

A Formal Concurrency Model Based Architecture Description Language for Synthesis of Software Development Tools

Wei Qin
wqin@princeton.edu

Subramanian
Rajagopalan
sr@princeton.edu

Sharad Malik
sharad@princeton.edu

Department of Electrical Engineering
Princeton University
Princeton, NJ 08544

ABSTRACT

Rapidly increasing design and manufacturing non-recurring engineering (NRE) costs are prompting a shift in electronic design from hardwired application specific integrated circuits (ASICs) to the use of software on programmable platforms. However, in order to minimize the power and performance overhead of such processors, we are seeing the introduction of domain or application specific processors such as network and communication processors. The design of such specialized processors requires software development tools such as simulators and compilers. In order to quickly develop these tools for multiple design points under consideration, it is highly desirable to have them synthesized from formal processor descriptions written in Architecture Description Languages (ADLs). In this paper, we present the Mescal Architecture Description Language (MADL). MADL features a two-layer structure, a core layer and an annotation layer. The core layer is based on a formal and flexible microprocessor model – the operation state machine (OSM), which enables MADL to express the concurrency at the operation execution level for a wide range of architectures. We address the challenges faced in designing the core layer to combine the OSM model with techniques for achieving compact processor descriptions. The annotation layer features a generic syntax that allows creating annotation schemes to specify implementation dependent or tool specific information. To show the effectiveness of MADL, we present an MADL-based simulator synthesis framework that has been used to generate efficient cycle accurate simulators and instruction set simulators with very low development effort. We also describe our annotation schemes that enable the extraction of architecture properties for use in instruction scheduling and integer-linear-programming based register allocation. Our experimental results demonstrate the efficacy of MADL as a practical and promising language for the development of programmable platforms.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

LCTES'04, June 11–13, 2004, Washington, DC, USA.
Copyright 2004 ACM 1-58113-806-7/04/0006 ...\$5.00.

Categories and Subject Descriptors

I.6.2 [SIMULATION AND MODELING]: Simulation Languages;
D.3.4 [PROGRAMMING LANGUAGES]: Processors—*Retargetable compilers*

General Terms

Design, Languages, Verification

1. INTRODUCTION

Increasingly complex system functionalities and shrinking process feature sizes are changing how electronic systems are implemented today. Hardwired application specific integrated circuit (ASIC) solutions are no longer attractive due to sharply rising non-recurring engineering (NRE) costs to design and manufacture the chips. The NRE costs have increased the break-even volume beyond which these hardwired parts are preferred over programmable solutions. Thus, increasingly we are seeing a shift toward systems implemented on programmable platforms. This shift is helped by the development of application/domain specific processors which attempt to reduce the power and performance overhead of programmability by providing micro-architectural features matched to the domain specific computational requirements. However, there is relatively little available in terms of design tools for the software environment (e.g. simulators, compilers) for these processors and thus these tend to be hand crafted – a fairly low productivity task and especially limiting during processor design space exploration when a large number of design points need to be evaluated. Therefore it is highly desirable that the software design tools can be synthesized automatically from high level processor specifications.

Though microprocessors¹ are diverse in their computation power and their underlying micro-architectures, they share some common general properties inherited from their common ancestor – the von Neumann computer. They fetch instruction streams from memory, decode instructions, read register and memory states, evaluate instructions as per their semantics and then update register and memory states. As a result of this commonality, there exists potential to develop high level data models to abstract a wide range of micro-processors. These abstract data models can then be used to assist the development of design automation tools that prototype, synthe-

¹We use this term broadly to include application-specific processors such as digital signal processors, as well as more traditional general purpose processors.

size, verify and program microprocessors. One example of such an abstract data model is the register-transfer-list used by the compiler community for code generation.

Architecture description languages (ADLs) were created to convey abstract processor model information to the software development tools. The differing emphases placed by the ADL designers in supporting different tools and different architecture families make them highly diverse in their syntaxes and semantics. From our experience, the quality of an ADL is largely determined by its underlying processor model. Partially due to the lack of flexible processor models that are capable of expressing precise data and control semantics of microprocessors at a high abstraction level, no existing ADL can be simultaneously used for cycle accurate simulator and compiler generation for a reasonably large architecture range. Our ongoing research is focused on addressing this problem.

After experimenting with various processor models and software development tools, we proposed the operation state machine (OSM) formalism as a microprocessor model [14]. The OSM model provides a high level but powerful abstraction mechanism to model the data and control semantics of a microprocessor. Based on the model, we have designed a two-layer architecture description language named Mescal Architecture Description Language (MADL) to support accurate modeling of a broad range of architectures. In this paper, we present the structure of the language, as well as demonstrate its use in the development of software tools or tool-components such as the cycle accurate simulator, the instruction set simulator, the register allocator and the instruction scheduler. The main contributions of the paper are:

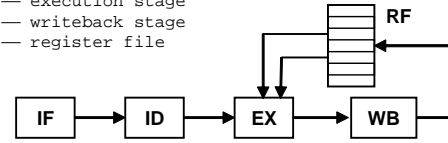
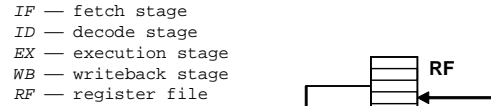
- A layered approach to architecture description language design that separates the key architectural properties from the tool-dependent information.
- A flexible and extensible ADL with demonstrated capability to extract architectural properties for use in simulators and important compiler optimizers.
- A retargetable simulator synthesis framework for quick synthesis of efficient cycle accurate and instruction set simulators.

The paper is organized as follows. Section 2 reviews the OSM model and describes its new extension for data flow modeling. We then present the design of MADL in Section 3. Section 4 describes the generation of several software tools from MADL descriptions, and Section 5 contains experiment results. Section 6 discusses related work in the field. Finally, Section 7 provides some concluding remarks.

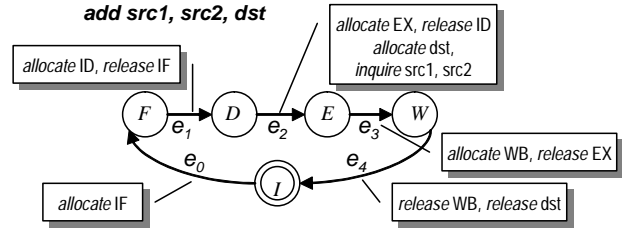
2. OSM MODEL: A REVIEW

An architecture model largely determines the quality of the ADL built on top of it. In this section, we give a brief overview of the OSM model that forms the basis of MADL [14].

