaboration. PD-XML allows for both high-level and low-level architectural specifications required to support a toolchain for design space exploration. PD-XML consists of three intuitive entities, describing: (a) the storage components available in a design, (b) the instructions supported by an architecture, and (c) the resources afforded by the microarchitecture implementation. PD-XML is not specific to any one architecture, compiler or simulation environment and hence provides greater flexibility than related machine description methodologies. We demonstrate how PD-XML can be interfaced to to existing description methodologies and tool-flows. In particular, we show how PD-XML specifications can be translated into appropriate machine descriptions for the parametric HPL-PD VLIW processor and for the flexible instruction processor approach.

1. INTRODUCTION

During the past three decades, the microprocessor has proliferated many aspects of daily life with a scope and depth that was hard to imagine during its early development. For microprocessors, the periodic doubling in the number of transistors that can be fabricated on a chip often meant a hundred percent increase in performance every year and a half, at no additional cost [1]. The advance in silicon design and fabrication technology is leading to an evolution towards *custom computing solutions*: recent years have witnessed the emergence of a plethora of customized processors that are *embedded* within products such as media players, digital cameras, network routers, and set-top boxes. It is believed [1] that the trends of customization will continue to evolve in the lower market tiers; and gradually over a period of years, the trend will creep upward into higher tiers of the market, including personal computers.

Today, several commercially available strategies afford competing degrees of customization and scalability, ranging from variations of traditional embedded processors such as the ARM and MIPS, to wholly configurable processors such as the Tensilica Xtensa architecture and ARC Cores. Numerous other companies such as Philips, StarCore, ST Microelectronics and Texas Instruments have announced embedded VLIW cores for high performance embedded computing. Common to each of these commercial solutions is the need to overcome high *non-recurring engineering costs* (NRE) which are key hurdles in designing custom solutions in the short *time-to-market* characteristic of the industry. To this end, an embedded system is often assembled using a specialized core to carry out number crunching functions required by a given application, and leverages the latest technology developments available in commercial-off-the-shelf (COTS) embedded processors for less computationally demanding tasks [1, 15]. All the while, the design process is subject to various stringent constraints of size, power consumption, timing and performance. Hence, an intensive *design space exploration* is conducted to find a desirable solution for a specified set of constraints. The exploration or design process typically proceeds by first discovering a new candidate architecture and measuring its costs. Next, a toolchain consisting of a compiler and simulator for the candidate architecture is generated and used to assess the relative merits of the target hardware for a given application. The process is repeated until a satisfactory solution is discovered.

In this paper, we propose a *processor description extensible markup language* (PD-XML) as a means of describing instruction processors in general, with the specific aim of enabling their rapid prototyping, evaluation and eventual design and implementation. It can be used to support methods and tools for developing and optimizing instruction set architectures and their implementations, such as Trimaran [21], FIP [17] and architecture assembly [10]. In particular, PD-XML allows for extensible descriptions for both instruction set architectures and their microarchitecture implementations. The framework follows a specification format which may be easily simulated or even synthesized. This will permit the rapid exploration of the architectural space and will complement several well-founded methodologies that have emerged to ameliorate the engineering costs associated with exploring the design space of custom computing components [11, 16, 17]. The contributions of our work are as follows:

1. We propose PD-XML, a generic and extensible methodology for describing, simulating and implementing instruction set architectures. Information is organized into two entities: one about storage, and the other about the instruction set.
2. We extend PD-XML to cover descriptions of microar-

chitectures, by including information about the resources associated with a given microarchitecture.

3. We demonstrate how PD-XML can be used to support existing description methodologies and tool-flows. In particular, we show how a specification realized in PD-XML may be translated into a machine description for the parametric HPL-PD VLIW processor [7] and for the FIP approach [17].

The rest of the paper is organised as follows. Section 2 provides an overview of our approach. Section 3 discusses related work. Section 4 and 5 respectively cover ISA and microarchitecture descriptions in PD-XML. Section 6 describes TRIMARAN and FIP interfaces for PD-XML, and Section 7 presents a summary and future work. Appendix A contains an example of ISA and microarchitecture descriptions in PD-XML.

