ISDL: An Instruction Set Description Language for Retargetability

Abstract

We present the Instruction Set Description Language, ISDL, a machine description language used to describe target architectures to a retargetable compiler. A retargetable compiler is capable of compiling application code into machine code for different processors. The features and flexibility of ISDL enable the description of vastly different architectures such as an ASIP VLIW processor and a commercial DSP microprocessor. For instance, unlike other machine description languages, ISDL explicitly supports constraints which define valid operation groupings within an instruction, increasing the range of specifiable architectures. We have written a tool which, given an ISDL description of a processor, can automatically generate an assembler for it. Ongoing work includes the development of an automatic code-generator generator.

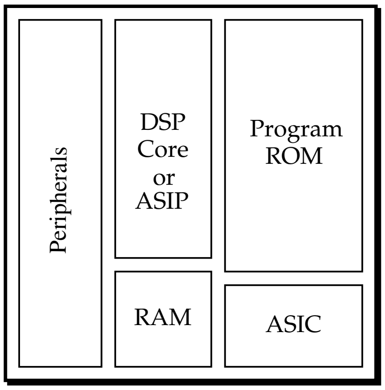

Figure 1: **A heterogeneous system-on-a-chip**

1 Introduction

1.1 Embedded Systems

For a variety of reasons, manufacturers profit from integrating an *entire system* on a single integrated circuit (IC) [1] [2]. Such a high level of integration has brought new challenges in the design of digital systems. Designing an entire system with a custom integrated circuit is now neither economical nor practical. 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. These programmable components, called *embedded processors* or *embedded controllers*, can be general-purpose microprocessors, off-the-shelf digital signal processors (DSPs), in-house application-specific instruction-set processors (ASIPs), or microcontrollers. Systems containing programmable processors that are employed for applications other than general-purpose computing are called *embedded systems*. Figure 1 presents one such system, consisting of a DSP core or an ASIP, a program ROM, RAM, application-specific circuitry (ASIC), and peripheral circuitry.

The advantages of incorporating software components are twofold: faster turnaround time due to the use of predefined processors, and increased flexibility due to field programmability. Moreover, various applications of the same genre may share the same hardware structure, with differences reflected in the software component; therefore, it is possible to use the same mask for all of these applications, thus substantially reducing the cost of manufacturing.

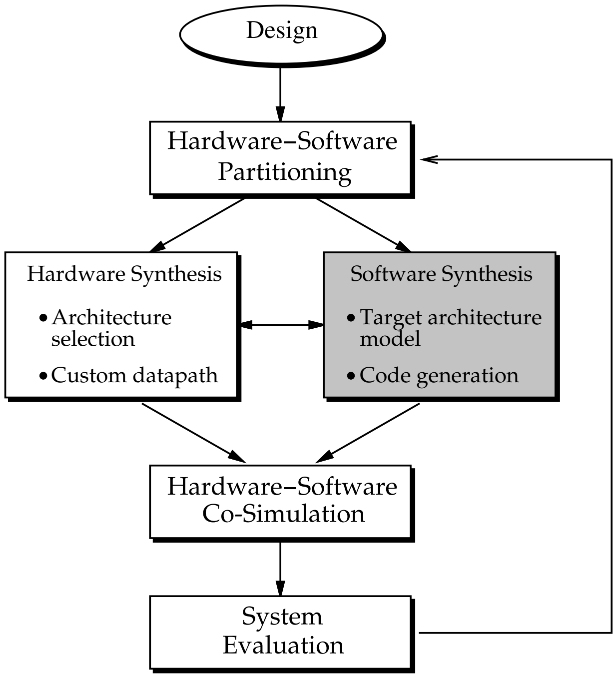

Figure 2: **A generic hardware–software co-design methodology**

1.2 Hardware–Software Co-Design

Rather than designing the software and hardware components of an embedded system separately, *hardware–software co-design* is more cost effective and results in a shorter time to market. There have been proposals for the hardware–software co-design of digital systems (e.g., [3], [4], and [5]). The simplified view of a generic co-design methodology is shown in Figure 2.

In this design flow, designers first determine which parts of the functionality of the system will be implemented in hardware and which parts in software. They then proceed to design each of the hardware and software components. The system is simulated and evaluated with a hardware–software co-simulator. If the results of the simulation meet design specifications (e.g., correctness and timing constraints), then the design is accepted. Otherwise, the designers may re-partition the original design specification and reiterate the same process. Under this methodology, tools for *code generation* and *hardware–software co-simulation* have become essential parts of the designer's tool-box.