The OSM model divides the processor specification into the operation layer and hardware layer. The operation layer models each machine operation as a finite state machine, which communicates with the hardware layer through a formalized token transaction protocol. In the hardware layer, structural or data resources are modeled as tokens. Function units interacting closely with the operations are modeled as token managers, which are effectively resource arbiters that allocate tokens to the state machines based on their resource management policies. This model provides for a natural way to express concurrency at the two layers and the mapping between them.



(a) A 4-stage pipeline



(b) OSM for the add instruction

Figure 1: An OSM model example

Consider the simple example in Figure 1 which shows a 4-stage pipelined processor and a state machine modeling its “add” operation. The states of the state machine represent the execution statuses of the instruction. The state machine starts from the initial state I . At each control step (a clock cycle or phase), it will try to advance its state by one step. When it returns to state I , the instruction retires. In order to advance the state along an edge in the state diagram, the state machine must satisfy the transition condition on the edge. Such condition is defined as the conjunction of the outcomes of a list of token transactions, such as the ones shown in the text boxes in Figure 1(b). We defined four types of token transactions – *allocate*, *inquire*, *release* and *discard*. These control primitives help model pipelining behaviors including structural, data and control hazards that are common in modern microprocessors.

In order to model the data flow in the processor, we introduce two data communication primitives – *read* and *write*. An OSM can read the value of a token either if it is *allocated* the token or if it successfully *inquires* about the token. It can write to a token only if it is *allocated* the token. An OSM can then exchange data values with the token managers through these two primitives. In the example of Figure 1, the state machine can read source register operand values on edge e_2 and write its destination operand value on edge e_4 . Besides these data communication primitives, we also include as part of the OSM model a set of computation operators that can be used to evaluate instruction semantics. The communication and computation primitives complement the original four control primitives. They are new to this framework and are part of the new contributions of the paper. All control, data and computation primitives are called *actions* and they are all bound to the edges of the state machines.

Modern processors are typically pipelined and hence they may have more than one operation executing simultaneously. In the simulation model of such a processor, multiple such state machines may be alive at the same time, each representing an operation in the pipeline. They compete for resources through their token transaction requests and transition their states concurrently. They constitute the operation layer of the model.

The hardware layer of the processor is not shown in Figure 1(b). It includes 4 token managers modeling the pipeline stages, each of

which controls one token that represents the ownership of the corresponding pipeline stage. In addition, we model the register file as a token manager containing one token for each register. The token managers respond to the token transactions required by the state machines. All managers contain a set of interface functions corresponding to all types of transactions. These interface functions implement the resource assignment policies of the token manager and control the execution flow of the state machines.

The finite state machine is a highly flexible model capable of modeling arbitrarily connected execution paths of an operation, including those of superscalar processors with out-of-order issuing. For example, Figure 2 shows part of a state machine that can be used to model the “add” operation in an out-of-order processor. In this example, “RS” represents a token manager modeling a reservation station and “RB” is a token manager for the reorder buffer. After the operation has been decoded, to leave state *D*, the state machine has two options. It can either proceed to state *E*, the execution stage, if an execution unit, a reorder buffer entry and the source operands are available, or enter state *R*, the reservation station stage, to wait for these resources. Such out-of-order issuing behavior cannot be modeled by other pipeline models such as the pipeline diagram, but can be easily modeled by the state machine. The flexibility of the operation layer as well as the extensibility of the hardware layer allows the OSM model to capture the precise execution semantics of microprocessors for cycle accurate simulator generation.

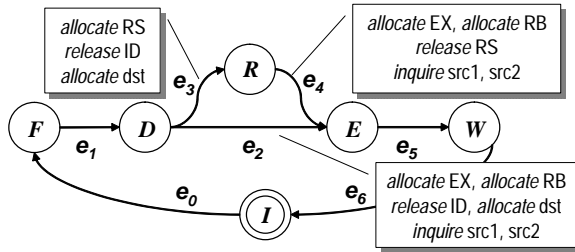


Figure 2: OSM example for out-of-order issuing

From the point of view of compiler developers, with proper hints, it is easy to analyze the state diagram and the *actions* to extract useful properties such as operation semantics and resource usage information. Such information is essential for compilers and its automatic extraction can be very helpful for retargetable compiler development.

In summary, the OSM model provides for a flexible yet formal means to model microprocessors. It is suitable as the underlying semantic model of a machine description language. For details of the OSM model, we refer readers to [14].

3. MADL

Based on the OSM model described in Section 2, we designed MADL, an ADL that can be used to describe a broad range of microprocessors including scalar, superscalar, VLIW and multi-threaded architectures. The goal of the description language is to assist the generation of software tools such as simulators and compiler optimizers.

To make MADL simple yet extensible, we adopted a two-layer description structure for microprocessor modeling. The first layer is the core language that describes the operation layer of the OSM model. It contains specifications of instruction semantics (the *actions*), binary encoding, assembly syntax, execution timing, and resource consumption. This layer forms the major part of a processor

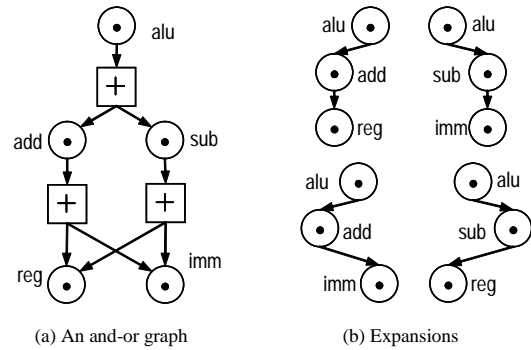


Figure 3: And-or graph example

description. In section 3.2, we will describe how we can achieve concise descriptions in this layer by combining the OSM semantic model with an effective syntax model, namely the and-or graph.

The second layer of MADL describes information that is dependent on software tools and their implementations. It either supplements the core description by providing tool specific information or assists the software tools to analyze and extract processor properties. This layer is called the annotation layer. The annotation syntax is generic and hence flexible and extensible to use. Unlike the core layer, its semantics are subject to interpretation by the software tools that use the information.

3.1 The and-or graph: a review

A good syntax model is important to the usability of an ADL. In this section, we review the and-or graph model that is commonly used to describe instruction encoding, assembly syntax and evaluation semantics. It appears in different forms in ADLs including nML [4], ISDL [7] and LISA [12]. An and-or graph is a directed acyclic graph with only one source node. It is composed of a number of *elements*. An *element* is either an and-node and all its or-node children, or a leaf and-node. Figure 3(a) shows an example of an and-or graph with five *elements*. An *expansion* of an and-or graph can be obtained by short-circuiting each or-node with an edge from its parent to one of its children. Figure 3(b) shows all possible *expansions* of the graph, each of which corresponds to an instruction. Compared to enumerating instructions in their expanded forms, the and-or graph model is much more compact as it factorizes common properties to the upper levels, thereby minimizing redundancy.