2. OVERVIEW OF APPROACH

Our description framework is based on an extensible markup language known as XML [2]. XML provides a set of guidelines and conventions for structuring and representing data – in our case instruction processor description. It is generic and easily extensible, and its merits as an architecture description language have been discussed in the past [13]. In addition, XML is widely popular with the World Wide Web Community as a means for structured information exchange and collaboration. Furthermore, by virtue of using XML as a building block for the proposed framework, it is possible to leverage various publicly available open-source tools designed to work with XML. Thus, PD-XML subsumes many of the existing processor description languages, and overcomes many of their disadvantages and drawbacks. In particular, (*i*) PD-XML is not tied to any one architecture, compiler or simulation environment, (*ii*) it is capable of representing both high-level as well as low-level specifications required to support a design space exploration toolchain, and (*iii*) it does not require expert-level know-how to read, understand and extend.

PD-XML consists of a collection of four main entities. The first captures information about components that store information, such as registers, register files, stacks, external memory, or block RAMs on FPGAs; hence it shall be called the *store* entity. The second entity describes the instruction set and shall be called the *inst* entity; it may include pseudo instructions which are decomposed by a compiler into several operations that are executed by the processor. The third is the *resource* entity and it contains information about physical resources available in a microarchitecture such as ALUs, cache control units as well as fetch and decode units. The fourth, *instance* declares an instance of a resource. An instruction set architecture description of a processor mainly involves the *store* and *inst* entities to be described in Section 4; the microarchitecture description requires all four entities, to be explained in Section 5.

A microarchitecture description comes in two flavours: high level and low level. A high-level microarchitecture description makes explicit the resources associated with each instruction, enabling simulation to take place. A low-level microarchitecture description contains detailed information

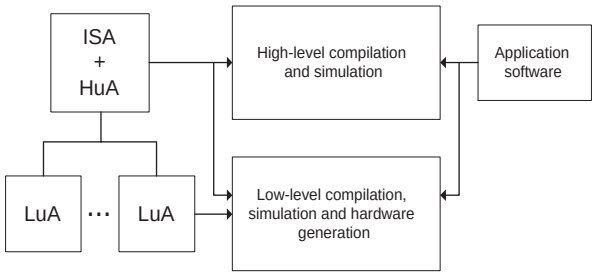

Figure 1: An (ISA, HuA) pair, where ISA denotes an instruction set architecture description and HuA denotes a high-level microarchitecture description, can be used for high-level design exploration involving application software. For detailed design development and implementation, an LuA (low-level microarchitecture) description is also required. The box labeled "high-level compilation and simulation" will be elaborated in Figure 2.

the datapath and control, allowing design optimisation, evaluation and implementation. Note that an ISA can be implemented by multiple high-level microarchitectures, and a high-level microarchitecture can be implemented by multiple low-level microarchitectures (Figure 1).

While PD-XML affords several advantages compared to previously published related work outlined in the following section, it is necessary to demonstrate its utility and flexibility to interface with other machine description methodologies. PD-XML is designed to provide a vehicle for the easy and rapid extension of existing frameworks where potentially substantial resources, and investments have already been committed. To this end, in Section 6 we shall demonstrate how PD-XML can be used to support two development approaches: first, an existing machine description language used in TRIMARAN [21], an infrastructure for research on explicitly parallel instruction computers (EPIC); second, design libraries for flexible instruction processors [17].

3. RELATED WORK

Various hardware description languages have been proposed in the past. Languages such as VHDL [19] and Verilog [12] allow engineers to describe their designs in a high-level language for which a compiler can synthesize the circuitry. Unfortunately, such languages tend to revolve around low-level details and it is generally well-known that even slight design variations require well-trained experts who are familiar with the architecture as well as the description language. To help alleviate the problems commonly associated with low-level description languages, a new generation of hardware description languages such as SystemC [20] and Handel-C [6] have emerged. However, these are intended as general-purpose hardware description languages and do not provide the necessary specifics to facilitate the easy and direct description of instruction set architectures or microarchitectures.

Many *architecture description languages* (ADLs) have been proposed to describe both hardware and software architectures. Examples of hardware architecture description languages include HMDES [4] and Pebble [8]. Such languages

are specific to a particular architecture domain or a compilation/simulation infrastructure. In contrast, PD-XML is a generic meta-language that is extensible and may be used to describe widely-varying architectures such as superscalar or VLIW processors, including their multi-clustered variants. The need for a meta-description of software architectures – the very same role that PD-XML is intended to play – has been a well-studied subject in software engineering [13].

