ISDL: An Instruction Set Description Language for Retargetability

George Hadjiyiannis
Department of EECS, MIT
ghi@rle-vlsi.mit.edu

Silvina Hanono
Department of EECS, MIT
silvina@rle-vlsi.mit.edu

Srinivas Devadas
Department of EECS, MIT
devadas@rle-vlsi.mit.edu

Abstract— **We present the Instruction Set Description Language, ISDL, a machine description language used to describe target architectures to a retargetable compiler. The features and flexibility of ISDL enable the description of vastly different architectures, in particular VLIW architectures. ISDL explicitly supports constraints that define valid operation groupings within an instruction, increasing the range of specifiable architectures. We have written a tool that, given an ISDL description of a processor, automatically generates an assembler for it. Ongoing work includes the development of an automatic code-generator generator.**

I. Introduction

A. Embedded Systems

For a variety of reasons, manufacturers profit from integrating an *entire system* on a single integrated circuit (IC). As time-to-market requirements place greater burden on designers for fast design cycles, programmable components are introduced into the system, and an increasing amount of system functionality is implemented in *software* relative to hardware. Systems containing programmable processors that are employed for applications other than general-purpose computing are called *embedded systems*.

B. Hardware–Software Co-Design

Rather than designing the software and hardware components of an embedded system separately, *hardware–software co-design* (e.g., [1]) is more cost effective and results in a shorter time to market.

In this design methodology, designers partition the system functionality into hardware and software. Additionally, a target processor is chosen from existing processor designs, or an ASIP (Application Specific Instruction-Set Processor) is designed to execute the software. The hardware, software, and ASIP are implemented and the resulting system is evaluated using a hardware–software co-simulator. The partitioning and processor design are repeated until an acceptable system is developed. Under this methodology, tools for *code generation* and *hardware–software co-simulation* have become essential parts of the designer's tool-box.

As the complexity of embedded systems grows, programming in assembly language and optimization by hand are no longer practical except for time-critical portions of the program that absolutely require it. Further, hand coding virtually eliminates the possibility of changing the processor architecture. An automatic code generation methodology will be most useful if it can be easily adapted to generating code for different processors. This property, commonly called *retargetability*, is discussed in the following section. We argue that in order to be able to explore the processor design space, an automatically retargetable compilation strategy is required.

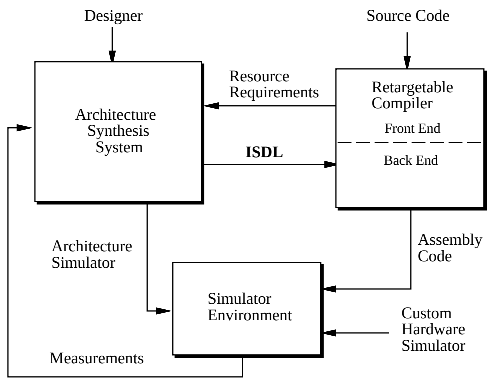

Fig. 1. **The design flow for an ASIP**

C. A Machine Description Language for Easy Retargetability

A retargetable compiler receives as input the program corresponding to the application as well as the *machine description* of the target processor. The machine description includes an instruction set specification and some architectural information. The code generator produces code that can run on the target processor optimized for speed and size. By varying the machine description (i.e., using an ASIP) and evaluating the resulting object code one can effectively explore the design space of both hardware and software components. Our ASIP design methodology is illustrated in Figure 1.

The machine description language is a critical component in the design flow and is the focus of this paper. Ideally, the machine description language must be able to perform several functions:

- Specify a wide variety of architectures.
- Explicitly support constraints that define valid operation groupings within an instruction.
- Be easily understandable and modifiable by a compiler writer or hardware architect.
- Support the automatic generation of a code generator.
- Support the automatic generation of an assembler.
- Support the automatic generation of an Instruction Level Simulator (**ILS**).
- Provide adequate information to allow for code optimizations.

Various proposed machine description languages lack support for one or more of the above features (see Section II for a review). We have developed a machine description language, ISDL (Instruction Set Description Language), which has all of the above features. We have complete ISDL descriptions for a powerful ASIP VLIW (Very Long Instruction Word) architecture and the Motorola 56000 (see Section IV). We have also developed a tool that automatically generates an assembler given any ISDL description, an important step in an automatically retargetable compilation methodology.