Most, if not all, instruction set architectures (ISA) organize instructions into hierarchical classes. All instructions in a class share properties such as encoding format, assembly syntax, etc. Hence, the and-or graph model is a natural choice to model ISAs, especially orthogonal ISAs such as RISC. For the toy instruction set shown in Figure 3, since the operands and the opcodes are orthogonal, they can be separated into different elements. The common elements among different instructions are then merged to form the and-or graph.

3.2 The core layer

Due to the significant advantages of the OSM architecture model and the and-or graph syntax model, we chose them as the foundations of the core layer of MADL. Thus, the task of designing our ADL is to combine the and-or graph syntax model with the OSM semantic model to create a description language that is expressive and efficient to describe encoding, assembly syntax, semantics, timing information of the instructions and their interactions with the hardware layer.

Note that currently our focus on the core layer is restricted to the operation layer and does not include the hardware layer (mainly the token managers). The execution models of the token managers are currently organized as a C++ template library. We are still in the search of suitable syntaxes for describing token managers in future versions of MADL as this may allow the MADL compiler to potentially analyze the token managers for verification and extraction of other information.

In the rest of this section, we will describe the core layer of MADL using an example of the 4-stage pipelined processor of Figure 1 with a toy instruction set shown in Figure 3.

As mentioned in Section 2, the edges of an OSM are statically bound with the *actions*. Since different types of operations differ in their *actions*, it would be a natural thought to adopt separate state machines to model different operation types. However, the actual type of an operation cannot be resolved until it is fetched and decoded, which occurs during the execution of the state machine. On the other hand, we cannot decide which type of state machine to execute until we know the actual type of the operation. Such mutual dependency creates a bootstrapping problem.

To solve the problem, we can use a polymorphic state machine that can model all types of operations. Figure 4 shows one such state machine for the example processor. In such a state machine, each path from state *F* to state *I* represents the execution path of one type of operation. The state machine is capable of fetching and decoding the operation and choosing to execute along the right path that corresponds to the actual type of the operation. An artificial token manager can be implemented to help steer the execution path. It should be noted that although the state machine is polymorphic, at one time it still represents only one operation. Multiple such state machines are needed to model the operations in a pipelined processor.

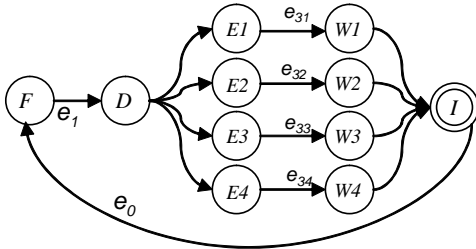


Figure 4: Polymorphic state machine for the toy ISA

In general, the paths of the polymorphic state machine may have heterogeneous topologies. For real-world instruction set architectures with hundreds of operations, the state machine may be very large and cumbersome to specify. Moreover, although we may still apply the and-or graph to describe the *actions* of all operations, it is not directly applicable to specify the static binding relationships between the *actions* and the edges if most of the edges are not sharable across operations.

To avoid the potential state explosion problem in descriptions, we adapted the OSM model to its dynamic version. In the dynamic OSM model, the *actions* are no longer statically a part of the state diagram. Instead, they are bound to the edges dynamically during the execution of the state machine. In such a dynamic model, two different operations can refer to the same set of edges in the *action* description. As long as the syntax ensures that no conflict occurs during binding, the same state diagram can be shared by all operations in the description. This makes it possible for us to fully apply the and-or graph model to specify the *actions*.

In the dynamic OSM model, the state diagram is simply a directed graph. We call such a bare diagram a *skeleton*. To apply the and-or graph model, we describe operations based on *syntax operations*, each of which corresponds to one element in the and-or graph. All *syntax operations* in one *expansion* of the and-or graph form one operation. At run time, the *actions* of the *syntax operations* in the *expansion* will bind to the *skeleton* to form a finite state machine that models the operations.

```
#this skeleton defines
#variables rdst,v_src1,v_src2,v_dst,
#fbuf,dbuf,ebuf,wbuf,rbuf
USING diagram_a;

OPERATION alu
VAR oper : {add, sub}; #an or-node
iw : uint<14>; #i-word
TRANS e0 : {fbuf = IF[]}; #allocate IF
e1 : {dbuf = ID[], #allocate ID
iw = *fbuf, #read i-word
!fbuf} #release IF
+oper = iw; #decode & bind
e2 : {ebuf = EX[], #allocate EX
rbuf = RF[rdst], #allocate dst
!dbuf}; #release ID
e3 : {wbuf = WB[], #allocate WB
!ebuf}; #release EX
e4 : {*dst_buf = v_dst, #write dst
!rbuf, #release dst
!wbuf}; #release WB

OPERATION add
VAR rs1 : uint<4>; #assigned from CODING
src2 : {imm, reg}; #an or-node
CODING 1 rdst rs1 src2;
EVAL +src2; #decode per bit0-4 of CODING
TRANS e2 : {v_src1 = *RF[rs1]};
#inquire&read src1
e3 : v_dst = v_src1 + v_src2;
#compute addition

OPERATION imm
VAR v_im : uint<4>; #assigned from CODING
CODING 1 v_im;
TRANS id_ex : v_src2 = (uint<32>)v_im;

OPERATION reg
VAR rs : uint<4>; #assigned from CODING
CODING 0 rs;
TRANS e2 : {v_src2 = *RF[rs]};
#inquire&read src2

OPERATION sub
.....
```

Figure 5: Example ADL description

Figure 5 shows an example based on such a description scheme for the 4-instruction toy processor. The first USING statement indicates that the following *syntax operations* are based on the *skeleton* named “diagram.a”, whose structure is the same as the state diagram shown in Figure 1. A *skeleton* definition (not shown here) contains a list of states and edges, and a list of variables that can be accessed by all the *syntax operations* bound to it. The rest of the example defines a set of OPERATIONS, each corresponding to a *syntax operation*. An OPERATION contains sections such as VAR where local variables are defined, CODING where binary encoding is specified, EVAL where the initialization of local variables can be

performed, and TRANS where *actions* are defined with respect to the edges in the *skeleton*. In a VAR section, besides variables of normal arithmetic data types, a special type of variable that corresponds to the or-node in the and-or graph can also be defined. An example of such a variable is the “oper” variable in “alu” which corresponds to the top-most or-node in Figure 3(a). In the TRANS sections, control and data *actions* are enclosed within curly braces, while computations are not.

