A Unified Formal Model of ISA and FSMD

Jianwen Zhu, Daniel D. Gajski

Technical Report ICS-99-06
Feburary 1999
(Version as of October 14, 1999)

Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425, USA

jzhu@ics.uci.edu
gajski@ics.uci.edu

Abstract

In this paper, we develop a formal framework to widen the scope of retargetable compilation. The goal is achieved by the unification of architectural models for both the processor architecture and the ASIC architecture. This framework enables the unified treatment of code generation and behavioral synthesis, and is being used in our experimental codesign environment to drive system-on-a-chip synthesis from an object oriented language.

Contents

1 Introduction

For billion-transistor chip design, synthesis, reuse, and exploration are three important vehicles to help reduce design effort and improve system performance. In this setting, the conventional wisdom of Y chart [7], which defines synthesis as the generation of structure, or architecture, from the behavior, can be refined as the generation of low level representation from the behavior, to *configure* a reused architecture. This view is readily understood for programmable devices, where software compilation is the generation of instruction stream to "configure" the reused processors. With a little bit more thinking, behavior synthesis can be viewed as generation of a finite state machine to "configure" the data path, where the data path itself can be parameterized in terms of the numbers and types of the functional and storage units available. When the system design starts from a uniform specification in a programming language, the entire system synthesis task can then be best viewed as the familiar retargetable compilation.

It is thus necessary to establish the **architectural model** before any synthesis tasks can be carried out. Note that the purpose of an architectural model is neither collecting every piece of information one need to build the actual hardware, for example, the detailed netlist; nor repeating every piece of information one can find in a traditional architectural manual. Instead, the architectural model should serve as guidance for the corresponding synthesis tools and capture all (and only) the necessary architecture-dependent information one needs to know in order to "configure" the architecture for a given behavior.

Several issues need to be discussed for the architecture modeling.

- **Scope of architectural models:** The architectural model is needed at all abstraction levels. At the system level, the architecture is physically a network of processing elements, each of which might be a programmable device with an instruction set architecture (ISA); or a dedicated device with finite state machine with a data path (FSMD) architecture. The architectural model at this level must provide necessary information for system partitioning as well as interface synthesis.

  When we proceed to synthesize behavior mapped to each processing element, we need more detailed information for the corresponding devices. For an ISA architecture, the architectural model must provide the available instruction set, the storages, the calling conventions, frame layout, instruction level parallelism, etc. For the FSMD architecture, the available data path components, and sometimes their interconnections have to be provided.

- **Architecture configurability:** Architectural models vary on their hardware configurability. On the one extreme, an architectural model can have no constraints on the number and connection of architecture resources. For example, in logic synthesis, the synthesized network can have a virtually unlimited number of gates as needed. On the other extreme, for example, an ISA architecture, the hardware is fully determined. Many models fall into the category of partially constrained architecture. For example, traditional behavioral synthesis takes an architectural model which constrains the number and types of data path components (allocation). Their interconnections, however, are not constrained. This is in contrast to behavioral synthesis with data path cores, where the data path is fully constrained.

- **Forms of architectural models:** Architectural models exist in the form of APIs in most retargetable software compilers. Although efficiently implemented, they tend to be *ad hoc* and implementation-dependent. Another problem is that they are often difficult for human to create, modify and maintain. Recently, many instruction set architecture description languages have been developed to offer conciseness, intuitiveness, and verifiability, and more importantly, the automatic generation of compiler back-ends and simulation tools. Unfortunately, no language is established on a formal mathematical model. Lacking a formal model, the architecture specifier tends to be overwhelmed by the language syntax and the amount of information one must capture in a typical architecture, without appropriate understanding of their relationship. In fact, we found some specification languages are merely collection of C code for tools such as simulator and instruction selector for each instruction. The lack of a formal model also makes it difficult to construct target dependent compiler optimizations.

In this paper, we demonstrate the first step toward our goal of developing a formal architectural model for system-on-a-chip by a unified treatment of its most important components: the ISA and FSMD architecture. The rest of this paper is organized as follows. Section 2 reviews the related works and highlights our contributions. Section 5 discusses the behavioral models that are relevant to the subsequent discussion of architecture models. Section 6 discusses the instruction set architecture. Section 7 discusses the FSMD architecture. Section 8 describes the algorithms which unite the FSMD and ISA architecture.

