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. 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 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 (FIP) approach targeting reconfigurable implementations.*

1 Introduction

In this paper, we propose a *processor description extensible markup language* (PD-XML) [4] as a means for 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 [6] and FIP [5]. 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 [3, 5].

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 intuitively organized into two entities: one for storage, and the other for the instruction set. (2) We extend PD-XML to cover descriptions of microarchitectures, 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 and for the FIP approach [5].

2 Overview of approach

Our description framework is based on an extensible markup language known as XML. XML is generic, easily extensible and is widely popular with the World Wide Web Community as a means for structured information exchange and collaboration. 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 three main entities. The first captures information about components that store information, such as register files, stacks, external memory, or block RAMs on FPGAs; hence is called the `store` entity. The second entity describes the instruction set and is 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, fetch and decode units. An instruction set architecture (ISA) description of a processor mainly involves the `store` and `inst` entities (Section 3); the microarchitecture description requires all three entities (Section 4).

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 regarding 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).

3 Describing ISA

An ISA description should: (1) expose the capabilities of an instruction set to the programmer and compiler writer, as

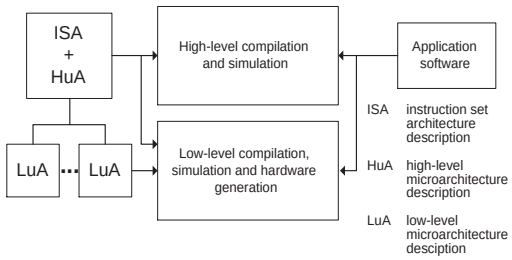

Figure 1. An (ISA, HuA) pair can be used for high-level design exploration involving application software. For detailed design development and implementation, an LuA description is also required. The box labeled "high-level compilation and simulation" will be elaborated in Figure 2.

well as (2) provide a functional specification of the instruction set for implementation by microarchitectures. As such an ISA description is made up of `store` and `inst` entities. The bit-width of registers and instruction formats are included at this stage to allow for the generation of binary code.

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 instructions to high level microarchitecture descriptions. The following is an example of a `store` entity:

```
<store type="RegFile" name="r">
 <doc>
  registers used for general computation
 </doc>
 <ISA>
  <bitsize>32</bitsize>
  <depth>5</depth>
  <index>0..31</index>
 </ISA>
</store>
<store type="RegFile.r" name="t">
 <ISA>
  <index>8..15,23..25</index>
 </ISA>
</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 `ISA` tag is used to group together relevant information. 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 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 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. The second `store` entity defines 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 – which defines an instruction called ADD – is shown below:

