APDL: A Processor Description Language For Design Space Exploration of Embedded Processors

N. Honarmand[1], H. Sohofi[1], M. Abbaspour[2] and Z. Navabi[1]
[1] CAD Laboratory, School of Electrical and Computer Engineering, University of Tehran, Tehran, IRAN
[2] School of Electrical and Computer Engineering, Shahid Beheshti University, Tehran, IRAN

{nima, h_sohofi}@cad.ece.ut.ac.ir, maghsoud@ipm.ir, navabi@ece.neu.edu

Abstract—This paper presents Anahita Processor Description Language (APDL) for generation of retargetable processor design tool sets. The emphasis is on the applicability of the generated tools in the design space exploration (DSE) phase of designing a new embedded processor. APDL descriptions can be used for generating cycle-accurate instruction set simulators, assembler/disassembler tools, production quality compilers and architecture verification tools. The paper first investigates the features required for a language to be useful for DSE and then presents APDL constructs along with code samples.

Index Terms— architecture description languages, application specific architectures, retargetable compilation, retargetable instruction set simulator.

1. Introduction

The proliferation of the embedded electronic systems in different branches of technology has fueled rapid growth of industry sectors like telecommunication and automotive industries, medical instruments, military equipment, etc. This effect has created a vast demand for electronic systems and resulted in a competitive and fast-growing market for embedded systems. In this setting, the ability to deliver new products within a short period of time becomes crucial for remaining in business.

Also, shrinking feature sizes in IC fabrication technology, which for several decades have shown exponential growth in transistor count and performance (Moore's law [1]), has made it possible to put more and more functionality on a single silicon die. This, in turn, has caused an increase of complexity in modern IC designs, and more bugs and more design re-spins before delivering a working product. At the same time, the increasing mask cost, due to newer fabrication technologies, discourages multiple design spins and calls for the less-error-prone design techniques. All these challenges encourage the design-reuse in electronic system design.

Because of the looser coupling between different system components, design reuse is much easier in software-based systems than hardware systems. One can use the same hardware, i.e., the processor, for different designs by just reprogramming it. Another equally important problem is to incorporate new features in the old designs and to introduce improved versions to the market rapidly. This necessitates frequent design revisions that are easier to do in software.

To address the performance requirements of the software-based design methodology, embedded system designers have turned to use techniques like Instruction Set Extensions (ISEs), Digital Signal Processors (DSPs) and Application Specific Instruction Processors (ASIPs) [2]. In the former two, the processor designer chooses a general purpose architecture and modifies it, i.e., adds some application specific features to it, to improve its applicability to the specific problem. In the latter, the designer tailors, almost from scratch, a processor's architecture to a class of closely-related applications.

Before finalizing the processor architecture, the designer needs to measure the figures of merit for different alternative architectures. This process is usually referred to as Design Space Exploration (DSE). To do this rapidly and easily, the designer needs several design automation tools, like instruction set simulators (ISS), high level language (e.g., C) compilers, hardware generators and architecture verification tools.

Naturally, the designer expects to be able to use a single description to feed all these different tools, because the requirement of providing several models of the design arises the issue of consistency between different descriptions and thus is not desirable. Conventionally, languages designed for this goal are called Architecture Description Languages (ADL). In computer science, the term ADL has been used for languages describing architecture of both software and hardware systems. In this paper, we use this term to refer to languages describing the structural and behavioral aspects of instruction set processors.

This work presents Anahita Processor Description Language (APDL) which is the processor description formalism behind the Anahita Processor Design Suite, currently under development in our research team. APDL has been designed as a small yet powerful language to aid the DSE during the design of new or modified embedded processors, ASIPs and DSPs.

The rest of the paper is organized as follows: Section 2 provides the goals driving the current structure of our proposed language. Section 3 surveys some of the previous works. Section 4 provides an introduction to the major features of the APDL and Section 5 concludes the paper.