In order to understand the process of dynamic binding of *actions* onto edges, consider the *syntax operations* shown in Figure 5 and the *skeleton* shown in Figure 1. During run time, the top-level *syntax operation* “alu” will be associated with the *skeleton* first. It will bind its *actions* specified in the TRANS section onto the corresponding edges of the *skeleton*. In this simple example, the *actions* include reading the instruction word from the token manager “IF”, which models the instruction fetcher, when leaving state *F* and then decode the or-node variable “oper” with the instruction word. Depending on the value of the instruction word, decoding will resolve “oper” to one of the two *syntax operations* “add” and “sub”. If the result is “add”, then “add” will also bind its *actions* to the edges of the same *skeleton*. Meanwhile it will further decode its or-node variable “src2” based on the lowest order five bits of its encoding value. If the result is “imm”, then “imm” will bind its *action* onto the *skeleton*. The combined result of “alu”, “add”, “imm” and the *skeleton* “diagram.a” models one add-immediate instruction. Other combinations such as “alu”, “sub” and “reg” share the same *skeleton* in the description. The decoding statement ensures that at run time, only one *expansion* is bound to the *skeleton*.

The introduction of the dynamic OSM model enables us to apply the and-or graph model to greatly ease the description. As with the original version, the dynamic OSM model is also executable. However, its execution is less efficient due to the overhead incurred by dynamic binding. Therefore the MADL compiler will perform a simple transformation to convert the dynamic model to the original version. For each *expansion* in the and-or graph, the compiler will duplicate a portion of the state diagram that is reachable from the decoding edge without going through state *I*. It will then statically bind all *actions* of the *expansion* onto the duplicated sub-diagram. The result of the transformation is an expanded state machine with an overall topology similar to that shown in Figure 4.

For more details of the core layer syntax, we refer interested readers to its manual [13].

3.3 The annotation layer

Annotations appear as *paragraphs* in an MADL description. Figure 6 shows the syntax in Backus-Naur Form. A paragraph contains an optional namespace label and a list of declarations and statements. The label specifies the tool-scope of the paragraph and can be used to filter irrelevant annotations. Paragraphs without a label belong to the global namespace.

One of the features of the syntax is that it supports the declaration of variables and their relationships. The scope of the variables is determined by the location of the annotation paragraph in the description and its namespace. The relation operators currently supported include normal comparison operators and the set containment operator (“<:” in syntax). Section 4 shows how these operators can be effectively used to express irregular architecture constraints.

The annotation paragraphs are associated with syntax elements of the core description. After the MADL description is parsed, the tools can access an annotation paragraph via the pointer to the intermediate representation object of the syntax element that it is associated with. A set of APIs are defined for such access.

```

annot_paragraph ::= claus_list
                 | :id: claus_list //with namespace

claus ::= decl | stmt

decl ::= var id:type //variable
       | define id value //macro

stmt ::= id (arg_list) //command
       | val op val //relationship

arg ::= id = value

val ::= id | number | string
      | (val_list) // tuple
      | {val_list} // set

type ::= int<width> | uint<width> | string
       | (type_list) // tuple type
       | {type} // set type

```

Figure 6: Annotation syntax in BNF

```

MANAGER
CLASS
    unit_resource : void -> void;
    $:COMP: SCHED_RESOURCE_TYPE(size=1);
    $$:SIM: USE_CLASS(name="untyped_resource",
                      param="1");$$
.....

```

Figure 7: Two-layer description example

An annotation paragraph can either be in a single-line format or in a block format. The former is preceded by a “\$” and runs through the end of the line while the latter is enclosed within a pair of “\$\$”s. Figure 7 gives an example of an MADL description. It shows part of a MANAGER section. The third line defines a token manager class “unit_resource” and its type property. The two subsequent annotation paragraphs are attached to this manager class and provide additional information to different tools. The first paragraph informs the tool in the “COMP” namespace (in this case the compiler) that this manager class can be treated as a resource unit by a scheduler. Such information is not essential to the processor architecture, but a user-supplied hint for the compiler. The second paragraph notifies the “SIM” tool (in this case the cycle accurate simulator) that this manager uses the template class “untyped_resource” with a template argument of “1” as its C++ implementation in the simulator. Note that the interpretation of the annotation commands are solely up to the tools. Any change in the tool implementation may affect the annotation description but neither its syntax nor the core layer. This feature insulates the MADL syntax from the frequent modifications and extensions of the software tools, thereby lengthening the lifetime of MADL.

4. SUPPORT FOR TOOLS

In this section, we describe how MADL can be effectively used to describe information necessary for various software tools. In particular, we show that the core layer is powerful enough to model a wide range of architectures, and that the annotation layer is flexible enough to represent information needed by a variety of tools, specifically, the cycle accurate simulator, the instruction set simulator, the register allocator, and the reservation table based scheduler.

4.1 Cycle accurate simulator

Cycle accurate simulators are important for the verification of programmable platform designs as well as the application softwares that run on the platform. Since the OSM model is inher-

ently an executable model, it is straightforward to synthesize cycle accurate simulators from MADL descriptions.

Figure 8 shows the structure of our cycle accurate simulator synthesis framework based on MADL. The inputs to the framework are the MADL description, the token manager implementations in C++ and the peripheral unit implementations in C++. The peripheral units are those hardware components that do not directly interact with the state machines in our model. They communicate with the state machines indirectly through token managers.

As mentioned in Section 3, the core layer is used to specify the operation layer information. Thereby the MADL compiler can extract all state machine details from the *skeleton* and *syntax operation* descriptions and translate them into state machine classes in C++. As significant reuse is expected for the token managers and the peripheral units, they are organized as C++ template class libraries. New token managers or function units can be customized by specializing and extending the library classes or from scratch. As shown in the example of Figure 7, we can use annotations to specify pointers to the actual token manager implementation. The synthesis framework will follow these pointers to statically instantiate the token manager objects. The MADL description and the customized token managers represent the major development effort for targeting the framework to a new processor.

The state machine classes and the token manager classes are instantiated in the cycle accurate simulator. The simulator also contains instances of the relevant peripheral units such as the cache models, as well as the cycle-driven simulation kernel and the memory emulator.