2 Related Works

The earliest forms of explicit architecture model descriptions are various code generator generator (CGG) [3] [1] [5]. The CGGs typically use tree pattern specifications to drive the automatic generation of the instruction selector. However, such specification is often tied to a particular compiler implementation, for example, a particular intermediate representation. A more recent effort is the the set of computer system description languages in the Zephyr project, where [14] is devoted to the description of binary encoding of instruction set, [13] focuses on the semantics of instruction sets, and [2] describes the calling conventions. However, the models are not integrated and different aspects of the same architecture are scattered in different specifications in different languages. In addition, there is no explicit support of instruction level parallelism (ILP). The machine description language MDES [8] of the impact compiler, seems to be the most sophisticated in this aspect. In MDES, the architecture model is accurate enough to describe the superscalar, VLIW, as well as new architecture features such as predicated and speculative execution.

Architecture descriptions for irregular processors, for example, the DSP processors and application-specific instruction set processors (ASIPs), have also received intense interest in the recent years. Representative works are MIMOLA [12], where a hardware description language is used to describe the structural model of the processor; as well as nML [4] and ISDL [9], which take a more traditional approach. Representing more recent efforts, the Expression [10] architecture description language uses both the behavioral level model and the structural model to describes the processor architecture. In addition to the accurate modeling of ILP, Expression also features the explicit description of memory hierarchy.

A widely used ASIC architecture model is FSMD, where a formal definition can be found in [6]. However, the simple model given in [6] does not handle advanced features such as procedure call. How the ASIC interacts with its environment is also left unspecified. Furthermore, the model serves better as description of the synthesized ASIC, rather than the model which guides synthesis. In general, the architectural information captured in most behavioral synthesis systems is limited to allocation table, in the form of graphical user interface or compiler pragmas. Little work has been done on defining an architectural model flexible enough to describe a partially constrained architecture.

3 A Formal Approach

Our approach, as presented in this paper, is unique in the following aspect:

- **completeness**: Unlike previous works discussed, which focus on the ISA architecture, our model also covers the FSMD architecture. Our future work will extend this work to include the communication architecture, with a unified treatment of the local communications, that is, the calling conventions, as well as the system wide communications.
- **uniformness**: Our model unifies the apparently different FSMD and ISA architecture, this effectively helps to unify the software compiler and behavioral synthesizer. In fact, with our synthesis infrastructure, a behavioral synthesizer appears just as yet another target in the back-end.
- **formality:** Our model is the first to formally define the essential elements as well as their relationship in the architecture.

Despite its uniqueness, our work is in many respects inspired by the previous works. The concept of implicitly representing the instruction set of a processor using a structural description, was first proposed in MIMOLA [12] and detailed in [11]. Expression [10] also uses the same concept to reduce the size of processor specification. Our work differs in that we extend this concept to the FSMD architecture, and our structural specification is abstract, parameterized and partially constrained. Our model for ILP is an abstraction from the one in MDES [8].

4 Notations

We employ a set theoretic notation for the presentation of the formal model. The form of the notation is made deliberately to resemble an imperative programming language so that the notation is intuitive for developing algorithms later.

In our notation, a set is most conveniently conceived as data types in an imperative programming language. A set is denoted as an identifier. Some sets are predefined, for example, *int* is defined to be the set of integers. Most other sets are defined in terms of other sets:

- The notation $A \cup B$ represents the union of set A and set B.
- The notation $A \cap B$ represents the intersection of set A and set B.
- The notation 2^A represents the power set of set A.
- The notation $A \times B$ represents the cartesian set of set A and set B.
- The notation $A \mapsto B$ represents a function from set A to set B.
- The notation A^* represents language with alphabet A. Hence bit^* represents the set of binary strings.
- The notation

$$\begin{array}{lll} \textbf{set} & D\ \{ & \\ & a & : A; \\ & b & : B; \\ & c & : C; \\ & \} & \end{array}$$

  defines set D as the set of tuples $\langle a, b, c\rangle \in A \times B \times C$.

The notation $a : A$ declares that an element a is a member of A.