2. Goals and Requirements

Two different aspects of ASIP designs, namely irregular hardware structures and the need for aggressive code optimization, have greatly impacted the current structure of APDL. Irregular data paths, multiple instruction pipelines and split register files are among the features commonly found in state-of-the-art DSPs and ASIPs. As a result, designing the control logic becomes one of the most difficult and error-prone tasks during DSP or ASIP design. Also, such irregular structures impose many constraints on possible combinations of operations in the instruction word of the processor. It is difficult to consider all these combinations in a hand coded assembler or code generator of a compiler. Since nearly all of these constraints arise from resource conflicts between processor operations, the information required to detect such constraints could easily be extracted from the processor description, provided that enough information regarding the resource requirements of each operation is given in the description.

On the other hand, power-consumption considerations in modern embedded applications discourage the embedded processor designers from using many dynamic ILP extraction and hazard/resource conflict resolution features which are common in today's high-end desktop or server processors [3]. To compensate for the performance loss due to lack of such features, embedded system designers should rely on aggressive code optimization by compiler or code scheduler. To do this, the code scheduler needs detailed information about the behavior of processor operations like when an operation is going to read (write) a value from (to) a register or memory. This way, the compiler will be able to effectively utilize the delay slots between producer and consumer operations and increase the performance.

3. Previous Works vs. APDL

3.1. Previous Works

Conventionally, ADLs have been classified into three major categories:

- Structural ADLs which focus on the hardware components of the processor and their interconnection (MIMOLA [4])
- Behavioral ADLs which mainly focus on the functional semantics of the processor instruction set (nML [6] and ISDL [9])
- Mixed ADLs which consider both structure and behavior and provide constructs to express their interactions (LISA [11], EXPRESSION [12], MADL [13], ArchC [14] and RADL [15])

MIMOLA [4] focuses on describing the structure of the target processor with HDL-like constructs. In [5], authors reported techniques to extract the instruction set (IS) of the processor by processing this structural description. The difficulty of extracting IS information from complicated control unit and data path descriptions makes this an unsuitable approach for retargetable code generation. Also, the MIMOLA approach will not suit the requirements of the DSE phase. During DSE, the designer is unlikely to be willing to deal with detailed structural implementation of the processor. Instead, he or she is interested to begin with a mixture of coarse-grained IS and structural decisions and evaluate their impact on performance parameters of interest.

nML [6] is an elegant formalization for describing the IS of a processor that has been used by the Belgium-based Target company [7] as the formalization behind its CHESS/CHECKERS processor design tool suite. nML provides constructs for hierarchical and concise operation descriptions. nML ignores the temporal resource requirements of the operations and thus is not quite suited for the DSE phase of ASIP design. Being a behavioral ADL, nML ignores detailed temporal resource requirements of the processor operations. Also, in nML, designer should explicitly enumerate all the operation combinations that form valid instructions. This is not a feasible task for large ASIP designs.

ISDL [9], targeted mainly towards VLIW processors, follows the same path as nML although it provides, through description of constraints, the ability of invalidating some combinations of operations in the instruction word. The drawback of this method is that the designer should manually extract and code the invalid combinations in the description, which is a tedious and error-prone task for complex irregular architectures.

LISA [11], EXPRESSION [12] and MADL [13] are examples of mixed-paradigm ADLs. In LISA, the designer should provide a detailed and explicit description of behavior and interaction of operations in different stages of the processor pipeline. Though a good feature for generation of cycle accurate instruction set simulators, it is a drawback for DSE. During DSE the designer should not be engaged in error-prone and time-consuming task of modeling the control unit which is by far the most error-prone and time-consuming task in high level processor design.

EXPRESSION [12], on the other hand, provides features which are more suitable for DSE. Especially, through the description of pipeline stages, it provides the notation of operation-to-resource mapping. One major feature of EXPRESSION which is not found in other ADLs is the ability to describe the memory subsystem in the same processor description. Despite these, there are several major drawbacks in EXPRESSION. First, it lacks the hierarchical operation description style that makes its descriptions lengthy. Second, the timing model of EXPRESSION is bound to the concept of pipeline, and temporal behavior and resource requirements of the operations are indirectly described through instruction-to-pipeline and pipeline-to-resource mappings. Third, it describes the semantics of the processor operations by providing a mapping between the operations of the target machine and those of a generic machine. This makes the language tool-dependent and results in lengthy descriptions.