Compilers for software written in high-level languages are becoming indispensable in the design of embedded systems. The traditional approach to high-quality embedded software has been to write the code in assembly language. 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. These processors can be different commercial DSP cores, or application-specific instruction processors. This property, commonly called *retargetability*, is discussed in the following section.

1.3 A Machine Description Language for Easy Retargetability

In the hardware–software co-design iteration (see Figure 2), the target architecture itself may be changing. This is because in order to minimize cost and power consumption and to increase performance of the system, the architecture should be matched to the application. This can be achieved by designing an ASIP, generating code for it with a compiler, and then testing the code for desired performance. The results of this design process can then be used to make architectural changes to the ASIP to improve performance. Hand-coding is not an option since it eliminates the capability of iterating over the hardware specification of the ASIP. We argue that in order to be able to explore the design space, an automatically retargetable compilation strategy is required.

A retargetable compiler receives as input the program corresponding to the application as well as the *machine description* of the ASIP. The machine description includes an instruction set specification and some architectural information. The code generator produces code that can run on the ASIP optimized for speed and size. By varying the machine description 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 3.

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 a simulator.

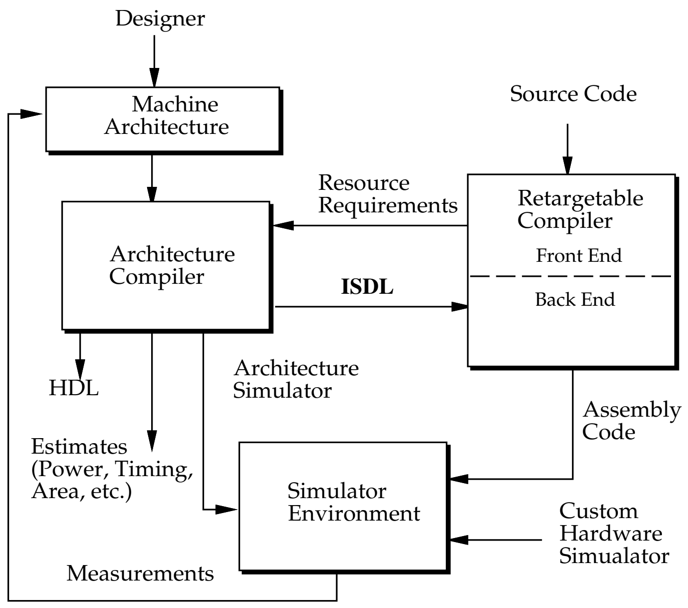

Figure 3: **The design flow for an ASIP**

- 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 2 for a review). We have developed a machine description language, ISDL (Instruction Set Description Language), which has all of the above features. Using ISDL, we have written machine descriptions for ASIPs as well as commercial DSP cores. In particular, we have complete ISDL descriptions for a powerful ASIP VLIW (Very Long Instruction Word) architecture and the Motorola 56000 (see Section 4). We have also written a tool that automatically generates an assembler given any ISDL description, an important step in an automatically retargetable compilation methodology.

1.4 Organization of this paper

The rest of this paper will focus on the details of ISDL. Section 2 presents previous work on retargetability and machine description languages. Section 3 presents a description of the ISDL language and its components. In Section 4, we further illustrate the language components using portions of the 56000 ISDL description. Section 5 describes our assembler generator tool and related issues. Conclusions and ongoing work on ISDL are presented in Section 6.

2 Related Work

This section presents a brief survey of compiler literature that pertains to retargetability and machine description languages.

2.1 Retargetability for General-Purpose Processors

There are two issues involved in retargetable compilers: code generation and optimization. One approach to generating code is to perform pattern-matching on expression trees. This approach is also capable of performing local optimizations. Cattell designed a framework in which code generators based on pattern-matching are automatically derived from a machine description language [6]. The landmark paper by Aho et al. [7] established the foundation in which several modern dynamic-programming code-generator generators find their origin. One such code-generator generator is IBURG [8], employed in the compiler framework LCC [9]. The dynamic-programming methodology yields only locally optimal code (i.e., within the expression tree) which is often insufficient.

More recently, Bradlee proposed a retargetable scheduler, MARION, that is used for RISC architectures [10]. It uses a machine description that models pipelines and superscalar instruction issues, and schedules instructions to effectively utilize the features and to reduce conflicts.