D. Organization of this paper

Section II presents previous work on retargetability and machine description languages. Section III presents a description of the ISDL language and its components. In Section IV, we further illustrate the language components using portions of the 56000 ISDL description. Section V describes our assembler generator tool and related issues. Conclusions and ongoing work on ISDL are presented in Section VI.

II. Related Work on MDLs for Embedded Processors

We briefly review three representative research projects in the area of code generation for embedded systems: MIMOLA [2], CHESS [3], and FLEXWARE [4]. The proceedings of the Dagstuhl Workshop [5] contain a collection of papers documenting several other contributors' efforts.

The MIMOLA design system is an environment for hardware–software co-design and includes a retargetable microcode compiler. The MIMOLA microcode compiler infers rules for code generation directly from a structural description (e.g., a net-list) of the target architecture instead of a behavioral description (e.g., the instruction set). The advantage of this approach is that it provides a single machine description for both the synthesis of the target architecture and the generation of microcode. However, MIMOLA descriptions are generally very low level, and therefore laborious to write and modify.

CHESS is a retargetable code generation environment for fixed-point DSPs and ASIPs; it was developed in the context of the CATHEDRAL II high-level synthesis system [6]. The target machine is described using the language nML [7]. The nML language is attractive because it allows the user to specify the target architecture in a way that parallels instruction-set descriptions found in a user's manual. In contrast to MIMOLA, the machine description contains behavioral as well as structural information. This enables the code generator to recognize more optimization opportunities.

FLEXWARE consists of two components: a code generator, CODESYN [8], and an instruction-set simulator, INSULIN [9]. The machine description for CODESYN consists of three components: the *instruction set*, the *available resources* and their classification, and an *interconnect graph*.

None of the systems mentioned above provide support for explicit constraints. Without explicit constraints, descriptions for architectures with Instruction Level Parallelism become very laborious to write because every legal combination of microinstructions must be explicitly listed. In addition, deriving a set of constraint clauses, in a form usable by the compiler, is very difficult.

Also, none of the systems automatically generate assemblers, although nML provides enough support for doing so.

III. The Instruction Set Description Language

An ISDL model of the target processor is generated either by hand or by a high-level CAD tool. The compiler front end receives a source program written in C or C++. It then parses the source program and generates an intermediate format description in SUIF [1] [10]. The compiler back end takes the SUIF code as well as the ISDL description as inputs and produces assembly code specific to, and optimized for, the target processor. The ISDL description is also used to create an assembler (see Section V). The automatically generated assembler transforms the code produced by the compiler to a binary file that is used as input to the Instruction Level Simulator (ILS).

The goal of our system is to support a wide variety of architectures. The main focus of ISDL is on VLIW (Very Long Instruction Word) architectures; however, it also supports standard microcontrollers, and custom datapath DSP cores. In particular, it must support multiple functional units, different interconnect topologies, complex instructions, resource conflicts, pipelining idiosyncrasies, etc. Our system also supports automatically generated architectures. Such architectures cannot be guaranteed to have clean instruction sets (i.e., instruction sets where every operation combination is valid). In order to accommodate for this, ISDL supports explicit constraints that define which operation groupings are valid. The compiler can therefore avoid generating invalid instructions by ensuring that each instruction meets these constraints. Note that some commercial architectures also require such constraints (e.g., the Motorola 56000 cannot perform a `Move` to the top of the hardware stack within the last three instructions of a `DO` loop).

An ISDL description consists of six sections:

- Instruction Word Format
- Global Definitions
- Storage Resources
- Instruction Set
- Constraints
- Optional Architectural Details

Each of these is described below along with their high level syntax definitions.

A. Instruction Word Format

The *Instruction Word Format* section defines the hardware instruction word. The instruction word is separated into multiple fields each containing one or more subfields. The bitwidth of each subfield is also provided. The instruction word is assembled by concatenating all the subfields in the order specified in this section.

Note that the division into subfields is a convenience to the designer. The subfield division may be arbitrary; however, careful subfield division can make later parts of the machine description easier to write.

