# A Novel Approach for Flexible and Consistent ADL-driven ASIP Design

Gunnar Braun, Achim Nohl
CoWare, Inc.
Dennewartstrasse 25-27
52068 Aachen, Germany
gunnar@coware.com

Weihua Sheng, Jianjiang Ceng,
Manuel Hohenauer, Hanno Scharwächter,
Rainer Leupers, Heinrich Meyr
Institute for Integrated Systems
Aachen, Germany

## ABSTRACT

Architecture description languages (ADL) have been established to aid the design of application-specific instruction-set processors (ASIP). Their main contribution is the automatic generation of a software toolkit, including C compiler, assembler, linker, and instruction-set simulator. Hence, the challenge in the design of such ADLs is to unambiguously capture the architectural information required for the toolkit generation in a single model. This is particularly difficult for C compiler and simulator, as both require information about the instructions' semantics, however, while the C compiler needs to know *what* an instructions does, the simulator needs to know *how*. Existing ADLs solve this problem by either introducing redundancy or by limiting the language's flexibility.

This paper presents a novel, mixed-level approach for ADL-based instruction-set description, which offers maximum flexibility while preventing from inconsistencies. Moreover, it enables capturing instruction- and cycle-accurate descriptions in a single model. The feasibility and design efficiency of our approach is demonstrated with a number of contemporary, real-world processor architectures.

**Categories and Subject Descriptors:** C.0 [Computer Systems Organization]: General – *Modeling of computer architecture*; D.3.2 [Programming Languages]: Language Classifications – *Design languages*, *LISA*; I.6.5 [Simulation and Modeling]: Model Development – *Modeling methodologies*; J.6 [Computer Applications]: Computer-Aided Engineering – *Computer-aided design (CAD)*

**General Terms:** Design, Languages

**Keywords:** ADL, ASIP, Embedded Processors

## 1. INTRODUCTION

Recently, architecture description languages (ADL) have been established as an efficient solution for the design of application-specific instruction-set processors (ASIP). The main contribution of such languages is the automatic software toolkit generation from ADL processor models, i.e., automatic retargeting of C compiler, assembler, linker, and instruction-set simulator on instruction- and cycle-accurate level. The most challenging tasks in the design of ADLs is to capture the architectural information required for the tool generation in a consistent and unambiguous way. In particular, the C compiler and the instruction-set simulator (possibly on different abstraction levels) require the specification of the instructions' semantics, however, from very different points of view. While a C compiler generally only needs information about *what* an instruction does, the simulator needs the details *how* the instruction performs its particular task. As it turned out to be a difficult if even impossible task to derive the *what* from the *how* (or vice versa), none of today's ADLs solves this problem satisfactory. Examining the variety of ADLs, one can distinguish two major approaches: either redundancy is introduced by providing the semantical information separate from the behavioral information (compiler and simulator specification), or the set of architectures that can be modeled is reduced by introducing a more formalized but inflexible description of the instruction behavior.

In this paper, we present a novel approach that neither sacrifices flexibility nor introduces redundancy. Besides providing an elegant solution for the problem presented above, it allows for the generation of instruction- and cycle-accurate simulator from a single model, and furthermore enables the generation of a consistent instruction-set documentation (not covered in this paper). The solution presented in this paper is based on an extension of the LISA architecture description language accompanied by an efficient modeling methodology.

The rest of the paper is organized as follows: section 2 elaborates the problem and discusses the approaches of related work. Section 3 defines the design criteria of the language extension and shows how such are addressed within scope of LISA. Section 4 shows how the simulator generator utilizes the information provided in the model, and reveals the relevant implementation details. A methodology for incorporation of instruction-set and microarchitecture information in a single model description, namely the generation of an instruction-accurate (IA) simulator from a cycle-accurate (CA) model, is discussed in section 5. Section 6 presents results on modeling efficiency and IA simulator generation, while the generation of the C compiler will be presented in a separate paper, as it would exceed the scope of this paper. Section 7 concludes the paper.

## 2. RELATED WORK

Architecture description languages have been developed to support the design of instruction-set processors. The major contribution of ADLs is the automatic generation of software toolkit, system interfaces, and even synthesizable RTL code. The high degree of automation reduces the design effort significantly, and thus allow for an efficient architecture exploration. Most ADLs known today have originally been designed to aid the automation of a particular piece in the puzzle, and have then been extended to address


*DAC'04*, June 7–11, 2004, San Diego, California, USA.

larger parts of the embedded processor design. As a result, most ADLs are well-suited for, for instance, the automatic generation of a C compiler while they impose major restrictions on, or are even incapable of, the generation of a cycle-accurate simulator.

