Journal of VLSI Signal Processing 43, 235–246, 2006

DOI: 10.1007/s11265-006-7273-3

Modeling Instruction Semantics in ADL Processor Descriptions for C Compiler Retargeting

JIANJIANG CENG, WEIHUA SHENG, MANUEL HOHENAUER,
RAINER LEUPERS, GERD ASCHEID AND HEINRICH MEYR
Institute for Integrated Signal Processing Systems, Aachen
University of Technology (RWTH), Sommerfeldstr. 24 WSH Aachen, Germany

GUNNAR BRAUN
CoWare, Inc. Dennewartstrasse 25-27, Aachen Germany

Abstract. Today's Application Specific Instruction-set Processor (ASIP) design methodology often employs centralized Architecture Description Language (ADL) processor models, from which software tools, such as C compiler, assembler, linker, and instruction-set simulator, can be automatically generated. Among these tools, the C compiler is becoming more and more important. However, the generation of C compilers requires high-level architecture information rather than low-level details needed by simulator generation. This makes it particularly difficult to include different aspects of the target architecture into one single model, and meanwhile keeping consistency.

This paper presents a modeling style, which is able to capture high- and low-level architectural information at the same time and make it possible to drive both the C compiler and the simulator generation without sacrificing the modeling flexibility. The proposed approach has been successfully applied to model a number of contemporary, real-world processor architectures.

Keywords: architecture description language, processor model, C compiler retargeting, embedded processor design, electronic system level

1. Introduction

Today, an increasing number of *A*pplication *S*pecific *I*nstruction-set *P*rocessors (ASIPs) are employed as building blocks in embedded *S*ystem-*o*n-*C*hip (SoC) designs. Since the architectures and the instruction-sets of ASIPs are highly optimized for specific applications or application domains like image processing or network management, they are very efficient in executing such target applications. On the other hand, compared to a hardwired *A*pplication-*S*pecific *I*ntegrated *C*ircuit (ASIC), ASIPs' programmable architectures are much more flexible. Therefore, ASIPs are getting more and more attractive due to their balance between performance and flexibility.

To design an ASIP, one of the most important steps is *architecture exploration*. In the architecture exploration phase, designers need to find out the optimum architecture for the target application. First, the application is profiled to determine critical portions that require dedicated hardware support by using application specific instructions. This is also often referred to as hardware-software partitioning. After that, the instruction-set is defined based on the result of partitioning and profiling. Then, the micro-architecture of the processor is designed to implement the

instruction-set. The process is partially or fully repeated when different design space parameters are exploited until the design requirement is met.

Such iterative architecture exploration process demands that each time when the architecture is tuned, software tools such as compiler, assembler, linker and simulator should be updated and available as soon as possible so that the tuning result can be examined to find out the potential improvement for the next iteration. It is obvious that manually retargeting these software tools is a time-consuming, tedious, and error-prone work. For this reason, *A*rchitecture *D*escription *L*anguages (ADLs) are developed to aid the design of ASIPs. Different from hardware description languages like VHDL or Verilog which are mainly designed to model and simulate hardware, ADLs characterize processor architecture from different high-level aspects, e.g. instruction behavior, assembly syntax, binary coding, and so on. Base on ADL processor models, required software tools are automatically generated which significantly improves design efficiency.

However, the generation of different software tools requires heterogeneous information on the architecture. It is challenging to make one description to fit all requirements. This is especially true for the C compiler generation and the instruction-set simulator generation. Instruction-set simulators need the knowledge in detail about how the architecture executes an instruction, e.g. internal data manipulations, side effects calculation, cycle-accurate pipeline activities, etc. In contrast, C compilers view these instructions from a much higher, semantics-oriented abstraction level. What they need to know is the purpose of instructions rather than their execution on the micro-architecture level. Due to these orthogonal requirements, it is difficult to include information for both software tools into one single model.

In this paper, we describe an extension of LISA 2.0 [1], a widespread industrial ADL, towards the modeling of instruction semantics for C compiler retargeting. The design of this extension aims at enabling the description of high-level instruction behavior with minimum design effort. With this extension, embedded processor designers can generate a C compiler conveniently from a LISA processor model. Moreover, our approach is not only useful for the C compiler generation. In [2], we described the technique of simulator generation based on the work described in this paper and the related model consistency issues. Combined with the C compiler generation capability, our approach fulfills the demand for consistent-tool generation from a single ADL model, meanwhile not sacrificing flexibility. As the tool generation exceeds the scope of this paper, here we will focus on the new language constructs.

The rest of this paper is organized as follows: Section 2 shortly discusses the approaches of related works. An overview of the LISA ASIP design methodology is given in Section 3. Section 4 reviews several important principles in the LISA language. Section 5 describes in detail the design criteria and the extension of the language, which is the core of this paper. Section 6 presents the modeling results of several real-world processors. Finally, Section 7 concludes the whole paper.

2. Related Work

Within the last decade, a variety of ADLs has been developed to support ASIP design. However, not all of them support the generation of compilers. One important architecture-specific component in compiler is the code selector. It performs the task of translating the intermediate representation (IR) of the applications into assembly instructions. To generate a code-selector for a processor architecture, the knowledge of instruction *semantics*, i.e. what instructions do, is needed. Because most of the ADLs known today were originally designed to automate the generation of a specific processor design tool, e.g. simulator, and later extended to other tools, different modeling styles were developed to support the generation of code-selectors. Based on the nature of the information provided, these ADLs can be classified into three categories: structural, behavioral and mixed.