B. Global Definitions

The second section of an ISDL description contains a list of definitions used in the later sections. The definitions consist of three main types: *Tokens*, *Non-terminals*, and *Split functions*. These definitions also help create an automatically generated assembler.

Tokens are a symbolic representation of the assembly syntax within the parser. Tokens are used to represent entities such as register and memory bank names, immediate constants, etc. In addition, we allow groupings of syntactically related tokens. In order to differentiate between the elements in a group, these tokens return a value identifying the particular element (e.g., register names such as R0 to R15, can be abbreviated as one token whose value corresponds to the register number).

Non-terminals have several purposes. First, syntactically unrelated tokens can be grouped together in a non-terminal for convenience; allowing a large number of possible alternatives in an instruction to be factored out into a non-terminal. For example, suppose that `SRC` and `DEST` can each be one of seven different options. In the following instruction

```
Move SRC DEST
```

49 different rules are required to describe all possible syntax combinations. With non-terminals, only three rules are required: one for the instruction, and one each for the `SRC` and `DEST` non-terminals.

Non-Terminals also contain an `action` field which allows the inclusion of arbitrary C code to be executed along with every rule. This is a very powerful feature that permits the inclusion of architecture specific routines in our generated tools.

Split function definitions are used to automatically create functions that can take a long bitfield (such as a long memory address, or

[1] Stanford University Intermediate Format

immediate data) and split it up into existing subfields of the instruction word. These functions can then be used in non-terminal actions and bitfield assignment commands (see Section III-D) in order to assign the correct values to the subfields.

C. Storage Resources

The *Storage* section lists all storage resources visible to the programmer. It lists the names and sizes of the memory, register files, and special registers. This section is used by the compiler to determine the available resources and how they should be used.

The different storage resource specifications and their parameters are: (depth in words, width in bits)

- Memory(depth, width)
- RegFile(depth, width)
- Register(width) - Single registers used for data computation.
- CRegister(width) - Control and Status registers that may cause side effects.
- Stack(depth, width) - Used for hardware stacks only.
- ProgramCounter(width)
- Wire(width) - Interconnect for datapath.

D. Instruction Set

The *Instruction Set* section is split into *Fields* corresponding to the separate operations that can be performed in parallel within a single instruction. This supports the description of VLIW architectures. Some fields may be optional. A list of all possible operations is provided for each of these fields.

Each operation description consists of the following elements:

- **Operation Name** - Assembly mnemonic for each operation.
- **Operation Parameters** - A list of tokens or non-terminals.
- **Bitfield Assignment Commands** - A set of commands which manipulate the bitfields. They may include operation parameters as values.
- **RTL Description** - This describes the effect of the operation on the storage resources. The compiler uses the RTL description to make operation selection decisions.
- **Costs** - Multiple costs are allowed including operation execution time, code size, costs due to resource conflicts, etc.
- **Timing** - Information describing when the various effects of the operation take place (e.g., because of pipelining).

E. Constraints

The Instruction Set section describes a number of fields that can generally be executed in parallel. However, there are certain combinations of operations that may not be executable by the hardware. The *Constraints* section is used to make these combinations visible to the compiler so that the compiler can avoid generating such illegal operation combinations.

Constraints are described as a set of Boolean rules, all of which must be satisfied for an instruction to be valid. Constraints can be time shifted to show conflicts in instructions issued at different times. Wild cards can be used to simplify the constraint descriptions. Also, variables are used to enforce any restriction where different parts of a single constraint must match.

We have identified three types of constraints:

- **Datapath Conflicts** - Two parallel operations try to use the same datapath resources (e.g., competition for the bus).
- **Bitfield Conflicts** - Two parallel operations try to set the same bitfield in the instruction word.
- **Syntactic Constraints** - Constraints that do not correspond to hardware conflicts, but are artifacts of the assembler syntax.

