Techniques for Accurate Performance Evaluation in Architecture Exploration

George Hadjiyiannis, Srinivas Devadas

Abstract— **We present a system that *automatically* generates a cycle-accurate and bit-true Instruction Level Simulator (ILS) and a hardware implementation model given a description of a target processor. An ILS can be used to obtain a cycle count for a given program running on the target architecture, while the cycle length, die size, and power consumption can be obtained from the hardware implementation model. These figures allow us to accurately and rapidly evaluate target architectures within an architecture exploration methodology for system-level synthesis.**

In an architecture exploration scheme, both the ILS and the hardware model must be generated automatically, else a substantial programming and hardware design effort has to be expended in each design iteration. Our system uses the ISDL machine description language to support the automatic generation of the ILS and the hardware synthesis model, as well as other related tools.

Keywords— **architecture exploration, instruction set, cycle-accurate simulation, hardware synthesis.**

I. Introduction

Embedded systems typically require low cost and low power consumption. To reduce manufacturing cost and power consumption, it is important to match the architecture of the processing engine to the application at hand. A simple way of designing such a processing engine is architecture exploration by iterative improvement (see Figure 1). In this approach, the application code is analyzed, and an initial target architecture is generated and described in a machine description language. The application code is then compiled for this target architecture and executed on an Instruction Level Simulator (ILS) where performance measurements and utilization statistics are gathered. A hardware model of the target architecture is used to derive the length of the cycle and the physical costs (such as die size and power consumption). These measurements allow one to evaluate the architecture and make improvements. A new architecture is generated based on these improvements and the process repeated until no further improvements can be made.

Such a synthesis scheme can only be effective if the design evaluation tools (compiler, ILS, hardware model, assembler and disassembler) can be *automatically* generated from the machine description. Automatic generation of the design evaluation tools allows rapid evaluation of candidate architectures, increasing the coverage of the design space while shortening the design time and, thus, the time-to-market. The machine description language forms the most important part of the system. Ideally, it should support the *automatic* generation of all the design tools rather than being optimized for just one or two. It should also support a wide variety of architectures.

The authors are with the Laboratory for Computer Science, Massachusetts Institute of Technology, 545 Technology Square, Cambridge, MA 02139, USA.

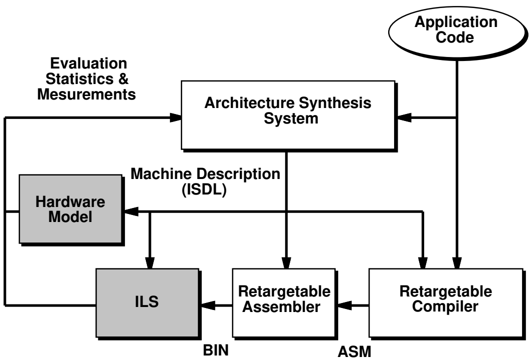

Fig. 1. **Architecture Exploration by Iterative Improvement**

We are developing a system such as the one illustrated in Figure 1 based on the ISDL machine description language[1], [2]. ISDL is specifically designed to support the automatic generation of all the tools including a retargetable compiler. It is also designed to cover as wide a range of *instruction set architectures* as possible, and in particular Very Long Instruction Word (VLIW) architectures. In this paper we focus on the generation of two of the tools from ISDL: the ILS and the hardware implementation model (corresponding to the shaded boxes in Figure 1). The design of the retargetable compiler is covered in [3]. The design of the assembler generator is briefly described in [4].

A. Organization of this paper

Section II presents a brief overview of the ISDL machine description language, with emphasis on the features that make simulator generation and hardware synthesis possible. Section III presents the GENSIM simulator generation system. Section IV presents the methodology we use to synthesize hardware from an ISDL description. Section V presents preliminary experimental results using our system. Section VI presents some previous work in simulator generation and hardware synthesis from machine description languages, and compares these systems to our own, ISDL-based approach.

II. The Instruction Set Description Language (ISDL)

The key component in an architecture exploration system is the machine description language used to describe the candidate architecture to the retargetable design evaluation tools. Our methodology uses ISDL[2] [4] [5], a machine description language that was specifically designed for this task. ISDL is a behavioral language that explicitly lists the instruction set of the target architecture. It is based on an attributed grammar in which production rules are used to abstract common patterns in operation definitions. ISDL attempts to cover a wide range of architectures, and places special emphasis on Very Long Instruction Word (VLIW) architectures.

A. Structure and Syntax of ISDL

ISDL views the processor as a set of state elements (collectively referred to as the *state*) and a set of operations that transform the state. This section provides only a brief description of ISDL, and a short example. For a complete description of ISDL including larger examples refer to [5], [2].

Each ISDL description consists of six sections: format, global definitions, storage, instruction set, constraints, and optional architectural information. Below are detailed explanations of the function of the relevant sections.

A.1 Format Section

The format section divides the binary instruction word into a set of contiguous, non-overlapping fields, each of which is further divided into a set of contiguous, non-overlapping subfields. The first subfield listed forms the MSB of the instruction word and the rest follow in order.

A.2 Global Definitions Section

The global definitions section defines a set of abstractions that are used in later sections of the ISDL description. The two main types of abstractions are tokens and non-terminals:

1. *Tokens* represent the syntactic elements of the assembly language of the architecture. They can also group together syntactically related entities (such as the register names in a register file). Tokens are provided with a return value that identifies the different options.
2. *Non-Terminals* abstract common patterns in operation definitions (e.g., addressing modes). Each non-terminal definition consists of the non-terminal name and a list of options. Each option consists of the same six parts that make up an operation definition (see section II-A.4)[1].

A.3 Storage Section

The storage section of a description explicitly lists all visible storage elements in an architecture. These elements collectively make up the state of the processor. Each storage definition consists of the storage name, the type, and the size. ISDL recognizes the following types: instruction memory, data memory, register file, register, control register, memory-mapped I/O, program counter, and stack. Sizes consist of a width in bits and, for addressed types, a depth in locations. ISDL also allows the definition of aliases. These are alternative names for arbitrary sub-parts of the processor state.

[1] The only differences between a non-terminal option and an operation is that non-terminal options do not have names, and non-terminals have a return value associated with them that behaves like a binary instruction of varying width.

A.4 Instruction Set Section

The *Instruction Set* section lists all of the operations available on the target processor. It groups the operations into mutually exclusive sets called *Fields*. Each field roughly corresponds to the operations that can be performed on a single functional unit. A VLIW instruction consists of a group of operations, one from each field.

The instruction set section consists of a list of field definitions. Each field definition consists of a number of operation definitions. Each operation definition consists of the following elements:

- **Syntax:** This declares the assembly syntax of the operation. It consists of an operation name followed by a list of parameters, each of which is the name of a token or a non-terminal.
- **Bitfield Assignments:** The bitfield assignments define the assembly function for the operation. The assignments are a set of statements that assign the appropriate binary values to the subfields defined in the instruction word format section. The bitfield assignment statements may make use of the return values of the tokens and non-terminals in the operation's parameter list.
- **RTL Action:** This describes the effect of the operation on the processor state using an RTL type language. It may make use of the return values of tokens and the RTL action clause of non-terminals appearing in the operation's parameter list.
- **RTL Side Effects:** This describes any side effects of the operation using the same RTL language as the RTL action description. It may make use of the return value of tokens and the RTL side effects clause of non-terminals in the parameter list.
- **Costs:** Multiple costs are permitted including operation execution time, code size, costs due to resource conflicts, etc. ISDL predefines three cost parameters:

1. `Cycle` declares the number of cycles that the operation requires to execute on the hardware.
2. `Size` declares the number of instruction words needed to represent the operation.
3. `Stall` declares the number of stall cycles that will be inserted if the next instruction attempts to use the results of the operation.

The cost parameters are defined as a set of arithmetic expressions resulting in numerical values. The arithmetic expressions can include arithmetic operators (i.e., `+`, `-`, `*`, `/`, and `%`) and relational operators (e.g., `==`, `<`, `>`). They may also use the return value of tokens and the cost modifiers of the non-terminals in the operation's parameter list. Furthermore, the arithmetic expression may use the cost clauses of operations in other fields and the values of storage references.

- **Timing:** The timing parameters describe when the various effects of the operation take place. ISDL predefines two timing parameters:

1. `Latency` specifies the number of instructions (including the one containing the current operation) that must be fetched before the results of the current operation become available.

2. `Usage` specifies the number of instructions (including the one containing the current operation) that must be fetched before the corresponding functional unit becomes available again.
The timing clauses are complex arithmetic expressions resulting in numerical values. The timing arithmetic expressions obey the same syntax and semantics as the cost clauses.