Other related work involving instruction processor descriptions include ISDL [5], an instruction set description language, and MAML [3] and MESCAL [14], architecture description languages. ISDL supports the description of the ISA and microarchitecture of a processor, while MAML and MESCAL can be used for pipelined processor designs. PD-XML includes the capabilities available in these languages. In addition PD-XML captures the ISA and microarchitectures at multiple levels of abstraction, enabling an ISA to be mapped onto various microarchitectures.

4. DESCRIBING ISA

An ISA description contains information about the memory or storage capabilities and the related instructions. As such an ISA description is made up of *store* and *inst* entities. This allows an ISA designer to concentrate on the functionality of the processor. The bit-width of registers and instruction formats are included at this stage to allow for the generation of binary code.

To summarize, the ISA description should:

1. expose the capabilities of an instruction set to the programmer and compiler writer,
2. provide functional specification of the instruction set for implementation by microarchitectures.

In PD-XML, the instruction repertoire of an ISA can be obtained by inspecting the list of *inst* entities. The list of *store* entities provide information on the storage resources available to the ISA. Criterion (1) and (2) can be satisfied by the information contained in the two lists. For example, the *opcode* attribute and the *inst_format* tag provide information on how to translate assembly code into machine code. The *behav* tag specifies the functional capability of the instruction, and can be used to directly map onto a high level microarchitecture description.

Multiple issue architectures can be simulated at the ISA level by composing instructions in parallel and issuing them as a single sequential instruction. Technicalities such as MultiOp-P and MultiOp-S or EQ and LEQ machines can be dealt with in the *behav* section. The following is an example of a *store* entity:

```
<store type="RegFile" name="r">
 <doc>
  registers used for general computation
 </doc>
 <bitsize>32</bitsize>
 <depth>5</depth>
 <index>0..31</index>
</store>
```

The *store* entity has two fields, *type* and *name*. Here, we declare a register file called r. The *doc* field provides documentation for the entity. The *bitsize* field provides information on the number of bits (size) of the number that can be represented by a single register. The *depth* field provides information on the depth of the register file; here it says that the register file can be accessed by a pointer that is 5 bits wide, so there are 2^5 registers in this register file, r[0]..r[31]. The *index* field shows the indexing count for this register file. In other words the depth value tells the compiler the number of bits required to index into this register file, and the index value tells the compiler the actual index positions into the register file. This allows for a single physical register file to be logically split up into separate smaller register files.

At this level of abstraction no decisions are made as to the physical implementation of components. However, it is often convenient to segregate the use of registers within a register file. For example, there are 32 general purpose registers in the MIPS ISA description, which are split up into different categories of usage; temporary store, argument store, reserved for stack, frame pointer etc. This can be defined by creating a *store* entity with a type that refers to a previously defined entity.

```
<store type="RegFile.r" name="t">
 <doc>
  temporary registers, not preserved
  across calls. t[0]-t[10]
 </doc>
 <index>8..15,23..25</index>
</store>
```

Here we define an alias to the register file r: the t registers are referred to like a register file, but indices for t registers are translated into indices for r registers, using the convention outlined in the index section. Here t[0] is an alias of r[8]. An example of an *inst* entity:

```
<inst opcode="ADD">
 <doc>
  Definition for the add instruction
  add rd,rs,rt
  rd = rs + rt;
  where rd,rs,rt are from the r register file
 </doc>
 <in type="RegFile.r">in1,in2</in>
 <out type="RegFile.r">out1</out>
 <inst_format>
  000000::in1::in2::out1::xxxxx::100000
 </inst_format>
 <behav>r[out1] = r[in1] + r[in2];</behav>
</inst>
```

Here we define an instruction called ADD. It takes three operands that contain indices for the RegFile.r register file.

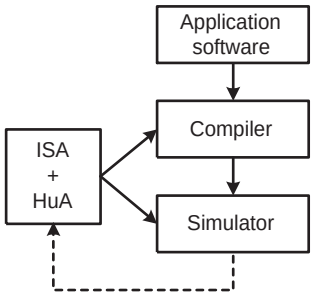

Figure 2: The ISA and HuA (high-level microarchitecture) descriptions can be used to produce high-level compilation and simulation tools. The dotted line indicates that the simulation result can be used to refine the ISA and HuA descriptions.

The names in1, in2 and out1 are labels that refer to indices into the *store* entity RegFile.r. The *inst_format* tag gives the binary instruction format for the ADD instruction. The first six bits correspond to the opcode while the last six bits corresponds to the function code, as defined by the MIPS instruction set. The '::' operator denotes concatenation. 'x' denotes bits whose value we don't care about.

