

Processor Description in APDL for Design Space Exploration of Embedded Processors

N. Honarmand¹, H. Sohofi¹, M. Abbaspour² and Z. Navabi¹

¹ CAD Laboratory, School of ECE, University of Tehran, Tehran, IRAN

² School of ECE, Shahid Beheshti University, Tehran, IRAN

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

Abstract

This paper presents modeling of embedded processors in Anahita Processor Description Language (APDL). APDL is a language 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 new embedded processors. APDL introduces a new level of abstraction for processor description. This language can be used for generation of tools such as compilers, architecture verification tools, instruction set simulators, and hardware generators. In particular, it provides constructs to explicitly model the interaction of instructions in the processor's code sequence. The paper first investigates the features required for a language to be useful for DSE and then presents APDL constructs along with code samples and a case study.

1. Introduction

The proliferation of 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 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. On the other hand, shrinking feature sizes in IC fabrication technology has caused an increase of complexity in modern IC designs, and thus, 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. To address the performance requirements of software-based design methodology, embedded designers have turned to use techniques like Instruction Set Extensions (ISEs), Digital Signal Processors (DSPs) and Application Specific Instruction Processors (ASIPs) [1].

Before committing to a specific processor architecture, the ASIP or DSP designer should measure the figures of merit for different alternative architectures, a process 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 would like 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 checking between different descriptions and thus is not desirable. Conventionally, such languages are called architecture description languages (ADL).

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 design space exploration (DSE) during the design of new or modified embedded processors.

The rest of the paper is organized as follows: Section 2 provides the goals driving current structure of APDL. Section 3 surveys some of the previous works and compares them with APDL. Section 4 provides an introduction to the major features of the APDL. Section 5 provides a case study and Section 6 concludes the paper.

2. Goals and Requirements

Two different requirements in ASIP design process, namely irregular hardware structures and the need for aggressive code optimizations, have greatly impacted the current structure of APDL. Irregular data paths, multiple instruction pipelines and split register files are among the common features in DSPs and ASIPs. As a result, control logic design is a difficult and error-prone task in 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 and error-prone to consider all these combinations in a hand coded assembler or code generator. Since nearly all of these constraints arise from resource conflicts between processor op-

erations, the information required to detect such constraints should be incorporated in the description in a way that allows for automatic generation of such tools.

On the other hand, Power-consumption considerations in modern embedded applications discourage the embedded processor designers from using techniques like dynamic scheduling and hazard/resource conflict resolution [2]. To compensate for the performance loss due to lack of such features, embedded designers should rely on aggressive code optimizations by code scheduler. To do this, the code scheduler needs detailed information about the internal behavior of processor instructions like when an instruction is going to read (write) a value from (to) a register or memory. This way, it will be able to effectively utilize the delay slots between producer and consumer instructions and increase the performance.

3. Related Works Versus APDL

3-1. Previous Work

Conventionally, ADLs have been classified into three major categories:

- Structural ADLs which focus on hardware components of a processor (MIMOLA [3])
- Behavioral ADLs which mainly focus on the functional semantics of the processor's instruction set (nML [5] and ISDL [8])
- Mixed ADLs which consider both structure and behavior and provide constructs to express their interactions (LISA [10], EXPRESSION [11] and MADL [12])

MIMOLA [3] focuses on describing the structure of the target processor with HDL-like constructs. In [4], 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.

nML [5] is an elegant formalization for describing the IS of a processor used by the Belgium-based Target company [6] in its CHESS/CHECKERS processor design tool suite. nML provides constructs for hierarchical and concise operation descriptions. Being a behavioral ADL, it ignores detailed temporal resource requirements of the operations. Also, in nML, designer should explicitly enumerate all the operation combinations that form valid instructions, an infeasible task for large ASIP designs. ISDL, [8] and [9], targeted mainly towards VLIW and DSP processors, follows the same line as nML although it provides, through description of constraints, the ability of invalidating some operation combinations in the instruction word. Here, the designer should manually extract and code invalid operation combinations, a tedious and error-prone task for complex irregular architectures.