2.2 Retargetability in Embedded Processors

We briefly review three representative research projects in the area of code generation for embedded systems: MIMOLA [11], CHESS [12], and FLEXWARE [13]. The proceedings of the Dagstuhl Workshop [14] contain a collection of papers documenting several other contributors' efforts.

The MIMOLA design system was originally conceived as a design environment for hardware structures using the MIMOLA hardware description language [15]. It later evolved into an environment for hardware–software co-design and includes a retargetable microcode compiler [11] [16]. One of the key features that distinguishes the MIMOLA project from others is that the 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 uniform mechanism of machine description for the various tasks in the entire design process. That is, the same machine description is used 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. In addition, MIMOLA does not support explicit constraints, nor does it allow for automatic assembler generation.

CHESS [12] 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 [17]. The target machine is described using the language nML [18]. A graphical representation of the architecture called the *instruction-set graph (ISG)* is then derived from the nML description. The ISG contains structural information such as connectivity and timing characteristics of the processor resources.

The nML mechanism appears 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. nML, like MIMOLA does not support explicit constraints. While it appears possible to automatically generate an assembler from an nML description, the nML publications do not make a claim to this effect.

FLEXWARE consists of two components: a code generator, CODESYN [19], and an instruction-set simulator, INSULIN [20]. FLEXWARE is a tool-set developed in response to the results of the survey conducted by Paulin et al. regarding trends and requirements in DSP design environments [13].

The machine description for CODESYN consists of three components: the *instruction set*, the *available resources* and their classification, and an *interconnect graph.* Like MIMOLA, CODESYN does not have support for explicit constraints, and it does not automatically generate an assembler.

3 The Instruction Set Description Language

Figure 4 demonstrates how ISDL is used in our design methodology. 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] [21]. 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 5). The automatically generated assembler transforms the code produced by the compiler to a binary file that is used as input to the simulator.

[1] Stanford University Intermediate Format

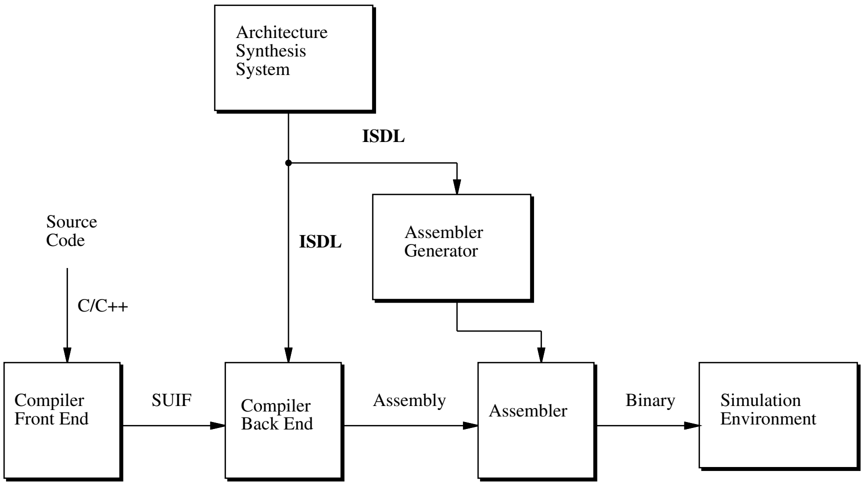

Figure 4: **High level system**

The goal of our system is to support a wide variety of architectures. These include standard microcontrollers, custom datapath DSP cores, and VLIW (Very Long Instruction Word) processors. 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
- Assembly Syntax
- Constraints
- Optimization Information

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

3.1 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.

```
Instruction Word Format:
<field_list>    ::= <field> <field_list> | <field>
<field>         ::= <field_name> '=' <subfield_list> ';'
<subfield_list> ::= <subfield> ',' <subfield_list> | <subfield>
<subfield>      ::= <subfield_name> '[' bitwidth ']'
```

Note that the division into subfields is a convenience to the designer. The subfield division may be arbitrary; however, there are certain subfield divisions which make later parts of the ISDL easier to write.

3.2 Global Definitions

The second section of an ISDL description contains a list of definitions used in the later sections. The definitions consist of four main types: *Tokens*, *Non-terminals*, *Split functions*, and *Macro definitions*. With the exception of macro definitions, these definitions are designed to help create an automatically generated assembler using Lex and Yacc [2] [22].