A set of examples can better illustrate the use of the costs and timing parameters to describe the effect of pipelines:

- **Case 1:** Consider a simple architecture with no pipelining where each instruction completes before the next one is fetched and requires three cycles to do so. Thus, `Cycle` = 3 and `Stall` = 0 because it is not possible for any operation to stall (no pipeline). `Latency` = 1 because the next instruction can use the results of the current operation.
- **Case 2:** Consider an architecture with a simple four-stage pipeline with no protection (no bypass logic or stall capability). The architecture can execute one instruction per clock cycle for all instructions. Thus, `Cycle` = 1 because each instruction effectively takes one clock cycle. `Stall` = 0 since stalls are not necessary in an unprotected pipeline. `Latency` = 4 because the current instruction plus three additional instructions must be fetched before another instruction can use the results of the current operation.
- **Case 3:** Consider an architecture with a four-stage pipeline with full protection (bypass logic). The architecture can execute one instruction per clock cycle for all instructions. Thus, `Cycle` = 1 since each instruction effectively takes one clock cycle. `Stall` = 0 since stalls are not necessary because of the bypass logic. `Latency` = 1 because the bypass logic guarantees that the results will be available to any subsequent instruction.
- **Case 4:** Consider an architecture with a four-stage pipeline protected by bypass logic for all operations except loads. Loads are protected by a single stall cycle. For the load instruction `Stall` = 1 because one cycle will be inserted if the next instruction attempts to use the results of the current operation. `Cycle` = 1 since all instructions effectively take one instruction, except for the case of a load with a stall which was already taken into account. Finally, `Latency` = 1 since all operations (including loads) are protected, thus the next instruction will be able to use the results of the current operation.
- **Case 5:** Finally, consider an architecture with a four-stage pipeline and a branch-if-zero instruction which flushes two stages of the pipeline if the branch is taken. The effect of the pipeline flush would appear as an additional two cycles added to the `Cycle` cost of the operation if the branch is taken. Therefore, `Cycle` = $(R == 0)*2+1$, `Latency` = 1 since the next operation will be fetched from the target of the branch, and `Stall` = 0. The `Cycle` cost is 1 if register R is not zero (i.e., the branch is not taken) and 3 if R is zero (i.e., the branch is taken).

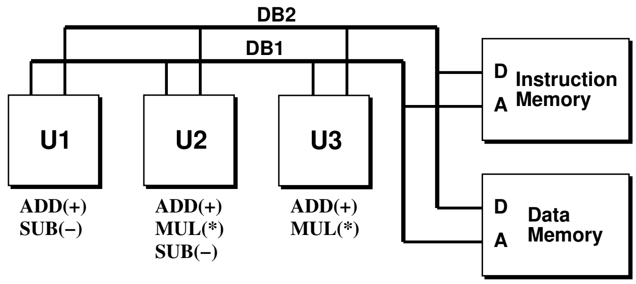

Fig. 2. **The SPAM VLIW-1 Architecture**

A.5 Constraints Section

In ISDL an instruction is formed by grouping together operations, one from each field. Not all such combinations are valid. The constraints section describes the valid combinations by listing a set of constraints which must *all* be satisfied by each instruction in order for the instruction to be considered valid. If a single constraint is violated then the instruction is invalid.

Constraints are completely independent of each other. The presence or absence of a constraint, generally speaking, declares different architectures. Constraints therefore represent another dimension in ISA exploration.

Constraints allow all operation definitions to be treated as orthogonal throughout the description resulting in much more concise and intuitive descriptions. Constraints can also provide information about the underlying implementation of the instruction set, thus helping to generate efficient hardware.

B. ISDL Example

An extended example based on the simple architecture of Figure 2 is used to better illustrate the features of ISDL. It is a VLIW architecture with three functional units U1, U2, and U3. Each functional unit has its own register file consisting of four 16-bit registers. The architecture also includes a data memory of 4096 16-bit locations and an instruction memory capable of storing 4096 44-bit instructions. The register files and the two memories are connected through two buses: DB1 and DB2. This architecture can perform three data operations and two data transfers in parallel. In this processor, U1 can perform addition and subtraction, U2 can perform addition, subtraction, and multiplication, and U3 can perform addition and multiplication.

The instruction word for the example architecture is shown in Figure 3. Each of the functional units has its own field in the instruction word. Each field consists of an opcode, two source register identifiers, and one destination register identifier. Each of the buses also has its own field in the instruction word consisting of the databus source and destination identifiers.

The Format section for this example architecture is shown below. It describes the components of the instruction word.

```
Section Format
```

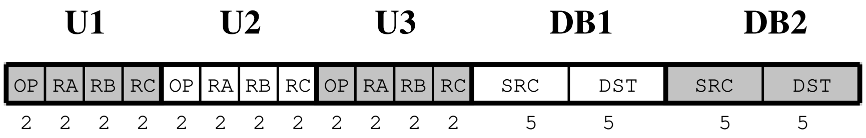

Fig. 3. **The Instruction Word of the SPAM VLIW-1 Architecture**

```
U1      = OP[2], RA[2], RB[2], RC[2];
U2      = OP[2], RA[2], RB[2], RC[2];
U3      = OP[2], RA[2], RB[2], RC[2];
DB1     = SRC[5], DEST[5];
DB2     = SRC[5], DEST[5];
```

This description specifies that the instruction word is divided into five fields `U1`, `U2`, `U3`, `DB1`, and `DB2`. Each field is further divided into subfields, and each subfield is annotated with its length in bits. The concatenation of each subfield in MSB (most significant bit) to LSB (least significant bit) order results in the instruction word shown in Figure 3.

The following is the complete Storage section description for the example architecture.

```
Section Storage

Instruction Memory INST          = 0x1000 x 0x2C
Memory DM                        = 0x1000 x 0x10
RegFile U1                       =    0x4 x 0x10
RegFile U2                       =    0x4 x 0x10
RegFile U3                       =    0x4 x 0x10
ProgramCounter PC                =          0x10
```

Each of the storage units (memories and register files) is explicitly listed along with the number of entries it contains and the width of each entry. For individual registers, such as the Program Counter, the size describes the width of the register in bits. Note that the instruction memory is explicitly identified, and that the program counter must be included even though it is implied by the instruction set.

A sample token definition, which is part of the Global Definitions section, is presented below:

```
Section Global_Definitions

//    assembly        token       value
Token "U1.R"[0..3]    U1_R        { [0..3]; };
...
```

The line beginning with the keyword `Token` defines a token that groups the syntactic entities `U1.R0`, `U1.R1`, `U1.R2`, and `U1.R3` as denoted by the assembly syntax declaration `"U1.R"[0..3]`. These are actually the names of the registers in the register file of unit U1. The token is named `U1_R` and can be referred to in non-terminal and operation definitions using that name. The return values are zero, one, two, and three respectively (i.e., they are the index of the corresponding register) as denoted by the return value entry {`[0..3];`}. In the full description of the example architecture, two additional tokens exist that define the register names for the other two register files in the same manner.

The following set of non-terminal definitions are also part of the Global Definitions section:

```
Non_Terminal U1_RA: U1_R { $$ = U1_R; }
                    {U1[U1_R]} {} {} {} ;
Non_Terminal U1_RB: U1_R { $$ = U1_R; }
                    {U1[U1_R]} {} {} {} ;
Non_Terminal U1_RC: U1_R { $$ = U1_R; }
                    {U1[U1_R]} {} {} {} ;
...

Non_Terminal SRC:
        U1_R { $$ = 0x00 | U1_R; }
                    {U1[U1_R]} {} {} {} |
        U2_R { $$ = 0x04 | U2_R; }
                    {U2[U2_R]} {} {} {} |
        U3_R { $$ = 0x08 | U3_R; }
                    {U3[U3_R]} {} {} {} ;

Non_Terminal DEST:
        U1_R { $$ = 0x00 | U1_R; }
                    {U1[U1_R]} {} {} {} |
        U2_R { $$ = 0x04 | U2_R; }
                    {U2[U2_R]} {} {} {} |
        U3_R { $$ = 0x08 | U3_R; }
                    {U3[U3_R]} {} {} {} ;
```

The first line defines a non-terminal named `U1_RA`. This name can be used to refer to it in operation and other non-terminal definitions. This non-terminal consists of a single option. The single token `U1_R` forms the assembly syntax for this option. The return value of this option, which can be used in the bitfield assignments of operations, is the same as the return value of the token as denoted by the statement { `$$ = U1_R;` }. The RTL action corresponding to this non-terminal is simply a reference to the appropriate storage location as denoted by the RTL action statement {`U1[U1_R]`}. It specifies that the non-terminal refers to a register in register file `U1` indexed by the return value of the `U1_R` token. The next set of braces contains the RTL side effects of the non-terminal. An empty side effects statement denotes that there are no side effects. The next two sets of braces contain the costs and timing modifiers of the non-terminal option. Undefined cost or timing modifiers imply a value of zero.