LISA [10], EXPRESSION [11] and MADL [12] are examples of mixed-paradigm ADLs. In LISA, designer should provide a detailed and explicit description of behavior and interaction of operations in different stages of processor pipeline. Though a good feature for generation of cycle accurate instruction set simulators, this feature 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. EXPRESSION [11], on the other hand, provides features more suitable for DSE. Especially, through the description of pipeline stages, it provides the notion of operation-to-resource mapping. One major feature of EXPRESSION not found in other ADLs is the ability to describe the memory subsystem in the same processor description. There are several major drawbacks in EXPRESSION, though. First, it lacks hierarchical operation description which makes its descriptions lengthy. Second, it describes the semantics of the instructions by providing a mapping between operations of the target machine and a generic machine. This makes the language somehow tool-dependent and cumbersome to use. Third, 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-pipeline and pipeline-resource relationships. MADL [12] uses an 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 extract control information required by tools like compilers.

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 processor operations. 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. 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 [10], the operation behavior in different pipeline stages cannot be generally used to extract control information required by an optimizing compiler.

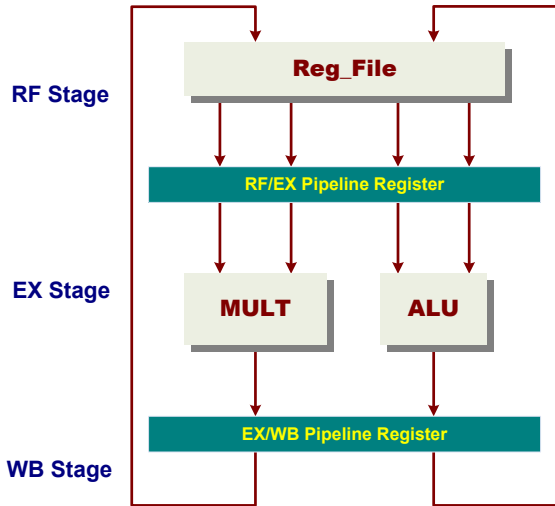


Figure 1. Datapath of the original processor

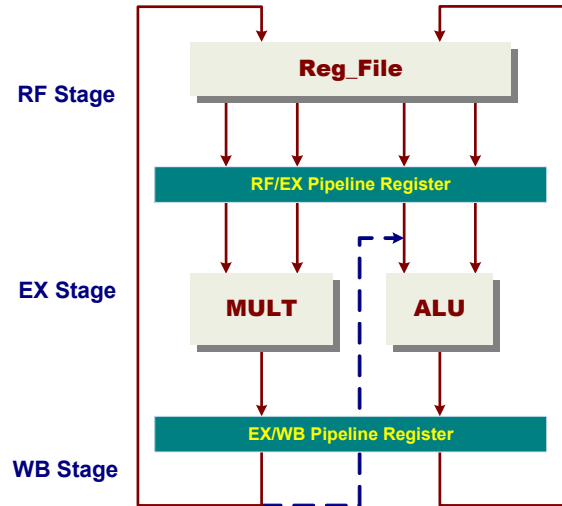


Figure 2. Datapath of the modified processor

```

#define RF 0
#define EX 1
#define WB 2

type reg_range is 0 to 31;
type int32 is int<32>;

resource RF_READ_PORT[4], RF_WRITE_PORT[2];
resource ALU, MULT;

storage REG_FILE[31][32];

operation mult_reg_src_0 (addr : reg_range) is
  val := REG_FILE[addr] |RF_READ_PORT[0],RF,1|;
end operation;

operation mult_reg_src_1 (addr : reg_range) is
  val := REG_FILE[addr] |RF_READ_PORT[1],RF,1|;
end operation;

operation alu_reg_src_0 (addr : reg_range) is
  val := REG_FILE[addr] |RF_READ_PORT[2],RF,1|;
end operation;

operation alu_reg_src_1 (addr : reg_range) is
  val := REG_FILE[addr] |RF_READ_PORT[3],RF,1|;
end operation;

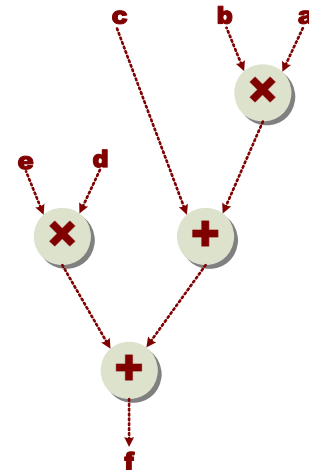
operation mult_op (s0 : mult_reg_src_0; s1 : mult_reg_src_1; dst_addr: reg_range) is
  action := {
    REG_FILE[dst_addr] |RF_WRITE_PORT[2],WB,1| := int32(s0'val) *|MULT,EX,1|
    int32(s1'val); }
end operation;

operation add_op (s0 : alu_reg_src_0; s1 : alu_reg_src_1; dst_addr: reg_range) is
  action := {
    REG_FILE[dst_addr] |RF_WRITE_PORT[1],WB,1| := int32(s0'val) +|ALU,EX,1|
    int32(s1'val);
  }
end operation;

instruction ins is (m_op : mult_op; a_op : add_op) end instruction;