An algorithm can be then specified as a function over a set of parameters, each of which belongs to certain sets, and returns a value in another set. For example,
function $func(a : A, b : B) : C$ {
...
}
represents a function $func$ which takes $a \in A$ and $b \in B$ as parameters and returns a value in C. Constructs such as conditional, loop and expressions can be used in the body of the function to specify the computation.

5 Behavioral Model

It is important in the architectural model to associate an architecture resource with the behavior that it can implement. We define our model of behavior in terms of trees.

Definition 5.1 *A* ***tree*** *over an alphabet V is a member of*

$$\begin{array}{lll} \textbf{set} & Tree\langle V\rangle\ \{ & \\ & kind & : V; \\ & rank & : int; \\ & kids & : Tree\langle V\rangle^*; \\ & \} & \end{array}$$

, where kind is the type of the tree root, kids is a sequence of trees representing the successors of the root, and rank is the number of the successors. □

A piece of behavior, called the **behavior pattern**, can then be defined as follows:

Definition 5.2 *A* ***behavior pattern*** *$lhs \leftarrow rhs$ over the* ***terminal*** *set Σ_t and* ***nonterminal*** *set Σ_n is a member of*

$$\begin{array}{lll} \textbf{set} & BP\langle \Sigma_t, \Sigma_n\rangle\ \{ & \\ & lhs & : \Sigma_n; \\ & rhs & : Tree\langle \Sigma_t \cup \Sigma_n\rangle.; \\ & \} & \end{array}$$

The set of nonterminal that appears in rhs is called the ***operands*** *of the tree. Note that $\forall n \in Tree\langle\Sigma_n\rangle, n.rank = 0$. And typically, the terminal set Σ_t is taken as $T \times O$, where T is the set of basic* ***data types****, and O the set of* ***opcodes****.* □

It is trivial to design Algorithm 5.1 to access the operands of the given behavior pattern, and Algorithm 5.2 to expand a behavior pattern at a given operand, or to replace it with another behavior pattern.

Algorithm 5.1 *Locating operands.*
function $numOfOperands(bp : BP\langle\Sigma_t, \Sigma_n\rangle)$
$: int;$
function $operand(bp : BP\langle\Sigma_t, \Sigma_n\rangle, i : int)$
$: BP\langle\Sigma_t, \Sigma_n\rangle;$
□

Algorithm 5.2 *Expansion of behavior patterns.*
function $expand(bp, kid : BP\langle\Sigma_t, \Sigma_n\rangle, i : int)$
$: BP\langle\Sigma_t, \Sigma_n\rangle;$
□

Definition 5.2 is adopted by many tree-based CGGs such as BURG [5]. Typically, the nonterminal set Σ_n is taken as the **value holders**, such as storages, in the architecture model. This effectively makes the behavior pattern architecture-dependent, and hence a large set of behavior patterns has to be captured for every architecture.

Alternatively, the behavior pattern can be orthogonally decomposed into the computational part, called **value transform**, and the communication part, called **value transfer**. The value transform models the function of behavior pattern to transform values, and is hence architecture-dependent. The value transfer only models how the pattern transfer values between value holders.

Definition 5.3 *A* ***value transform*** *is a member of* $Tree\langle\Sigma_t \cup \{k_i\}\rangle$. *A* ***value transfer*** $lhs \leftarrow rhs$ *is a member of* $\Sigma_n \times \Sigma_n^*$.

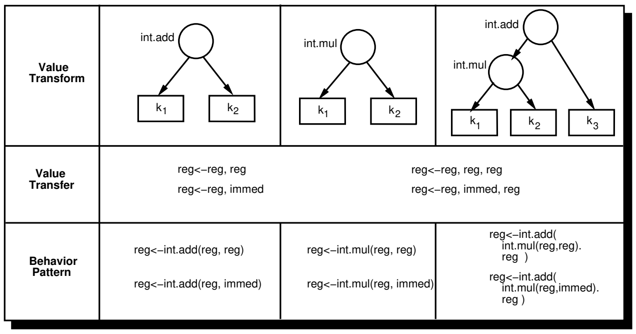

Figure 1. Behavior patterns.

Example 5.1 shows some examples of value transforms and transfers, and behavior patterns as their combination.