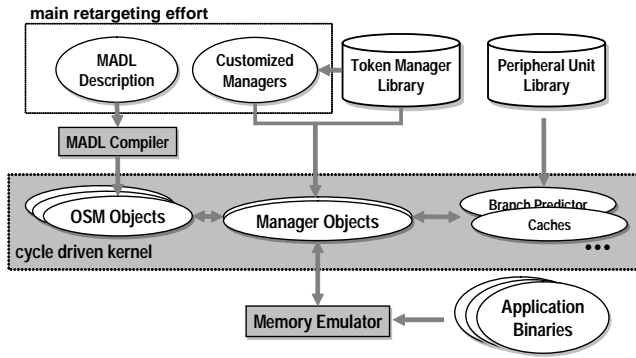


Figure 8: Cycle accurate simulation framework

It is worth noting that the use of MADL improves the productivity of simulator development since MADL descriptions are very compact. Moreover, the synthesized simulators most often have better simulation performance than hand-coded OSM simulators. The reason is that when coding by hand, programmers commonly have to trade code speed for code simplicity. For instance, in our hand-coded OSM simulators, all OSM action classes are derived from a common base class. Such use of the C++ polymorphism feature greatly simplifies the implementation as the actions can be treated in a homogeneous fashion, e.g. their pointers can be stored in an array. However, this introduces many virtual function calls at run-time that slow down the simulation speed notably. While in a synthesized simulator, code complexity concerns are irrelevant and the fastest option is always favored.

4.2 Instruction set simulator

Instruction set simulators (also called functional simulators) are another important class of software tools that help verify the func-

tionality of application binaries. With two minor modifications, our simulation framework can also be used to generate instruction set simulators.

First, since the instruction set simulator has no notion of time or operation concurrency and executes the operations sequentially, only one state machine instance is needed in the simulation kernel to model the operation stream. Therefore the simulation kernel becomes a simple loop that iteratively activates the state machine instance. In each iteration, the state machine instance will execute the complete semantics of one operation from its fetch to its retirement without interruption. Second, the token managers used for the instruction set simulator are simplified by removing all timing related control semantics. Therefore, by supplying a set of simplified token managers, the same MADL description written for the cycle accurate simulator can be used to generate the instruction set simulator.

4.3 Register Allocator

Register allocation is often considered to be one of the most important optimizations in a compiler. An allocator that produces excessive spill code or too many false dependences can potentially destroy benefits of other optimization phases. Hence it is very critical to have a good register allocator and also to provide accurate information to the register allocator from the processor specification. One of the most widely used register allocators is the graph coloring based allocator. It is elegant, retargetable and effective for RISC-style ISAs with a large number of homogeneous registers. However, for CISC-style architectures and many embedded processors such as DSPs, graph coloring based allocators do not work well. This is due to the existence of highly irregular ISAs optimized for code density, small number of registers and heterogeneous register banks. While several attempts have been made to develop efficient register allocators for such processors, very few are retargetable, efficient and easy to use. In our work, apart from a graph coloring based allocator, we also built an Integer Linear Programming (ILinP) based allocator developed by Appel and George [2]. The AMPL [5] based ILinP model is simple to use, expressive to describe constraints, retargetable and guarantees an optimal solution. Although we may use different allocators for different domains, the information needed by the allocators is very similar. In this section, we describe the various inputs to a register allocator and how they can be specified in MADL.

The basic input to register allocators are the different types of register banks, both physical and logical. Properties of a register bank include data type of the bank, number of registers and overlap information such as that of single and double precision registers commonly seen in modern microprocessors. Figure 9(a) gives a general overview of the nature of information that we need to capture. The bank information, enclosed within the dotted box, includes the eight registers “R0” to “R7”, the register banks “bank1” and “bank2”, and register overlap information as indicated by the double arrow between registers “R2” and “R4”. Note that a register may be in multiple logical banks like “R3” or may not be a part of any bank like “R7”.

Apart from all the register and register bank related information, the allocators also need to know how the operands of various operations are constrained. This includes the set of register candidates that an operand may be assigned from, operands that have been pre-allocated and sets of operands that may have to be allocated together. In Figure 9(a), this information is represented by the thick dotted arrows. For example, operands “o1” and “o2” of “oper1” are constrained to be assigned from register banks “bank1” and “bank2” respectively, operand “o3” of “oper2” needs to be pre-

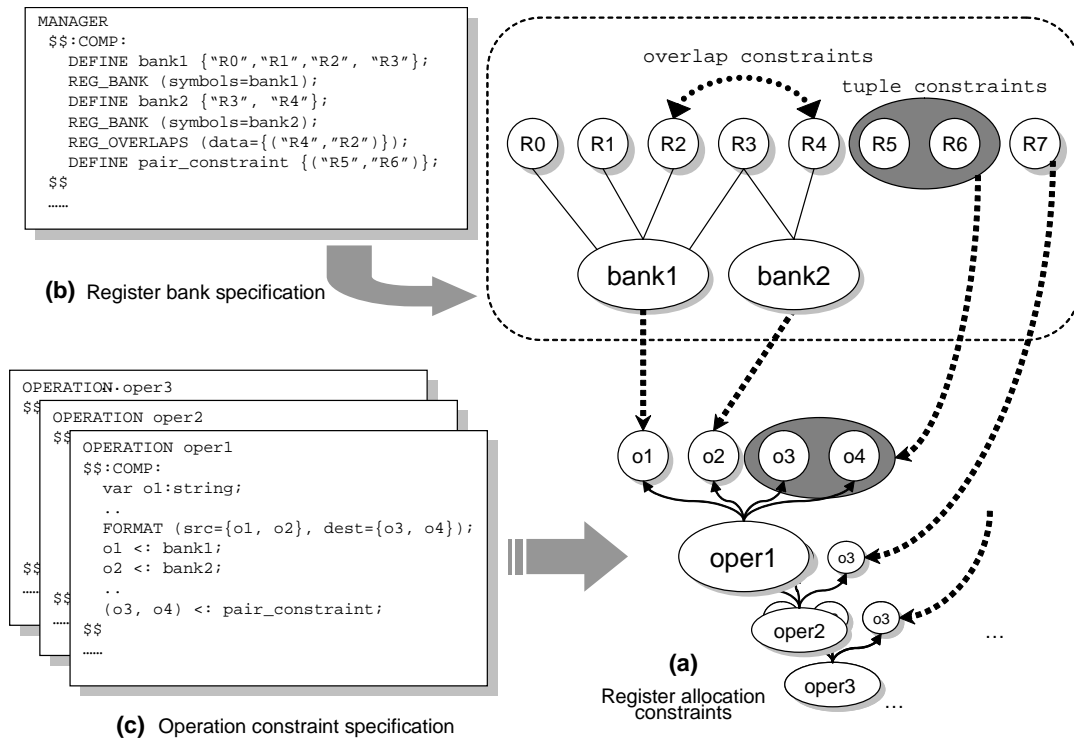


Figure 9: Register constraint specification in MADL

assigned to “R7” and the operand pair (“o3”, “o4”) of “oper1” must be assigned the register pair (“R5”, “R6”). In general, for the last constraint there may be more than one option for assignment. All the constraints are collectively termed as *allocation constraints*.