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). The tokens are defined in a syntax similar to that of Lex.

```
Tokens:
<token_list>    ::= <token_def> <token_list> |
<token_def>     ::= Token <assembly_syntax> <token_name> <token_type>
                    <token_value>
```

In addition to the tokens explicitly stated in the ISDL, there are other tokens which are automatically defined. These include all the instruction names, integers, hexadecimals, floating point

[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.

numbers, strings, names (a sequence of alphanumeric characters beginning with a letter and containing no white space), and single characters.

Non-terminals have several purposes. First, syntactically unrelated tokens can be grouped together in a non-terminal for convenience. If there are a large number of possible alternatives in an instruction, they can be factored out into a non-terminal. The following example demonstrates this point:

```
Move X:(addressing mode) Y:(addressing mode)
```

Suppose that `addressing mode` could be one of seven different options. In the instruction above, this would require 49 different rules to describe all possible syntax combinations if one did not have non-terminals. With non-terminals, it requires the use of two rules, one for the instruction and one for the `addressing mode` non-terminal.

```
Non_Terminals:
<non_terminal_list>    ::= <non_terminal_def> <non_terminal_list> |
<non_terminal_def>     ::= Non_Terminal <non_terminal_type> <non_terminal_name>
                           <non_terminal_options>
<non_terminal_options> ::= <non_terminal_opt> '|' <non_terminal_options> |
                           <non_terminal_opt>
<non_terminal_opt>     ::= <element_list> <action>
<element_list>         ::= <element> <element_list> | <element>
<element>              ::= <token_name> | <non_terminal_name> | <quoted_char>
```

The second advantage of non-terminals is that they allow the definition of new grammar rules, not necessarily related to any instruction. Finally, the `action` portion of non-terminals 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 immediate data) and split it up into existing subfields of the instruction word. These functions can then be used in non-terminal actions and operation assignment commands (see Section 3.4) in order to assign the correct values to the subfields.

```
Split Functions:
<split_list>    ::= <split_func> <split_list> |
<split_func>    ::= Split.<name> <subfield_list>
<subfield_list> ::= <subfield_name> '+' <subfield_list> | <subfield_name>
```

The *Macro definitions* type defines standard text replacement macros for syntactic convenience. We use the C pre-processor to implement this feature.

```
Macro Definitions:
<definition_list>       ::= <definition> <definition_list> |
<definition>            ::= '#'define <macro_name> <macro_substitution> |
                            '#'define <macro_name> '(' <param_list> ')'
                                <macro_substitution>
<param_list>            ::= <name> , <param_list> | <name>
```

3.3 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)** - An arbitrary number of memory banks is allowed. The instruction memory is not explicitly defined.
- **RegFile(depth, width)** - An arbitrary number of register files is allowed.
- **Register(width)** - Single registers used for data computation.
- **CRegister(width)** - Control and Status registers. Accessing these may cause side effects (e.g., Accessing I/O registers causes external ports to assume new values).
- **Stack(depth, width)** - Used for hardware stacks only.
- **ProgramCounter(width)** - The program counter has to be explicitly declared in order to make it visible to the RTL description, which is described in Section 3.4.

3.4 Assembly Syntax

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

```
Assembly Syntax:
<field_list>            ::= <field> <field_list> | <field>
<field>                 ::= Field <default_assignment> <field_name> ':'
                            <operation_list>
<operation_list>        ::= <operation> <operation_list> | <operation>
```

```
<operation>             ::= <operation_name> <param_list> <assignment_commands>
                            <RTL_desc> <costs> <timing>
<default_assignment>    ::= <assignment_commands> |
```

Each operation description consists of the following elements:

- **Operation Name** - Assembly mnemonic for each operation.
- **Operation Parameters** - A comma separated 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).

For optional fields, a default bitfield assignment command must be provided.

3.5 Constraints

The Assembly Syntax 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.

```
Constraints:
<constraint_list>       ::= <constraint> <constraint_list> |
<constraint>            ::= '(' <constraint> ')' |
                            <constraint> '|' <constraint> |
                            <constraint> '&' <constraint> |
```

```
                                '~' <constraint> |
                                <instr_def>
<instr_def>                 ::= <instr_name> <param_list> |
                                '[' <int> ']' <instr_name> <param_list>
<param_list>                ::= <param_item> ',' <param_list> | <param_item> |
<param_item>                ::= <param_name> | '*' | '@' <variable_name>
```

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.