The MIMOLA [3] language belongs to the structural ADL category. Its modeling style is similar to that of the VHDL hardware description language. The instruction set information is extracted from the *R*egister-*T*ransfer *L*evel (RTL) module netlist for use in code selector generation [4]. The software programming model of MIMOLA is an extension of PASCAL.

ISDL [5] is classified as behavioral ADL. It provides the means for hierarchical specification of instruction sets. During the code selector generation, the correlation between the target processor operations and the compiler basic operations comes from the behavior description of each instruction [6]. Because ISDL cannot model the structural details for pipelining, cycle accurate simulator generation is not possible.

nML [7] and EXPRESSION [8] are mixed ADLs. nML was designed from its beginning to provide a formalism for instruction set modeling. The instruction set of the processor is described in a hierarchical style. The roots of the hierarchy are instructions, and the intermediate elements are partial instructions. Both instructions and partial instructions have *action* sections, which describe the behavior of instructions. Although nML is claimed as a behavioral/structural ADL, it lacks the capability of describing detailed micro-architecture structures, which limits the capability of simulator generation. The code-selection from EXPRESSION processor models relies on a so-called "Generic Machine" [9], which has a RISC instruction set architecture similar to that of the MIPS. *Operation-mapping* sections in EXPRESSION processor models are exclusively used to define the mapping between target processor instructions and one or more generic machine instructions on assembly level. The EXPRESS compiler first translates the input application to generic instructions, which are then replaced by target instructions.

The LISA 2.0 language [10] belongs to mixed behavioral/structural ADLs. The language allows the description of the micro-architecture behavior with arbitrary C/C++ code, which achieves high flexibility of modeling and very fast simulation speed for a broad range of contemporary RISC, VLIW, NPU, DSP, and ASIP architectures.

3. System Overview

The work described in this paper is based on the *LISATek Processor Designer*, an embedded processor design platform available from CoWare, Inc [11]. The core of the platform is the LISA 2.0 ADL. It supports the automatic generation of efficient ASIP development tools such as instruction-set simulator, debugger/profiler, assembler, and linker. Furthermore, the platform also provides the capability of generating VHDL, SystemC and Verilog hardware description language models for hardware synthesis. Because the problem of model inconsistency and the need to use various special-purpose description languages are avoided by using one single ADL model, this ADL-driven ASIP design approach can achieve very high design efficiency (Fig. 1).

In [12], we have described our overall LISA 2.0 based C compiler generation framework. Using ACE's CoSy system [13] as a backbone, it enables semi-automatic retargeting of C compilers from LISA processor models. A restriction of this earlier version is the need for manual interaction to specify the code-selector description in the compiler backend. The extension described in the following sections makes this manual interaction largely superfluous, and thus, permits to generate the code selector from a LISA model.

4. Lisa Language

As the core of the LISATek processor design platform, the LISA language provides processor designers the capability of describing an embedded processor with a high level language. Basically, a LISA processor model consists of two parts: resource declarations and operations. Resource declarations define the processor resources like registers, buses, memories, pipelines, etc. The major part of a model consists of operations. `OPERATION` is the basic element of the instruction set description. Depending on the abstraction level of the model, namely *I*nstruction *A*ccurate (IA) or *C*ycle *A*ccurate (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 has one or several sections describing the attributes of the operation according to its purpose. For example, if an operation contains a part of an instruction, it usually contains a `CODING` section that specifies a part of the instruction's binary coding.

The operations in a LISA processor model are organized hierarchically. The binary coding, assembly syntax, behavior and timing information are distributed over a number of operations inside the model. One of

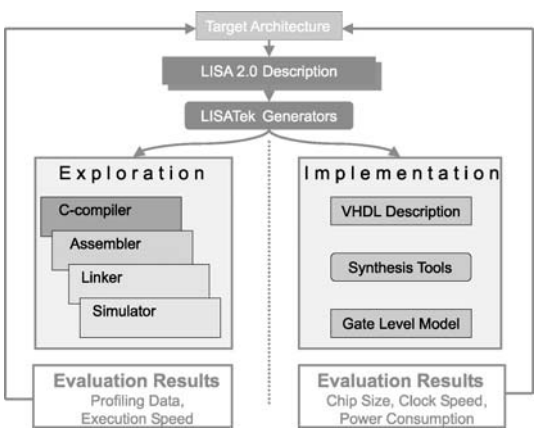


Figure 1. LISA 2.0 based ASIP design flow.

```
OPERATION arithm IN pipe.ID{
    DECLARE{
        GROUP opcode = { ADD || SUB || ... };
        GROUP Rs1, Rs2, Rd = { reg_32 };
    }
    ...
    BEHAVIOR{
        PIPELINE_REGISTER(pipe, ID/EX).src1 = GPR[Rs1];
        PIPELINE_REGISTER(pipe, ID/EX).src2 = GPR[Rs2];
    }
}
OPERATION ADD IN pipe.EX{
    ...
    BEHAVIOR{
        int op1 = PIPELINE_REGISTER(pipe, ID/EX).src1;
        int op2 = PIPELINE_REGISTER(pipe, ID/EX).src2;
        PIPELINE_REGISTER(pipe, EX/WB).dst = op1+op2;
    }
}
OPERATION SUB IN pipe.EX{
    ...
    BEHAVIOR{
        int op1 = PIPELINE_REGISTER(pipe, ID/EX).src1;
        int op2 = PIPELINE_REGISTER(pipe, ID/EX).src2;
        PIPELINE_REGISTER(pipe, EX/WB).dst = op1-op2;
    }
}
```

Figure 2. LISA operation examples.

the advantages of the hierarchical structure of operations is that the commonality of instructions can be easily exploited. Figure 2 shows three example operations `arithm`, `ADD` and `SUB`. Because `ADD` and `SUB` use the same type of operands, their operand fetching behavior is modelled in the operation `arithm` which belongs to the higher hierarchy. Their relationship is realized through the definition of GROUPs, whose members correspond to a list of alternative operations. In the example, the information about coding, timing and syntax is omitted for simplicity.