Now we describe how all the register information and operand constraint information can be specified in MADL using annotations. In the example shown in Figure 9, Figure 9(b) shows how the register information corresponding to the dotted box of Figure 9(a) can be specified in the MANAGER section of MADL. The *DEFINE* keyword is used to define a macro as the set of register symbols in physical or logical banks. The annotation command “REG_BANK” takes a symbol set and instantiates a register bank. In order to specify all the pairs of registers that overlap, the command “REG_OVERLAPS” is used. Figure 9(c) shows how the OPERATION sections of MADL can be annotated on to specify the various operand constraints corresponding to the lower half of the Figure 9(a). “FORMAT” is used to define the source and destination operands. The set containment operator is then used to constrain each one of the operands specified in “FORMAT”, for example, $o1 <: bank1$ constrains operand “o1” to be assigned a register from “bank1”. *Tuple constraints* can also be specified using the set containment operator as shown in Figure 9.

The irregular register allocation constraints are largely introduced by non-orthogonal instruction encodings designed to achieve high code density. However, it is not always possible for tools to infer automatically such constraints from the encodings. Therefore a proper annotation scheme is necessary.

As stated earlier, to demonstrate the capabilities of MADL, we extracted the processor specific ILinP specification for the Hiperion DSP [6]. We chose the Hiperion architecture since it is a highly irregular DSP with all the aforementioned register constraints. The practical experiences for this are discussed in Section 5.

4.4 Scheduler

The instruction scheduler is another important component of a compiler as it helps reduce the execution time of a program by effectively hiding instruction latencies. This is particularly true for embedded processors as many of them are statically scheduled and fully depend on the compiler for best performance. Reservation table based schedulers not only offer good performance, but are also highly retargetable. Hence they are among the most widely used types of schedulers. In this section, we describe how we can extract information for a reservation table based scheduler from an MADL description.

In order to build a reservation table, the scheduler needs to obtain information including physical or virtual resources that are exported to the scheduler such as issue slots and functional units, all possible execution paths for each operation, resource usages of each operation describing the time slots when different resources will be used, the layout of source and destination operands of each operation and their respective read and write latencies. It also requires register overlap information to detect data hazards.

Figure 10 gives an overview of how reservation tables can be extracted from an MADL description for an example operation. Figure 10(a) shows a part of a MANAGER section in which a token manager type, “unit_resource”, containing a single resource is defined. “IF” is defined as an instance of such a type. As a scheduling resource instance, it is assigned an alias “f_slot”. All other physical resources are similarly defined by annotations on their respective manager classes and instances. The MADL compiler can extract these physical resources by searching for annotation commands “SCHEM_RESOURCE” and “SCHEM_RESOURCE_TYPE” in the MANAGER sections.

Figure 10(c) shows the state machine for the example operation. The state diagram and the actions associated with its edges

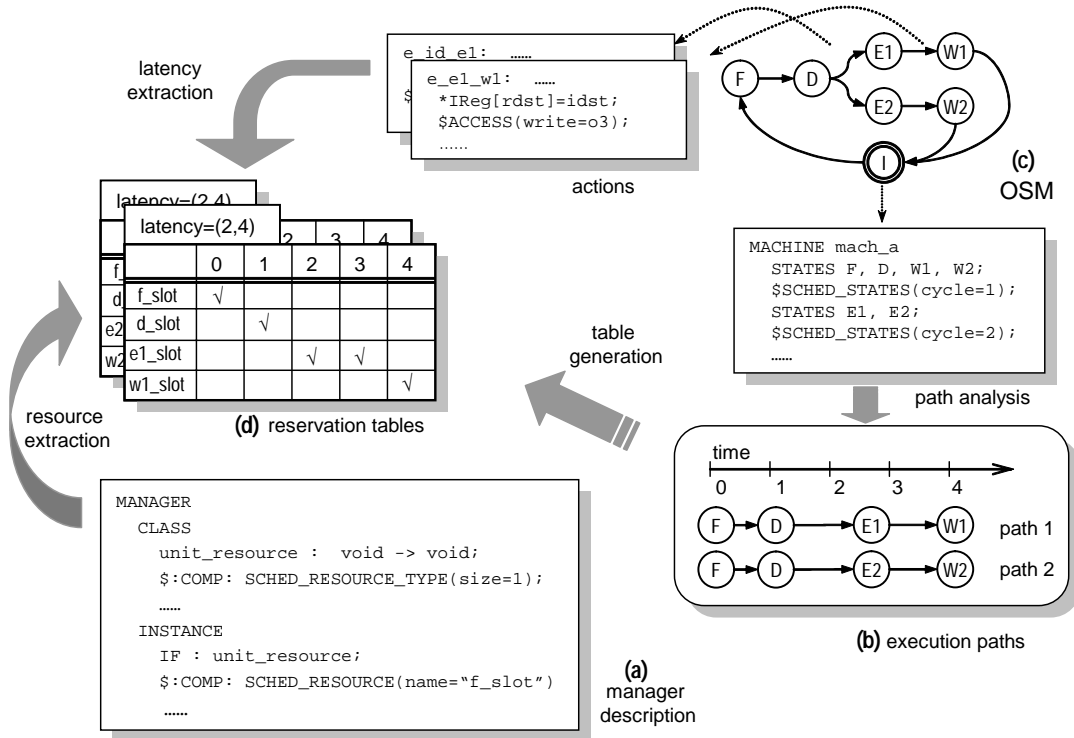


Figure 10: Overview of reservation table extraction from MADL

are all specified in the MACHINE and OPERATION sections in an MADL description. By analyzing the connectivity of the state diagram and its associated “SCHED_STATES” annotation commands, we can extract the relevant execution paths and nominal latencies. Moreover, by analyzing the actions on the edges, we can determine the resources used at each state. Take the OSM in Figure 1(b) for example. When the state machine proceeds from state *I* to *F*, the *allocate* request to the fetch manager is fired. Therefore we can immediately deduce that the resource “f_slot” is used by the state machine at state *F*. The resource is given up when the state machine proceeds to state *D* and fires the *release* action. Hence by examining all the *allocate*, *release* and *discard* actions along all execution paths, we can obtain the resource usage information of the operation.

The last few pieces of information needed to build the reservation table are the number and type (source/destination) of operands, and their read or write latencies. The “FORMAT” annotation statements in OPERATION sections are used to describe names and types of operands as shown in the example of Figure 9(c). In Figure 10, operand read/write activities are annotated by the “ACCESS” commands on the edges where the corresponding *read/write* actions occur. We can combine operand access information with the path latency information to get the operand latencies.

Finally, we now have all the information that we need to build the reservation tables shown in Figure 10(d).