3.6 Optimization Information

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 correct code, however, it can help the compiler generate better code. Examples of such optimizations are the use of delay slot instructions, and branch prediction hints.

4 An ISDL Example

We have written ISDL descriptions for an aggressive ASIP VLIW architecture, and for the Motorola 56000 DSP processor. Figure 5 shows the structure of the Motorola DSP56000. It consists of three main processing units that operate in parallel: the ALU performs all computation operations, the Address Generator Unit can generate up to two addresses simultaneously for memory transfers, and the Program Controller is responsible for instruction fetching and flow control. It also has three memory banks: the Program ROM which contains the executable code and program constants, the X-memory, and the Y-memory. The architecture is designed around two sets of buses: the address buses (XAB, YAB, and PAB) and the data buses (XDB, YDB, PDB, and GDB). The architecture is capable of performing two address calculations, two memory transfers, and an ALU operation within a single instruction.

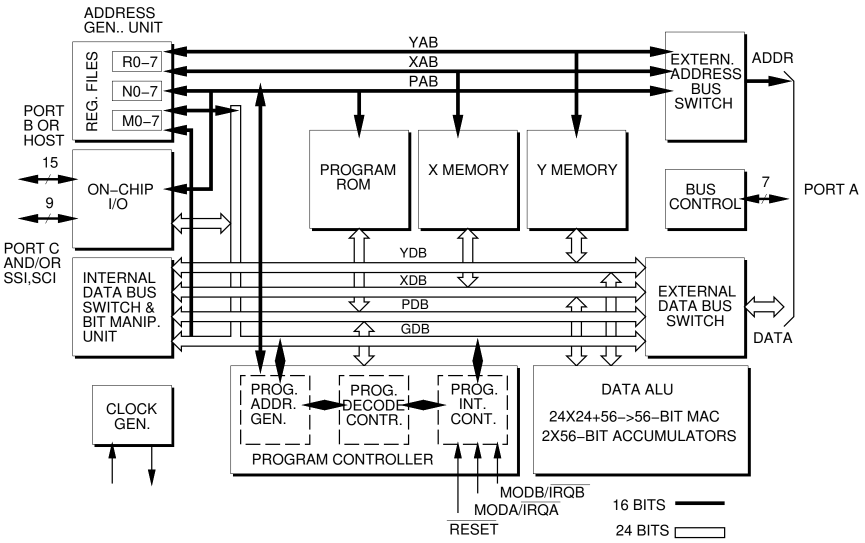

Figure 5: **The Motorola DSP56000 Architecture**

The ALU contains two source registers, X and Y, which can be subdivided into registers X0/1 and Y0/1. The destination registers consist of two accumulators, A and B, which can be subdivided into A0-A2 and B0-B2. All ALU arithmetic is fixed-point fractional. A shifter at the output stage of the ALU takes care of scaling if necessary. The address generator unit contains three register files: R, N, and M. It supports various addressing modes including bit-reversed addressing. The program control unit contains the PC, a number of control and status registers, and a hardware stack. It supports zero-overhead looping and nested loops.

The instruction word for the 56000 is 24 bits long and divided into an 8-bit op-code (LSB) and a 16-bit parallel data bus move operation. Some instructions require the use of an additional 24-bit word to store immediate addresses or data.

Figure 6 shows the structure of the ASIP VLIW processor. We designed this processor internally not as an example of a competitive DSP architecture, but rather as an over-aggressive processor that would allow us to explore the limits of our system. It consists of five functional units that can operate in parallel: the Controller which is responsible for instruction fetch and control flow, two identical and completely independent address generators, an ALU that can perform a number of operations on integer types, and a floating point MAC (multiply-accumulate) unit. The ALU, MAC, and address generators each have a 4-port register file. All data paths in this processor are 32 bits wide. This processor, unlike the DSP56000, is a load-store architecture requiring all operations to take place on registers. However, the architecture contains two specialized data paths between the memory and the main processing units (ALU and MAC), as well as a global bus. This