In BEHAVIOR sections, the instruction behavior is described on the micro-architecture level, i.e. any change in the processor state should be correctly modelled here. Since there are an unlimited number of possible processor behaviors, the LISA language allows the use of arbitrary C/C++ code inside BEHAVIOR sections to describe them. This makes the LISA language very flexible in modeling processor behavior. Meanwhile, since the code can be directly compiled into executable code without language translation overhead, simulators generated from LISA processor models can achieve very high performance. On the other hand, the use of C/C++ description makes it difficult to extract compiler semantics from the behavior of instructions, which are typically even distributed over different instruction pipeline stages or operations like the example in figure 2. The compiler requires a high-level model of the target machine in contrast to the detailed LISA model that in particular must capture cycle and bit-true behavior of machine operations. An example for this semantic gap is given in Figure 3. The `LDI` operation shifts the 16-bit immediate value `Imm16` to the left by `Off` bits and moves the result and the index of the destination register `Rd` into a pipeline register. The compiler generator would have to find out that this operation can be used for two purposes: it can be used to move 16-bit unsigned immediate values into registers if `Off`=0 or it can be used to move 32-bit immediate values into registers; the immediate is first shifted to the right by 16-bit, then the `LDI` operation is used with `Off`=16, then the lower 16 bit of the immediate are added to the result.

```
OPERATION LDI IN pipe.ID { ...
  SYNTAX {
    "LDI" Rd Off Imm16
  }

  BEHAVIOR {
    unsigned result = (unsigned short) Imm16 << Off;
    PIPELINE_REGISTER(pipe,EX/WB).result = result;
    PIPELINE_REGISTER(pipe,EX/WB).destination = Rd;
  }
  ...
}
```

Figure 3. C++ based behavior description of LISA models.

The analysis is significantly aggravated by the multitude of possible C++ code behavior descriptions that have the same semantic meaning. In the example of figure 3 a cast to `unsigned short` is used to extract the lowest 16 bit of the immediate operand. It is equally possible to do this with a bit-mask (`Imm16 & 0xFFFF`) or with subsequent shift operations (`((unsigned int) Imm16) << 16 >> 16`). It is also extremely difficult to support the semantic analysis of C++ code containing pointer arithmetic or object oriented concepts like virtual functions, templates, etc. In pipelined architectures, things will become even more complicate. Their instruction behavior description is normally distributed over different pipeline stages. It is quite often that features like register bypassing need significantly more C code than what is needed for calculating the result of the instruction. Taking all these into account, it is nearly impossible to extract instruction semantics from BEHAVIOR sections. For this reason we

introduced SEMANTICS section into the LISA 2.0 language.

5. Semantics Section

Before the SEMANTICS section was introduced into the LISA language, the language itself has been evolving for several years. A lot of processor models have been developed with it and used by different users. Adding a new section to the language would possibly cause some incompatibility to the old models. Keeping this incompatibility to a minimum was a goal from the very beginning of the design of the SEMANTICS section. Moreover, the modelling flexibility should not be sacrificed so that a broad range of instruction set architectures can be described. Additionally, since there are a lot of existing users of the LISA language, the syntax of the new section should be made simple and easy to learn so that extending an existing LISA processor model with SEMANTICS sections should not be difficult for old LISA users. Furthermore, although the major motivation of the SEMANTICS section is to enable the automatic compiler generation, the use of the section itself should not require too much compiler knowledge from user. Otherwise, the user might prefer manually retarget a compiler himself rather than using the new language construct.

Of course, since the section is to provide the information that cannot be analyzed from C/C++ behavior code, the ambiguity of C code should be avoided in any case. For most operations, there should be only a single, concise way to define the semantics. Inevitably, the use of SEMANTICS sections introduces certain redundancy, because the instruction behavior is now described not only in BEHAVIOR sections but also in SEMANTICS sections. However, with simple and concise constructs, this redundancy can be reduced to minimum.

5.1. Register and Immediate Operands

To describe the behavior of an instruction in SEMANTICS sections, the first element we need to take care of is the operand used by the instruction. Most of the time, such operands are registers or immediate values which are explicitly specified in the assembly syntax. In SEMANTICS sections, such operands are treated as *modes*. *Modes* do not perform any computation or data assignment. They are used to describe access to the processor's resources or immediate operands. In the SEMANTICS section, there are two modes available, the *register* mode and the *immediate* mode .

```
OPERATION reg_32{
    DECLARE{
        LABEL index;
    }
    SYNTAX { "R"~index }
    CODING{ index = 0bx[4] }
    SEMANTICS{
        _REGI(GPR[index])<0..31>;
    }
    ...
}
```

Figure 4. Register mode example.

