# A Methodology for Accurate Performance Evaluation in Architecture Exploration

George Hadjiyiannis
ghi@caa.lcs.mit.edu

Srinivas Devadas*
devadas@mit.edu

Laboratory for Computer Science
Massachusetts Institute of Technology
545 Technology Square
Cambridge, MA 02139, USA

**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.

## 1 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.

*Corresponding Author

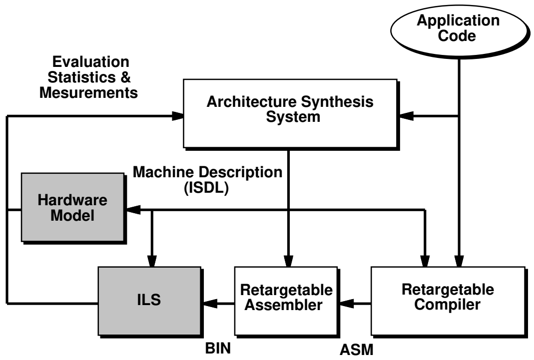


Figure 1: **Architecture Exploration by Iterative Improvement**

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. We are developing a system such as the one illustrated in Figure 1 based on the ISDL machine description language[1]. 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 architectures as possible, and in particular Very Long Instruction Word (VLIW) architectures. In this paper we focus on the methodology which allows us to generate 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 [2]. The design of the assembler generator is briefly described in [3].

## 1.1 Organization of this paper

Section 2 presents a brief overview of the ISDL machine description language, with emphasis on the features that make simulator generation and hardware synthesis possible. Section 3 presents the GENSIM simulator generation system. Section 4 presents the methodology we use to synthesize hardware from an ISDL description. Section 6 presents some previous work in simulator generation and hardware synthesis from machine description languages, and compares these systems to our own, ISDL-based approach. Section 7 presents preliminary experimental results using our system.

# 2 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[1] [3] [4], 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.

## 2.1 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. For a complete description of ISDL including some examples refer to [4].

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.

### 2.1.1 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 2.1.3)[1].

[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.

### 2.1.2 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.

### 2.1.3 Instruction Set Section

ISDL places special emphasis on VLIW architectures. An instruction in a VLIW architecture consists of a combination of operations, one for each functional unit. To model this, the instruction set of an architecture is described as a list of fields, each of which is a list of operation definitions. A field roughly corresponds to the set of operations that can be performed on a single functional unit. Thus, operations within a field are mutually exclusive and cannot be used in parallel (since they all map to the same functional unit). To form a VLIW instruction, operations are selected (one from each field) and grouped together.

The instruction set section of an ISDL description consists of a list of fields, each of which consists of a field name and a list of the operations within that field. An operation definition consists of six parts:[2]

1. *Operation Syntax:* The operation name and a list of parameters (token or non-terminal names) separated by commas.

2. *Bitfield Assignments:* Assignments which set the instruction word bits to the appropriate values.

3. *Operation Action:* A set of RTL-type statements that describe the effect of the operation on the processor state.

4. *Operation Side Effects:* RTL-type statements that describe the side-effects of the operation (such as setting a carry bit).

5. *Operation Costs:* These describe the performance and size implications of each operation. ISDL predefines three costs:

   (a) `Cycle:` the number of cycles the operation takes on hardware in the absence of stalls.

[2] Note that these correspond to the six parts of the definition of a non-terminal option because non-terminals abstract common patterns in operation definitions.

   (b) `Stall:` the number of additional cycles that may be necessary during a pipeline stall.

   (c) `Size:` the number of instruction words required for the operation.

6. *Operation Timing:* These are a set of assignments that define the timing of the operation effects. ISDL pre-defines two timing parameters:

   (a) `Latency:` This describes when the results of the operation become available.

   (b) `Usage:` This describes when the functional unit becomes available.