MADL [13] uses a state-machine based formalism to represent the progress of operations in the processor. To model the interaction of operations with hardware components, it introduces the concept of token managers which grant operations the permission to use hardware components. In MADL, the behavior of token managers can be described in an arbitrary procedural code that makes it difficult to automatically extract control information required by tools like compilers.

ArchC [14] is more suited for generation of retargetable instruction set simulators and assembler/disassembler tools. The lack of formal semantics and using arbitrary C code for behavior description is one of the major drawbacks of ArchC that makes it inappropriate for code generation and DSE. RADL [15], a dialect of LISA, is intended to be used for modeling complex pipeline behavior and bears the same drawbacks as LISA.

3.2. Comparison with APDL

To fulfill the requirements depicted in Section 2, we devised a new abstraction level for describing the temporal behavior of the processor operations and their interaction with hardware resources. This description, which we refer to as Timed Register Transfer Level (T-RTL), considers the behavior of operations as a timed set of read/write/compute events. Each of these events starts at a specific time, spans one or more clock cycles and has some associated resource requirements. APDL provides notations to conveniently specify these features for each event. Through T-RTL, APDL provides a unified and concise syntax to simultaneously represent different aspects of operation behaviors and avoid redundancies or limitations which would occur if loosely related semantic entities were to be used for this purpose.

T-RTL helps APDL to be *analyzable*. By analyzability, we mean that different tools, from compilers to ISS generators, can readily extract all the provided information. This is not the case with many other mixed-paradigm languages. For example, in LISA [11] and RADL [15], the operation behavior in different pipeline stages cannot be generally used to extract control information required by an optimizing compiler.

Also, the T-RTL representation can be regarded as a generalized form of the pipeline-oriented description style of languages like LISA [11] or EXPRESSION [12]. The implementation style of the processor, whether it is of a pipelined or multi-cycle or single cycle from, can be extracted from the T-RTL operation descriptions. And, if the processor has, for example, a pipelined structure, pipeline control signals like stall and squash can be automatically extracted from the operation descriptions. Section 4.4 demonstrates the usage of T-RTL in APDL descriptions.

```
type reg_read_range is 0 to 31;
type reg_write_range is 1 to 30;
type reg_write_range_link is 1 to 31;
type int16 is int<16>;
type uint16 is unsigned int<16>;
type int32 is int<32>;
type uint32 is unsigned int<32>;
type int26 is int<26>;
```

Figure 1. Example type declaratoins in APDL

4. The APDL Language

This section describes the elements of an APDL description. To avoid misinterpretation, syntactical constructs of APDL are represented in italic font whenever necessary. An APDL description consists of eight classes of entities: *data type declarations*, *resources*, *storages*, *attributes*, *expressions*, *statements*, *operations* and *instructions*. *Operations* are the backbone of the descriptions in APDL. Most of the important design data are provided through operation descriptions. Each operation can have attributes describing different aspects of the operation including its behavior, binary image and assembly syntax. *Resources* are used to express structural properties of the design and, through their appearance inside operation descriptions, convey enough information for detecting possible resource conflicts between different operations. *Storages*, a special class of resources, represent non-volatile storage elements like registers, register files or memories. *Attributes* provide an elegant and uniform syntactical representation to describe different aspects of operations, resources and storages. What follows discusses the details of these and other APDL entities.

4.1. Data Type Declarations

APDL is a strongly typed language. Every data type in the design should be one of the followings:

- A signed/unsigned integer with a fixed, arbitrary bit-width, e.g., `int<48>`, which is a signed 48-bit integer data type;
- An integer range, e.g., `1 to 31`;
- A floating point data type with fixed, arbitrary bit widths for the mantissa and the exponent, e.g., `float<51,12>` which is equivalent to `double` data type of C++
- An enumeration type definition, e.g., `{false, true}` for a boolean enumeration

Figure 1 shows some type declarations from a DLX [3] processor model.