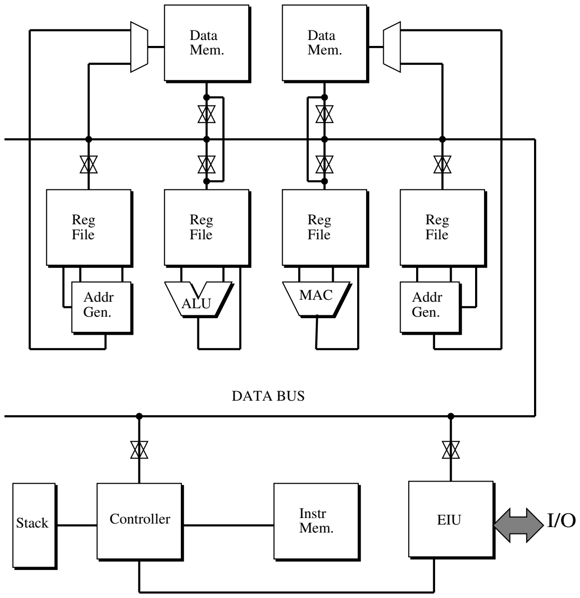

Figure 6: **The ASIP VLIW Architecture**

processor can perform three data transfers, two addressing operations, two computation operations, and a flow control operation within a single instruction. With the use of software pipelining, the load-store architecture has no negative impact on performance.

The address generators have multiple addressing modes. Their main use is for accessing the data memory, although they can also be used for integer computation. The ALU performs a wide variety of arithmetic and logical operations on integer types. It can also perform shift operations. The MAC unit performs various operations (including a multiply-accumulate operation) on floating point types. The controller contains a PC and a Jump Register (JR). The jump register is used for flow control and zero-overhead looping. It also includes a 16-entry hardware stack. The instruction word is 99 bits long and does not require additional words.

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.

```
Section Global_Definitions
```

```
//      assembly        token   type    value
Token   X[0..1]         X_R     ival    { yylval.ival = yytext[1] - '0'; };
Token   Y[0..1]         Y_R     ival    { yylval.ival = yytext[1] - '0'; };
Token   BA              BA_D    ival    { };
Token   A10             A10_D   ival    { };
Token   X:              XMEM    ival    { };
...
```

Text between the `//` and the end of the line is a comment. The next five lines define a list of tokens. Each definition consists of the keyword `Token`, the syntax of the token as it appears in assembly, a symbolic name for the token, the type of value returned by the token, and a piece of Lex dependent code that returns the appropriate token value. For example, the first line defines a token with the symbolic name `X_R` whose value is an integer. The assembly syntax allowed is `X0` or `X1`, and the values returned are 0 or 1 respectively.

```
//           type assembly      action
Non_Terminal ival IMM:          INT {$$ = INT; };
Non_Terminal ival SIMMD:        SID INT {$$ = INT; };
Non_Terminal ival PPA:          '<' '<' INT {$$ = INT & 0x3F; };
Non_Terminal ival XYSRC:        X_D {$$ = 0; } | Y_D {$$ = 1; };
Non_Terminal ival ALUSRC:       A_D {$$ = 1; } | B_D {$$ = 1; } |
                                X_D {$$ = 2; } | Y_D {$$ = 3; } |
                                X_R {$$ = 4 + 2 * X_R; } |
                                Y_R {$$ = 5 + 2 * Y_R; };
...
```

The above lines define non-terminals. The definitions consist of the keyword `Non_Terminal`, the type of the returned value, a symbolic name as it appears in the assembly, and an action that describes the possible token or non-terminal combinations and the return value associated with each. For example, the last 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`, 3 for `Y_D`, 4 for `X0`, 6 for `X1`, 5 for `Y0`, and 7 `Y1`.

```
Split.ADDR      DBM.OP+DBM.MODE+Main.OP;
...
```

The right hand side of this example represents the concatentation of `DBM.OP`, `DBM.MODE`, and `Main.OP`. This section is used to generate a function which splits a long input into shorter bitfields, as specified by the right hand side.

```
Section Storage

//                       = depth x width
Memory X                 = 0x1FF x 0x18
RegFile AGU_R            = 0x8 x 0x18
Register LA              = 0x10          // Loop Address
Register SP              = 0x6           // Stack Pointer
Register X0              = 0x18
CRegister SR             = 0x10
ProgramCounter PC        = 0x10
Stack SS                 = 0xf x 0x20
...
```