#### 2.1.4 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 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.

## 3 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.

### 3.1 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

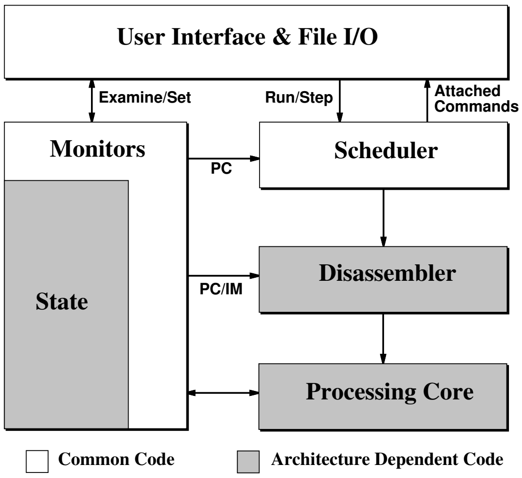


Figure 2: **Internal Structure of the XSIM Simulators**

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.

### 3.2 Simulator Structure

Figure 2 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.

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.

## 3.3 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.

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

### 3.3.2 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 3):

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:

| 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 | $a'_7$ | $a'_6$ | $a'_5$ | $a'_4$ | $a'_3$ | $a'_2$ | $a'_1$ | $a'_0$ | $c'_7$ | $c'_6$ | $c'_5$ | $c'_4$ | $c'_3$ | $c'_2$ | $c'_1$ | $c'_0$ |

Figure 3: **Operation Signatures**

- "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[3]:

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

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[4]. 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 4 shows this algorithm in pseudo-code.

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.

### 3.3.3 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.

[3] For brevity, we will not attempt to describe here why this axiom holds. Suffice to say that is is true for *every* architecture we have come across.

[4] 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.

```
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
```

Figure 4: **Disassembly Algorithm**

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 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.

# 4 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).

## 4.1 Hardware Synthesis from ISDL

In our methodology, the architecture synthesis system produces instruction sets instead of architectures[5]. 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. If any implementation-specific information was missing from the original ISDL description, the HGEN compiler will provide it at this time.

The above methodology only uses a single description avoiding consistency issues. Also, the granularity at which changes can be made is much finer[6] 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 4.1.1.

We feel that direct synthesis from ISDL has compelling advantages, including the fact that it will benefit

[5] Architectures can be thought of as implementations of the instruction set.

[6] Individual changes are made at the level of an RTL operation.

more from improvements in other CAD tools (such as silicon compilers). Also, the resource sharing problem can be solved using a combinatorial optimization strategy.

#### 4.1.1 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 data-paths to handle the `load` and `store` operations even though it is possible to implement these with the same bus that implements the `move`.

#### 4.1.2 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 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 `subtract`) *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.

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

Figure 5: **Resource Sharing Algorithm**

Once we have the entries in the matrix, we can simply create maximal cliques[7] 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 5 shows this algorithm in pseudo-code.

### 4.1.3 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 4.1.1. 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.

[7] 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.

### 4.2 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[8]. They both implement the same function (reversing the assembly function). 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 3 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.

## 5 Experimental Results

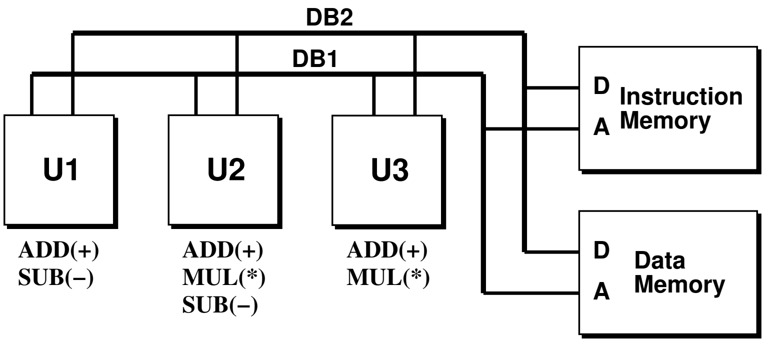


Figure 6: **The SPAM VLIW-1 Architecture**

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.