Especially the synthesis of compiler and simulator from a single model reveals an interesting problem, since both require a same kind of information, but from a different perspective, or *view*. A C compiler, or better, the compiler's code selector, relies on the knowledge of each instruction's purpose (and its constraints) in order to select the appropriate assembly instruction(s) for a given piece of C code – in other words, the instruction *semantics*. In contrast, a cycle-accurate simulator is based on the detailed *behavior* of an instruction, i.e, *how* the instruction is executed on the processor, including the description of possibly occurring pipeline stalls or operand forwarding. As it is obviously impossible to create microarchitecture behavior from a semantic description, it is likewise impossible to extract the abstract, semantic information from a microarchitecture description without imposing major restrictions on the class of architectures that can be modeled at all.

After examination of today's state of the art, the existing ADLs can be separated into three categories: those which are capable of generating *either* compiler *or* CA simulator for contemporary processors, those which introduce redundancy by separating semantic and behavioral description, and those which combine both views into a very restrictive grammar, hence sacrificing flexibility.

The first category includes languages as ISDL [6] or LISA 1.0, a former version of LISA used by Axys Design Automation [1]. ISDL has been reported to generate efficient compilers, however, is not capable of describing processors on cycle-accurate level. Hence, CA simulators cannot be generated. LISA 1.0 allows for the generation of fast, compiled simulators on IA and CA level, but nothing has been published on C compiler generation yet.

The EXPRESSION [4] ADL belongs to the second group as C compiler (EXPRESS) as well as simulator (SIMPRESS) are automatically generated, however, the views for both are separated into two different sections of the architecture description. While the simulator employs the information in a *behavior section*, the compiler relies on the existence of a so-called *operation mapping*, which associates the instruction with a tree of primitive operations similar to the nodes in a compiler's intermediate representation (IR). Naturally, this aids the task of code selection, but at the expense of the incorporation of redundant, purely compiler-specific information into the ADL.

By far the most approaches can be found in the third category. ADLs such as nML [8], MIMOLA [15], and PEAS-III [9] define a very restricted grammar for the modeling of the instruction behavior, mostly by offering a set of predefined operators which can be combined into a behavioral description. For each operator, the functional behavior (for simulation) as well as its semantics (for compilation) are defined and usually contained in additional libraries. A further step is taken by a number of approaches employing so-called *parameterizable generic processor cores*, such as Tensilica's Xtensa [3, 14] architecture. Such cores usually have a fixed instruction-set, which can be extended by adding predefined or user-defined instructions. As software toolkit and synthesizable RTL model are available for the fixed part of the core, only the additional instructions must be taken into account, and still, most of the user-defined instructions cannot be utilized by the compiler's code selector. The approaches in this category are rather suitable for the design of domain-specific processors than for ASIPs as they pose too little flexibility on the modeling of complex pipelines and execution schemes.

Recently, Weber et al. formalized the problem of capturing multiple architecture *views* in a single architecture model, and presented an analytical solution approach applied to a channel encoding processor [17]. However, details on how the semantic consistency is kept are not available, and no results have been published on the C compiler generation yet.

```
OPERATION ADD {
  DECLARE {
    GROUP src1, dst = { reg };
    GROUP src2 = { reg || imm };}
  SYNTAX { "add" dst "," src1 "," src2 }
  CODING { 0b0000 src1 src2 dst }
  BEHAVIOR {
    dst = src1 + src2;
    if (  ((src1 < 0) && (src2 < 0))
       || ((src1 > 0) && (src2 > 0) && (dst < 0))
       || ((src1 > 0) && (src2 < 0) && (src1 > -src2))
       || ((src1 < 0) && (src2 > 0) && (-src1 < src2)))
       )
    { carry = 1; }}}
```

**Figure 1: LISA Operation (IA)**

LISA 2.0, which this work is based on, belongs to the first category [5]. LISA offers a high degree of flexibility by allowing the usage of C for the description of the instruction behavior. This enables the generation of high-speed simulators on IA and CA level for a broad range of contemporary RISC, VLIW, NPU, DSP, and ASIP architectures. On the other hand, a C description – possibly on microarchitecture level – cannot be used for the extraction of the instruction semantics.