The next two lines define non-terminals identical to U1_RA except that they are named differently. The reason for defining identical non-terminals with different names is so that they can be distinguished when used in the same operation definition.

The next non-terminal defined is named SRC and consists of three options. The syntax of the first option is U1_R which represents the registers in the U1 register file. Similarly, the second and third options represent the registers of register files U2 and U3. This non-terminal can be used to represent a register in any of the three register files. The return value of the non-terminal is defined as follows: for registers in the U1 register file, the return value is the constant 00 concatenated with a two bit value representing the index of the register, for registers in the U2 register file the return value is the constant 01 concatenated with the value representing the index of the register, and for registers in the U3 register file the return value is the constant 10 concatenated with the value representing the index of the register. The RTL action for each non-terminal option is a reference to the appropriate storage location, and there are no side effects or costs and timing modifiers specified. An identical non-terminal named DEST is also defined for use in conjunction with the SRC non-terminal when describing databus move operations.

Below is a portion of the Instruction Set section for the example architecture:

```
#define ADDm(x,y)    ADD(x,y,16,"trn")

Section Instruction_Set

Field U1f:
  U1_add  U1_RA, U1_RB, U1_RC
    { U1.OP = 0x0; U1.RA = U1_RA;
        U1.RB = U1_RB; U1.RC = U1_RC; }
    { U1_RC <- ADDm(U1_RA,U1_RB); }
    { }
    { Cycle = 1; Size = 1; Stall = 0; }
    { Latency = 1; Usage = 1; }
  ...
Field U2f:
  ...
Field U3f:
  ...
```

This section defines the three functional unit operations, and the memory and databus operations. The functional unit definitions consist of three field definitions, one for each functional unit. Each field lists all of the operations that the corresponding functional unit supports. For brevity, a single operation, namely an add on unit U1, is presented. The syntax of the operation is shown on the first line of the operation definition. It consists of the operation name U1_add followed by a list of three register names denoted by the non-terminals U1_RA, U1_RB, and U1_RC as parameters. The following is an example of an operation of this type:

```
U1_add U1.R0, U1.R2, U1.R2
```

The first set of braces in the operation definition contain the bitfield assignments (i.e., the bits assigned to the various subfields in the instruction word to denote this operation). In this case, the subfields of the U1 field are assigned the following values: the OP subfield is assigned the value 0 which is the opcode for the add operation, and the RA, RB, and RC subfields are set to the return values of the corresponding non-terminals. These are actually the indices of the corresponding registers in the register file.

The next set of braces contain the action of the operation in RTL. For this operation, the value of the register corresponding to the first parameter is added to the value of the register corresponding to the second parameter, and the result is stored in the register corresponding to the third parameter of the operation[2].

The third set of braces describe the side effects of the operation. An empty side effects statement denotes that there are no side effects. Note that in ISDL, the program counter, PC, is implicitly incremented in order to fetch the next instruction and thus does not appear in the operation side effects statement. Also note that explicit manipulation of the program counter denotes a control flow operation.

Finally, the costs and timing parameters of the operation are provided. This add operation takes one cycle to execute, and requires at most one instruction word. This operation will not introduce any stall cycles if a subsequent instruction attempts to access the result of the current operation. The results of this operation are available to all subsequent operations, and functional unit U1 is immediately available to perform another operation.

A load operation from the data memory field is shown below:

```
#define DMdata 0x0C
#define DMaddr 0x0D
#define REG SRC
#define LOC DEST
Field DMf:
  DM_ld  REG, LOC
    { DB1.SRC = DMdata; DB1.DEST = REG;
      DB2.SRC = LOC; DB2.DEST = DMaddr; }
    { REG <- DM[LOC]; }
    {}
    { Cycle = 1; Size = 1; Stall = 1; }
    { Latency = 1; Usage = 1; }
```

The main difference to note in this operation definition is the Stall cycle cost. A value of one means that if the next instruction attempts to use the result of the load operation, then the pipeline will be stalled for one cycle and the cycle count for this operation must be increased by one.

[2] The reference U1_RC inside the RTL action of the operation description refers to the RTL value of the non-terminal U1_RC. This is given by the corresponding RTL action of the non-terminal definition - in this case a reference to the appropriate register.

Finally, a portion of the Constraints section of the example VLIW architecture description is shown below:

```
Section Constraints

// SRC and DEST cannot be the same on either bus
~( DB*_move U@[1].R*, U@[1].R* )
```

The third line declares a constraint that is violated if the instruction contains a databus move operation on either bus (represented by `DB*_move`), *and* the source and destination come from the same register file (represented by `U@[1].R*, U@[1].R*` where `@[1]` is a variable that must match in both its instances). This is not possible to execute in hardware since each register file only has one port attached to each databus. Therefore, a constraint is used to disallow such an operation.

III. The Gensim Simulator Generator

In order to be able to evaluate the suitability of a candidate architecture for a particular application, it is necessary to be able to simulate the program on the candidate architecture. This makes it possible to verify performance, determine the utilization of individual architecture features and functional units, and suggest possible improvements to the architecture.

We present a tool called the Gensim system, that automatically generates an Instruction Level Simulator given an Isdl description of a candidate architecture. This simulator (called an Xsim simulator) can then be used to execute a program in order to measure performance, verify correctness and evaluate the suitability of the architecture.

A. Simulator Features

The Xsim simulators are cycle-accurate and bit-true by construction. They also provide fast execution times and perform disassembly off-line to improve speed. They provide both a graphical user interface and a command-line interface with full batch-file support. They also provide full debugging support (e.g., breakpoints, state monitors and attached commands). Finally, they can create an execution address trace which is either written into a file or directly to a processing program.

These features make it possible to use the Xsim simulators for detailed evaluation of candidate architectures. At the same time, they make the simulators easy to use both manually as well as automatically.

B. Simulator Structure

Figure 4 shows the structure of an Xsim simulator. The simulator consists of six parts:

1. **User Interface and File I/O:** This part implements both the command line interface as well as the graphical interface. It also implements the interfaces to the operating system and the file-system of the underlying platform.
2. **Scheduler:** The scheduler is responsible for sequencing the instructions during execution, managing breakpoints, dumping the execution traces to a file or processing program, and dispatching attached commands back to the user interface for processing.

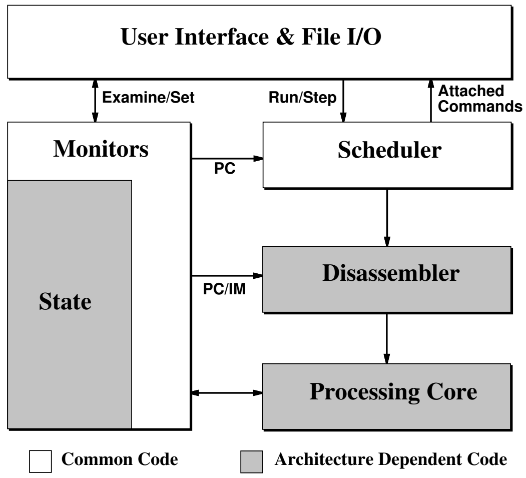

Fig. 4. **Internal Structure of the Xsim Simulators**

3. **State Monitors:** These provide a set of hooks that can detect whenever any user-defined portion of the state changes, and print a diagnostic message to that effect.
4. **State:** This is a set of data structures that emulate the state of the target architecture.
5. **Disassembler:** The program to be simulated must be disassembled in order to determine which operations correspond to each input instruction. The simulator contains a built-in disassembler which disassembles the program off-line at load time.
6. **Processing Core:** Each operation and Isdl non-terminal option have an RTL action (and an RTL side-effect) associated with them. These get translated to a set of routines that emulate those actions. The processing core consists of the collection of these routines.

C. Simulator Generation

All of the simulator code is written in C, with the exception of the graphical user interface which is written in Tcl/Tk. The user interface, state monitors, and scheduler code is common to all architectures and is implemented as a library. The state data structures, disassembler, and processing core routines are specific to each architecture and are generated as C source code from the Isdl description. The C source can then be compiled and linked with the common library to create an executable program for the simulator. This executable is specific to an architecture but can load different programs for the same architecture (unlike compiled code simulators).