For convenience, the labels in1, in2, and out1 above refer to 5-bit numbers (derived from the depth value of the RegFile.r object) and acceptable values for this 5-bit number are numbers between 0 and 31 (derived from the index tag of the RegFile.r object). The *behav* field contains the behavioural information that outlines the operation of this instruction in a high-level manner.

Figure 2 shows that the ISA description can be used to produce compilation and simulation tools to facilitate high-level design exploration. Note that multiple high-level microarchitectures can be used to implement each ISA; details of a high-level microarchitecture description will be explained in the next section.

5. DESCRIBING MICROARCHITECTURE

The microarchitecture section captures resource dependence in the processor, so that a cycle accurate description can be developed. This level contains descriptions of resources that are not directly accessible by a programmer, such as the fetch module or the program counter. The following describes the requirements of microarchitecture descriptions, followed by an explanation of high-level and low-level microarchitecture descriptions in PD-XML.

5.1 Requirements

The purpose of a microarchitecture description is to include implementation constraints to enable effective implementation and evaluation. From experience it should:

1. expose the hardware capabilities of an instruction processor,
2. expose the resource dependencies,
3. allow the compiler to be further optimized,

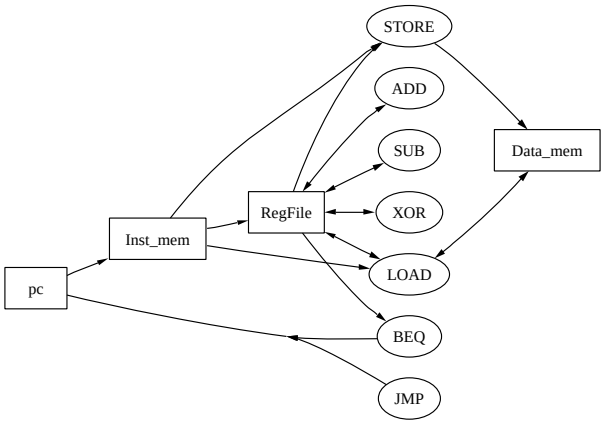

Figure 3: A graph representation of a high level microarchitecture description.

4. provide enough information for cycle-accurate simulation of the microarchitecture.

In PD-XML, the *uA* field can be written in two levels of abstraction: high-level microarchitecture and low-level microarchitecture. From the *uA* definition, we can deduce dependence information, which can be drawn as a DOT graph [9]. Each node in the graph maps to a physical implementation block. This exposes the hardware capabilities of the processor. The *in* and *out* tags capture dependence information. This is reflected in the edges of the graphs. Section 5.4 and 5.5 show how information for further optimizing a compiler can be captured. Simulation of a microarchitecture can be done in two levels. The high level microarchitecture can be simulated using information captured in the *behav* tags in the ISA and low level microarchitecture can be simulated with information from the *struct* tags.

5.2 High-level microarchitecture

The high level microarchitecture description closely resembles the ISA specification, where instructions are segregated. The *uA* description at this level is organized around the instructions in the processor and only data flow between the *store* and *inst* entities are shown. Figure 3 shows a graph representation of part of a MIPS processor. The *uA* field contains information about the modules and provides information such as data dependence, allowing pipelining and scheduling to take place. An example of the *uA* definition follows:

```
<store type="RegFile" name="r">
 <uA>
  <in>Inst_mem</in>
  <out>ADD;XOR;SUB;LOAD;STORE;BEQ</out>
 </uA>
</store>
```

This shows that RegFile takes information from Inst_mem and provides information for LOAD, ADD etc. As another example, consider the ADD module:

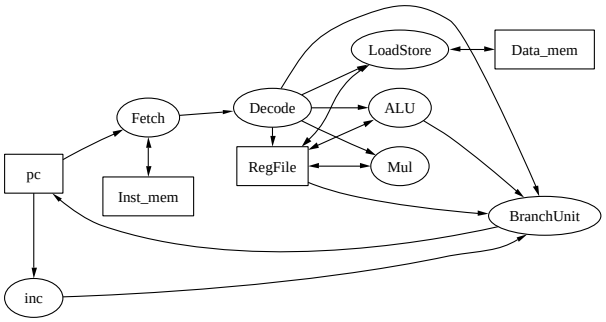

Figure 4: A graph representation of a processor with shared functional units.

```
<inst opcode="ADD">
 <uA>
  <in>RegFile.r</in>
  <out>RegFile.r</out>
 </uA>
</inst>
```

Behavioural information can be incorporated in the same way as in the low-level microarchitecture definition shown in the next section. Latency information can be incorporated manually or can be determined by simulation.