This is an example of the storage section. It contains instantiations of memory, register files, single registers, control registers, the program counter, and stacks.

```
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; }
    JCLR IMM, MEMS, ADRM, ADDRESS
                                      { Main.OP = 0x80|(IMM&0x1F)|(MEMS<<6);
                                        DBM.OP = 0x0A;
                                        DBM.MODE = 0x40 | ADRM;
                                        ADDITIONAL(0,Split.ADDR = ADDRESS); }
                                      { switch(MEMS.ADRM[IMM]) {
                                          case 0: PC <- ADDRESS;
                                          default: PC <- PC + 1; }
                                      }
                                      { cycle = 6 + jx; size = 2; }
                                      { latency = 2; }
...

Field { DBM.OP = 0x20; DBM.MODE = 0x00; } DBM:
    Move SIMMD, DMOVE                 { DBM.OP = 0x20 | DMOVE;
                                        DBM.MODE = SIMMD & 0xFF; }
                                      { DMOVE <- SIMMD; }
                                      { dbm = 0; }
                                      { latency = 1; }
...
```

The `Field` keywords denote operations that can be performed in parallel. The presence of the assignment commands { `DBM.OP = 0x20; DBM.MODE = 0x00;` } on the `Field` line of DBM declares that field as optional and sets the default values. In this example, the `JCLR` operation takes four parameters. The `ADDRESS` parameter is treated as a long constant and is passed through the *split address* function. The `Main.OP` bitfield is set to the result of `0x80|(IMM&0x1F)|(MEMS<<6)`. The `ADDITIONAL` keyword denotes that an additional word is required for this operation (to hold the `ADDRESS`).

Note that the `JCLR` operation assigns values to the `DBM` bitfields. This conflicts with the corresponding assignments of the `Move` operation. Thus, these two operations cannot be performed in parallel, and this conflict should appear in the Constraints section.

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 `JCLR` operation denotes a conditional jump by performing the appropriate arithmetic on the Program Counter.

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. For example, the `ADC` operation takes two cycles to complete unless it is grouped with a `Move` operation that requires additional cycles. It also takes one instruction word unless it is grouped with a `Move` that requires additional words. The particular `Move` operation shown does not require additional cycles or words.

The last set of brackets in each operation contains timing information.

```
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.

5 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 simulator. Figure 4 (page 8) shows the high level view of our system. The simulator requires a compiled binary as input. The availability of an assembler allows assembly programs to be written and tested on the simulator, 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 a simulator.

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 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). The actual assembler consists of the parser, and a shell script that processes the input files through the C preprocessor before parsing them. The preprocessing allows for macro substitutions and file include mechanisms. The latter enables us to include a common kernel which acts as an operating system into our compiled code. The kernel performs various initialization tasks such as setting up trap and exception vectors, enabling interrupts, etc.

The Lex file contains the lexical analyzer for the parser, and a symbol table. It defines the following tokens:

- Predefined types such as integers, hexadecimals, floats, strings, etc.
- Each of the operation names
- The tokens defined in the global definitions section of the ISDL description
- Labels used in the symbol table to represent symbolic addresses

The Yacc file contains the main driver and the actual parser. The main driver performs initializations for file I/O, and does some cleanup in the case of a syntax error. The parser contains a top level rule that performs the two passes. The first pass fills in the symbol table and produces a listing file as a debugging aid. The two pass system enables forward and reverse references in the labels.

```
word:           '{' Main_f DBM_f '}'
                        { ... } ;
```

The second pass contains a top level rule, shown above, that composes an instruction out of the *fields* described in the assembly syntax section of the ISDL description. Each *field* is a subrule which consists of a list of possible patterns, one for each *operation* described in the specification for that field. The action for each pattern is the *assignment command* that was given for that operation. For example, the description for `Field Main` (see Section 4) results in the following rule:

```
Main_f:         Main_ADC  XYSRC ',' ACC ';'
                        { Main_OP = 0x21 |($4 <<0x3 )|($2 <<0x4 );
                          }

        |       ...