4.2. Resource Declarations

Resources are used to express resource requirements of the processor operations. In addition, the designer can, through resource attributes, provide more information regarding different aspects of each resource, if the processing backend understands these attributes.

```
    resource ALU;
    resource FP[7];
    resource Mult[4];

    storage reg_file[31][32];
    storage LO[32];
    storage HI[32];
```

Figure 2. Example resource and storage declarations in APDL

In APDL, resources can be multi-dimensional arrays. An obvious application of a one dimensional resource array could be to describe a pipelined functional unit. Another application could be to describe a set of identical functional units that can be interchangeably used by an operation. Figure 2 shows the declaration of one scalar and two array resources. The first one is intended to represent the integer ALU of a processor. The second one can represent a 7-stage pipelined floating point unit and the third one might represent a set of 4 identical multipliers in the processor.

4.3. Storage Declarations

Storages are a special class of resources in APDL. They are intended to represent non-volatile storage elements of the processor like registers, register files and memories. Their semantics are the same as those of the resources. In addition, they can appear as operands in expressions. Like resources, storages can be declared as multi-dimensional arrays. Unlike many other languages, in APDL no data type is associated with storages. They represent raw spaces in which one can save the bit pattern of any expression, whether it is of an integer or a floating point or an enumeration type.

Figure 2 shows the declaration of one register file and two standalone registers from the DLX model. All of these registers have a width of 32 bits and the register file has a depth of 31. Note that although nominal DLX register file has a depth of 32, but since register R0 always returns a 0 value upon reading, it is not a real, non-volatile storage element and thus the register file has 31 actual registers.

4.4. Expressions and Statements

Expressions and *statements* are used to describe the behavior of the processor operations. Every statement is either a conditional assignment or a reference to a statement attribute of a sub-operation (more on operations and different kinds of attributes later). Every conditional assignment has two major parts: 1) an optional condition expression and 2) a T-RTL assignment to some storage element(s). If the condition expression is present, the assignment should take place only if the condition expression evaluates to true. Figure 3 shows an example of an assignment statement in the `action` attribute of **`operation`** `add`.

The set of supported expressions in APDL includes conditional, relational, shift, mathematical, logical, bit-concatenation and type conversion expressions. They are chosen to support all RTL operations found in common hardware description languages.

There are two kinds of expressions in APDL: **resources** and **simple**. Resourced expressions use the T-RTL description style and can include *resource usage clauses* for any operation or operand while simple expressions are plain RTL ones.

A resource usage clause contains one or more *resource usage declarations*. Each resource usage declaration has three clauses: 1) the resource to be used, 2) the clock cycle at which the computation of the expression begins (start time), and 3) the number of clock cycles required.

The *resource clause* can be a completely specified resource name like `ALU` or `FP[3]`, or an incompletely specified resource name like `Mult[?]`. The latter form provides a facility for the designer to indicate that he or she wishes to indicate the resource as one of a group of resources without exactly specifying which one. Such incomplete specifications can be used for design optimization similar to the way that don't cares are used in logic optimization. Inclusion of such features increases the usefulness of APDL for DSE.

The start time and the required clock cycles are either integer constants or, similar to incomplete resource specifications, might be left unspecified. In the latter case, the backend tools can decide their values based on the context in which the resource usage clause appears.

In Fig. 3, the `|ALU,EX,1|` clause which succeeds the `+` indicates that this addition takes place at cycle `EX` (which is the second clock cycle), uses the resource named `ALU`, and takes one clock cycle to execute. The fact that in DLX, register R0 always returns zero upon reading and cannot be used as a non-volatile storage location has been addressed in the description of `reg_src` and `reg_dst` operations through the use of conditional expressions and integer range types.

4.5. Operations and Attributes

Operations are the focal point of any APDL description. *Attributes* describe different aspects of operations, as well as resources and storages.

APDL provides for hierarchical operation descriptions, i.e., the description of one operation can refer to the description of other ones as sub-operations. An APDL operation does not necessarily represent a complete (or standalone) operation of the target processor. Partial behaviors like reading (writing) from (to) memories can be described as separate operations. In fact, because basic operations like reading from register file might be part of many computational operations, one might benefit from encapsulating them into separate operations and using them as sub-operations of higher-level operations. Figure 3 shows **`operation`** `add` which uses instances of `reg_src` and `reg_dst` as sub-operations.