5.3 Low-level microarchitecture

While the high-level microarchitecture describes the resource dependences of the ISA, the low-level microarchitecture (also known as the *implementation*) captures how the ISA may be realized. This is divided into two parts:

1. *LL1* The first part describes the types of resources found in the datapath and the instances of these resources. Instances may also contain additional implementation dependent information. For example, in the case of a cache, different cache parameters and policies can be described.

2. *LL2* The second part describes resources that control the flow of instructions. Information regarding implementation details such as whether the processor is EPIC or superscalar, UAL, NUAL, MultiOp-P, MultiOp-S, EQ or LEQ are also encapsulated in this part of the machine description.

5.4 LL1 - data path

The *uA* field can also be written to conform to more conventional processor architectures, with shared computation units, such as the ALU and branch units. At this level of description, the microarchitecture description is resource oriented and contains control information. The *inst* tag is no longer used and the *resource* and *store* tags are used as type definitions. The *instance* tag is used to declare an instance of a resource or store entity. The following description of the microarchitecture can be seen as a refinement of the first example.

```
<resource type="arith" name="ALU">
```

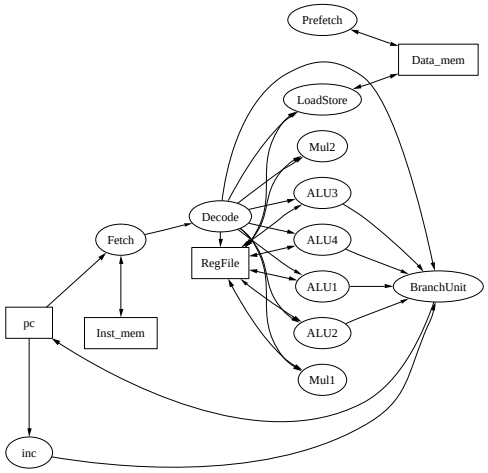

Figure 5: A graph representation of the MIPS ISA with multi-issue microarchitecture.

```
 <doc>The ALU unit for adds, sub, xor ... </doc>
 <uA>
  <in type="RegFile.r">in1,in2</in>
  <in type="Decode">aluop</in>
  <out type="RegFile.r">out1</out>
  <out type="BranchUnit">branch</out>
  <implements>
   ADD,XOR ...
  </implements>
  <struct>
    ...
  </struct>
 </uA>
</resource>
```

The *resource* entity at this level contains fields that relates the information in the *inst* field to this resource. The *in* and *out* fields provide connection information. The *implements* field denotes what instructions are supported. The *struct* field provides information on how to build each of the modules. This can be captured in Pebble [8], VHDL or other similar hardware description languages.

Figure 4 shows a single issue implementation of the MIPS ISA, while Figure 5 shows a multi-issue implementation. The high level microarchitecture description provides an easy but inefficient way to implement the ISA. The ISA can then be mapped into different low level microarchitecture implementations depicted in Figure 4 and 5.

This format allows microarchitectures of different levels of abstraction and functionality to be coupled with an ISA description. Conversely several ISA definitions can be mapped onto a microarchitecture description.

```
<instance type="arith.ALU" name="ALU0">
 <doc>...</doc>
</instance>
<instance type="arith.ALU" name="ALU1">
```

```
 <doc>...</doc>
</instance>
<instance type="store.cache" name="ICACHE">
 <doc>...</doc>
 <size>8Kbyte</size>
 <line_size>...</line_size>
 <associativity>...</associativity>
 <replacement_policy>...</replacement_policy>
</instance>
```

Instances contain information about a particular instance of a resource. The example above shows the instantiation of two ALU units and a cache.