The following sections present a semantic extension for LISA, which enables the automatic generation of C compilers, while keeping the model consistency by using the same information for simulator generation as well. Unlike the above approaches, this does not affect the flexibility, as C can still be used for the description of very irregular instructions and microarchitecture details. Hence, on IA level, consistency is kept by using the same information for simulator and compiler generation. On CA level, semantic and behavior description are separated, which enables the automatic generation of IA and CA simulator from a single model, while still being able to generate C compiler and instruction-set manual.

# 3. A SEMANTIC EXTENSION FOR LISA

LISA captures the instruction-set description in so-called *operations*. Depending on the abstraction level of the model – namely IA or CA – an operation may describe an entire instruction, a part of an instruction, e.g., an immediate operand, or even a piece of a functional unit, e.g., a stage of a pipelined multiplier. Each operation may contain a number of sections describing the attributes of the operation dependent on its purpose. As an example, an operation modeling (a part of) an instruction usually contains a SYNTAX section specifying the instruction's assembly syntax, while an operation describing an instruction fetch unit would only contain a BEHAVIOR section defining the functional behavior of the unit. An exemplary LISA operation is shown in figure 1.

The operation (describing a simple add instruction) illustrates the problem introduced in the previous sections: even for this relatively simple operation, it is nearly impossible to extract the operation's semantics from the behavioral description, in particular, when considering that due to the flexibility of C, the presented description is only *one* way to model an add with carry flag computation.

From the example, we can derive the requirements for a semantic operation description: uniqueness, simplicity, and flexibility. For most operations, there should be only a single, concise way to define the semantics, while the grammar of a SEMANTICS section should be flexible enough to describe complex operations.

## 3.1 Micro-operation Approach

The MIMOLA ADL [15] employs a set of so-called *micro-operations* to describe a processor's instruction-set. Micro-operations are primitive operations similar the instructions of a RISC ISA, which allow to model simple instruction by means a single micro-operation, and complex instructions (as found in CISC machines) by a combination of such. Although the micro-operation approach has turned out to be unsuitable for the description of complex microarchitectural behavior, it has proven feasible and complete for the specification of instruction semantics. As the description of the microarchitecture is left to the existing BEHAVIOR section in our approach, we adapted the idea for the definition of a SEMANTICS section.

```
OPERATION ADD {
  DECLARE {
    GROUP src1, dst = { reg };
    GROUP src2 = { reg || imm };}
  SYNTAX      { "add" dst "," src1 "," src2 }
  CODING      { 0b0000 src1 src2 dst }
  SEMANTICS   { _ADDI[_C] ( src1, src2 ) -> dst; }}

OPERATION reg {
  DECLARE {
    LABEL index;
  SYNTAX      { "R" index=#U4 }
  CODING      { index=0bxxxx }
  SEMANTICS   { _REGI(R[index])<0..31> }}
```

**Figure 2: LISA Operation with Semantics**

Figure 2 shows the operation from figure 1 using the SEMANTICS section instead of the BEHAVIOR section. A single statement consisting of a single micro-operation is required to precisely describe the purpose of the operation. The micro-operator `_ADDI` defines the integer addition, while the `_C` in square brackets specifies that the carry flag is affected by the operation. A comma-separated list of operands follows in parenthesis, and finally, the pointer (`->`) specifies the location for the result.

The operands of the micro-operator can be either terminal elements, such as integer constants, or other operations. In the latter case, the respective operations (here: `reg` and `imm`) must contain a SEMANTICS section on their own. In the example, the SEMANTICS section of the `reg` operation defines the semantic type of the operand – here, a 32-bit integer register specified as array `R` in LISA's RESOURCE section (not shown).

In general, each operand of a micro-operation can be represented as a 3-tuple $(u, v, w)$ consisting of the value/resource $(u)$ and a bit-field specification represented by bit offset $(v)$ and bit width $(w)$. The corresponding 3-tuple for operation `reg` is $(u, v, w) = (R[index], 0, 32)$. The same formalism can be applied to operation `ADD`. As no explicit bit-field specification is given in its SEMANTICS section, the expression shares the specification of the operands. In other words, the addition of two operands $(a, 0, 32)$ and $(b, 0, 32)$ results in the 3-tuple $(c, 0, 32)$, where $c$ is the result of the 32-bit addition of $a$ and $b$. A constraint for the `_ADDI` micro-operator is that both operands have the same bit width. If that constraint is not met, the respective operand has to be extended to match the width of the second operand by means of an explicit sign/zero extension. Two separate micro-operations `_SXT` and `_ZXT` serve that purpose.