### 5.1 ISDL Descriptions

A number of architectures were described in ISDL in order to prove the power and flexibility of the language as well as to serve as experiment platforms for the results presented in Sections 5.2 and 5.3. These four

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

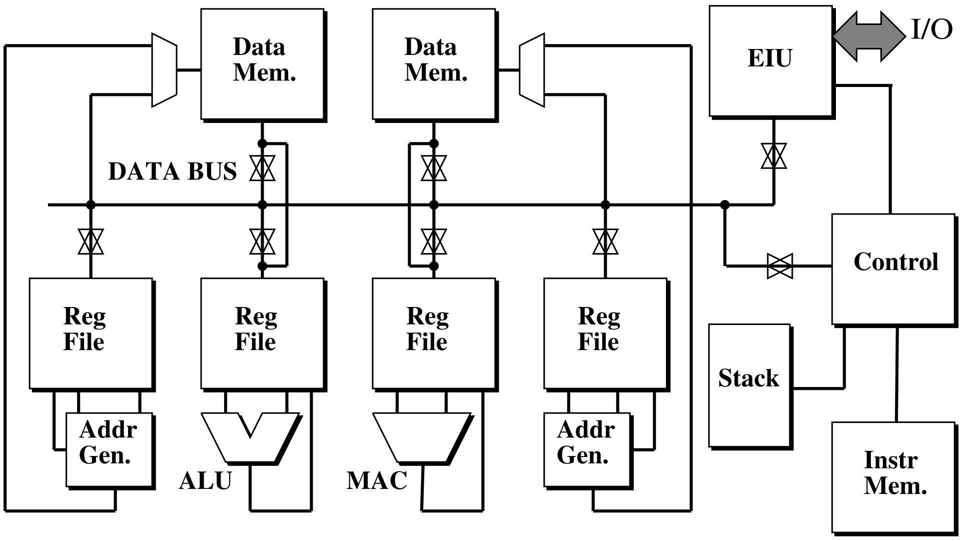


Figure 7: **The SPAM VLIW-2 Architecture**

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 6.

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

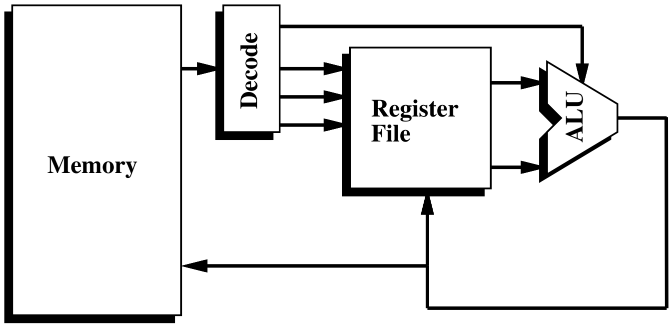


Figure 8: **The SPAM RISC Architecture**

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 7.

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 8.

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 9.