Example 5.1 *Let* $\Sigma_t = \{int.add, int.sub, int.mul, int.div\}$. *Let* $\Sigma_n = \{reg, immed\}$. *Then* $int.add(k_1, k_2)$ *is a value transform representing the addition of values of int type.* $int.add(int.mul(k_1, k_2), k_3)$ *is a value transform representing the* MAC *operation of three values of int type. Both* $reg \leftarrow reg, reg$ *and* $reg \leftarrow reg, immed$ *are value transfers.* $reg \leftarrow int.add(reg, reg)$ *and* $reg \leftarrow int.add(reg, immed)$ *are behavior patterns.*

The decoupling of value transforms from value transfers makes it possible to capture them separately. Since value transforms are architecture-independent, they can be shared by all the architecture models. However, for convenience, we will use the first definition for subsequent discussions.

6 ISA Architecture

An instruction set architecture is characterized by its instructions, storages, the instruction level parallelism, as well as communication schemes such as calling conventions. Definition 6.1 gives our formal model of an ISA architecture.

Definition 6.1 *An* ***ISA architectural model*** *is a member of*

$$
\begin{array}{lll}
\textbf{set} & ISA\ \{ & \\
 & m, n & : int; \\
 & S & : 2^{Store}; \\
 & I & : 2^{Instrn}; \\
 & SC & : 2^{I \times I \times int}; \\
 & TC & : I \times I \mapsto int \times int \times int; \\
 & CA & : COMM; \\
 & \} &
\end{array}
$$

where m *is the size of the instruction word;* n *is the number of pipeline stages;* S *is the set of* ***stores;*** I *is the set of instructions;* SC *is the* ***spatial constraint,*** TC *is the* **temporal constraint**, *and* CA *is the communication architecture.* □

To simplify the model, we assume that the size of the instruction word is constant: For example, for a typical 32-bit RISC processor, m is 32; for a four-issue VLIW processor, m may be 128; for a four-issue superscalar processor, m is 32. The irregularity of CISC processors causes some problems, but it can be easily handled in the implementation. The model for communication architecture $COMM$, however, will not be covered in this paper.

6.1 Stores

Much the same way as the addressing modes one can find in the architectural manual, stores (Definition 6.2) model processor storage resources such as register files and memories. The information of interest is the set of cells (Definition 6.3) that it contains as well as the way they can be *accessed* and *allocated*.

Definition 6.2 *A store is a member of set*

$$
\begin{array}{lll}
\textbf{set} & Store\ \{ & \\
 & C & : 2^{Cell}; \\
 & base & : Cell; \\
 & offset & : int; \\
 & AS & : bit^*; \\
 & AF & : AS \times T \mapsto AS \times int; \\
 & \} &
\end{array}
$$

, where C *is the set of* ***cells*** *that the store contains; base is a cell representing the base of addressing;* $offset$ *is an integer representing the additional offset used in the addressing;* AS *is a finite set of* ***allocation state;*** *and* AF *is the* ***allocation function.*** □

Definition 6.3 *A cell is a member of*

$$
\begin{array}{lll}
\textbf{set} & Cell\ \{ & \\
 & n & : int; \\
 & \} &
\end{array}
$$

, *where* n *is the number representing its offset in the store.* □

A store is called a **finite store** if the number of its cells $|C|$ is finite, it is called **infinite store** otherwise. Register files are usually finite stores since they contain a fixed number of cells: the registers. The immediate stores, fall into the category of infinite stores, since their cells are created "on-demand". Although memory stores are physically finite, it is conceptually simpler to consider them to be infinite. Note that the fields $base$ and $offset$ make sense only in the memory store.

The allocation of cells in each store is characterized by its allocation state space. The allocation state space of a register file can be modeled as a bit vector, where each bit corresponds to a register of the smallest granularity. For example, a 32-bit register in a typical RISC machine occupies one bit in the vector. A register pair used to hold a 64-bit value, on the other hand, occupies two bits in the vector. The allocation state space of a memory store is usually modeled after the alignment status of the current available memory location. For example, state 100 represents that the current location is byte aligned, but not word or double word aligned. On the other hand, 111 represents that the current location is double word aligned, and hence is automatically byte and word aligned.

Given the current allocation state and a data type, the allocation function can allocate a new cell by updating the allocation state and output the allocated cell number.