```

Figure 4. APDL description of the datapath of Figure 1



```

int a, b, c, d, e, f;
f = a*b + c + d*e

```

Figure 3. Sample input code and its DFG

Also, the T-RTL representation can be regarded as a generalized form of the pipeline-oriented description style of languages like LISA [10] or EXPRESSION [11]. The implementation style of the processor, whether it is of a pipelined or multi-cycle or single cycle form, 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 description

4. The APDL Language

4-1. Processor Model's Abstraction Level

In the APDL view, a processor is a programmable element that executes the semantics associated with a sequence of *instructions*. Each instruction consists of a set of processor *operations*. In the APDL terminology, the term *operation* can be used to refer to any data transfer or manipulation behavior inside the processor. Conventionally, a processor is divided into datapath and control units. Each processor operation might use several hardware elements in the datapath during its lifetime. The control unit decides when an operation should use which hardware element. To do so, the control unit needs to know about the exact behavior of each operation and also about the interaction of active operations to preserve dependencies and prevent conflicts among them. A DSE-friendly processor description should provide enough information about the datapath and control unit of the processor.

In APDL, the designer does not provide explicit descriptions for the control unit and datapath. Instead, he or she describes the behavior of processor operations in T-RTL. The designer combines the described operations to form the processor instructions. Detailed information regarding the structure of datapath and functionality of the control unit will be inferred automatically from these descriptions. T-RTL descriptions provide an implicit model of the processor's datapath and control unit. In other words, in an APDL description, the designer implicitly gives the datapath and control unit requirements which should be fulfilled in order to implement the described operation semantics.

In all other behavioral or mixed ADLs, each operation is described in terms of its operands and the storage elements of the processor. In this scheme, if the designer wants to include some inter-instruction control rules like forwarding, he or she should do so using the storage elements of the processor. Generally, this description style is not amenable to automatic extraction of control information in a way that can be effectively used by high level tools like compilers. Such tools need a fairly high level view of what is inside the processor. To solve this problem, some languages like MADL [12] have opted to use annotations to convey such information. But this method suffers from the problem of redundancy. Such annotations provide redundant semantics which have already been described in a different part of the design. If the two sets of redundant information are not con-

sistent, the design would not be causal, i.e., there could be no feasible realization of the processor.

In APDL, on the other hand, we chose to explicitly model the interaction between different instructions in the code sequence. We use the concept of *control flags* to model such dependencies. This method of description provides enough information both for tools like compilers which need high level information about processor behavior and tools like hardware generators or instruction set simulators which should implement the described behavior in hardware or software. Section 5 provides an example of control flags.

4-2. APDL Constructs

An APDL description consists of *data type declarations*, *resources*, *storages*, *expressions*, *statements*, *flags*, *attributes*, *operations* and *instructions*. What follows briefly discusses these APDL entities. Figures 4 and 5 show examples of these entities. For a more in-depth description, the reader might refer to [13]. APDL is a strongly typed language. For each operation, the types of its arguments are checked at the time of analysis. Every data type used in the design should be declared before use, like `int32` in Figure 4. *Resources* are used to express resource requirements of operations. *Storages* represent non-volatile storage elements of the processor like registers, register files and memories.

Expressions and *statements* are used to describe the behavior of processor operations. Every statement is either a conditional assignment or a reference to a statement attribute of a sub-operation. 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 evaluates to true. Figure 4 shows an example of assignment statement in *action* attribute of operation `add_op`. There are two kinds of expressions in APDL: *simple* and *resourced* expressions. Resourced expressions use T-RTL description style and can include resource usage clauses. 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) resource in use, 2) start time and 3) the number of clock cycles required. The start time and required clock cycles are either integer constants or might be left unspecified. In Figure 4, the `|ALU, EX, 1|` clause which succeeds the `+` operation indicates that this add operation will take place at 1st clock cycle (because `EX` has been defined as 1) and will use the resource named `ALU`. *Flags* are used to describe inter-instruction dependencies. Next section provides a description of flags in context of a case study.