Many embedded processors including DSPs have ISAs that are optimized for code density as well as performance. Their narrow instruction width introduces irregular ILP constraints, where the encoding allows only certain combinations of operations to be issued in parallel. In order to use a reservation table based scheduler, we use a pre-processing tool, the Artificial Resource Allocator (ARA) [15], to translate such encoding constraints to regular re-

source based constraints using artificial resources. The ARA takes the set of all combinations of operations that may be issued in parallel from MADL and augments the reservation table appropriately.

5. RESULTS

In order to demonstrate the effectiveness of the MADL-based software tool development approach, we modeled three architectures for software tool generation. They are the StrongARM core, the Fujitsu Hiperion DSP and an 128-bit subword parallel media processor named PLX [10]. Each of these represents a class of processors commonly used in embedded systems. The StrongARM core is a 5 stage pipelined implementation of the ARM V4 architecture. Though generally viewed as a RISC processor, the ARM ISA contains some CISC features such as the load-multiple instruction. Hiperion is a 16-bit DSP with an irregular ISA. Its instruction encoding is optimized mainly for code size and therefore is not fully orthogonal. PLX is a research architecture developed by a group of colleagues. It contains more than 200 integer and floating-point instructions. Its MADL model was developed by its designers.

We wrote a specification for the StrongARM in MADL to generate the cycle accurate simulator, the instruction set simulator and the reservation table for the scheduler. The Hiperion description is used to generate both simulators, the register allocator and the scheduler. The PLX description supports generating both simulators. Table 1 shows some statistics of the processor models. This contains the line counts of the MADL descriptions, the number of annotation paragraphs, the line counts of the token manager implementations in C++ for both types of simulators, and line counts of the synthesized C++ code from MADL for both types of simulators. The MADL compiler optimizes the synthesized cycle accurate simulators to minimize their code sizes without negatively affecting

their performance. This results in smaller source code size of the cycle accurate simulator than that of the instruction set simulator in the StrongARM and PLX cases.

| | StrongARM | Hiperion | PLX |
|------------------------|-----------|----------|--------|
| MADL lines | 2,001 | 3,346 | 3,389 |
| annot. para. | 135 | 368 | 10 |
| token managers | 494 | 417 | 349 |
| token managers (ISS) | 319 | 417 | 281 |
| synthesized code | 45,707 | 16,804 | 20,764 |
| synthesized code (ISS) | 62,816 | 12,243 | 27,110 |

Table 1: Description statistics

We performed all our experiments on a P4 2.8GHz Linux system with 1GB memory. The compilation of the MADL description in each case takes less than 10 seconds. The building of the synthesized simulators were done by g++3.3.2. The building time is less than 2 minutes for each simulator with g++ optimization switches “-O3 -finline-limit-1500 -fomit-frame-pointer”.

Table 2 shows the average simulation speed of all simulators. We used a mix of SPECINT benchmarks and MediaBench [9] to evaluate the StrongARM simulators. The Hiperion model was tested with the DSP-stone benchmarks and the PLX model was tested with several hand-coded assembly kernel loops.

| | StrongARM | Hiperion | PLX |
|------------------------|-----------|----------|------|
| cycle accurate (MHz) | 2.60 | 3.82 | 2.01 |
| ISS (Million inst/sec) | 8.26 | 11.5 | 2.40 |

Table 2: Simulation speed

In general, the synthesized cycle accurate simulators are faster than their hand-coded counterparts. Figure 11 compares the simulation speed of several StrongARM simulators, namely, the SimpleScalar target for StrongARM [3] (configuration file *salcore.cfg*), our hand-coded OSM simulator (*sima*) for StrongARM [14], and the simulator synthesized from MADL. The results show that the synthesized simulator has obvious simulation speed advantage, beating the best hand-coded simulator by 80% on average. The synthesized PLX cycle accurate simulator is also faster than the hand-coded simulator written by its designers, which runs at an average speed of 0.76MHz.

Currently the speed of the instruction set simulators are slower than those hand-coded ones. In the StrongARM case, the synthesized simulator runs at about 1/3 the speed of our hand-coded simulator. There are two main reasons for the difference. First, the synthesized simulator utilizes bit-true data types for state variables and for computation, while the hand-coded simulator uses native data types. The overhead of using the bit-true data types slows down the simulation speed. Such overhead also exists in synthesized cycle accurate simulators, but they are less sensitive to the overhead as data copying and computation constitute only a tiny fraction of their execution time. Second, since we use the same MADL description to generate the instruction set simulator as well as the cycle accurate simulator, the synthesized instruction set simulator executes the operations following the state diagrams designed for cycle accurate simulation. So for each operation, it will undergo the normal stages of fetching, decoding, evaluation and result write-back. In contrast, the the hand-coded simulator does not have to follow these stages and can thereby benefit from some processor specific optimizations. In the StrongARM case, the hand-coded simulator evaluates the predicate operand of an instruction prior to fully decoding it. If the predicate operand is false, the instruction will be

skipped and its decoding/evaluation can be saved. Such early evaluation of the predicate operand is not possible in the cycle accurate simulator due to timing constraints.

The speed of the instruction set simulators can be improved by using different MADL descriptions from those used to generate cycle accurate simulators. However, we believe that it is valuable to synthesize both types of simulators from the same description as it improves productivity and preserves consistency. We expect to improve the instruction set simulation speed by implementing more aggressive optimizations in the MADL compiler.

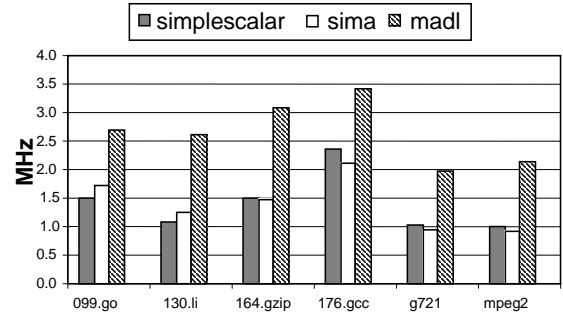


Figure 11: StrongARM simulation speed comparison

The MADL specifications for Hiperion and StrongARM include the annotations needed for compiler optimizations as shown in Table 1. From the core and the annotation layers of both descriptions, we extracted all the respective reservation table information such as scheduling resources, resource usages and operand latencies in a low level format. For the StrongARM, 6 physical resources and 548 reservation tables were generated, whereas, 5 physical resources, 7 virtual resources and 164 reservation tables were generated for the Hiperion. The virtual resources for Hiperion were generated by the ARA tool for irregular ILP constraint modeling.

The bulk of the annotation paragraphs for the Hiperion, however, are for the register allocator tool due to the irregular nature of the architecture. From the Hiperion MADL description, we automatically extracted the register allocation constraints for an *AMPL* based ILinP model. The total number of processor specific *AMPL* constraints generated is 660. In the presence of such a large number of constraints, automating the extraction of constraints from a machine description is of great value.