The *register* mode defines (allocatable) register resources and their width. The names of registers are derived from the SYNTAX section. Figure 4 gives a *register* mode example. The `reg_32` LISA operation works as a register wrapper in a LISA model. Accordingly, its SEMANTICS section contains a *register* mode specification. `_REGI` is the keyword of the *register* mode. In the brackets, `GPR` is the name of the processor resource array which should be used as register file. `index` is a LISA LABEL which represents a set of bits in the complete instruction binary coding. Here, its value is used to index the resource array, which in turn indexes a register. From the expression `0bx[4]` in the CODING section, we know that `index` is 4 bits wide. Therefore, there are 16 register available by this register wrapper operation. The bit-width of the registers are given by the notation `<0..31>` which means this is a 32 bit register file with LSB(Least Significant Bit) at bit 0 and MSB(Most Significant Bit) at bit 31.

Except for the explicit register usage through the *register* mode, it is also possible that an instruction reads or modifies a register which does not appear in the syntax at all. For example, a call instruction might implicitly save the return address in a predefined register so that later an indirect jump can be performed to return from the called function or procedure. Then, neither the assembly syntax nor the binary coding of this instruction contains the information about the use of this register. It is implicitly hard coded in the behavior of the instruction. For this case, the required register resource can be directly used in SEMANTICS section without creating a wrapper operation with *register* mode. E.g. if the the resource element maps to the implicitly read/written register is `GPR[15]`, then we can simply directly use `GPR[15]` in its SEMANTICS section.

```
OPERATION imm8{
    DECLARE{
        LABEL value;
    }
    CODING{ value = 0bx[8] }
    SEMANTICS{
        _IMMI(value);
    }
    ...
}
```

Figure 5. Immediate mode example.

Simpler than the *register* mode, the *immediate* mode defines an immediate value usually carried in the instruction binary coding. Its bit-width can be therefore derived from the CODING section. In Figure 5, the operation `imm8` wraps an 8-bit instruction coding to be used as immediate value. The keyword `_IMMI` in its SEMANTICS section starts an *immediate* mode statement. Its operand `value` specifies the LISA LABEL from which the immediate value can be derived. The range of the value can be deduced from its bit-width which is 8 here. Depends on how the bits are treated, the range can be either 0 to 255 as unsigned integer or −128 to 127 as signed integer(two's complement).

For the generation of the compiler, especially the code selector description, *modes* directly lead to the generation of an important component, *non-terminal*(details are described in [14]). For a LISA processor model, wrapper operations containing *register* and *immediate* modes normally serve as input operands for other operations.

5.2. *Micro-operations*

The MIMOLA ADL [3] employs a set of so-called *micro-operations* to describe a processors 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 micro-architectural behavior, it has proven feasible and complete for the specification of instruction semantics. As the description of the micro-architecture is left to the existing BEHAVIOR section in our approach, we adapted the idea for the definition of SEMANTICS sections.

After the examination of the instruction set architectures of a number of modern processors, we see that the high-level behavior of most of the instructions used in these processors are normally either arithmetic calculations based on several operands or branches. The calculations carried out by the instructions can be further decomposed into one or several primitive operations. These primitive operations are modelled with micro-operations.

Generally, a micro-operation contains four parts, micro-operator, side-effects declaration, operands and bit-field specification. The operands of the micro-operators can be terminal elements, e.g., integer constants, `OPERATION`s, or other micro-operations. The constraint of the `OPERATION`s used as operands is that they must contain a SEMANTICS section. Side-effects in SEMANTICS sections are all predefined to avoid ambiguity. The behavior of four commonly used flags are provided, carry, zero, negative and overflow flags. If an instruction has any of the predefined side-effects, they can be declared by putting the corresponding shortcut after the micro-operator. Of course, these four flags cannot cover all possible side-effects which might appear in a processor architecture. In case an instruction has side-effects not fitting to any of the four predefined flags, there are two possible ways to go depending on the side-effects should be compiler aware or not. If the side-effect will change compiler known resources like post increasing an operand or address register, then such behavior should be modelled explicitly with additional SEMANTICS statements(see Section 5.3 for how to model complex operations). On the other hand, if the affected processor resource is a status register which is invisible to the compiler, then modeling it in SEMANTICS sections is no more appropriate. The BEHAVIOR section is more suitable in this case. The bit-field specification provides the bit-width and offset information of the micro-operations. It is compulsory for those micro-operations whose output bit-width cannot be deduced from their operands, such as sign/zero extension. Besides, allowing the use of micro-operations as the operands of other micro-operations is very useful in modeling complex operations.

Figure 6 shows the `ADD` operation of Fig. 2 with SEMANTICS section added. One statement is used to describe the semantics of an `ADD` instruction. The `_ADD` symbol is a micro-operator representing an integer addition. The following `_C` specifies that the carry flag is affected by the operation. `Rs1` and `Rs2` in the

```
OPERATION ADD IN pipe.EX{
    DECLARE{
        GROUP Rs1, Rs2, Rd = { reg_32 };
    }
    ...
    SEMANTICS{
        _ADD|_C|(Rs1, Rs2)<0,32>->Rd;
    }
}
```

Figure 6. An example of SEMANTICS section.

```
OPERATION SWAP{
    DECLARE{
        GROUP Rs = { reg_32 };
    }
    ...
    SEMANTICS{
        Rs<0,16>->Rs<16,16>;
        Rs<16,16>->Rs<0,16>;
    }
}
```

Figure 8. Multiple statements.

parentheses are the operands of the addition. They are GROUPs with one member, the `reg_32` operation. That means, the semantics of the operands is defined in the SEMANTICS section of `reg_32`. The `<0,32>` after the brackets explicitly specifies that the result of the addition is 32-bit wide and bit 0 is the first bit. Assignments in SEMANTICS sections are specified with `->` with the source on its left hand side and the destination on the right. Compared with the BEHAVIOR sections shown in Figure 2, the description in SEMANTICS section is much simpler.