APDL has two types of operations: **single** and **group**. A single operation declaration, like `reg_src` and `add` in Fig. 3, has an argument list whose elements must either be an instance of a sub-operation or an instance of a declared

```
#define ID 1
#define EX 2
#define WB 4

type reg_read_range is 0 to 31;
type reg_write_range is 1 to 30;
type int32 is int<32>;

resource rf_read_port[2], rf_write_port, ALU;
storage reg_file[31][32];
fixed inherited operation attribute rf_read_port_number;

operation reg_src ( addr : reg_read_range ) is
   val :=
      (addr == 0)
          ? 0 |rf_read_port[rf_read_port_number],ID,1|
          : reg_file[addr] |rf_read_port[rf_read_port_number],ID,1| ;
end operation;

operation reg_dst ( addr : reg_write_range ) is
   val := reg_file[addr] |rf_write_port,WB,1|;
end operation;

operation add ( s0: reg_src, s1: reg_src, d: reg_dst ) is
   s0'rf_read_port_number := 0;
   s1'rf_read_port_number := 1;
   action := {
       d'val := int32(s0'val)  +|ALU,EX,1|  int32(s1'val);
   }
end operation;

operation sub . . . end operation;
.
.
operation all_ops is {add, sub, . . . } end operation;

instruction ins is {all_ops} end instruction;
```

Figure 3. Exmaple code demonstrating different aspects of operation declarations in APDL

data type. In the latter case, the argument represents one immediate operand of the operation. A group operation declaration is a list of operations, like `all_ops` in Fig. 3. Every reference to such a group operation can be substituted with a reference to each of the grouped sub-operations. This operation hierarchy declaration has been inspired by the nML [6] language, although there are major semantic differences in the usage of sub-operations.

There are three types of attributes in APDL: **expression**, **statement** and **fixed**. The allowed "values" of these attributes are *expressions*, *statements* and *constant values*, respectively. Also, there are two classes of attributes in APDL: **inherited** and **synthesized**. Inherited attributes of an operation *OP* should be defined in the declaration of those operations of which *OP* is a sub-operation, while synthesized attributes of *OP* are defined in the declaration of *OP* itself. Figure 3, from the DLX description, shows an example of inherited attributes as `rf_read_port_number` which is used in the description of `reg_src` and `add` operations. The DLX register file has two read ports, one for the first register operand and the other for the second register operand of register operations. Declaration of `rf_read_port` resource declares two resources, one for each read port of the register file. Since the `add` operation uses two instances of `reg_src` as sub-operations and these instances should use different read ports, the declaration of `add` uses inherited attribute `rf_read_port_number` to provide its sub-operations with the required information. The idea of using attribute grammars for ADL design has been previously used in languages like nML [6] and LISA [11]. However, all of these languages only use synthesized attributes, while we consider inherited attributes as a powerful aid for writing concise descriptions.

Fixed and expression attributes of one entity can be referenced in the definition of expression or statement attributes of another entity, while statement attributes can only be referenced in the statement attributes of other entities. In Fig. 3, two predefined operation attributes `action` and `value` describe the behavior of declared operations.

4.6. Instructions

Instructions are top-level constructs which program the processor. Every processor operation can be executed if it is used inside an instruction. In APDL, the designer can define multiple instructions. Each instruction consists of one or more operations that should be executed in parallel; no inter-locking is allowed inside an instruction to resolve possible resource conflicts or data/control hazards. There is no restriction on the number of instructions and the number of operations in an instruction.

Figure 3 shows the declaration of the `ins` instruction. Every instance of `ins` can contain one operation of kind `all_ops` which has been declared as a group operation. `all_ops` is intended to contain all the high-level operations of the processor.

Based on the temporal behavior of the operations, some combinations of the operations inside an instruction might be invalid because of, for example, resource conflicts. These restrictions depend on how the instructions have been defined, and will be automatically extracted by the processing tool. This will relieve the designer of manually specifying such constraints and let him focus on other aspects of the design.

4.7. Processor Timing Model

APDL uses a formal timing model for instructions in the program sequence. In this model, there is a **lock** signal associated with each instruction in the program. At each instruction cycle, one new instruction should be issued in the processor. Also, the time of every instruction inside the processor, i.e., those issued in the previous cycles, should be advanced unless the lock signal is activated for that instruction.