The generic 3-tuple operand representation allows for a very compact instruction-set description while keeping the number of required micro-operations small. But there is another advantage: micro-operations can be used as operands for other micro-operations. This mechanism is further on called *chaining* and is discussed is the following paragraphs.

```
OPERATION DMAC {
  DECLARE {
    GROUP src1, src2, dst1, dst2 = { reg };
  SYNTAX      { "dmac " dst2 ":" dst1 "," src1 "," src2 }
  SEMANTICS   {
    _ADDI ( _MULUU( src1, src2 )<0..31>  , dst1 ) -> dst1;
    _ADDI ( _MULUU( src1, src2 )<32..63> , dst2 ) -> dst2;}}
```

**Figure 3: Chaining and Parallelizing**

## 3.2 Complex Operations

It is obvious that only a limited set of RISC instructions can be expressed by single micro-operations. In order to model complex instructions, two mechanisms are employed that combine two or more micro-operations, namely, chaining and parallelizing. Both are illustrated in figure 3.

The example in the figure shows the description of a dual multiply-accumulate instruction (`DMAC`), which carries out a 32x32 multiplication and accumulates the high and low word of the 64-bit result in two separate registers. Each line of the SEMANTICS section in figure 3 describes one of the MAC operations. The two MACs are executed in parallel, which is reflected in the semantic description by the following rule:

> *All statements in a single* SEMANTICS *section are evaluated in parallel,*

where a statement is defined as a (chain of) micro-operations concluded by a semicolon. The individual MAC operations are expressed as additions taking the result of a multiplication as first operand, thus building a micro-operation *chain*. Chained expression are evaluated in a sequence, as defined by the following rule:

> *In a chained micro-operation, the innermost micro-operation (operand) is evaluated before its embracing micro-operation (operator).*

The bit-field specification in angle brackets is required to meet the constraint of matching bit widths of the operands of `_ADDI`.

With those two mechanism to combine primitive operations into complex operations, it has been possible to describe most instructions of five architectures under examination, namely Infineon's PP32 network processor [11], STMicroelectronics' ST220 VLIW multimedia core [7], CoWare's LTRISC core [2], ARM's ARM7 core, and Texas Instruments' C54x digital signal processor. One of the few exceptions is the `FFS` instruction of the PP32, which computes the first occurrence of a set bit in a register. For such instructions, either a BEHAVIOR section or a special *intrinsic* micro-operation can be used.

The micro-operation approach presented in this section has two major advantages. Firstly, only a small set of micro-operations ($\approx 25$) is sufficient to describe nearly all instructions of the architectures mentioned above by means of SEMANTICS sections without changing the operation structure of the (already existing) models. Secondly, the chaining mechanism avoids the usage of temporary variables (as allowed in the BEHAVIOR section), which guarantees a tree structure for each semantic statement. Such trees are extremely well-suited for C compiler generation, as most code selectors of today's compilers are based on tree grammars.

# 4. TOOLKIT GENERATION

The introduction of the semantic extension in the previous section builds the foundation for the automatic generation of a C compiler. However, in order to maintain the model consistency, it is mandatory that each operation in an instruction-accurate (IA) model *either* contains a SEMANTICS *or* a BEHAVIOR section. Obviously,

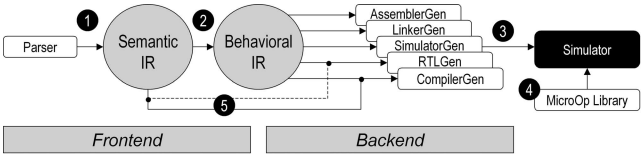


**Figure 4: Simulator Generation Work-flow**

```
SEMANTICS {
  _ADDI[_Z,_C] ( Rs1<4..7>, Rs2<20..23> ) -> Rd<5..8>;
}

BEHAVIOR {
  Rd = ( Rd & (~BITMASK(5,4)) ) |
    ( BIT_EXTRACT
       ( SEM_ADD ( Rs1,  4, 4,    /* op1 3-tuple */
                   Rs2, 20, 4,    /* op2 3-tuple */
                   CARRY_FLAG | ZERO_FLAG
                 ),
         0, 4        /* result is also a 3-tuple */
       ) << 5
    );
}
```

**Figure 5: Semantics to Behavior Translation**

this can only be achieved by using the semantic information for simulator generation, too. Hence, this section shows how an IA simulator can be generated from the semantic information. The integration of semantic information into cycle-accurate (CA) models and its consequences are discussed in section 5. The C compiler generation will be addressed in a separate paper, as it would by far exceed the scope of this work.