        |       Main_JCLR  IMM ',' MEMS ',' ADRM ',' ADDRESS ';'
                        { Main_OP = 0x80 |($2 &0x1F )|($4 <<0x6 );
                          DBM_OP = 0xA ;
                          DBM_MODE = 0x40 |$6 ;
                          word_swap(0) ;
                          split_ADDR($8) ;
                          word_swap(0) ;
                          }
        |       ...
        ;
```

Furthermore, a rule is added for each of the non-terminals specified in the global definitions section.

Using Lex and Yacc for the assemblers implies that the assembly syntax must be unambiguous with single token lookahead (since parsers generated by Yacc can only look one token ahead). In addition, the fact that the assembler preprocesses the input files with the C pre-processor imposes additional syntax restrictions. In particular, the # symbol (commonly used in most assemblers to denote an immediate data value) may not appear in assembly instructions.

Overall, our system may force the use of a slightly modified assembly syntax for commercial processors. We do not consider this to be a problem since our input will be coming from our compiler, which will generate this modified syntax. The binary file created by assembling this modified syntax is identical to the output of a commercial assembler.

6 Conclusions and Ongoing Work

Having described the details of ISDL, we now compare it to the machine description languages referred to in Section 2. Maril, the language used in Bradlee's MARION system, has most of the required features specified in Section 1.3, but it is limited to RISC architectures and does not support constraints or automatic assembler generation. MIMOLA is too low level for retargeting compilers, and it also 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 architecture design space until an architecture that matches the application at hand is found.

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

References

[1] G. Goossens, F. Catthoor, D. Lanneer, and H. De Man. Integration of Signal Processing Systems on Heterogeneous IC Architectures. In *Proceedings of the 6th International Workshop on High-Level Synthesis*, pages 16–26, November 1992.

[2] F. Depuydt. *Register Optimization and Scheduling for Real-Time Digital Signal Processing Architectures*. PhD thesis, Katholieke Universiteit Leuven, October 1993.

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

[4] A. Kalavade and E. A. Lee. A Hardware–Software Codesign Methodology for DSP Applications. *IEEE Design & Test of Computers*, pages 16–28, September 1993.

[5] D. E. Thomas, J. K. Adams, and H. Schmit. A Model and Methodology for Hardware–Software Codesign. *IEEE Design & Test of Computers*, pages 6–15, September 1993.

[6] R. Cattell. Automatic Derivation of Code Generators from Machine Descriptions. *ACM Transactions on Programming Languages and Systems*, 2(2):173–190, 1980.

[7] A. Aho, M. Ganapathi, and S. Tjiang. Code Generation Using Tree Matching and Dynamic Programming. *ACM Transactions on Programming Languages and Systems*, 11(4):491–516, 1989.

[8] C. W. Fraser, D. R. Hanson, and T. A. Proebsting. Engineering a Simple, Efficient Code-Generator Generator. *ACM Letters of Programming Languages and Systems*, 1(3):213–226, September 1992.

[9] C. W. Fraser and D. R. Hanson. *A Retargetable C Compiler: Design and Implementation*. Benjamin/Cummings Publishing Company, Redwood City, California, 1995. ISBN 0-8053-1670-1.

[10] D. G. Bradlee, R. R. Henry, and S. J. Eggers. The Marion System for Retargetable Instruction Scheduling. In *Proceedings of the 1991 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 229–240, June 1991.

[11] 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.

[12] D. Lanneer, J. Van Praet, A. Kifli, K. Schoofs, W. Geurts, F. Thoen, and G. Goossens. CHESS: Retargetable Code Generation for Embedded DSP Processors. In P. Marwedel and G. Goossens, editors, *Code Generation for Embedded Processors*, chapter 5, pages 85–102. Kluwer Academic Publishers, Boston, Massachusetts, 1995.

[13] 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.

[14] 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.

[15] G. Zimmermann. The MIMOLA Design System: A Computer Aided Digital Processors Design Method. In *Proceedings of the 16th Design Automation Conference*, pages 53–58, 1979.

[16] P. Marwedel. Tree-Based Mapping of Algorithms to Predefined Structures. In *Proceedings of the 1993 IEEE/ACM International Conference on Computer-Aided Design*, pages 586–593, November 1993. Extended version: Technical report 431, Lehrstuhl Informatik XII, University of Dortmund, Germany. January 1993.

[17] G. Goossens, D. Lanneer, M. Pauwels, F. Depuydt, K. Schoofs, A. Kifli, M. Conero, P. Petroni, F. Catthoor, and H. De Man. Integration of Medium-Throughput Signal Processing Algorithms on Flexible Instruction-Set Architectures. *Journal of VLSI Signal Processing*, 9(1):49–65, 1995.

[18] 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.

[19] 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.

[20] 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.

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

[22] J. Levine, T. Mason, and D. Brown. *lex & yacc*. O'Reilly & Associates, Inc., 1992.