The following three sections describe how we generate the state, disassembler and processing core of an Xsim simulator.

C.1 State Generation

Generating the state data structures is a simple matter of allocating sufficient memory for each storage element defined in the Isdl description, and copying the rest of the information in the definition (such as the width, depth,

Field 1	I_{19}	I_{18}	I_{17}	I_{16}	I_{15}	I_{14}	I_{13}	I_{12}	I_{11}	I_{10}	I_9	I_8	I_7	I_6	I_5	I_4	I_3	I_2	I_1	I_0
op1 a, b	1	x	x	x	a'_7	a'_6	a'_5	a'_4	b'_7	b'_6	b'_5	b'_4	b'_3	b'_2	b'_1	b'_0	a'_3	a'_2	a'_1	a'_0
op2	0	1	x	1	1	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
op3 c, a	0	0	0	x	a'_7	a'_6	a'_5	a'_4	a'_3	a'_2	a'_1	a'_0	c'_7	c'_6	c'_5	c'_4	c'_3	c'_2	c'_1	c'_0

Fig. 5. **Operation Signatures**

and type) to the data structure. All accesses to state are automatically routed through the monitors code.

C.2 Disassembler Generation Algorithm

The ISDL bitfield assignments provide the *assembly function*. This is a function that, for a given operation (or non-terminal option) and a given set of parameters, provides the values of the relevant bits of the instruction word. In order to generate a disassembler we need to *reverse* this function (i.e., given the values of the bits in the instruction word we must first identify the operation and then provide the values of the parameters).

To derive the disassembly function from the bitfield assignments of a description we use the following model (see figure 5):

We associate with each operation in every field a *signature*. This is an image of the instruction word with symbols entered into each bit. The following symbols are used:

- "Don't care" entries (represented by an "x") imply that the assembly function for this operation does not set the corresponding word bit.
- The constant "0" or "1" implies that the assembly function for this operation sets the corresponding bit to the given constant.
- A parameter symbol (such as "a'_0") implies that the assembly function for the operation sets the corresponding bit to a *function* of the value of one of the parameters.

Our methodology is based on the following axiom:

Axiom 1: Each parameter symbol in a signature is a function of a *single* parameter only.

What this axiom means is that no bit in the instruction word will be a function of more than one parameter and therefore no back-tracking will be necessary in order to decode the parameters. All architectures known to us obey the parameter decode axiom. It is easy to see why this axiom would hold for any realistic architecture:

- Encoding parameters in a fashion that violates the parameter decode axiom does not confer any advantages in terms of performance and rarely confers any advantage in terms of code size since most parameters are powers of 2 and will fit exactly in a given number of bits.
- Back-tracking in the disassembly algorithm translates to very expensive exponential lookup tables in the decode logic.

Given the signature of each operation in a field and the axiom above we can reverse the assembly function as follows:

We attempt to match the constant part of the signature for each operation against the current instruction word. The match is guaranteed to be unique for a decodeable assembly function[3]. We can then reverse the encoding of each parameter symbol bit in the instruction word to obtain the original parameter value. The axiom above guarantees that the encoding is reversible. Most of the time the encoding can be reversed symbolically (i.e., dealing with multiple bits at the same time). Figure 6 shows this algorithm in pseudo-code.

```
Generate signatures for each operation in each field
Generate signatures for each option in each non-terminal

disassemble(i)
    for each f in description
        disassemble_field(i, f)
    end

disassemble_field(s, f)
    for each operation o in f
        if signature of o matches s
            for each parameter p in o
                case (p)
                    token: reverse s to get token value t
                    non-terminal: reverse s to get return value r
                        disassemble_ntl(r, p)
            end
        return OK
    end
    return ILLEGAL INSTRUCTION

disassemble_ntl(s, n)
    for each option o in n
        if signature of o matches s
            for each parameter p in o
                case (p)
                    token: reverse s to get token value t
                    non-terminal : reverse s to get return value r
                        disassemble_ntl(r, p)
            end
        return OK
    end
    return ILLEGAL INSTRUCTION
```

Fig. 6. **Disassembly Algorithm**

Note that neither the assembly nor the disassembly function are complete (valid for all inputs). However, the constraints describe invalid inputs to the assembly function, while invalid inputs to the disassembly function are allowed to result in undefined behavior since they should never occur in a valid program.

C.3 Processing Core Generation

The processing core is merely a collection of routines that correspond to the RTL statements in the description. These RTL statements are translated to C functions that carry out the actions described in the RTL. These functions are then compiled into the processing core as a collection of routines, and get called by the scheduler when the instruction gets executed. However, there is a certain amount of book-keeping that needs to be done in order to guarantee bit-true, cycle-accurate results. First of all, we must ensure that *all* RTL statements read their input values before *any* RTL statement writes its results. This

[3] Note that the number of matches that need to be performed is at most the number of operations in the instruction field. Therefore the number of matches grows linearly with respect to the size of the original ISDL description and can never be too large.

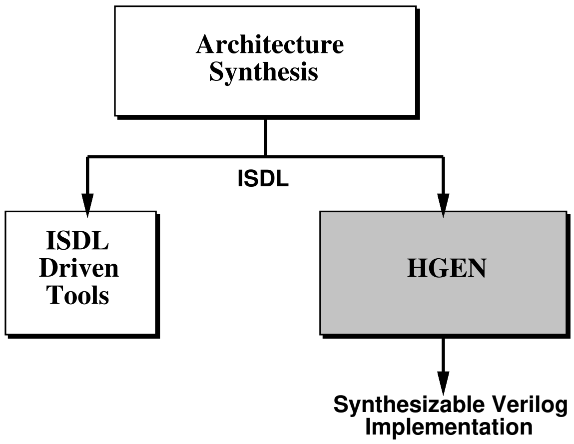

Fig. 7. **Hardware Synthesis from ISDL**

is achieved by dividing the cycle into two distinct phases. During the first phase, all RTL statements read their values from the state and write their results into temporary storage. During the second phase, the temporary storage is written back to state. Furthermore, we must ensure that write-backs to state are delayed by the right number of cycles (determined by the `Latency` parameter of ISDL). Also, the RTL for side-effects conceptually takes place *after* the RTL for actions (while still in the same cycle). To achieve this we divide the evaluation phase into an *action evaluation phase* and a *side-effects evaluation phase*. Finally, to ensure cycle accuracy, we need to take into account the stall cycles. Since there is no explicit model of a pipeline in ISDL, there is no model of the pipeline in the simulator either. Instead, stall cycles are computed from the static instruction stream and are added to the normal cycle count as needed.

The order in which the RTL statements are executed is dictated by the original statements in ISDL. The overall timing of the sequence of statements is computed from the timing parameters in the ISDL description.

IV. The HGEN Hardware Synthesis System

In order to fully evaluate an architecture we need to obtain an estimate of its physical costs (e.g., silicon area or power consumption). At the same time, while the ILS provides performance measurements in terms of cycles, the length of the cycle is still necessary to obtain an accurate measure of performance. Both the cycle length and the physical costs can be determined by synthesizing a hardware model for the architecture. We consider a description of the architecture in synthesizable Verilog to be a sufficient hardware model. This description can then be used to map to any kind of underlying technology using modern CAD tools (silicon compilers).

A. *Hardware Synthesis from* ISDL

In our methodology, the architecture synthesis system produces instruction sets instead of architectures[4]. The output of the architecture synthesis system is an ISDL description, possibly with some implementation-specific details (such as timing information) missing. This ISDL description is used to drive both the ISDL-based evaluation tools, and an ISDL-to-hardware compiler (called HGEN). The output of the HGEN compiler is synthesizable Verilog which can then be used to create a hardware implementation in any kind of underlying technology.

The above methodology only uses a single description avoiding consistency issues. Also, the granularity at which changes can be made is much finer[5] than methodologies based on parameterized architectures, which makes architecture exploration much more effective. Finally, the design of the instruction set is decoupled from the design of the hardware implementation providing an additional degree of freedom in finding good solutions. The main disadvantage of this approach is that it is subject to the resource sharing problem which is described in section IV-A.1. This approach makes hardware generation more complex, but results in easy-to-write ISDL descriptions.

We feel that direct synthesis from ISDL has compelling advantages, including the fact that it will benefit more from improvements in other CAD tools (such as silicon compilers). Also, the resource sharing problem can be solved using a combinatorial optimization strategy.