5.3. Modeling Complex Operations

Many DSP processor architectures have instructions doing combined computation, e.g. multiply and accumulate. Such behavior is captured in SEMANTICS sections with *chaining*, i.e. using a micro-operation as the operand of another micro-operation. A simple example of a `MAC` operation can be found in Figure 7. `_MULUU` is the micro-operator that denotes the unsigned multiplication. Its result is used as one of the operands of the `_ADD`, which forms a micro-operation chain. The chaining mechanism helps to describe some complex operations without introducing temporary variables, and keeps the statements in a tree-like structure. Such a structure is suitable for code selectors, because most of the code selection algorithms use the tree-pattern matching technique [15, 16].

With *chaining*, most of the RISC instructions can be modelled with one statement, but obviously this is not enough for those instructions transferring data to multiple destinations. They are modelled with multiple statements in SEMANTICS sections. If a SEMANTICS section contains multiple statements, they are assumed to execute in parallel. That means, a preceding statement's result cannot be used as the input of the following statement. To illustrate this, an example is given in Figure 8. The `SWAP` operation swaps the content of a register by exchanging the higher and lower 16 bits. Because the execution is in parallel, the data in the register are exchanged safely without considering sequential overriding.

Another kind of important behaviors used in modern processors is conditional execution, i.e., an instruction is executed according to certain conditions. In order to model such instructions, `IF-ELSE` statements can be used. A total of 10 comparison operators are defined in SEMANTICS sections to model all kinds of conditions. Besides, logical `AND` (&&) and `OR` (||) operators can be used to combine different conditions. In Figure 9, the `_EQ` operator checks whether its two operands are equal or not. According to the result, the `IF` statement will execute the behavior specified in the braces.

```
OPERATION MAC{
    DECLARE{
        GROUP Rs1, Rs2, Rd = { reg_32 };
    }
    ...
    SEMANTICS{
        _ADD(_MULUU(Rs1, Rs2)<0,32>, Rd) -> Rd;
    }
}
```

Figure 7. Micro-operation chaining.

```
OPERATION CADD{
    DECLARE{
        GROUP Rs1, Rs2, Rd = { reg_32 };
    }
    ...
    SEMANTICS{
        IF(_EQ(_CF,1)){ _ADD(Rs1, Rs2)->Rd; }
    }
}
```

Figure 9. IF-ELSE statement.

```
OPERATION DCT2d{
    DECLARE{
        GROUP Rs = { reg_32 };
    }
    ...
    SEMANTICS{
        "_DCT2d"(Rs);
    }
}
```

Figure 10. Intrinsic micro-operation.

ASIPs often employ special instructions to accelerate target applications. These instructions perform specific operations frequently used in application source codes. Traditionally such special instructions are accessed by compiler known functions, a.k.a. *intrinsics*. With intrinsics, time consuming parts of the code can be executed fast by specific hardware support. However, the behaviors of special instructions are very heterogeneous, their C behavior code can vary from a few lines to several hundreds. Normally, such complex behaviors are impossible to describe with micro-operations. Even if they would be modelled with primitive micro-operations, the C compiler can hardly make use of them. For this reason, the semantics of special instructions is viewed as special user defined micro-operations in SEMANTICS sections. We also call these user defined micro-operations *intrinsics*. Figure 10 gives an example of intrinsics micro-operation. The operation `_DCT2d` operation is supposed to perform a 2 dimensional discrete cosine transformation. It is normally impossible to map C code to such instructions with normal tree-pattern matching techniques [16]. Intrinsics is actually the only feasible way here. Then, to describe the semantics of this instruction, it is no more important for the C compiler to know what calculation will be done, the important information is to which compiler known function this instruction belongs and how many parameters should be expected. In the SEMANTICS section of the example, the string `_DCT2d` enclosed in double quotes is the name of the user-defined micro-operation or intrinsics. `Rs` is its operand. With the given information, the compiler then knows that a function with the prototype "`void _DCT2d(void*);`" (the parameter can also be other data types) should map to this instruction. For the compiler generation, this is enough. However, since intrinsic micro-operations do not contain any concrete instruction behavior description, for simulator generation, their actual behavior must be specified explicitly in the BEHAVIOR sections with C/C++ code.

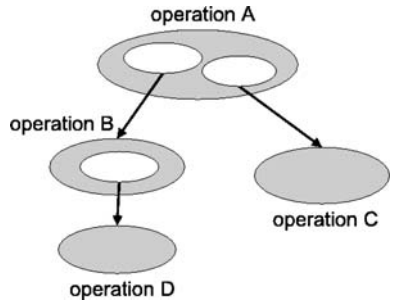


Figure 11. Operation hierarchy 1.

5.4. Semantics Hierarchy

The hierarchical definition of instruction semantics relies on the LISA operation hierarchy. In section 4 we have briefly illustrated the operation hierarchy. Here, some more details are discussed before going into hierarchical semantics.

The LISA operation hierarchy is built by referencing other operations. The specification of non-terminal operations on a higher level of the hierarchy is completed by the referenced terminal operations on the lower levels as shown in Figure 11. Here, the specification of the operation `A` is non-terminal and includes two references to the operations `B` and `C`. The specification of the operations `C` and `D` is terminal.