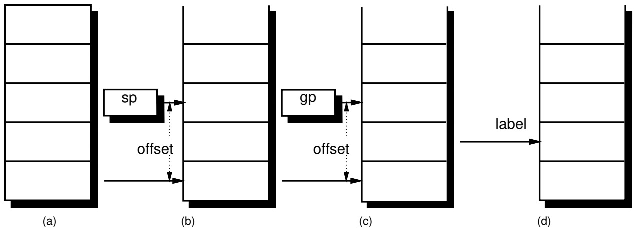

Figure 2. Stores: (a) general purpose register file (b) stack (c) globals (d) absolute addressing.

Example 6.1 *Figure 2 (a) shows a general purpose register file; (b) shows a stack memory which is addressed by the stack pointer sp plus an offset; (c) shows the global memory which is addressed by the global pointer gp plus an offset; (d) shows the memory store which is addressed by the absolute immediate values.* □

6.2 Instructions

The instruction (Definition 6.4) models processor computational resources. The information of interest is its semantics and binary encoding. Modeled as a behavioral pattern, the instruction semantics helps to identify the behavior that the instruction can implement. When a behavior is identified to be implemented by a particular instruction, the storage allocation must be performed to determine the **destination**, which is essentially a cell within the store corresponding to the left hand side of the pattern; and the set of **sources**, which are cells within the stores corresponding to the operands of the pattern.

The instruction encoding is modeled by the *opcode*, which is a bit pattern of size m; the *dest*, which is a field (Definition 6.5) specifying the offset and width of the instruction destination; and *srcs*, which are fields strings for instruction sources.

Definition 6.4 *An instruction is a member of set*

$$
\begin{array}{lll}
\textbf{set} & \mathit{Instrn}\ \{ & \\
 & \mathit{pattern} & : BP\langle O \times T, \mathit{Store}\rangle; \\
 & \mathit{opcode} & : \mathit{bit}^*; \\
 & \mathit{dest} & : \mathit{Field}; \\
 & \mathit{srcs} & : \mathit{Field}^*; \\
 & \} &
\end{array}
$$

, *where pattern is the instruction semantics, opcode, dest, srcs is the instruction encoding.* □

The field is characterized by its offset and width.

Definition 6.5 *A **field** is a member of*

$$
\begin{array}{lll}
\textbf{set} & \mathit{Field}\ \{ & \\
 & \mathit{offset}, \mathit{width} & : \mathit{int}; \\
 & \} &
\end{array}
$$

, *where $offset$ is its offset within the instruction word, and $width$ is the width of the field.* □

Given a field and an integer number, Algorithm 6.1 can be used to compute the corresponding bit pattern. The instruction encoding can then be computed as the bitwise-or of its opcode, and the field patterns of its destinations and sources, given the result of storage allocation. Note that for some static memory stores, the storage offset is resolved only at link time, where techniques such as relocation must be used.

Algorithm 6.1 *Field pattern.*
function $getFieldPattern(field : Field, cell : int) : bit^*$**;**
□

6.3 Constraints

The temporal constraint (Figure 3 (b)) models temporal parallelism between the instructions. Modern microprocessors are always pipelined, and hence allow the interleaved execution of instructions. However, the possibility of interleaving is limited by various forms of dependency between instructions: the flow-dependency (read after write), the anti-dependency (write after read), and the output-dependency (write after write). The situation is further complicated by the processor's bypassing capability. To complicate matters even more, some processors allow bypassing within the same functional unit, but not across. Our model of temporal constraint maps any pair of instruction i_1, i_2, into a triple of numbers $TC(i_1, i_2) = \langle d_{flow}, d_{anti}, d_{output} \rangle$, where each number indicates the minimum number of cycles that i_2 should be scheduled after i_1 for flow, anti and data dependency respectively.

The spatial constraint models (Figure 3 (a)) spatial parallelism between the instructions. Modern microprocessors, whether a superscalar or VLIW architecture, contain multiple functional units to allow the simultaneous execution of instructions. Such parallelism is limited only by the processor resources. An entry $\langle i_1, i_2, d \rangle$ in the spatial constraints indicates that i_2 cannot be scheduled at d cycles later than i_1. On the other hand, if $\forall d, \langle i_1, i_2, d \rangle \notin SC$, then i_1, i_2 can be scheduled with a delay of any cycles, or, can be executed in parallel. Note that it is always the case that $d \geq 0 \wedge d < n$, where n is the number of pipeline stages.