## 5.2 Experimental Results on Simulator Generator

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

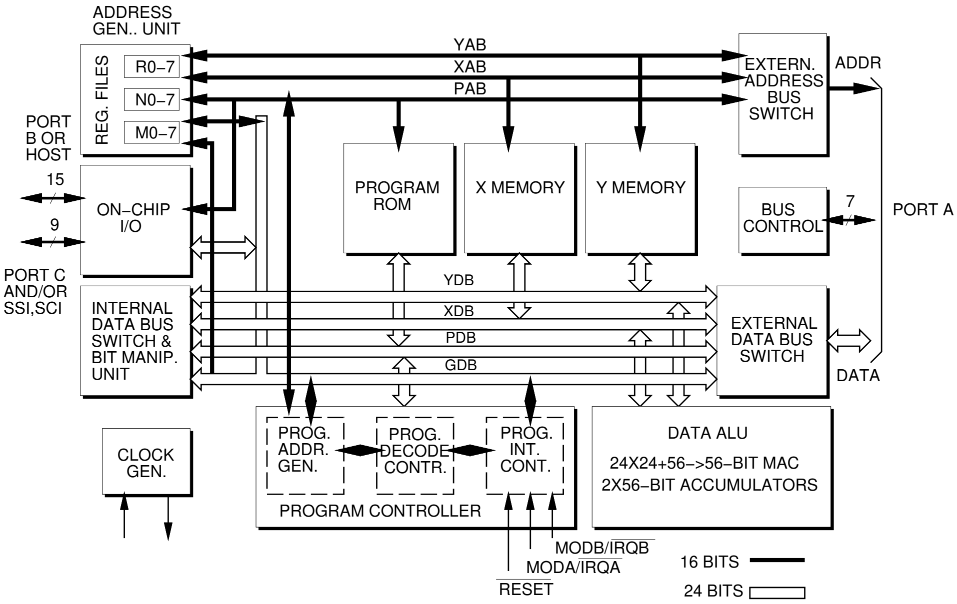


Figure 9: **The Motorola 56000 Architecture**

- **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 and, in particular, all the architecture classes that form the target of the ARIES system.

- **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, 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 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).

| 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 1: **Architectures Used for Simulator Generation**

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 1 shows a summary of the architectures for which simulators were generated and lists the architectural features for each one.

Generation speed was also measured for all of the above architectures. Table 2 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.

| 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 2: **Simulator Generation Speed Results**

- 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 architecture[9].

We also measured the simulation speed for all of the simulators generated by GENSIM. The results are shown in Table 3. 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 4. Speedups will be in the range of 10-50X for most archictures.

Note the anomalies in the figures in Table 3. While in terms of the fetch rate the five versions of the SPAM RISC were the top performers, in terms of the actual operations per second, these were the worst performers.

[9] Note that in all the cases listed in Table 2, 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 | NA | 46.19† | 0.092† |

Table 3: **Simulation Speed Results**

| Model | Speed (cycles/sec) | Speedup |
|---|---|---|
| XSIM (ILS) Simulator | $4.5 \times 10^6$ | 34.4 |
| Synthesizable Verilog | $13.1 \times 10^4$ | 1 |

Table 4: **Simulation Speeds for XSIM vs Hardware Model**

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

Table 5: **Hardware Synthesis Statistics**

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[10]. The code was compiled without optimizations which lead to substantially reduced speed[11].

### 5.3 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 5 shows these numbers for the first three architectures. The Verilog model was synthesized using the Synopsys toolkit and the LSI 10K technology libraries.

## 6 Related Work

### 6.1 Mimola

The MIMOLA[5] design system was created as a high-level design environment for hardware, based on the MIMOLA hardware description language[6]. 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

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

[11] 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.

much easier to synthesize hardware from the descriptions.

### 6.2 nML

The nML machine description language[7] is a high-level machine description language that can be used to support automatically generated tools. It was used in the CHESS[8] system for retargetable code-generation as well as a variety of other tools[9]. 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.

### 6.3 LISA

The LISA[10] 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 also be possible although we are unaware of any publications describing such a system. However, LISA is 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.

### 6.4 HMDES/Playdoh

HMDES[11] is a machine description language that was developed specifically for the TRIMARAN compiler system. It is based on a parameterizable architecture called PLAYDOH[12]. 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.

## 7 Conclusions 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 even for large, complex designs. The run-time of the tool itself is reasonable, and is dominated by the time taken by the silicon compiler.

Future work includes 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* $34^{th}$ *Design Automation Conference*, pages 299–302, June 1997.

[2] S. Hanono and S. Devadas. Instruction Selection, Resource Allocation, and Scheduling in the AVIV Retargetable Code Generator. In *Proceedings of the* $35^{th}$ *Design Automation Conference*, pages 510–515, 1998.

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

[4] 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).

[5] 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.

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

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

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

[9] 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*, pages 303–306, 1997.

[10] 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.

[11] J. C. Gyllenhaal, W. W. Hwu, and B. R. Rau. HMDES Version 2.0 Specification. Technical Report IMPACT-96-3, University of Illinois, Urbana, 1996.

[12] V. Kathail, M. S. Schlansker, and B. R. Rau. HPL PlayDoh Architecture Specification: Version 1.0. Technical Report HPL-93-80, Hewlett-Packard Laboratories, 1994.