There are two ways to define the hierarchy relation between operations, using INSTANCEs or GROUPs. An instance produces a single branch originating from the current operation and pointing to the inferior operation. Regarding the operation `A` in the tree of the previous example, the operations `B` and `C` are inferior operations that must be declared as an instance. The idea is to divide the whole description into logical parts of the programmable architecture similar to function calls in the programming language `C`. Then two or more operations may reuse the same operation.

GROUPs are used to reference one selected operation out of a given set of sub-operations which are used in the same context. Figure 12 shows how groups structure references to alternative sets of operations. Operation `A` includes one instance to operation `B` and another reference which is based on a group. This group defines three options to describe a certain aspect of operation `A`. These members of the group are the operations `C`, `D`, and `E` which represent the possible options.

With operation hierarchy, the complete processor behavior with regards to one instruction is actually

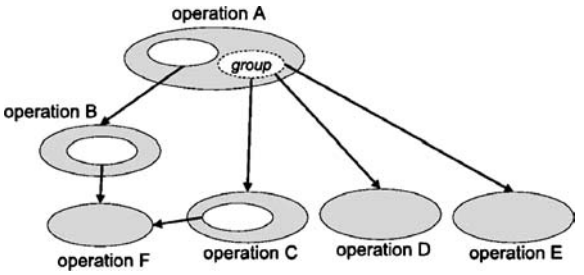


Figure 12. Operation hierarchy 2.

distributed in different operations so that the common behavior does not need to be described repeatedly. The same idea is applied to the SEMANTICS sections. Figure 13 shows the hierarchical SEMANTICS sections added for the three operations in Figure 2. In the `arithm` operation, the GROUP `opcode` is used as micro-operator instead of predefined ones, which means that the concrete micro-operators can be found in the SEMANTICS sections of the GROUP members. Accordingly, the SEMANTICS sections of the `ADD` and `SUB` operation contain simply a micro-operator. The similarity of the `ADD` and `SUB` operations' semantics is well exploited here to simplify the description.

A SEMANTICS section can return not only a micro-operator but also a complete micro-operation expression. In Figure 14, the SEMANTICS sections of the `SHL` and `SHR` operations do not contain a complete statement with assignment but micro-operators with

```
OPERATION arithm IN pipe.ID{
    DECLARE{
        GROUP Rs1, Rs2, Rd = { reg_32 };
        GROUP opcode = { ADD || SUB || ... };
    }
    ...
    SEMANTICS{ opcode|_C|(Rs1, Rs2)->Rd; }
}

OPERATION ADD IN pipe.EX{
    ...
    SEMANTICS{ _ADD; }
}

OPERATION SUB IN pipe.EX{
    ...
    SEMANTICS{ _SUB; }
}
```

Figure 13. Hierarchical operators.

```
OPERATION ADD IN pipe.EX{
    DECLARE{
        GROUP Rs1, Rd = { reg32 };
        GROUP opd = { SHL || SHR };
    }
    ...
    SEMANTICS{ _ADD(Rs1, opd)->Rd; }
}

OPERATION SHL IN pipe.EX{
    DECLARE{
        GROUP Rs2 = { reg_32 };
        GROUP imm = { imm8 };
    }
    ...
    SEMANTICS{ _LSL(Rs2, imm); }
}

OPERATION SHR IN pipe.EX{
    DECLARE{
        GROUP Rs2 = { reg_32 };
        GROUP imm = { imm8 };
    }
    ...
    SEMANTICS{ _LSR(Rs2, imm); }
}
```

Figure 14. Hierarchical operands.

operands (`_LSL` and `_LSR` are logical left and right shift micro-operators). As a result, the semantics of these two operations are not self-contained, because the data sink is missing. The use of these two operations is actually doing operand pre-processing for the `ADD` operation, which can be seen in its SEMANTICS section. The `opd` GROUP, which contains the previous two operations, is used as one of the operands of the `_ADD` micro-operation. Thereby, depending on the binary encoding of the instruction, one of the operand registers will be left or right shifted before addition.

In short, the formalism in SEMANTICS sections is very flexible and well integrated into the LISA 2.0 ADL. If the commonalities of instructions are fully exploited, their instruction semantics can be described with only a few lines of code, which reduces the user effort for writing instruction semantics.

5.5. Relation Between SEMANTICS and BEHAVOR

Though SEMANTICS and BEHAVIOR sections are similar in terms of describing the behavioral model of the processor, they are different in several ways: in BEHAVIOR sections, C/C++ code can be used

without limitation. However, in SEMANTICS sections, only a limited set of micro-operators (31 in total) are allowed, and their usages are fully formalized. Moreover, the operands in BEHAVIOR sections can be nearly all processor resources and arbitrary variables. On the contrary in SEMANTICS sections, only compiler related resources can be accessed directly, e.g. registers, memories, etc. All others are not allowed. SEMANTICS sections are intended to describe those compiler interested behaviors. On the other hand, information in BEHAVIOR sections are more or less simulator oriented.

From a modeling point of view, SEMANTICS sections and BEHAVIOR sections are still complementary to each other. Although the new section is mainly designed for compiler generation, it is not completely compiler specific. In principle, it is a formalized description of behavior. Therefore, instruction accurate simulators can be generated from SEMANTICS sections. However, even in the SEMANTICS section based simulator generation, the functional description of components which are not part of the instruction set, e.g. fetch unit, decoder, still relies on BEHAVIOR sections. Similarly, the behavior description of *intrinsics* operations can only given in BEHAVIOR sections, because in SEMANTICS sections they have only symbolic representatives.