Operations are the backbone of descriptions in APDL. Most of the important design data are provided through operation descriptions. *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. APDL has two types of operations: *Single* and *Group*. Each single operation declaration, like `alu_reg_src_0` and `add_op` in Figure 4, has an argument list whose elements must either be an instance of a sub-operation or an instance of a declared data type. In the latter case, the argument represents one operand of the operation. Each group operation declaration is a list of operations. Every reference to such a group operation can be substituted with a reference to each of the grouped sub-operations. *Attributes* provide an elegant and uniform syntactical representation to describe different aspects of operations, resources and storages. 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*. The idea of using attribute grammars [7] for ADL design has previously been used in languages like nML [5] and LISA [10]. However, all of these languages only use synthesized attributes, while we consider inherited attributes as a powerful aid for writing concise descriptions.

Instructions are top-level constructs which program the processor. 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. Based on the temporal behavior of the operations, some combinations of the operations in an instruction might be invalid because of, for example, resource conflicts. These restrictions will be automatically extracted by the APDL analyzer. This relieves the designer of the burden of manually specifying such constraints.

5. An Example Case Study

In this section, we provide a simple example to demonstrate the description of inter-operation dependencies in APDL. Figure 3 shows a C code fragment and its related data-flow graph. Suppose that values `a` to `f` are alive upon completion of this code fragment and, thus, should be assigned to different storage elements. Figure 1 shows the datapath of the processor on which this code should run and Figure 4 gives the APDL description of this processor. Each instruction goes through 3 stages named RF (for Register File), EX (for Execute) and WB (for Write Back) which are 0th to 2nd cycles of each instruction. As shown in Figure 4, each instruction of this processor has two operations, one multiplication for the MULT unit and one addition for ALU. Suppose values of `a` to `f` are assigned to R1 to R6, respectively.

At least three such instructions should be used to compile the C code of Figure 3 on this processor. Here is one possible instruction sequence:

- ```
(1) Mult R1,R2,R6 ; NOP
(2) Mult R4,R5,R7 ; Add R6,R3,R6
(3) NOP ; Add R6,R7,R6
```

Instructions (2) and (3) have data dependencies on their preceding instructions. Hence, they should wait for the completion of the previous instruction before starting execution. As shown in Figure 6(a), this sequence needs 9 cycles to complete on this architecture. Now suppose that the designer recognizes frequent use of multiply-add operations in the target application. Then he/she might decide that a forwarding path from the output of MULT to the input of ALU would be a reasonable modification to this architecture and changes the datapath to that of Figure 2. With the inclusion of this forwarding path, the above sequence would take 7 cycles to execute, as shown in Figure 6(b). Along the addition of the forwarding path to the datapath, the control unit should also be augmented. The control decision of using the forwarded input or the input coming from register file depends on the previous instructions in the code sequence. The use of the forwarded value should only be allowed if the first operand of the addition operation in the current instruction is the destination register of a multiplication operation in the immediately preceding instruction.

Figure 5 shows the necessary changes in APDL description for this purpose. 1) The designer should declare a register, named `WB_MULT_RES` in this example, to hold the forwarded value. In hardware implementation, this register could be mapped to a portion of the EX/WB pipeline register. 2) The declaration of operation `alu_reg_src_0` should change to use the forwarding path. Here, a flag, named `forward`, has been declared. Flags are boolean-valued expression which should be evaluated when an operation enters its 0<sup>th</sup> cycle. The declaration of `forward` has two different parts: first it declares that when the operation enters its 0<sup>th</sup> cycle, there should be an instruction of type `ins` at its 1<sup>st</sup> (EX) cycle. This instruction would be referenced as `i` in the rest of this flag declaration. Second, it declares a condition, after `:=`, as the value of the flag. The condition here states that the address of the destination register of the multiplication operation in `i` should be the same as the register address passed to this `alu_reg_src_0` operation. In fact, flag `forward` declares a condition among instructions in the code sequence which governs the use of the forwarding path. 3) The declaration of operation `mult_op` should also change to write the result of multiplication operation in `WB_MULT_RES` as well as `REG_FILE`. Here, the `?` in the resource usage clause of `WB_MULT_RES` indicates that designer would not like to indicate any explicit resource requirement for this operation and just wishes to specify its timing.