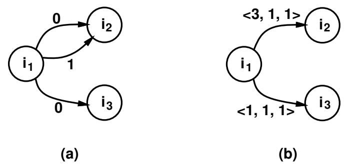

Figure 3. Constraints: (a) spatial constraints (b) temporal constraints.

7 FSMD Architecture

An ASIC is typically implemented in the FSMD architecture, which consists of a control path and a data path. The control path implements a finite state machine which generates a set of control signals, called the control word, at every clock cycle. The control word includes signals which specify the opcode of the functional units, the load signal of a register, the register number of the register file, or the enable signals of the bus driver. In combination, they specify a set of register transfer operations that should be performed by the data-path at a particular cycle.

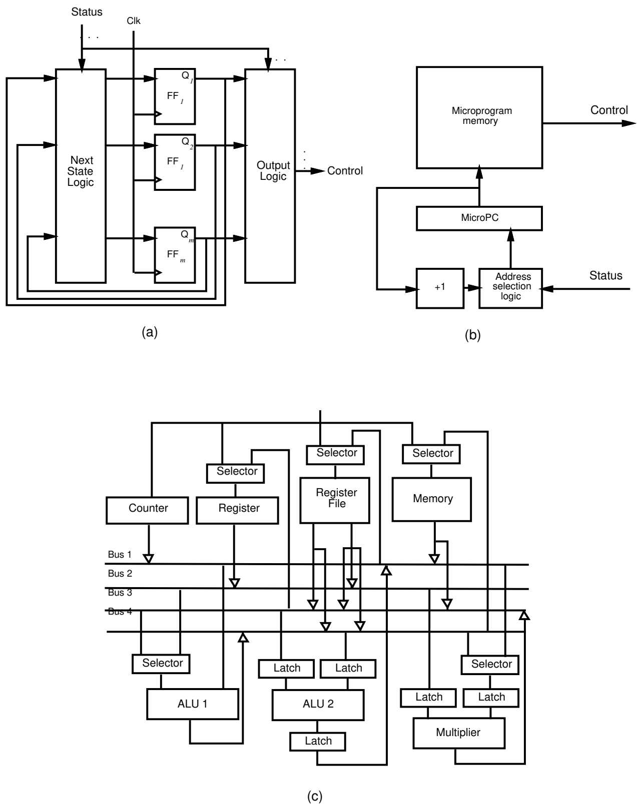

Figure 4. FSMD model (a) random logic based and PLA based control path (b) micro-code control path (c) data-path.

Our formal model of FSMD is given in Definition 7.1, which specifies the control path implementation style, the set of units and busses in the data path, as well as their interconnection.

Definition 7.1 *An* ***FSMD Architectural Model*** *is a member of*

$$\begin{array}{lll} \textbf{set} & FSMD\ \{ & \\ & c & : \{random, pla, rom\}; \\ & U & : 2^{Unit}; \\ & B & : 2^{Bus}; \\ & PMF & : \bigcup_{u \in U} u.P \mapsto B; \\ & CA & : COMM; \\ & \} & \end{array}$$

, where c is the control path implementation style, U is the set of units, B is the set of buses, PMF is the port

map function, and CA is the communication architecture. □

The controller implementation style can be random logic based or PLA based, as shown in Figure 4 (a), where a set of flip-flops are used to keep the states, and two pieces of combinational logic, the next state logic and output logic, are used to compute the next state and value of the control word, respectively. Alternatively, the controller can be microcode based, as shown in Figure 4 (b), where the control words are stored in a ROM and fetched according to the value of the microcode program counter.

The data path (Figure 4 (c)) contains a network of functional units, such as ALUs and multipliers; storage units, such as register files or memories; and steering units such as multiplexers and bus drivers. Definition 7.2 provides an abstraction for all of them. A unit contains a set of input and output data ports, which are eventually mapped to the buses in the data path. It also contains a set of control fields, which are connected to the control signals. For a storage unit, it also contains a store, as defined in Definition 6.2. The functions that the unit is able to perform are specified by its operations.