In the APDL view, the code scheduler and the control unit of the processor collaborate in determining the progress of different instructions in the program. This view comes from the fact that both the code scheduler and the control unit can resolve control and data dependencies between instructions. For example, consider two instructions *A* and *B* where *B* has a data dependency on *A*. *A* computes its result 5 cycles after it has been issued and, thus, *B* should be locked for 5 cycles before it can progress and use the result of *A*. There are different possibilities for how to provide for these 5 cycles: 1) code scheduler can create a 5-instrcution distance between *A* and *B* in the program sequence, or 2) control unit can lock *B* for 5 cycles before using the result of *A*, or 3) a hybrid of both techniques can be used.

In APDL, all the instructions in the program sequence are parallel and can be issued at the same time. However, code scheduler and control unit delay issuing instructions by locking them: code scheduler locks them by ordering them in the program sequence and control unit locks them by activating the associated lock signal during their execution in the processor. It should be noted that the concept of the lock signal has solely been introduced to formalize the timing model of the processor and in many cases it can be easily realized in the processor implementation. For example, for processors with a pipelined structure, having a stall signal for each pipeline stage is enough to implement this locking mechanism.

5. Summary and Future Works

This paper presented the design of Anahita Processor Description Language (APDL). APDL uses a new abstraction level, called Timed Register Transfer Level, to describe the temporal behavior of the processor operations and their interaction with hardware resources in the processor. This provides enough information to automatically generate the control unit of the target processor and enables automatic generation of aggressively optimizing compilers and cycle accurate instruction set simulators for the described processor. The authors believe that all of these features are strongly required for the DSE phase of designing embedded processors.

Currently, we have developed the APDL Analyzer, a tool which reads the APDL description and converts it to an intermediate format, and are working on a retargetable compiler backend, a cycle-accurate instruction set simulator, and an architectural verification tool based on APDL.

References

[1] Intel Corporation, “Moore's Law”, http://www.intel.com/technology/mooreslaw/index.htm (current April 2007).

[2] M.K. Jain, M. Balakrishnan and A. Kumar, “ASIP Design Methodologies : Survey and Issues”, in Proc. VLSID'01, p. 76.

[3] J.L. Hennessey and D.A. Patterson, “Computer Architecture: A Quantitative Approach”, 3rd Ed., Morgan Kaufmann Publishers, 2003.

[4] R. Leupers and P. Marwedel, “Retargetable code generation based on structural processor descriptions”, Design Automation for Embedded Systems, vol. 3, no. 1, 1998.

[5] R. Leupers et al, “Retargetable generation of code selectors from HDL processor models”, in Proc. EDTC’97, pp. 140-144.

[6] M. Freericks, “The nML machine description formalism”, Technical Report TR SM-IMP/DIST/08, TU Berlin CS Dept., 1993.

[7] Target Compiler Technologies, http://www.target.com

[8] J. Paakki, “Attribute grammar paradigms-- a high-level methodology in language implementation”, ACM Computing Surveys, vol. 27, no. 2, pp. 196-255, June 1995.

[9] G. Hadjiyiannis et al, “ISDL: An instruction set description language for retargetability”, in Proc. DAC’97, pp. 299-302.

[10] S. Hanono and S. Devadas, “Instruction selection, resource allocation, and scheduling in the AVIV retargetable code generator”, in Proc. DAC’98, pp. 510-515.

[11] S. Pees et al, “LISA-machine description language for cycle-accurate models of programmable DSP architectures”, in Proc. DAC'99, pp. 933-938.

[12] A. Halambi et al, “EXPRESSION: A Language for Architecture Exploration through Compiler/Simulator Retargetability”, in Proc. DATE'99, p. 485.

[13] W. Qin, S. Rajagopalan, and S. Malik, “A formal concurrency model based architecture description language for synthesis of software development tools”, in Proc. LCTES'04, pp. 47 – 56.

[14] R. Azevedo et al, “The ArchC Architecture Description Language and Tools”, International Journal of Parallel Programming, vol.33, no.5, pp. 453-484, October 2005.

[15] C C. Siska, “A processor description language supporting retargetable multi-pipeline DSP program development tools”, in Proc. ISSS’98, pp. 31-36.