A.1 The Resource Sharing Problem

The scope of each ISDL operation definition is independent of the scope of any other operation definition. This makes it non-trivial to deduce when hardware resources may be shared by multiple operations.

Consider a `move` operation that is implemented using a bus, and `load` and `store` operations that are mutually exclusive with the `move`. Additionally, the `move` operation resides in a different field than the `load` and `store` operations. A naive scheme would generate additional datapaths to handle the `load` and `store` operations even though it is possible to implement these with the same bus that implements the `move`.

A.2 Identifying Shared Resources

We have formulated a way of solving the resource sharing problem to allow ISDL-based hardware synthesis to be used efficiently. First we break up the RTL expressions for all operation definitions into a number of nodes, each of which can be mapped to a circuit. This collective set of nodes (let us say n nodes in total) is numbered with unique numbers from 1 to n. Then we create an $n \times n$ matrix A, with entries that are 1 or 0. A_{ij} is 1 if the nodes can be shared (i.e., they would never operate at the same time), and 0 if they cannot (because they have to operate in parallel). To determine

[4] Architectures can be thought of as implementations of the instruction set.
[5] Individual changes are made at the level of an RTL operation.

```
Label each operation in RTL with an integer

    for each i from 0 to n
      for each j from 0 to n
        A_ij = 0
        if i and j not in same operation
          if i and j functionally equivalent
            if i and j in operations in same field
            or constraint between i and j
              A_ij = 1
      end
    end

Generate maximal cliques for A
Generate hardware for maximal cliques
```

Fig. 8. **Resource Sharing Algorithm**

the entries in the matrix we can use the following set of criteria:

1. Nodes in the same RTL statement cannot be shared.
2. Nodes performing different tasks (e.g., a shift and an AND operation) cannot be shared. Pairs where one node is a subset of another (e.g., an `add` is a subset of a `sub`tract) *can* be shared assuming that the rest of the rules do not prevent it.
3. Nodes belonging to operations in the same field (or to options in the same non-terminal) will *never* be active at the same time so they can be shared.
4. Nodes that belong to operations in different fields will probably have to operate in parallel so they cannot be shared.

In addition to the above, constraints may be able to determine even more nodes that cannot operate in parallel (from Rule 4 above), so more sharing may be available if we take constraints into account.

Once we have the entries in the matrix, we can simply create maximal cliques[6] of the nodes that can be shared. These maximal cliques are then synthesized into circuits and the routing and glue logic is generated to complete the implementation. Figure 8 shows this algorithm in pseudocode.

The user can influence how much sharing exists by adding or deleting constraints. Thus, he/she can provide hints to the system. For example, if the user knows that bus transfers and memory operations will never occur in parallel, he/she can enter a constraint to that effect, and HGEN will use the bus to effect memory operations.

A.3 Obtaining Structural Information from ISDL

Although ISDL is a behavioral language and it contains no explicit structural information, a substantial amount of information about the structure of the underlying architecture can be extracted from various parts of the description. In particular, the costs and timing information exposes the underlying data-path pipelines to the instruction set. For example, an operation with a `Cycle` cost of 1, a `Stall` cost of 3, and a `Latency` of 1 implies a 4-stage data-path pipeline for the functional unit. Additionally, it implies no bypass logic for this particular operation. Similarly, an operation with a `Cycle` cost of 1, a `Stall` cost of 0, and a `Latency` of 1 implies a similar pipeline with full bypass logic. Similarly, the constraints express hardware restrictions and can therefore be used to deduce the structure of the underlying hardware. Consider the example described in IV-A.1. In this example we can connect the memory to the same bus as the `move` operation and avoid creating a new set of data paths for the `load` and `store` operations.

All operations in a field share the same pipeline, generally speaking. However, some results may be produced before others. For example, a MAC might have two pipeline stages (one for the multiply and one for the accumulate). One could use the same pipeline to do a multiplication and pull it out early using bypass logic; in effect, this behaves as a pipeline of length one for pure multiplies, but a pipeline of length 2 for MAC operations. We do not perform any consistency checks for timing of operations within a field for this reason, even though in most cases, the timing of operations within a field will be the same.

B. *Generating Decode Logic*

Note that there is a direct relationship between the disassembler generated for the GENSIM system and the decode logic to be used in hardware[7]. They both implement the same function (reversing the assembly function). In the disassembler decoding is done in software using `case` or `if` statements, in the hardware generator, it is done using logic gates.

We can therefore generate a complete implementation of the decode logic using the same approach we use to generate the disassembler for the GENSIM system. The process is as follows:

For each operation in a field we define a decode line which will be active if the operation is instantiated in the current instruction. We can then derive an equation for each decode line by simply examining the constants in the operation signature. For example, the equation for the operation `op2` in Figure 5 is $\bar{I}_{19}.I_{18}.I_{16}.I_{15}$. This results in an efficient two-level implementation. Similarly, logic can be generated from the decode functions that reverse parameter encodings. Finally a set of multiplexers and glue logic completes the decode circuit.

V. Experimental Results

This section presents some experimental results that were obtained using our system and analyzes these results to demonstrate the feasibility of the proposed process and the effectiveness of ISDL and the implemented tools.

A. ISDL *Descriptions*

A number of architectures were described in ISDL in order to prove the power and flexibility of the language as

[6] A clique is a set of nodes such that for any pair of nodes i and j in the clique, $A_{ij} = 1$. A maximal clique is a clique such that if any node is added to the clique, the resulting set of nodes is no longer a clique.

[7] In fact there is a very strong relation between generating a simulator and a hardware model: the synthesizable Verilog model is itself a simulator.

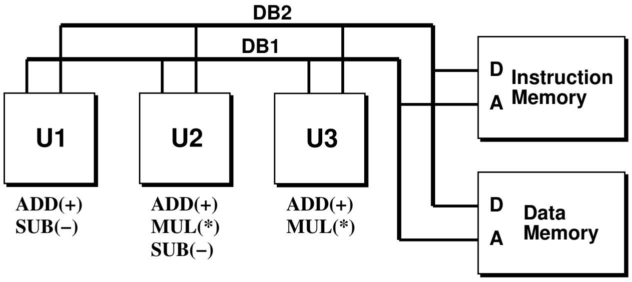

Fig. 9. **The SPAM VLIW-1 Architecture**

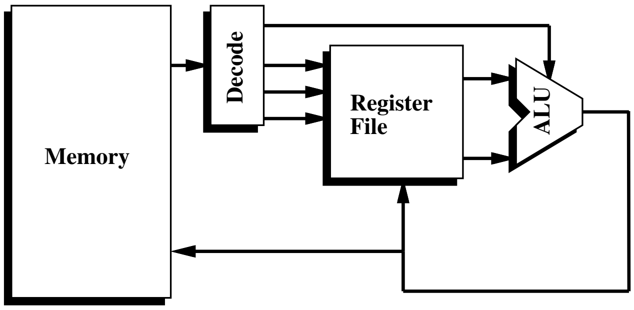

Fig. 11. **The SPAM RISC Architecture**

well as to serve as experiment platforms for the results presented in Sections V-B and V-C. These four architectures are:

1. **SPAM VLIW-1:** The VLIW-1 architecture is a 5-way VLIW architecture with three data processing units and two transfer buses. In addition, the architecture provides a separate instruction and data memory, either of which may be used to store data. Each data processing unit is very simple and can do a limited number of operations. Each data processing unit also contains an internal register file of 4 registers. The two transfer buses can be used in parallel to execute two register-file-to-register-file transfers in parallel, or can be used together to perform either a data memory transfer or an instruction memory transfer. In the case of memory transfers, one bus provides the address to the corresponding memory and the other performs the actual transfer. All data-paths are 16 bits wide and the instruction word is 44 bits wide. A block diagram of this architecture is shown in Figure 9.

2. **SPAM VLIW-2:** The VLIW-2 architecture is a 7-way VLIW architecture. It consists of a floating-point data processing unit, an integer data processing unit, a data memory for each unit, an address generator of each data memory, a transfer bus, an external interface unit, and a control unit. The control unit has a separate instruction memory and a hardware stack. The floating point unit contains its own own register file of 16 registers and an internal accumulator register. It is capable of all common floating point operations. The integer data processing unit can perform any arithmetic or logical operations and contains its own register file with 32 registers. Each address generator contains a dedicated register file with 8 registers and can perform simple indexing and incrementing operations. The two data memories can transfer data either over the bus or directly to their respective processing unit. If the data bus is not being used by a data memory, it can be used to transfer data between the registers files of the various or units. All data paths are 32 bits wide and the instruction word is 99 bits wide. A block diagram of this architecture is shown in Figure 10.