Definition 7.2 *A* ***unit*** *is a member of*

$$\begin{array}{lll} \textbf{set} & Unit\ \{ & \\ & P & : 2^{Port}; \\ & F & : 2^{Field}; \\ & S & : 2^{Store}; \\ & OP & : 2^{Operation}; \\ & \} & \end{array}$$

, where P is a set of ports; F is the set of control fields, S is its associated stores, OP is a set of operations. □

A control field (Definition 6.5) is characterized by its offset and width in the control word. While the width is a fixed value, the offset is determined only when the associated unit is instantiated in an FSMD.

The set of operations that a unit can perform is characterized by their behavior patterns, and the corresponding control configurations. Note that the non-terminal set of the behavior pattern is limited to the ports and store of the unit. The control configuration is characterized by a set of control fields, and the corresponding methods to compute the control values, which are specified either directly by integer numbers; or by stores. For those specified as stores, the control values are taken as the cell numbers as the results of storage allocation.

Definition 7.3 *An* ***operation*** *of unit u is a member of*

$$\begin{array}{lll} \textbf{set} & Operation\ \{ & \\ & p & : BP\langle O \times T, u.P \cup u.S\rangle; \\ & c & : 2^{u.F \times (int \cup u.S)}; \\ & \} & \end{array}$$

, where p is its behavior pattern, f is the control configuration in order to enable the operation. □

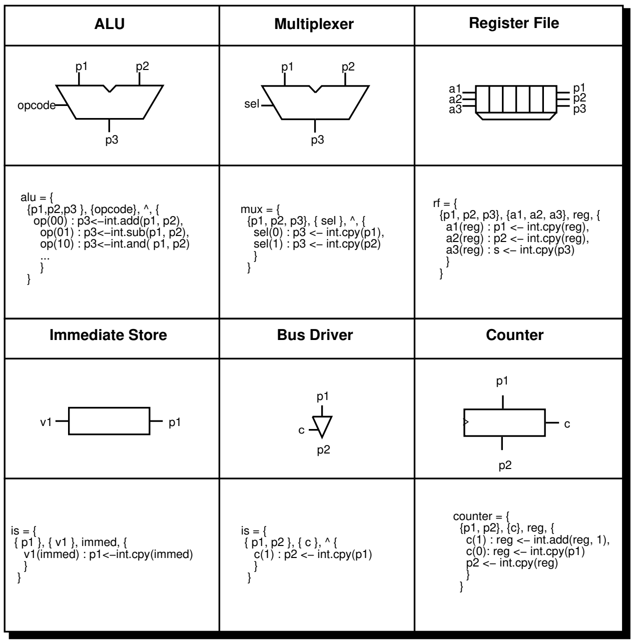

Figure 5. Units.

Example 7.1 *Figure 5 shows a unit library containing an ALU, a multiplexer, a register file and a bus driver.* □

In our model, the notion of bus goes beyond the group of physical wires. They are used to indicate the possible data transfers between the units. While a shared bus does imply one physical interconnection between the set of units, the corresponding steering units such as bus drivers and keepers can be automatically generated when the ASIC is finalized according to the application. As another example, an on-demand bus implies as many point-to-point connection as needed by the application.

Definition 7.4 *A* ***bus*** *is a member of*

$$\begin{array}{lll} \textbf{set} & Bus\ \{ & \\ & type & : \{shared, on\_demand\}; \\ & \} & \end{array}$$

where type is the type of the bus. □

To summarize, the information that characterizes an ASIC architecture captures only an initial, coarse idea of the implementation by specifying the controller style and an abstract netlist of the data-path. The exact, optimal form of register transfer level implementation will be generated by adapting the architecture to a particular application.

8 Bridging the Gap between FSMD and ISA

The FSMD model defined in Section 7, which naturally captures the ASIC architecture, has an apparently different structure than the ISA model defined in Section 6. To perform the mapping of behaviors to these two architecture, the code generator (for the ISA), and the behavioral synthesizer (for the FSMD), have to go through similar procedures such as control/dataflow analysis, target-independent optimizations, instruction selection (binding), scheduling and emission. Unifying these two architectural models can help unify these procedures, and consequently merge the different tools into one. Obviously, such unification can greatly reduce the development effort, and simplify the user interface of the synthesis tool.