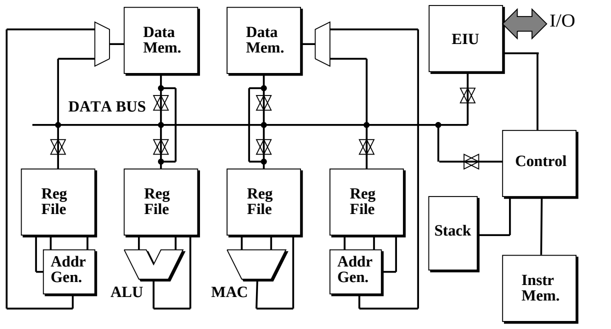

Fig. 2. **The ASIP VLIW Architecture**

F. Optional Architectural Details

The ISDL description can give the compiler some information about the hardware architecture in order for the compiler to make better machine dependent code optimizations. This section is not necessary for the compiler to generate good code, however, it can help the compiler find a few more optimizations to generate even better code. Examples of such optimizations are the use of delay slot instructions, and branch prediction hints.

IV. An ISDL Example

We have written ISDL descriptions for an aggressive ASIP VLIW architecture, and for the Motorola 56000 DSP processor. The Motorola DSP56000 consists of three main processing units that operate in parallel: an ALU, an Address Generator, and a Program Controller. The architecture is capable of performing two address calculations, two memory transfers, and an ALU operation within a single instruction.

Figure 2 shows the structure of the ASIP VLIW processor. It consists of five functional units that can operate in parallel: a Controller, two identical address generators, an ALU, and a floating point MAC (multiply-accumulate) unit. This processor can perform three data transfers, two addressing operations, two computation operations, and a flow control operation within a single instruction.

We provide portions of our ISDL description for the Motorola DSP56000 processor to illustrate the structure of ISDL, and how it is used.

```
Section Format

DBM     = OP[8], MODE[8];
Main    = OP[8];
```

This describes a 24 bit instruction word divided into three subfields: DBM.OP, DBM.MODE, and Main.OP. Each subfield is 8 bits long. DBM.OP is the MSB and Main.OP is the LSB.

```
Non_Terminal ival ALUSRC:
     A_D {$$ = 1; } | B_D {$$ = 1; } |
     X_D {$$ = 2; } | Y_D {$$ = 3; };
```

The above line defines a non-terminal. The definition consists of the keyword `Non_Terminal`, the type of the returned value, a symbolic name as it appears in the assembly section, and an action that describes the possible token or non-terminal combinations and the return value associated with each. Specifically, this line defines a non-terminal with the symbolic name `ALUSRC` whose return value is 1 for `A_D` or `B_D`, 2 for `X_D`, and 3 for `Y_D`.

```
Section Assembly

Field Main:
   ADC XYSRC, ACC
```

```
    { Main.OP = 0x21|(ACC<<3)|(XYSRC<<4); }
    { ACC <- ACC + XYSRC + CCR[0]; }
    { cycle = 2 + dbm; size = 1 + dbm; }
    { latency = 1; }
```

The `Field` keyword denotes operations that can be performed in parallel. In this example, the `ADC` operation takes two parameters. The `Main.OP` bitfield is set to the result of `0x21|(ACC<<3)|(XYSRC<<4)`.

The second set of brackets in each operation contain an RTL description of the effect of the operation. For example, the `ADC` operation has the effect of adding the contents of the accumulator to the source (X or Y) and to the carry bit (`CCR[0]`), and storing the result in the accumulator.

The third set of brackets in each operation contain a set of costs. In the example above, two costs are defined: a cycle count, and the resulting code size. The `ADC` operation takes two cycles to complete and one instruction word, unless it is grouped with a parallel operation that requires additional cycles and/or words. The last set of brackets in each operation contains timing information.

Note that if the `ADC` operation were to assign values to the `DBM` bitfields, then a bitfield conflict with `DBM` operations would result, and this should be reflected in the Constraints section.

```
Section Constraints
~(( REP *) & ([1] DO *,*))
```

This constraint denotes that any DO instruction is illegal when fetched as the next instruction after a REP instruction. The `[1]` indicates a time shift of one instruction fetch for the DO instruction.

See [11] for a more extensive example.

V. Automatic Assembler Generator

We required that ISDL be capable of automatically generating an assembler. This allows us to decouple the development of the compiler from that of the ILS. The ILS requires a compiled binary as input. The availability of an assembler allows assembly programs to be written and tested on the ILS, even when no compiler is available. Furthermore, the output of the compiler can be in assembly which is much more human readable than binary. Thus, some debugging can be performed without the use of an ILS.