## 4.1 IA Simulator

Figure 4 shows the work-flow of the simulator generation from a semantic instruction-set description. As can be seen from the figure, the generator (2) is in fact a SEMANTICS-to-BEHAVIOR translator, which translates each SEMANTICS section into a (set of) BEHAVIOR section(s). The generator can be understood as a *lowering engine* operating on an intermediate representation (IR) of the input description, that means, the abstract semantic IR is transformed into a lower-level behavioral IR on C level. The major advantage of the front-end implementation is the independence from the model compiler's backend, i.e., the already existing, powerful backend generators can be employed without any change in order to generate interpretive, compiled, or just-in-time cache-compiled simulators (JIT-CCS) [12] (3).

A sample translation is shown in figure 5. The SEMANTICS section in the figure is similar to the LISA description of the PP32 network processor, which has an instruction-set designed to operate on bit-fields instead of entire registers only. The following BEHAVIOR shows the result of the translation[1].

Firstly, it is noticeable that each micro-operation is translated into a function call. In the figure, `_ADDI` in the SEMANTICS section results in a call to `SEM_ADD` in the BEHAVIOR section. `SEM_ADD` takes three parameters – two 3-tuple operands `(Rs1,4,4)` and `(Rs2,20,4)`, and a specification of the affected flags resulting from the contents of the square brackets following the micro-operation. The (C) implementation of the function is found in an additional *micro-operation library*, which can be either inlined or eventually linked to the resulting simulator. It is furthermore noticeable that the micro-operation itself also represents a 3-tuple `(SEM_ADD,0,4)`. This is a prerequisite for the implementation of chaining (see section 3.2).

[1]Usually, the textual BEHAVIOR sections are not visible as the translation takes place on IR level as part of the front-end processing

```
1   OPERATION arithmetic {
2     DECLARE {
3       GROUP ArithOp = { ADD | SUB | MUL | ... };
4     SEMANTICS {
5       ArithOp ( Rs1, Rs2 ) -> Rd;}}
6
7   OPERATION ADD {
8     SEMANTICS { _ADDI }}
9
10  OPERATION arithmetic {
11    DECLARE {
12      GROUP ArithOp = { ADD | SUB | MUL | ... };
13    SWITCH (ArithOp) {
14      CASE ADD: { SEMANTICS { _ADDI ( Rs1, Rs2 ) -> Rd; }}
15      CASE SUB: { SEMANTICS { _SUBI ( Rs1, Rs2 ) -> Rd; }}
16      CASE MUL: { SEMANTICS { _MULU ( Rs1, Rs2 ) -> Rd; }}}}
```

**Figure 6: Semantics Hierarchy Transformation**

Finally, the assignment from `(SEM_ADD,0,4)` to `(Rd,5,4)` employs a number of bit-manipulation operations, as bitwise *and* (`&`), *or* (`|`), *negate* (`~`), *shift* (`<<`), and few macros as `BITMASK` and `BIT_EXTRACT`.

### 4.1.1 Complex Translations

In contrast to simple, single-statement translation as shown in figure 5, the complex operations require particular consideration. Besides *chaining* and *parallelism*, which are easily translated by employing nested function calls (chaining), or clock-sensitive register types instead of simple C variables (parallelism), there is a third type that requires more attention.

The SEMANTICS section allows to define non-terminals for micro-operations, such as LISA GROUPs. However, in contrast to the BEHAVIOR section, it is possible to pass parameters to such non-terminal micro-operation calls, as shown in lines 1-5 in figure 6. As LISA 2.0 forbids to provide parameters to operation calls in the BEHAVIOR section, a simple one-to-one mapping from SEMANTICS to BEHAVIOR is not possible.

Fortunately, besides operations and sections, LISA comprises so-called *control-flow statements*, which allow to bind sections to certain conditions. The SWITCH/CASE construct in lines 13-16 is such a control-flow statement. In order to generate the corresponding BEHAVIOR sections from the sematic description in lines 1-8, the operations `arithmetic` and `ADD` are first translated into an equivalent representation still employing SEMANTICS sections. This is achieved by completely removing the SEMANTICS section from operation `ADD`, and rewriting the operation `arithmetic` into the operation shown in lines 10-16. The SWITCH/CASE statement evaluates the GROUP, and thus selects only one of the three SEMANTICS sections depending on the operation chosen. This way, the non-terminal micro-operation call has been removed, and the individual SEMANTICS sections can be easily translated into BEHAVIOR sections (as explained in the previous section).