Moreover, having both the BEHAVIOR section and the SEMANTICS section in the same model has the benefit that a single model can provide multiple abstraction levels of instruction description. For different tools, this means each tool can choose the information meeting its requirements. E.g. in the generation of the C compiler especially the code selector needs the SEMANTICS section to understand the purpose of each instruction. On the other hand, the CA simulator generation uses solely the low-level C/C++ description in BEHAVIOR sections. Furthermore, the IA simulator generation incorporates the information in SEMANTICS section for instruction behavior and the description in BEHAVIOR section for fetch and decode units. Since the information for different tools exists in only one model, the design will be more manageable and less error prone.

6. Case Studies

In order to prove the concept described in this paper, overall five architectures have been examined, namely ARM's ARM7 core, CoWare's LTRISC core, STMicroelectronics' ST220 VLIW multimedia processor [17], Infineon's PP32 network processing unit, and Texas Instruments' C54× digital signal processor. The LTRISC processor is a very small RISC core, which is provided with CoWare's LISATek Processor Designer. The PP32 is an evolution of [18] and comprises instructions which are able to operate on bitfields. The existing LISA 2.0 models of these five processors have been extended with SEMANTICS sections. Although the SEMANTICS section is not intended for the extension of already existing models, our test approach proved that the new section does not impose any particular modeling style. This is very important to keep the flexibility of LISA 2.0.

Table 1. Semantics extension of five processors.

	ARM7	LTRISC	ST220	PP32	C54×
ISA	RISC	RISC	RISC	RISC	CISC
No. instructions	62	17	82	41	110
Design effort	4d	2d	10d	8d	15d

The result of extending the 5 LISA models is summarized in Table 1. Note that the design effort is calculated in man-days. It can be seen that the work for adding SEMANTICS sections scales with the number of instructions in the architecture. Note that although the PP32 has fewer instructions than the ARM7 processor, it takes more time to add SEMANTICS sections for the model. This mainly due to the fact that almost all instructions in PP32 can be executed conditionally. And, there are four different conditions can be selected. This makes the behavior of PP32's instructions more complex than that of the ARM7 processor, which leads to more modeling effort. Similarly, the complexity of the *I*nstruction-*S*et *A*rchitecture (ISA) also has influence on the effort. The RISC instruction-set requires less effort to model than the CISC instruction-set needs. Generally, the predefined micro-operations and the bit-field specification make the behavioral description code size in SEMANTICS sections significantly less than that of the C code in BEHAVIOR sections. Furthermore, the ambiguity of the C description is avoided, which is important for C compiler generation. In [14], details about SEMANTICS based C compiler generation techniques are discussed.

7. Conclusion

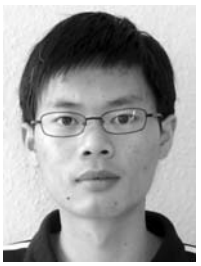
In this paper, we presented an approach for modeling instruction semantics based on an existing ADL with

the main purpose of C compiler generation. Our approach incorporates a new SEMANTICS section into the structure of LISA 2.0 without influencing the existing flexibility, and achieves a concise formalism for instruction-set description which is important for code selector generation. Besides providing instruction semantics for C compiler generation, it is also possible to generate instruction-set simulator and documentation with the information provided by SEMANTICS sections.

A further interesting result of our approach is that both instruction- and cycle-accurate descriptions of the processor architecture are able to coexist in a single model. This allows for a very high design efficiency on different abstraction levels, while maintaining consistency through using a single model during the entire design process. Our future research activities will be in the area of processor specific code optimization based on instruction semantics.