5.5 LL2 - control path

The control path of a microarchitecture can be described in a similar manner to the data path. The distinction is arbitrary and both control and data paths can be described by the same tags. The example below shows the declaration of two control resources: an instruction issue resource and a reservation station resource. Information required by the compiler is provided within the *uA* tags, such as the discipline and instruction window size. Control resources are instanced using the *instance* tag.

```
<resource type="control" name="inst_issue">
 <doc> Instruction issue controller </doc>
 <uA>
   <discipline>In_order</discipline>
   <instruction_window>10</instruction_window>
   <struct>
   ...
   </struct>
 </uA>
</resource>

<resource type="control" name="res_stat">
 <doc> Reservation stations </doc>
 <uA>
  <depth>5</depth>
  <discipline>In_order</discipline>
  <struct>
  ...
  </struct>
 </uA>
</resource>

<instance type="control.res_stat" name="ALU0_stat">
 <doc> The instance of the reservation station
       resource for ALU0
 </doc>
 <uA>
  <in type="control.inst_issue"></in>
  <out type="arith.ALU0"></out>
 </uA>
 ...
</instance>
```

6. INTERFACING PD-XML

This section covers two design and implementation flows to which PD-XML can be interfaced.

6.1 HMDES interface

In this section, we demonstrate how PD-XML may interface with and enhance HMDES [4], a powerful and complex machine description language used in the Trimaran research compiler and simulation environment. The architecture specifications in HMDES consist of six entities:

1. *format*, specifying the operands allowed by each type of operation,
2. *resource usage*, specifying how operations use the processor's resources when they execute,
3. *latency*, specifying how to calculate data-flow dependences between operations,
4. *operation*, describing the operations supported by the architecture and specifying their format with respect to operands, resources usage, and latency,
5. *register*, providing information necessary for register allocation performed by Trimaran,
6. *compiler*, a generic entity intended to communicate other information required by the compiler.

While HMDES provides a facility to model various specifications, it does often prove difficult to use. For example, to augment the ISA of a HMDES-modeled processor requires modifications at several different levels – namely, the *format*, the *resource usage*, the *latency* and the *operation* levels; and additional *compiler* flags may be necessary as well. Similarly, altering the microarchitecture often requires modifications to various entities. In contrast, PD-XML consists of only three components, each a self-contained and easily modified entity. Furthermore, most of the specifications and definitions required by the HMDES infrastructure can be directly inferred from the XML descriptions. In what follows, we highlight a few such examples.

Consider the declaration of register file `r` shown earlier in Section 4 and the equivalent HMDES code below. The `Register` section creates a total of 32 registers – r0 through r31 – which are subsequently assigned to the register file `r` in section `Register_File`. Specified in the former is the width of each register. Information as to whether a register is preserved across function calls can be described using similar annotations. Earlier we have shown how PD-XML supports the segregation of registers (for instance register file `t` in Section 4). This can be trivially translated to the equivalent HMDES specification illustrated below.

```
SECTION Register
{
  $for (N in $0..$31) {"r${N}"(width ($32));}
}

SECTION Register\_File
{
  r (registers($for (N in $0..$31) {"r${N}"}))
  t (registers($for (N in $8..$15) ...)
}
```

While the HMDES specification of register files is somewhat straightforward, the definitions of the ISA and the resource usage patterns (i.e. microarchitecture-level descriptions) are slightly more complex and are illustrated below.

```
SECTION Operand_Type
{
  FT_i(regfile(r));
  FT_l(regfile(L));
  FT_il(compatible_with(FT_i FT_l));
}

SECTION Operation_Format
{
  OF_ADD(src(FT_il FT_il) dest(FT_i));
}

SECTION Resource
{
  R_ALU0;
}

SECTION Operation
{
  ADD(format(OF_ADD) uses(R_ALU0) ...);
}
```

As shown above, in order to define an ADD operation, it is necessary to first define its format (`OF_ADD`). In this case, it requires two source operands, where each may be a register (`r`) or a literal (`L`), modeled as a pseudo-register file. In HMDES, it is often not possible to avoid some microarchitectural specifications when defining an operation. To this end, a `R_ALU0` resource is instantiated and declared to be capable of carrying out the operation of interest. Subsequently the actual operation is defined[1]. Referring back to the PD-XML specification for the ADD operation: it is readily apparent how the operation format may be inferred for the HMDES specification. However, whereas PD-XML affords an ISA-only specification – hence is not tied to a particular architecture implementation – it is now necessary to include some low-level details required to complete the HMDES specification. To this end, we extend the PD-XML description to include a pseudo-resource field which may be subsequently realized at the microarchitecture level.