6. RELATED WORK

Different ADLs emphasize different aspects of microprocessors and adopt different architecture models. Their design is commonly influenced by the tools that they intend to support. This section gives an overview of the ADLs designed for supporting both simulators and compilers.

MIMOLA [18] and UDL/I [1] were mainly designed to support hardware synthesis. They are based on the discrete event model of computation and describe micro-architectures at the register transfer level. They can be used to model arbitrary digital hardware and therefore are among the most flexible ADLs. Due to such low level of abstraction, it is inefficient to model complex microprocessors with these languages. Moreover, processor specifications in these languages can only be analyzed and utilized by compilers under certain assumptions that constrain the target architectures to only those with simple pipeline control.

ISDL [7] and the early version of nML [4] were mainly designed for retargetable compiler generation. nML was later extended to include architecture-template-based pipeline information [16]. Both

| ADL | range | prod. | eff. | ease |
|--------------|-------|-------|------|------|
| MIMOLA,UDL/I | ++ | - | - | - |
| ISDL,nML | - | + | + | + |
| EXPRESSION | - | + | + | + |
| LISA | + | + | + | - |
| MADL | + | + | + | + |

Table 3: ADL comparisons

languages utilize register-transfer-lists to abstract the behavior of individual instructions. These languages can support both simulator and compiler generation for a small range of architectures such as DSPs.

EXPRESSION [8] is a research language developed for the synthesis of both compilers and simulators. It uses a coarse grained netlist to model the pipeline structure. The language was later extended with ad-hoc functional abstraction models for hardware control specification [11].

LISA [12] is a pipeline description language. It utilizes the L-chart to model operation flow in the pipeline. It adopts a C-like syntax and several pipeline control APIs to support the specification of pipeline control logic. With manual assistance for instruction latency specification, instruction schedulers can be generated from a LISA description for several DSP and ASIP architectures [17].

In Table 3 we compare the existing ADLs and MADL through metrics such as the *range* of supported architectures, the modeling *productivity*, the *efficiency* of synthesized simulators, and the *ease* to extract architecture properties for compilers' use. MADL is less flexible than MIMOLA or UDL/I since it models only processor cores. However, on the whole MADL is very well balanced in all four aspects. Therefore, we believe that it is a practical and promising language for use in programmable platform development.

7. CONCLUSION

In this paper, we extend the OSM architecture model to include the communication and computation actions. The model naturally captures the concurrency at the operation and the hardware layer as well as the mapping between them. Based on the OSM model, we present an architecture description language MADL that contains a core layer and an annotation layer. The two-layer structure separates key architecture properties from volatile implementation dependent information, thereby lengthening the lifetime of MADL. The language can be used to model a broad range of architectures. We demonstrate that with proper annotation schemes, MADL descriptions can be used to assist the generation of efficient cycle accurate simulators, instruction set simulators, register allocators and instruction schedulers. Our ongoing work involves describing more architectures and extending the descriptions for use in more tools, particularly compiler components.

8. ACKNOWLEDGMENTS

This work is part of the MESCAL project of the Gigascale Silicon Research Center sponsored by DARPA/MARCO. We thank Prof. Ruby Lee, Xiao Yang, and Zhenghong Wang for their generous help in developing the PLX model. We also thank the anonymous reviewers for their invaluable comments.

9. REFERENCES

[1] H. Akaboshi. *A Study on Design Support for Computer Architecture Design*. PhD thesis, Department of Information Systems, Kyushu University, Japan, 1996.

[2] A. Appel and L. George. Optimal spilling for CISC machines with few registers. In *Proceedings of the Conference on Programming Language Design and Implementation*, 2000.

[3] T. Austin, E. Larson, and D. Ernst. Simplescalar: An infrastructure for computer system modeling. *IEEE Computer*, pages 59–67, Feb 2002.

[4] A. Fauth, J. V. Praet, and M. Freericks. Describing instructions set processors using nML. In *Proceedings of Conference on Design Automation and Test in Europe*, pages 503–507, Paris, France, 1995.

[5] R. Fourer, D. Gay, and B. Kernighan. *AMPL: A Modeling Language for Mathematical Programming*. Duxbury Press, 2002.

[6] Fujitsu Limited. *Hiperion II - Digital Signal Processor User's Manual*, 1998.

[7] G. Hadjiyiannis, S. Hanono, and S. Devadas. ISDL: An instruction set description language for retargetability. In *Proceedings of Design Automation Conference*, pages 299–302, June 1997.

[8] A. Halambi, P. Grun, V. Ganesh, A. Khare, N. Dutt, and A. Nicolau. EXPRESSION: A language for architecture exploration through compiler/simulator retargetability. In *Proceedings of Conference on Design Automation and Test in Europe*, pages 485–490, 1999.

[9] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. MediaBench: A tool for evaluating and synthesizing multimedia and communications systems. In *International Symposium on Microarchitecture*, pages 330–335, 1997.

[10] R. B. Lee and A. M. Fiskiran. PLX: A fully subword-parallel instruction set architecture for fast scalable multimedia processing. In *Proceedings of the 2002 IEEE International Conference on Multimedia and Expo (ICME 2002)*, pages 117–120, August 2002.

[11] P. Mishra, N. Dutt, and A. Nicolau. Functional abstraction driven design space exploration of heterogeneous programmable architectures. In *Proceedings of the International Symposium on System Synthesis*, Oct 2001.

[12] S. Pees, A. Hoffmann, V. Zivojnovic, and H. Meyr. LISA – machine description language for cycle-accurate models of programmable DSP architectures. In *Proceedings of Design Automation Conference*, pages 933–938, 1999.

[13] W. Qin. <http://www.ee.princeton.edu/MESCAL/madl.html>, 2004.

[14] W. Qin and S. Malik. Flexible and formal modeling of microprocessors with application to retargetable simulation. In *Proceedings of Conference on Design Automation and Test in Europe*, pages 556–561, 2003.

[15] S. Rajagopalan, M. Vacharajani, and S. Malik. Handling irregular ILP within conventional VLIW schedulers using artificial resource constraints. In *International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, 2000.

[16] Target Compiler Technologies N.V. <http://www.retarget.com>, 2004.

[17] O. Wahlen, M. Hohenauer, and R. Leupers. Instruction scheduler generation for retargetable compilation. *IEEE Design & Test of Computers*, 20(1):34–41, Jan 2003.

[18] G. Zimmerman. The MIMOLA design system: A computer-aided processor design method. In *Proceedings of Design Automation Conference*, pages 53–58, June 1979.