References

1. A. Hoffmann, R. Leupers, and H. Meyr. *Architecture Exploration for Embedded Processors with LISA*. Kluwer Academic Publishers, Boston, Jan. 2003. ISBN 1-4020-7338-0.
2. G. Braun, A. Nohl, W. Sheng, J. Ceng, M. Hohenauer, H. Scharwaechter, R. Leupers, and H. Meyr. A Novel Approach for Flexible and Consistent ADL-driven ASIP Design. *Proc. of the Design Automation Conference (DAC)*, Mar. 2004.
3. S. Bashford, U. Bieker, B. Harking, R. Leupers, P. Marwedel, A. Neumann, and D. Voggenauer. The MIMOLA Language, Version 4.1. Reference Manual, Department of Computer Science 12, Embedded System Design and Didactics of Computer Science, 1994.
4. R. Leupers and P. Marwedel. A BDD-based frontend for retargetable compilers. In *Proc. of the European Design and Test Conference (ED & TC)*, pages 239–243, 1995.
5. G. Hadjiyiannis, S. Hanono, and S. Devadas. ISDL: An Instruction Set Description Language for Retargetability. In *Proc. of the Design Automation Conference (DAC)*, Jun. 1997.
6. Silvina Hanono and Srinivas Devadas. Instruction selection, resource allocation, and scheduling in the AVIV retargetable code generator. In *Design Automation Conference*, pages 510–515, 1998.
7. A. Fauth, J. Van Praet, and M. Freericks. Describing Instruction Set Processors Using nML. In *Proc. of the European Design and Test Conference (ED & TC)*, Mar. 1995.
8. A. Halambi, P. Grun, V. Ganesh, A. Khare, N. Dutt, and A. Nicolau. EXPRESSION: A Language for Architecture Exploration through Compiler/Simulator Retargetability. In *Proc. of the Conference on Design, Automation & Test in Europe (DATE)*, Mar. 1999.
9. *EXPRESSION User Manual (version 1.0) http://www.ics.uci.edu/~express/documentation.htm*.
10. A. Hoffmann, T. Kogel, A. Nohl, G. Braun, O. Schliebusch, O. Wahlen, A. Wieferink, and H. Meyr. A Novel Methodology for the Design of Application Specific Instruction Set Processors (ASIP) Using a Machine Description Language. *IEEE Transactions on Computer-Aided Design*, 20(11):1338–1354, Nov. 2001.
11. CoWare Inc., http://www.coware.com. *LISATek product family*.
12. M. Hohenauer, H. Scharwaechter, K. Karuri, O. Wahlen, T. Kogel, R. Leupers, G. Ascheid, and H. Meyr. A Methodology and Tool Suite for C Compiler Generation from ADL Processor Models. *Proc. of the Conference on Design, Automation & Test in Europe (DATE)*, Mar. 2004.
13. ACE – Associated Computer Experts bv. *The COSY Compiler Development System http://www.ace.nl*.
14. J. Ceng, M. Hohenauer, R. Leupers, G. Ascheid, H. Meyr, and G. Braun. C compiler retargeting based on instruction semantics models. In *DATE*, pages 1150–1155, 2005.
15. A. Aho, R. Sethi, and J. Ullman. *Compilers, Principles, Techniques and Tools*. Addison-Wesley, Jan. 1986. ISBN 0-2011-0088-6.
16. A. Aho, M. Ganapathi, and S. Tjiang. Code generation using tree matching and dynamic programming. *IEEE Transactions on Programming Languages and Systems*, 11(4):491–516, Oct. 1989.
17. F. Homewood and P. Faraboschi. ST200: A VLIW Architecture for Media-Oriented Applications. In *Microprocessor Forum*, Oct. 2000.
18. X. Nie, L. Gazsi, F. Engel, and G. Fettweis. A new network processor architecture for high-speed communications. In *Proc. of the IEEE Workshop on Signal Processing Systems (SIPS)*, pages 548–557, Oct. 1999.



Jianjiang Ceng received his B. Eng. Degree in Electric Engineering from Shanghai Jiaotong University, Shanghai, China, in 1999 and the M. Sc. degree in Communications Engineering from the RWTH Aachen University of Technology, Aachen, Germany, in 2003. He is currently doing his PhD in the Institute of Integrated Signal Processing System of the RWTH-Aachen. His research interests are in embedded software, retargetable compilation, code optimization, with current activity focused on the heterogeneous multi-processor compilation and system simulation.
zeng@iss.rwth-aachen.de

Weihua Sheng received his M. Sc. degree in Communications Engineering from the RWTH Aachen University of Technology, Aachen, Germany, in 2003. He is currently working in Synopsys Shanghai as R& D engineer.
sheng@iss.rwth-aachen.de

Manuel Hohenauer received the diploma in electrical engineering from Aachen University of Technology (RWTH), Aachen, Germany, and is currently working towards the Ph.D. degree in electrical engineering at the same university. His research interests include retargetable code generation for embedded processors with main focus on machine description generation for retargetable compilers from architectural descriptions and retargetable code optimization techniques.
hohenau@iss.rwth-aachen.de

Rainer Leupers received the Diploma and Ph.D. degrees in Computer Science with honors from the University of Dortmund, Germany, in 1992 and 1997. From 1997-2001 he was a senior research engineer at the Embedded Systems group at the University of Dortmund. Between 1999-2001 he was also a project manager at ICD, where he headed the development of custom C compilers and other industrial software tool projects. In 2002, Dr. Leupers joined RWTH Aachen University as a professor for Software for Systems on Silicon. His research and teaching activities revolve around software development tools, processor architectures, and electronic design automation for embedded systems, with emphasis on C compilers for application specific processors in the areas of signal processing and networking. He authored several books and numerous technical papers on software tools for embedded processors, and he served in the program committees of leading EDA and compiler conferences, including DAC, DATE, and ICCAD. Dr. Leupers received several scientific awards, including Best Paper Awards at DATE 2000 and DAC 2002.
leupers@iss.rwth-aachen.de

Gerd Ascheid received the Dipl.-Ing. and Dr.-Ing. (PhD) degrees from the Aachen University in 1977 and 1984. PhD research focused on synchronization algorithms for digital receivers and their implementation using digital signal processing. This work was complemented by measurement and analysis of the non-linear behaviour of phase locked loops in noise. After the PhD he worked as Sr. Researcher at the Aachen University, continuing the research on synchronization and working on various industry consulting projects for European and US based companies. During that time he co-authored (jointly with Dr. Meyr) the book "Synchronization in Digital Communications", published by Wiley in 1990. Since April 2003 he is heading the Institute for Integrated Signal Processing Systems jointly with Prof. Meyr.
ascheid@iss.rwth-aachen.de

Heinrich Meyr received his M.Sc. and Ph.D. from ETH Zurich, Switzerland. He spent over 12 years in various research and management positions in industry before accepting a professorship in Electrical Engineering at Aachen University of Technology (RWTH Aachen) in 1977. He has worked extensively in the areas of communication theory, digital signal processing and CAD tools for system level design for the last thirty years. His research has been applied to the design of many industrial products. At RWTH Aachen he is a co-director of the institute for integrated signal processing system (ISS) involved in the analysis and design of complex signal processing systems for communication applications. As well as being a Fellow of the IEEE he has served as Vice President for International Affairs of the IEEE Communications Society meyer@iss.rwth-aachen.de