3. **SPAM RISC:** The RISC architecture is a small architecture implemented in the same spirit as most existing RISC processors (e.g., the Sun Microsystems SPARC processor). It contains a single data processing unit and a register file containing 32 registers. Register 31 is used to store the PC on calls and register 30 is by convention the stack pointer. The architecture supports all common data processing operations. All data-paths are 24-bits wide and the instruction word is 24-bits wide also. A block diagram of this architecture is shown in Figure 11.

4. **56000:** This is a description of the Motorola 56000 series DSP. It contains two 56-bit accumulators and two data registers each of which may be used as two 24-bit registers or one 48-bit register. It supports two data memories, each of which is 24-bits wide and which can be combined to a single 48-bit wide memory. It also contains three register files each of which contains 8 registers; these are used to support a rich set of addressing modes. The architecture supports common DSP data processing operations and up to two transfers in parallel with a data operation. The data paths vary in width according to the operation being executed and can be 24, 48, or 56 bits wide. The instruction is 24 bits wide. A block diagram of this architecture is shown in Figure 12.

Table I summarizes the main features of the descriptions. In the table, heavy opcode encoding implies that the the opcode for the addressing mode for each memory is combined with the opcode for the corresponding address generator in order to generate the final value to be inserted in the binary instruction. Op-code or field precedence in the table refers to an operation $o1$ from field X taking over bits belonging to the op-code of an operation $o2$ in another field Y to store large constants. Another issue that seems to be common in many DSP processors is that of register aliasing. For example in the Motorola 56000 DSP, the `X` register is 48 bits long but can also be accessed as two separate registers called `X0` and `X1`.

These four architectures represent a broad spectrum ranging from simple RISC style machines to commercial DSP processors, to highly-aggressive VLIW designs. Each architecture listed above was also used to generate several modified designs that were used to obtain the experimental results presented in Sections V-B. A number of the modified versions of the above architectures (except the 56000 which is already pipelined) were pipelined and provided with a range of timing-related features ranging from unprotected pipelines, to bypass logic, to stalls and delay slot instructions. This shows that ISDL is capable of describing a wide variety of architectures, both in terms of the classes of architectures it will allow and in terms of the features within each architecture.

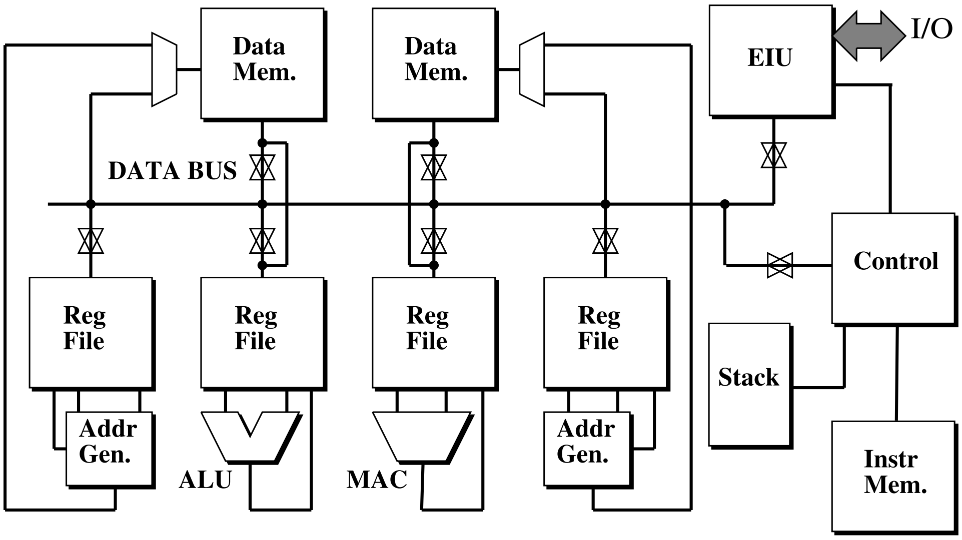

Fig. 10. **The SPAM VLIW-2 Architecture**

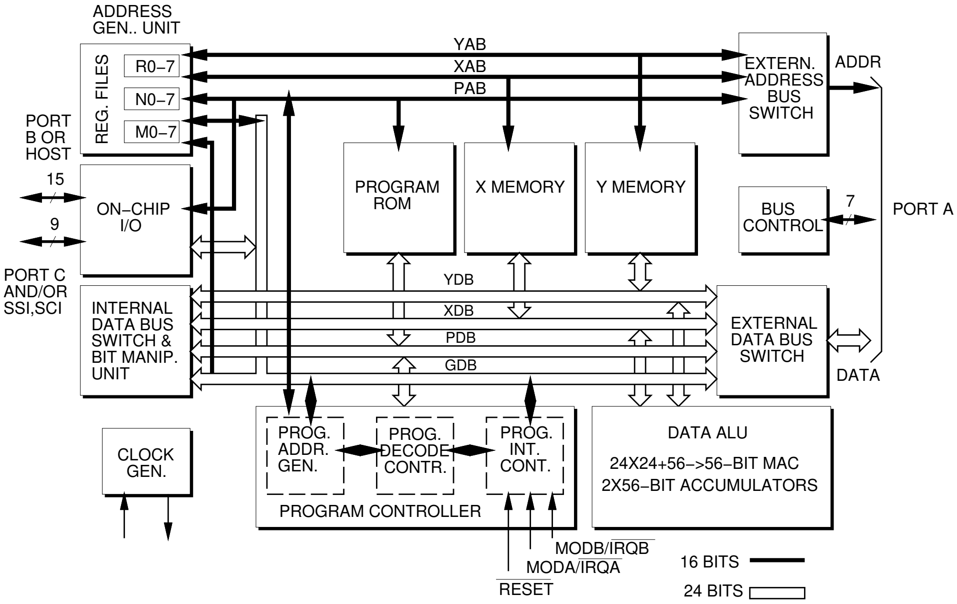

Fig. 12. **The Motorola 56000 Architecture**

Note that most architectures resulted in particularly concise descriptions ranging between 300 to 900 lines. The exception is the Motorola 56000 description. This architecture has a very heavily optimized instruction set with a very large number of operations; in fact the encoding of the instruction set appears to have been Huffman-encoded to allow as many operations as possible to be included in the 24-bit word. Additionally, many operations have multiple formats which are expressed as different operation definitions in the ISDL description. The result is a very large number of operations resulting in a large description. Furthermore, the 56000 description makes very heavy use of side-effects statements to set the condition code bits, with multiple such statements occurring in each operation definition. Note, however, that despite the large size of the description, it was developed in about 36 man-hours. This illustrates how easy ISDL descriptions are to develop.

Furthermore, several modified versions of each architecture (except the 56000) were created. Modifications ranged from the addition of a few operations, to the addition of up to 8 times the existing number of operations. Some modifications also involved a complete change in timing model for the associated architecture. Once again, modifications were simple to implement and local to each operation affected. In the cases where new operations created resource conflicts with existing operations, constraints were simply added to the architecture description. Most modifications were completed in less than 1 man-hour. This illustrates that ISDL maintains the orthogonality property of instruction sets (modulo the addition of constraints) and that modifications are easy to implement.

Architecture	SPAM VLIW-1	SPAM VLIW-2	SPAM RISC	56000
Class	VLIW	VLIW	RISC	DSP
Lines of ISDL	322	972	329	2286
Man-Hours to Generate	14	50	8	36
Fields	8	8	1	2
Operations	28	81	42	152
Non-Terminals	13	25	5	58
Constraints	6	16	0	43
Reg. Aliases	No	No	No	Yes
Heavy Op-code Encoding	No	Yes	No	No
Op-code Prec.	Yes	Yes	No	Yes

TABLE I
Summary of ISDL Descriptions of Base Architectures

Finally, the descriptions were used to generate functional tools, verifying that ISDL contains sufficient information to generate the required tools.

The ISDL descriptions of all four processors can be found in [6].

B. Experimental Results on Simulator Generator

This section presents experimental results related to the GENSIM system. The following are the parameters of interest:

- **Cycle-accuracy:** The ability to maintain correct cycle-counts (including stall cycles) and to ensure that results are written back when expected in the case of unprotected pipelines, as described in the costs and timings expressions of the input description.
- **Range of Architectures:** The ability to handle a wide variety of architectures.
- **Support for Architecture Features:** The ability to generate correct behavior for common architectural features such as zero overhead loops, setting of condition codes, heavy op-code encoding, field precedence, multi-cycle instructions of fixed length, etc.
- **Generation Speed:** The amount of time it takes to process an ISDL description and to generate a fully functional simulator from it.
- **Simulation Speed:** The speed at which the generated simulator runs on sample programs.