### 4.1.2 Intrinsics

As already mentioned in section 3.2, it is not possible to describe the semantics of *any* instruction with the formalism presented in section 3. In such cases, either a BEHAVIOR section or a particular *intrinsic* micro-operation can be used. A sample SEMANTICS section and the generated BEHAVIOR section for the previously mentioned `FFS` instruction are shown in figure 7. The quotation marks indicate an intrinsic micro-operation. From the generated BEHAVIOR section[2], it can be seen that a C function or macro `SEM_FFS` must be implemented by the user and added to the simulator library.

[2]For simplicity, the bit-extract and -mask operations have been left out in the figure

```
OPERATION FFS {
  SEMANTICS {
    "_FFS" ( Rs, ZeroOrOne, FromLeftOrRight ) -> Rd; }}
  BEHAVIOR { /* Generated from the above */
    Rd = SEM_FFS(Rd,ZeroOrOne,FromLeftOrRight);
```

**Figure 7: Intrinsic Micro-operation**

```
OPERATION ADD IN pipe.EX {
  DECLARE {
    INSTANCE  writeback;
    GROUP     src1, dst = { reg };
    GROUP     src2 = { reg || imm };}
  SYNTAX { "addc" dst "," src1 "," src2 }
  SEMANTICS { _ADDI[_C](src1, src2) -> dst; }
  BEHAVIOR {
    u32   op1, op2, result, carry;
    u1    c;
    if (forward) {
      op1 = PIPELINE_REGISTER(pipe,EX/WB).result;}
    else {
      op1 = PIPELINE_REGISTER(pipe,DC/EX).op1;}
    result = op1 + op2;
    carry = compute_carry(op1, op2, result);
    PIPELINE_REGISTER(EX/WB).result = result;
    PIPELINE_REGISTER(EX/WB).carry = carry; }
  ACTIVATION { writeback, carry_update }}
```

**Figure 8: LISA CA Operation**

The intrinsic approach has two major advantages: firstly, it provides enough information to the compiler generator in order to automatically generate a so-called *compiler-known function (ckf)*. Secondly, it allows to avoid BEHAVIOR sections, which is important when moving to cycle-accurate level (see section 5).

# 5. CYCLE-ACCURATE (CA) MODELING

In the previous sections, only instruction-accurate (IA) LISA models have been taken into consideration. It has been shown that the use of the SEMANTICS section obsoletes the need for an additional BEHAVIOR section for most operations, as the latter can be automatically generated. Since C compiler as well as instruction-set simulator are generated from the same, unique information, consistency is maintained, and, in addition, the modeling effort is much reduced due the presence of a micro-operation library. However, the question of the role of the SEMANTICS section in cycle-accurate (CA) models arises.

The difference between CA and IA models is made up by the way the behavior of an instruction is described. In an IA model, each instruction is self-contained, that means, it is assumed to have completed execution and written its results before the next instruction is decoded. In contrast, in a CA model instruction execution might be interleaved, for instance by means of an instruction pipeline. Hence, the behavioral description of an instruction is distributed over several operations, which are potentially executed at different clock cycles. Each of such operations gets the result from the previous operation, possibly performs a computation, and passes its results to the next operation. Figure 8 shows the implementation of the `ADD` operation from figure 1 in a CA model.

The first line of the excerpt indicates that the operation only describes `ADD`'s behavior in the execute (`EX`) stage of a pipeline. From the content of the BEHAVIOR section, two observations are noticeable:

1. It would be extremely difficult to extract the instruction semantics from a cycle-accurate behavior description in C.

2. It is impossible to generate a cycle-accurate behavior description from a specification of the instruction semantics (without assuming a fixed microarchitecture).

As a conclusion, in order to generate C compiler and CA simulator from a single model, it is inevitable to introduce a certain amount of redundancy into the description, i.e., to describe behavior *and* semantics at the same time. However, this relatively small overhead pays off enormously when considering that it brings birth to so-called *biabstract models*.

## 5.1 Biabstract Models

Biabstract models describe a single processor architecture on two abstraction levels at the same time, namely instruction- and cycle-accuracy. So far, few ADLs – including LISA 2.0 – are capable of modeling on different abstraction levels, but all of them require a completely separate model for each abstraction level. In practice, that means that two models have to be maintained, which only differ in the behavioral part while the specification of assembly syntax and instruction encoding is exactly the same in both models. The potentially arising inconsistency and maintenance cost eventually impairs the benefits of an ADL-based design approach.