The examples above are intended to highlight how PD-XML may easily and readily adapt to an existing machine description facility such as HMDES. Thus, we believe that it can be used to augment the Trimaran infrastructure which was conceived to explore the evolution of VLIW architectures. In particular, the machine-driven optimizing compiler and performance monitoring tools available in Trimaran may be easily retargeted to investigate the merits of architectural innovations via rapid prototyping using PD-XML. This is of increasing significance as VLIW architectures continue to proliferate at various tiers of the processor industry.

[1]Several details are omitted for clarity. We refer the interested reader to the HMDES technical report [4].

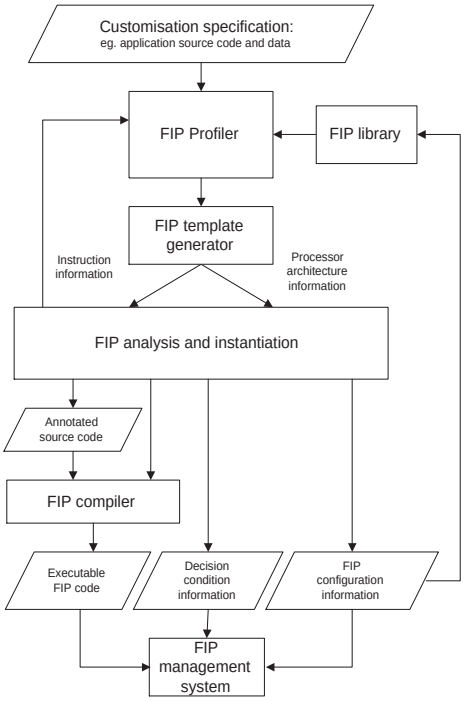

Figure 6: The design flow for the Flexible Instruction Processor (FIP) approach.

6.2 FIP Interface

The Flexible Instruction Processor (FIP) approach provides a mechanism for the systematic customization of instruction processors, targeting mainly reconfigurable devices [17]. This approach helps designers to tune hardware implementations to the characteristics of a system both at design time and at run time [18]. PD-XML provides a concise way to express both the ISA and the microarchitecture information required by the FIP approach.

Figure 6 shows a simplified FIP design flow. The FIP profiler takes in a custom specification, usually in C or in Java, as well as ISA information in PD-XML format from the FIP library. The ISA description is customized to the application source code provided in the custom specification. This information is then passed to the FIP template generator which creates a high-level microarchitecture in PD-XML that corresponds to the ISA description.

Next, the FIP template is put through an analysis phase where operations are optimized and custom instructions are introduced if appropriate [18]. After instancing, a FIP configuration is produced to program a reconfigurable device. A PD-XML description of the low-level microarchitecture is also passed to the FIP compiler, so that executable code can be produced for the FIP implementation.

7. SUMMARY

This paper introduces PD-XML, a meta-language for describing processors in general with an emphasis on embedded processors. PD-XML enables rapid high-level as well as low-level architectural specifications required to support a toolchain for design space exploration. The proposed ap-

proach should become increasingly important in the context of general-purpose systems design, and especially in embedded systems where the needs for application-specific architectures are increasing prevalent, and time-to-market is a primary concern. In addition, we have demonstrated how PD-XML can interface to and extend the flexibility of existing machine description methodologies such as HMDES used to model the parametric HPL-PD processor central to the TRIMARAN infrastructure. Current and future work includes the completion of the retargeting of our tools to support PD-XML, and the extension of PD-XML to support adaptive implementations that can be reconfigured at run time [18].

ACKNOWLEDGEMENTS

The authors acknowledge Mongkol Ekpanyapong, Georgia Institute of Technology, Anjani Kumar Tripathi, Banares Hindu University, and David Thomas, Imperial College, for their contributions to this paper. This work is supported in part by DARPA contract F30602-00-2-0564, A*STAR Project No. 012-106-0046, UK EPSRC projects GR/N 66599 and GR/R 55931, Celoxica Limited, Hewlett Packard Laboratories, and Yamacraw.

REFERENCES

[1] M. Bass and C. Christensen. The future of the microprocessor business. *IEEE Spectrum*, Apr. 2002.

[2] R. Cover. The XML cover pages. xml.coverpages.org.

[3] D. Fisher et al. Design space characterisation for architecture compiler co-exploration. In *Proc. CASES*, ACM, 2001.

[4] J. Gyllenhaal, W. Hwu, and B. R. Rau. HMDES version 2.0 specification. Technical Report IMPACT-96-3, University of Illinois, Urbana, 1996.