We have designed and implemented an automatic assembler generator. It receives an ISDL description as input, and produces an assembler which assembles the compiler's output to a binary file.

The assembler generator produces Lex and Yacc [2] files which, when compiled, result in an executable capable of parsing the assembly and generating the instruction words. This parser is a two pass parser capable of processing symbolic addresses (i.e., labels).

VI. Conclusions and Ongoing Work

Having described the details of ISDL, we now compare it to the machine description languages referred to in Section II. Mimola is too low level for retargetable compilers, and it does not support the definition of constraints. CodeSyn includes some but not all of the information provided in an ISDL description. In particular, it includes an instruction set description, a listing of the available resources, and an interconnect graph. However, it does not support constraints. Of all the languages studied, nML is the closest to ISDL. It provides most of the information included in ISDL, and supports the description of a very extensive range of architectures. Its one shortcoming with respect to ISDL is that it does not support explicit constraints. Instead, it requires the enumeration of all the valid instructions related to a conflict. However, even valid instruction enumeration cannot support time shifted constraints (i.e., constraints that span more than one instruction).

We are currently developing a code-generator generator which takes ISDL descriptions as input and produces the corresponding code-generator. The retargetability of the code-generators allows for the exploration of the processor design space until an architecture suitable for the application at hand is found.

Furthermore, we are developing a tool that automatically generates an ILS for the target architecture using ISDL descriptions.

Acknowledgments

This research is supported by NSF Contract MIP-9612632.

References

[1] R. K. Gupta and G. De Micheli. Hardware–Software Cosynthesis for Digital Systems. *IEEE Design & Test of Computers*, pages 29–41, September 1993.

[2] P. Marwedel. The MIMOLA Design System: Tools for the Design of Digital Processors. In *Proceedings of the 21th Design Automation Conference*, pages 587–593, 1984.

[3] D. Lanneer et al. CHESS: Retargetable Code Generation for Embedded DSP Processors. In *Code Generation for Embedded Processors*, pages 85–102. Kluwer Academic Publishers, 1995.

[4] P. G. Paulin, C. Liem, T. C. May, and S. Sutarwala. DSP Design Tool Requirements for Embedded Systems: A Telecommunications Industrial Perspective. *Journal of VLSI Signal Processing*, 9(1/2):23–47, January 1995.

[5] P. Marwedel and G. Goossens, editors. *Code Generation for Embedded Processors*. Kluwer Academic Publishers, Boston, Massachusetts, 1995. Proceedings of the 1994 Dagstuhl Workshop on Code Generation for Embedded Processors. ISBN 0-7923-9577-8.

[6] G. Goossens et al. Integration of Medium-Throughput Signal Processing Algorithms on Flexible Instruction-Set Architectures. *Journal of VLSI Signal Processing*, 9(1):49–65, 1995.

[7] A. Fauth, J. Van Praet, and M. Freericks. Describing Instruction Sets Using nML (Extended Version). Technical report, Technische Universität Berlin and IMEC, Berlin (Germany)/Leuven (Belgium), 1995.

[8] P. G. Paulin, C. Liem, T. C. May, and S. Sutarwala. CodeSyn: A Retargetable Code Synthesis System. In *Proceedings of the 7th International High-Level Synthesis Workshop*, Spring 1994.

[9] S. Sutarwala, P. G. Paulin, and Y. Kumar. Insulin: An Instruction Set Simluation Environment. In *Proceedings of the 1993 Conference on Hardware Description Languages*, pages 355–362, 1993.

[10] Stanford Compiler Group. *The SUIF Library*, version 1.0 edition, 1994.

[11] G. Hadjiyiannis, S. Hanono, and S. Devadas. ISDL: An Instruction Set Description Language for Retargetability. Technical report, MIT, 1996. (http://rle-vlsi.mit.edu/spam/pubs/ISDL-TR.html).

[2] Lex is a lexical analyzer generator, and Yacc is a parser generator. They are the standard Unix utilities used for creating compilers, assemblers, and similar applications.