```
<inst opcode="ADD">
 <doc>
  add rd,rs,rt
  rd = rs + rt;
  where rd,rs,rt are from the r register file
 </doc>
  <ISA>
  <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>
 </ISA>
</inst>
```

It takes three operands that contain indices for the `RegFile.r` register file. 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 and '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.

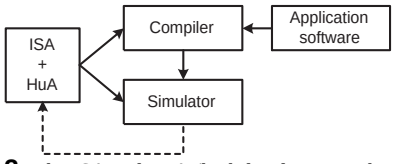

Figure 2. The ISA and HuA (high-level microarchitecture) descriptions can be used to produce high-level compilation and simulation tools. The dotted line indicates that the simulation result can be used to refine the ISA and HuA descriptions.

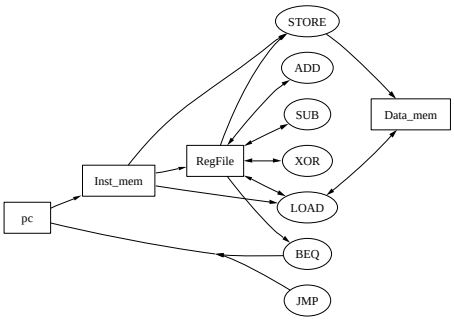

Figure 3. A graph representation of a high level microarchitecture description.

4 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 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 further optimisation of a compiler, (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 [2]. 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 4.2 shows 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.

4.1 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:

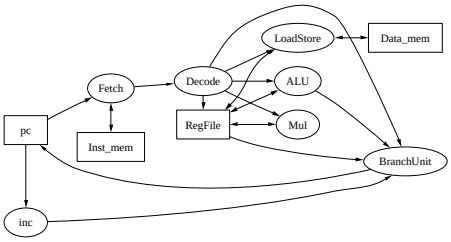

Figure 4. A graph representation of a processor with shared functional units.

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

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.

4.2 Low-level microarchitecture

While the high-level microarchitecture describes the resource dependences of the ISA, the low-level microarchitecture captures how the ISA may be realized. This is divided into two parts, LL1 and LL2. Two new entity tags are also introduced, `resource` which contains implementation details for resources such as the ALU, and `instance` which creates an instance of a resource.

LL1 – data path: LL1 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.

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.

LL2 – control path: LL2 describes resources that control the flow of instructions. Information regarding imple-

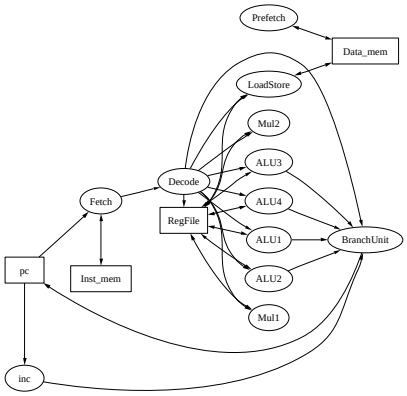

Figure 5. A graph representation of the MIPS ISA with multi-issue microarchitecture.

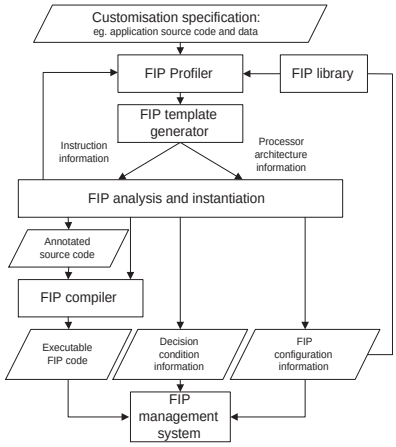

Figure 6. The design flow for the Flexible Instruction Processor (FIP) approach.

mentation details such as whether the processor is EPIC or superscalar, for example, are also encapsulated in this part of the machine description.

5 Interfacing PD-XML

HMDES Interface: PD-XML may be interfaced with and enhance HMDES [1], a powerful and complex machine description language used in the TRIMARAN research compiler and simulation environment. Altering the ISA and microarchitecture of a HMDES-modeled processor requires modifications at different levels of the description. In contrast, PD-XML consists of only three components, each a self-contained and easily modified entity. Most of the specifications and definitions required by the HMDES infrastructure can be directly inferred from PD-XML descriptions.

We believe that PD-XML 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.

FIP Interface: The Flexible Instruction Processor (FIP) approach provides a mechanism for the systematic customization of instruction processors, targeting mainly reconfigurable devices. This approach helps designers tune hardware implementations to the characteristics of a system both at design time and at run time [5]. 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 [5]. 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.

6 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 and low-level architectural specifications required to support a toolchain for design space exploration. 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 [5].

ACKNOWLEDGEMENTS

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

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

[3] K. Palem. Rapid design of custom embedded systems via architecture assembly. Technical report, Proceler Inc., Feb. 2002, http://www.proceler.com/pdfs/EETimes.pdf.

[4] S. Seng, K. Palem, R. Rabbah, W. F. Wong, W. Luk, P. Y. K. Cheung. PD-XML: Extensible markup language for processor description. Technical report, 2002/16, Department of Computing, Imperial College, UK, 2002.

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

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