## 6. Summary and Future Works

This paper presented the design of Anahita Processor Description Language (APDL). APDL uses a new abstraction level, called T-RTL, to describe the temporal behavior of processor operations and their interaction with hardware resources in the processor. Also APDL provides constructs to express interactions between instructions in the code se-

```

storage WB_MULT_RES[32];
operation alu_reg_src_0 (addr : reg_range) is
 flag forward(i : ins@EX) := i.m_op.dst_addr == addr;
 val := (forward
 ? WB_MULT_RES [|?,EX,1|
 : REG_FILE[addr] |RF_READ_PORT[2],ID,1|);
end operation;
operation mult_op (s0 : mult_reg_src_0; s1 : mult_reg_src_1; dst_addr: reg_range) is
 action := {
 WB_MULT_RES [|?,EX,1| := REG_FILE[dst_addr] |RF_WRITE_PORT[2],WB,1| :=
 int32(s0'val) +|MULT,EX,1| int32(s1'val);
 }
end operation;

```

Figure 5. Required changes in description of Figure 4 to implement architecture of Figure 2 (Gray color indicates the text unchanged from Figure 4)

|     | 0  | 1         | 2         | 3  | 4         | 5         | 6  | 7  | 8  |
|-----|----|-----------|-----------|----|-----------|-----------|----|----|----|
| (1) | RF | EX        | WB        |    |           |           |    |    |    |
| (2) |    | <b>RF</b> | <b>RF</b> | RF | EX        | WB        |    |    |    |
| (3) |    |           |           |    | <b>RF</b> | <b>RF</b> | RF | EX | WB |

(a)

|     | 0  | 1  | 2         | 3         | 4  | 5  | 6  | 7 | 8 |
|-----|----|----|-----------|-----------|----|----|----|---|---|
| (1) | RF | EX | WB        |           |    |    |    |   |   |
| (2) |    | RF | EX        | WB        |    |    |    |   |   |
| (3) |    |    | <b>RF</b> | <b>RF</b> | RF | EX | WB |   |   |

(b)

Figure 6. Execution steps of input C code on (a) original and (b) modified architectures. Bold-faced entries indicate stall cycles.

quence. These features provide enough information to automatically generate data path and control unit of a processor and enables automatic generation of aggressively optimizing compilers and cycle accurate instruction set simulators.

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

## 7. References

- [1] M.K. Jain, M. Balakrishnan and A. Kumar, "ASIP Design Methodologies : Survey and Issues," in Proc. VLSID'01, p. 76.
- [2] J.L. Hennessey and D.A. Patterson, "Computer Architecture: A Quantitative Approach", 3rd Ed., Morgan Kaufmann Publishers, 2003.
- [3] R. Leupers and P. Marwedel, "Retargetable code generation based on structural processor descriptions" Design Automation for Embedded Systems, vol. 3, no. 1, 1998.
- [4] R. Leupers et al, "Retargetable generation of code selectors from HDL processor models", in Proc. EDTC'97, pp. 140-144.
- [5] M. Freericks, "The nML machine description formalism", Technical Report TR SM-IMP/DIST/08, TU Berlin CS Dept., 1993.
- [6] Target Compiler Technologies, <http://www.target.com>
- [7] J. Paakki, "Attribute grammar paradigms-- a high-level methodology in language implementation", ACM Computing Surveys, vol. 27, no. 2, pp. 196-255, June 1995.
- [8] G. Hadjiyiannis et al, "ISDL: An instruction set description language for retargetability", in Proc. DAC'97, pp. 299-302.
- [9] S. Hanono and S. Devadas, "Instruction selection, resource allocation, and scheduling in the AVIV retargetable code generator", in Proc. DAC'98, pp. 510-515.
- [10] S. Pees et al, "LISA-machine description language for cycle-accurate models of programmable DSP architectures", in Proc. DAC'99, pp. 933-938.
- [11] Halambi et al, "EXPRESSION: A Language for Architecture Exploration through Compiler/Simulator Retargetability", in Proc. DATE'99, p. 485.
- [12] 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.
- [13] N. Honarmand et al, "APDL: A Processor Description Language For Design Space Exploration of Embedded Processors", available at [http://cad.ece.ut.ac.ir/~nima/pubs/apdl\\_intro.pdf](http://cad.ece.ut.ac.ir/~nima/pubs/apdl_intro.pdf)