Cycle-accuracy is guaranteed by construction. However, to verify that the behavior of the generated simulators is correct, a number of architectures and modifications were run on sample programs and the behavior and cycle counts were compared to those expected. The expected cycle counts were derived from manual counting and Verilog simulation. The generated simulators do indeed maintain cycle-accuracy to the instruction level (i.e., will maintain any timing semantics that are visible to the instruction set).

The simulator was used to describe a wide range of architectures with a number of common architectural features. The simulator generator successfully generated simulators for all of these, proving that the algorithms are flexible enough to handle both a wide range of architectures as well as a wide variety of commonly implemented features. Table II shows a summary of the architectures for which simulators were generated and lists the architectural features for each one. For more details on architectural features, see [6].

Generation speed was also measured for all of the above architectures. Table III summarizes the results obtained from the various architectures we examined. All run-times were measured on a Sun Microsystems Ultra-10 workstation with a 333 MHz processor, running Solaris 2.6. Most of the time spent on generating the simulator is actually spent determining op-code uniqueness. Note that despite the worst-case exponential run-time, our heuristics ensure that an exponential search is almost never used and therefore the run-times are very reasonable. Both the SPAM VLIW-1 architectures and the SPAM VLIW-2 architectures resort to a non-terminal expansion phase and the resultant search, but the heuristics make sure that the exponent is low (n^2) and each of the fields involved has only a few operations in it. The one exception is the case of the 56000 description. A number of factors conspire to make the exponential search unavoidable:

- The op-codes for operations within the same field are not in a constant place and therefore non-terminal expansion is necessary to resolve almost every single case.
- Since there are only two instruction fields in the description and one of them sometimes uses all the bits in the instruction word, the op-code subtraction heuristics do not perform well.
- Each field has a very large number of operations which means that the base in the exponential equation is really large.

Despite the above factors, the GENSIM system can generate a simulator in a reasonable time even for the 56000

Base	VLIW-1			VLIW-2			SPAM RISC					56000
Version	a	b	c	a	b	c	a	b	c	d	e	a
Class	VLIW			VLIW			RISC					DSP
ILP	x	x	x	x	x	x						x
Zero-Overhead Loops	x	x	x	x	x	x						x
Load/Store	x	x	x	x	x	x	x					
Direct-memory modes								x	x	x	x	x
Complex Instructions			x	x	x	x						x
Condition Codes												x
Hardware Stack				x	x	x						x
Harvard Architecture	x	x	x	x	x	x			x	x	x	x
Address Generators				x	x	x						x
Floating Point				x	x	x						
Stalls										x	x	
Delay Slots											x	
Heavy op-code Encoding				x	x	x						
Field Precedences	x	x	x	x	x	x						x
Register Aliasing												x
Predicated Execution												x

TABLE II
Architectures Used for Simulator Generation

Base	Version	Heavy Op-code Encoding	Precedences	Word (bits)	Generation Time (sec)
SPAM VLIW-1	a		x	44	< 1
	b		x	44	< 1
	c		x	44	< 1
SPAM VLIW-2	a	x	x	99	2.2
	b	x	x	99	2.3
	c	x	x	99	2.6
SPAM RISC	a			24	< 1
	b			24	< 1
	c			24	< 1
	d			24	< 1
	e			24	< 1
56000	a		x	24	181.27

TABLE III
Simulator Generation Speed Results

architecture[8].

We also measured the simulation speed for all of the simulators generated by GENSIM. The results are shown in Table IV. All run-times were measured on a Sun Microsystems Ultra-10 workstation with a 333 MHz processor, running Solaris 2.6. Note that simulation speeds for these example architectures range from 1.1 M to 4.5 M operations per second. These simulation speeds are high enough that realistic data samples can be run with the full application, thus increasing the accuracy of the evaluation process. We compared the speed of simulation using Xsim against the simulation of a hand-crafted Verilog model using a commercial simulator, for the SPAM VLIW-1 architecture. A speedup of over 30 was obtained, as shown in Table V. Speedups will be in the range of 10-50X for most architectures.

Many different application codelets were used for these benchmarks, and the results presented are the average results obtained. The codelets included n-tap FIR and IIR filters and various array-accumulate algorithms, including divide-and-conquer.

Note the anomalies in the figures in Table IV. While in terms of the fetch rate the five versions of the SPAM RISC

[8] Note that in all the cases listed in Table III, generating the simulator source code took less time than compiling the simulator source code with full optimization flags.

Base	Version	Fetch Cycles ($10^3/sec$)	Operations ($10^6/sec$)
SPAM VLIW-1	a	503	4.0
	b	563	4.5
	c	328	2.6
SPAM VLIW-2	a	292	2.3
	b	297	2.4
	c	452	3.6
SPAM RISC	a	1193	1.2
	b	1241	1.2
	c	1204	1.2
	d	1061	1.1
	e	1169	1.2
56000	a	46.19	0.092

TABLE IV
Simulation Speed Results

Model	Speed (cycles/sec)	Speedup
Xsim (ILS) Simulator	4.5×10^6	34.4
Synthesizable Verilog	13.1×10^4	1

TABLE V
Simulation Speeds for Xsim vs Hardware Model

were the top performers, in terms of the actual operations per second, these were the worst performers. This gives an indication of the overhead of the scheduler and the write-back mechanism. This overhead is shared over 8 fields in the two VLIW architectures while it is incurred by every operation of the RISC architecture. Also note the abnormally high performance of Version c of the SPAM VLIW-2 architecture. We believe this to be due to a better fit in the super-scalar issue mechanism of the workstation processor because of the reduced number of jumps at the end of the loop. We believe that the abnormally low speed of Version c of the SPAM VLIW-1 architecture is due to the extra computation that needs to happen for the load-and-increment and store-and-increment instructions. Finally, the abnormally low speed of the simulator for the 56000 architecture is an artifact of the compiler used to compile the generated simulators. The simulator for the 56000 was so large that the compiler could not perform optimizations on it[9]. The code was compiled without optimizations which lead to substantially reduced speed[10]. Turning on the optimization resulted in a five- to ten-fold improvement in performance for the other processor examples, and we conjecture that a similar improvement can be expected in the 56000 example as well.

[9] If we enabled compiler optimizations then the compiler would crash due to an internal error.

[10] The generated code is very inefficient but easy for the compiler to optimize - we count on compiler optimizations to yield the true performance our simulators are capable of.

C. Hardware Generation Results

For the Hgen system, the properties of interest are the die size and the cycle-length (the length of the critical path) of the generated model. Table VI shows these numbers for the first three architectures. The Verilog model was synthesized using the Synopsys toolkit and the LSI 10K technology libraries. The 56000 is not included in the results, because a fully automatic hardware generation was not possible. In particular, to produce the decode logic, manual intervention is required. (This is related to the problem encountered when attempting to optimize the simulator code.)

D. Architecture Exploration

We have performed many experiments on architecture exploration using the tools described in this paper. Exploration to make data processing more efficient and experiments to make control flow more efficient were conducted. Initial Isdl descriptions of the SPAM examples were evaluated, repeatedly recoded and evaluated using the Gensim and Hgen tools. These results are documented in [6] (Section 4.3).

VI. Related Work

A. Mimola

The Mimola[7] design system was created as a high-level design environment for hardware, based on the Mimola hardware description language[8]. The Mimola system was designed for development and evaluation of implementations at a much lower level than Isdl. The Mimola language is a structural description at a relatively low level, and thus results in unnecessarily long and complex descriptions, and in slower simulators (similar to simulation models written in Verilog). On the other hand, the low-level detail makes it much easier to synthesize hardware from the descriptions.

B. nML

The nML machine description language[9] is a high-level machine description language that can be used to support automatically generated tools. It was used in the Chess[10] system for retargetable code-generation as well as a variety of other tools[11]. nML is very similar to Isdl except in the way constraints are handled. nML can only describe valid instructions. Therefore, it must work around invalid combinations by using additional rules, resulting in longer and less intuitive descriptions. It is also unclear how well suited nML would be for hardware generation, since the constraints provide a lot of structural information used to generate efficient hardware.