A solution to this problem has already been presented in figure 8. Here, abstract semantics and microarchitecture behavior co-exist, while syntax and coding[3] are only described once. Furthermore, it has been shown in section 4.1 how an IA simulator can be generated from the information contained in the SEMANTICS section. Thus, for IA simulator generation, the existing (CA) behavior description captured in BEHAVIOR and ACTIVATION sections can be ignored, and an IA simulator can be generated from the translated SEMANTICS sections. On the other hand, by ignoring all SEMANTICS sections in the model, CA simulator generation can still take place from the existing BEHAVIOR and ACTIVATION sections.

## 5.2 Limitations

A major limitation of the above approach is that it relies on the fact that *all* operations can be described by means of the SEMANTICS section. Two problems arise:

1. As mentioned in section 3.2, there are instructions whose semantics cannot be expressed by means of micro-operations (e.g., `FFS`).

2. The SEMANTICS formalism is intended to be used for the description of *instructions*, while the functional description of components which are not part of the instruction-set, for instance, a fetch unit, will still employ BEHAVIOR sections.

The first problem is easily solved by the use of the *intrinsic* micro-operation presented in section 4.1.2. For the second problem, there are two possible solutions. The most practical approach is to employ preprocessor defines (`#ifdefs`) to make either one or the other BEHAVIOR section visible to the LISA model compiler. A potentially better solution is the use of parameterizable, functional models of instruction fetch unit and program sequencer. Especially on IA level, such components are usually simple to implement and do not vary much among different architectures. Most existing IA LISA models employ only two operations not describing instructions, namely `main` and `decode`. The feasibility of employing generic, functional models is presented in [10] (although in a different context).

# 6. RESULTS

In order to prove the concepts presented this thesis, the following three items need to be examined with respect to the SEMANTICS section.

[3]The CODING section is omitted in figure 8.

1. The *feasibility* to describe contemporary instruction-sets by means of the SEMANTICS section,
2. the associated *design effort*, and
3. the impact on *simulation performance* for IA models.

Overall, five architectures have been taken into account, namely ARM's ARM7 core, CoWare's LTRISC core, STMicroelectronics' ST220 VLIW multimedia processor [7], Infineon's PP32 network processing unit [11], and Texas Instruments' C54x digital signal processor. The LTRISC architecture is a very small RISC core ($\approx$ 14 instructions), which is provided with CoWare's Processor Designer [2]. The PP32 is an evolution of [11] and comprises a bit-level RISC instruction-set, that means, most instructions can operate on bit-fields rather than on byte- or word-aligned data only.

## 6.1 Feasibility

For comparison, existing LISA 2.0 models on instruction- and cycle-accurate level have been enhanced with SEMANTICS sections for compiler and simulator generation. Although the semantic extension is not primarily targeted for late incorporation into already existing models, this approach proved that the nature of the SEMANTICS section does not impose any particular modeling style – which is crucial with respect to the flexibility paradigm of LISA 2.0. All models taken into consideration have been enhanced without any (or only marginal) changes to the existing parts. Table 1 summarizes the results.

| | ARM7 | LTRISC32 | ST220 | PP32 | C54x |
|---|---|---|---|---|---|
| Abstraction level | IA | IA | CA | CA | CA |
| ISA | RISC | RISC | RISC[1] | RISC[2] | CISC |
| Data path | 32bit | 32bit | 32bit | 32bit | 16bit |
| No. operations | 108 | 39 | 121 | 151 | 408 |
| Design effort $\Delta$[3] | $4d$ | $2d$ | $6d$ | $10d$ | $15d$ |
| Avg. sim. perf.$\Delta$[4] | $-8\%$ | $-5\%$ | n/a | n/a | n/a |

[1] 4-issue VLIW [2] bit-field instructions [3] design effort for adding semantics to the existing models (in man-days) [4] relative to the performance of IA simulator generated from behavioral description

**Table 1: Model Statistics and Results**

## 6.2 Design Effort

From table 1, it can be seen that the design effort for adding SEMANTICS sections scales with the number of LISA operations and the complexity of the instructions (RISC vs. CISC, for instance). Generally, the effort for describing the instruction semantics is much less than for a behavioral description in C (on IA level), since the existing micro-operation library already defines, for instance, a 17x17 multiplication, while a behavioral description usually requires a significant amount of C code (which still has to be validated). Especially for the PP32, the explicit bit-field specification (compared to a typical and/or/shift description in C) reduces the design time enormously.