[5] G. Hadjiyiannis, S. Hanono, and S. Devadas. Isdl: An instruction set description language for retargetability. In *Proc. 34th Design Automation Conference*, 1997.

[6] The Handel-C programming language. www.embedded-solutions.ltd.uk/products/design_suite.

[7] V. Kathail, M. Schlansker, and B. R. Rau. HPL-PD architecture specification: Version 1.1. Technical Report HPL-9380 (R.1), Hewlett Packard Laboratories, Feb. 2000.

[8] W. Luk and S. McKeever. Pebble: a language for parametrised and reconfigurable hardware design. In *Proc. FPL*, LNCS 1482, Springer, 1998.

[9] Open source graph drawing software. www.research.att.com/sw/tools/graphviz.

[10] K. Palem. C-based architecture assembly supports custom design. In *EE Times*, Feb. 2002.

[11] K. Palem. Rapid design of custom embedded systems via architecture assembly. Technical report, Proceler Inc., Feb. 2002.

[12] S. Palnitkar. *Verilog HDL*. Prentice Hall, 1996.

[13] S. Pruitt, D. Stuart, W. Sull, , and T. Cook. The merit of XML as an architecture description language meta-language. xml.coverpages.org/ADL-meritofxml.html.

[14] W. Qin. Mescal architecture description. http://www.ee.princeton.edu/MESCAL/ar-de.html.

[15] B. R. Rau and M. Schlansker. Embedded computer architecture and automation. *IEEE Computer*, Apr. 2001.

[16] R. Schreiber, S. Aditya, S. Mahlke, V. Kathail, B. R. Rau, D. Cronquist, and M. Sivaraman. PICO-NPA: High-level synthesis of nonprogrammable hardware accelerators. *Journal of VLSI Signal Processing*, 2001.

[17] S. Seng, W. Luk, and P. Cheung. Flexible Instruction Processors. In *Proc. CASES*, ACM, 2000.

[18] S. Seng, W. Luk, and P. Cheung. Runtime Adaptive Flexible Instruction Processors. In *Proc. FPL*, LNCS 2438, Springer, 2002.

[19] S. Sjoholm and L. Lindh. *VHDL for Designers*. Prentice Hall, 1997.

[20] The open SystemC initiative. www.systemc.org.

[21] TRIMARAN: An infrastructure for research in instruction level parallelism. www.trimaran.org.

APPENDIX A

An example of a PD-XML ISA and microarchitecture description. The *implementation* tag relates an ISA with a microarchitecture. Comments are enclosed by *doc* tags.

```
<PD-XML>
 <implementation ISA="MIPS"
            uA="single-issue-superscalar"/>

 <instruction_set_architecture name="MIPS">
  <store type ="Register" name="pc">
   <ISA>
    <bitsize>32</bitsize>
   </ISA>
  </store>
  ... other store declarations

  <inst opcode="BEQ">
   <doc>
    Definition for branch if equal
    beq rs,rt,offset
   </doc>
   <ISA>
    <in type="RegFile.r">in1,in2</in>
    <in type="Inst_mem" bitsize="16">offset</in>
    <out type="Register.pc">pc</out>
    <inst_format>
     000100::in1::in2::offset
    </inst_format>
    <behav>
     ((r[in1]==r[in2])?(pc+=offset):(pc))
    </behav>
   </ISA>
  </inst>
  ... other inst declarations
 </instruction_set_architecture>

 <microarchitecture
             name="single-issue-superscalar"/>
  <store type ="Register" name="pc">
   <doc> Declaration of the PC register </doc>
   <uA>
    <struct>
     unsigned 32 pc;
    </struct>
   </uA>
  </store>
  ... other store declarations

  <resource type="branch" name="BranchUnit">
   <doc> Declaration of a branch unit </doc>
   <uA>
    <in>in1,in2</in>
```

```
    <out>out1</out>
    <implements>BEQ,BNE...</implements>
    <struct> ... </struct>
   </uA>
  </resource>
  ... other resource declarations

  <instance type="branch.BranchUnit" name="BR1">
   <doc> Instancing of a branch unit </doc>
   <uA>
    <in type="RegFile.r">in1,in2</in>
    <in type="Inst_mem" bitsize="16">offset</in>
    <out type="Register.pc">pc</out>
   </uA>
  </instance>
  ... other instances
 </microarchitecture>

</PD-XML>
```