C. LISA

The Lisa[12] language was developed as a machine description language specifically designed to support the automatic generation of very fast compiled-code simulators, that are cycle-accurate and bit-true. Given the structural content in a Lisa description, hardware generation should

Processor	Cycle (nsec)	Lines of Verilog	Die Size (grid cells)	Synthesis time(sec)	Die Size of Hand-coded Version
SPAM VLIW-1	32	1042	31443	827	27456
SPAM VLIW-2	28	405	4465	100	4232
SPAM RISC	22	287	3267	65	2998

TABLE VI
Hardware Synthesis Statistics

also be possible although we are unaware of any publications describing such a system. The early version of LISA was not well suited for generating code-generators and assemblers. If it was used in a system such as ours, a separate language would have to be used for code generation, thus resulting in consistency issues as well as making it harder to generate, describe, and evaluate architectures. However, these limitations have been addressed in a more recent version of the language [13].

D. HMDES/Playdoh

HMDES[14] is a machine description language that was developed specifically for the TRIMARAN compiler system. It is based on a parameterizable architecture called PLAYDOH[15]. PLAYDOH represents a very general class of architectures which includes features as complicated as predicated execution and complex instructions. While PLAYDOH is very general and can encompass a wide variety of architectures, it is still a parameterized architecture and thus has a limited scope. Similarly HMDES supports a parameterizable instruction set and therefore has a more restrictive scope than ISDL. Like nML, HMDES does not support constraints which may result in longer and less intuitive descriptions. Note, however, that HMDES, like LISA, contains a more extensive timing model than ISDL does.

E. Expression

The Expression language [16] developed at Irvine has been used for architectural exploration. Expression is a mixed behavioral/structural language unlike ISDL which is purely behavioral. It has been used to drive memory exploration and retargetable compilation.

VII. Conclusions, Limitations and Ongoing Work

The results show that the XSIM simulators are substantially faster than the corresponding behavioral Verilog simulation. This allows the use of more realistic simulation runs and provides ample justification for generating an additional model. Additional speedups can be obtained by a move to compiled-code simulators. Furthermore, the XSIM simulator provides a much more user-friendly interface in case the tool needs to be used independently from the rest of the system. This need will arise if a human programmer decides to optimize the output of the retargetable compiler by hand.

The results also show that the HGEN system can generate efficient hardware for medium-sized designs. The run-time of the tool itself is reasonable, and is dominated by the time taken by the silicon compiler.

One limitation of the current system was the reliance on C++ compilers to create the simulator executables and the inability of such tools to analyze large nested switch cases. Our approach is not specific to any language and the simulators could be more easily generated in assembly (where they would not have to rely on compiler optimizations), and would also perform faster. However, this would make the simulator generator non-portable as the back-end would have to produce code specific to a single platform.

Both the simulator generator and the hardware generator tool inherit the limitations of ISDL. For example, exceptions and interrupts are not modeled. This would be easy to do, but is not currently supported either in ISDL or in the tools. A more serious limitation is the inability of ISDL to describe what happens in hardware that is not visible to the instruction set, e.g., caching. An additional limitation is that ISDL cannot describe accurate timing for instructions with direct bypass paths to a subset of instructions, as opposed to all of the instructions. In this case, worst-case timing estimates have to be used. ISDL also cannot describe multi-cycle instructions of variable length.

Future work includes a new version of ISDL that overcomes some of the limitations of the current language, a compiled-code simulator generator for GENSIM, and pipeline optimizations for the HGEN system.

References

[1] G. Hadjiyiannis, S. Hanono, and S. Devadas, "ISDL: An Instruction Set Description Language for Retargetability," in *Proceedings of the 34th Design Automation Conference*, June 1997, pp. 299–302.

[2] G. Hadjiyiannis, S. Hanono, and S. Devadas, "ISDL: An Instruction Set Description Language for Retargetability and Architecture Exploration," *Design Automation of Embedded Systems: An International Journal*, vol. 6, no. 1, pp. 39–69, 2000.

[3] S. Hanono and S. Devadas, "Instruction Selection, Resource Allocation, and Scheduling in the AVIV Retargetable Code Generator," in *Proceedings of the 35th Design Automation Conference*, June 1998, pp. 510–515.

[4] G. Hadjiyiannis, S. Hanono, and S. Devadas, "ISDL: An Instruction Set Description Language for Retargetability," Tech. Rep., Massachusetts Institute of Technology, 1996, (http://www.ee.princeton.edu/spam/pubs/ISDL-TR.html).

[5] G. I. Hadjiyiannis, *ISDL: Instruction Set Description Language - Version 1.0*, MIT Laboratory for Computer Science, July 1998, (http://www.caa.lcs.mit.edu/~ghi/PostScript/ isdl_manual.ps).

[6] G. Hadjiyiannis, *An Architecture Synthesis System for Embedded Processors*, Ph.D. thesis, Massachusetts Institute of Technology, June 2000, (http://www.caa.lcs.mit.edu/~ghi/PostScript/ phd_thesis.ps).

[7] P. Marwedel, "The MIMOLA Design System: Tools for the Design of Digital Processors," in *Proceedings of the 21th Design Automation Conference*, 1984, pp. 587–593.

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

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

[10] D. Lanneer et al., "CHESS: Retargetable Code Generation for Embedded DSP Processors," in *Code Generation for Embedded Processors*. Kluwer Academic Publishers, 1995.

[11] M. A. Hartoog et al., "Generation of Software Tools from Processor Descriptions for Hardware/Software Codesign," in *Proceedings of the* 34^{th} *Design Automation Conference*, 1997, pp. 303–306.

[12] V. Zivojnovic, S. Pees, and H. Meyr, "LISA – Machine Description Language and Generic Machine Model for HW/SW Co-Design," in *Proceedings of 1996 IEEE Workshop on VLSI Signal Processing*, 1996.

[13] A. Hoffman, A. Nohl, S. Pees, G. Braun, and H. Meyr, "Generating Production Quality Software Development Tools Using a Machine Description Language," in *Proceedings of Design, Automation and Test in Europe Conference (DATE)*, March 2001.

[14] J. C. Gyllenhaal, W. W. Hwu, and B. R. Rau, "HMDES Version 2.0 Specification," Tech. Rep. IMPACT-96-3, University of Illinois, Urbana, 1996.

[15] V. Kathail, M. S. Schlansker, and B. R. Rau, "HPL PlayDoh Architecture Specification: Version 1.0," Tech. Rep. HPL-93-80, Hewlett-Packard Laboratories, 1994.

[16] A. Halambi, P. Grun, V. Ganesh, A. Khare, N. Dutt, and A. Nicolau, "EXPRESSION: A Language for Architectural Exploration through Compiler/Simulator Retargetability ," in *Proceedings of Design, Automation and Test in Europe Conference (DATE)*, March 1999.

George Hadjiyiannis received a BS in Electrical Engineering and Computer Science and a MS and Ph.D. in Computer Science from the Massachusetts Institute of Technology, Cambridge in 1993, 1995 and 2000 respectively. Since May 2000 he has held the position of CTO at Tenara Limited, of which he is a founder. Previously he held the position of Chief System Architect at netGenesis Inc. His current research interests include all aspects of computer architecture and design selection, software engineering management, inference engines for AI, VLSI design, and amorphous computing.

Srinivas Devadas received a B. Tech in Electrical Engineering from the Indian Institute of Technology, Madras in 1985 and a MS and Ph.D in Electrical Engineering from the University of California, Berkeley, in 1986 and 1988 respectively. Since August 1988, he has been at the Massachusetts Institute of Technology, Cambridge, and is currently a Professor of Electrical Engineering and Computer Science. He held the Analog Devices Career Development Chair of Electrical Engineering from 1989 to 1991. In 1992, he received a NSF Young Investigator Award. Prof. Devadas' research interests include VLSI CAD, VLSI testing, computer architecture, compilation techniques for embedded processors, formal verification, and security aspects of pervasive computing. Prof. Devadas has authored or co-authored numerous technical papers in journals and conferences, and has co-authored four books. He has received eight Best Paper awards at various conferences and journals, including the 1990 IEEE Transactions on CAD and the 1996 IEEE Transactions on VLSI Systems Best Paper awards. He has served on the technical program committees of several conferences and workshops including the Int'l Conference on Computer Design, the Design Automation Conference, and the Int'l Conference on Computer-Aided Design. He served as the Technical Program Chair for VLSI'99, and as General Co-chair for the ACM LCTES/SCOPES 2002 conference. He serves on the Editorial Board of ACM TODAES, Formal Methods in VLSI Design, and Design Automation of Embedded Systems journals. Prof. Devadas is a member of ACM.