## 6.3 Simulation Performance

For performance considerations, only the IA models have been taken into account, as the simulation of abstract semantics and micro-architecture behavior (in a CA model) are not comparable. Hence, IA simulators have been generated from the SEMANTICS and the existing BEHAVIOR sections of the ARM7 and the LTRISC models, and the performance of both simulators has been compared. All benchmarked simulators employ the JIT-CCS simulation technique.

From table 1, it can be observed that the performance of the simulators generated from the SEMANTICS description is slightly worse than for the simulators generated from the existing BEHAVIOR description. This is due to the generic implementation of the micro-operation behavior in the library, which contains a non-negligible overhead for native bit-widths, i.e., 8, 16, and 32-bit data. Although the overall design efficiency is much increased due to the enormous reduction in design effort, there is no doubt that the micro-operation library can be optimized in order to meet the performance of the BEHAVIOR-generated simulator.

# 7. CONCLUSION

In this paper, we presented a novel approach for solving the consistency problem that arises as soon as an architecture description language (ADL) serves multiple purposes. In particular, the automatic generation of C compiler *and* instruction-set simulator from a single model description either leads to a loss in modeling flexibility, or introduces a huge potential for inconsistencies. Our approach avoids both by presenting a mixed-level behavior/semantics modeling methodology supported by the underlying design tools.

Besides solving the consistency problem without any loss in flexibility on instruction-accurate (IA) level, the presented solution allows for the existence of so-called biabstract models, which incorporate instruction- and cycle-accurate model in a single architecture description.

Together with our previous work on automatic synthesis of instruction encoding [13] and generation of RTL descriptions [16], the proposed semantic extension of the LISA ADL allows for a very high design efficiency on abstract level, while maintaining consistency by means of a single model throughout the entire design process.

Our further research activities are in the area of C compiler retargeting, RTL code generation, and methodologies and tools for seamless IA-to-CA model refinement.

# 8. REFERENCES


[1] Axys Design Automation. *http://www.axysdesign.com*.
[2] Coware, Inc. *http://www.coware.com*.
[3] Tensilica. *http://www.tensilica.com*.
[4] A. Halambi, P. Grun et al. EXPRESSION: A Language for Architecture Exploration through Compiler/Simulator Retargetability. In *Proc. of the DATE conference*, Mar. 1999.
[5] A. Hoffmann, T. Kogel et al. A Novel Methodology for the Design of Application-Specific Instruction-Set Processors (ASIP) using a Machine Description Language. *IEEE Transactions on Computer-Aided Design*, Nov. 2001.
[6] G. Hadjiyiannis, S. Hanono et al. ISDL: An Instruction-Set Description Language for Retargetability. In *Proc. of the DAC*, Jun. 1997.
[7] F. Homewood and P. Faraboschi. ST200: A VLIW Architecture for Media-Oriented Applications. *Microprocessor Forum*, Oct. 2000.
[8] M. Hartoog, J.A. Rowson et al. Generation of Software Tools from Processor Descriptions for Hardware/Software Codesign. In *Proc. of the DAC*, Jun. 1997.
[9] M. Itoh, M. Imai et al. PEAS-III: An ASIP Design Environment. In *Proc. of the ICCD*, Sep. 2000.
[10] P. Mishra, N. Dutt, and A. Nicolau. Functional Abstraction Driven Design Space Exploration of Heterogeneous Programmable Architectures. In *Proc. of the ISSS*, Oct. 2001.
[11] X. Nie and L. Gazsi. A New Network Processor Architecture for High-Speed Communications. In *Proc. of the SIPS*, 1999.
[12] A. Nohl, G. Braun, et al. A Universal Technique for Fast and Flexible Instruction-Set Architecture Simulation. In *Proc. of the DAC*, Jun. 2002.
[13] A. Nohl and V. Greive et al. Instruction Encoding Synthesis for Architecture Exploration using Hierarchical Processor Models. In *Proc. of the DAC*, Jun. 2003.
[14] R. Gonzales. Xtensa: A Configurable and Extensible Processor. In *Proc. of the IEEE Micro*, Mar. 2000.
[15] S. Bashford, R. Leupers et al. The MIMOLA Language, Version 4.1. Reference Manual, Department of Computer Science, University of Dortmund, 1994.
[16] O. Schliebusch, A. Hoffmann, et al. Architecture Implementation using the Machine Description Language LISA. In *Proc. of the ASPDAC*, 2002.
[17] S. Weber, K. Keutzer et al. Multi-View Operation-Level Design – Supporting the Design of Irregular ASIPs. Technical Report UCB/ERL M03/12, UC Berkeley, Apr. 2003.