This goal is achieved by deriving an ISA model $\langle m, n, S, I, SC, TC, CA\rangle$ from the FSMD model $\langle c, U, B, PMF, CA\rangle$. Once such a unifying algorithm is found, we can elaborate the synthesizer as shown in Figure 6. Here, two APIs can be defined according to the ISA model and FSMD model defined in order to capture different processor and ASIC configurations. While the processor configurations can be captured directly into an ISA model, the ISA model of the ASIC configuration can be derived from the unifier. Synthesis tasks, such as binding, scheduling and register allocation can then consult a single data structure of ISA model which captures all the target dependent information. In the sequel, we discuss how such a unifying algorithm can be developed.

8.1 Deriving Instruction Layout

In the derived ISA model, the instruction word is viewed as the control word of the FSMD model. Hence n can be easily computed by summing up the width of all the control fields, as shown in Algorithm 8.1,whose side effect is to determine the offset of each control field of the FSMD units.

Algorithm 8.1 *Derivation of instruction layouts.*
function $deriveInstrnLayout(fsmd : FSMD)$
  $: int$ {

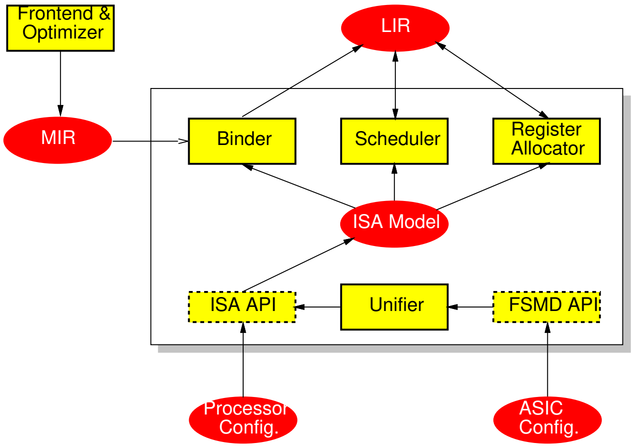

Figure 6. Synthesizer block diagram.

var $offset : int = 0;$

forall $u \in fsmd.U$ {
  forall $f \in u.F$ {
    $f.offset = offset;$
    $offset = offset + f.width;$
    }
  }
return $offset;$
}
□

8.2 Deriving Stores

The set of stores S can be computed simply by enumerating all the stores associated with the FSMD units.

Algorithm 8.2 *Derivation of stores.*
function $deriveStores(fsmd : FSMD)$
  $: 2^{Store}$ {
var $S : 2^{Store} = \emptyset;$

forall $u \in fsmd.U$ {
  if $(u.s \neq \perp)$
    $S = S \cup u.S;$
  }
return $S;$
}
□

8.3 Deriving Partial Instruction Set

The remaining problem is the derivation of the set of instructions as well as the spatial constraints. It can be solved by first deriving the **partial instruction sets** associated with the value holders, that is, the buses and the stores of the FSMD. Intuitively, a partial instruction stands for a storage-to-storage or storage-to-bus operation. In other words, a partial instruction associated with a value holder has a behavior pattern $lhs \leftarrow rhs$ which maintains the following invariant: lhs must be equal to the value holder itself and each operand of rhs must be stores. The partial instruction is also associated with encoding information as well as its resource usage, which is essentially a set of buses or units.

Definition 8.1 *A **partial instruction** in an FSMD architecture $\langle c, U, B, PMP, COMM\rangle$ is a member of*

$$
\begin{array}{lll}
\textbf{set} & PI\ \{ & \\
 & p & : BP\langle O \times T, \bigcup_{u\in U} u.P \cup u.S\rangle; \\
 & opcode & : bit^*; \\
 & dest & : Field; \\
 & srcs & : Field^*; \\
 & R & : B \cup U; \\
 & \} &
\end{array}
$$

, where p is the behavior pattern, $opcode, dest, srcs$ models the encoding, R is the set of resources used to execute the partial instruction.

Algorithm 8.3 composes a set of partial instructions into a new partial instruction.

Algorithm 8.3 *Composition of partial instructions.*

```
function composePI(
        fsmd : FSMD,
        u : Unit, op : Operation, kids : 2^PI
        ) : PI;

function getValueHolder(
        fsmd : FSMD,
        v : Port ∪ Store
        ) : Bus ∪ Store {
    if v instanceof Port
        return fsmd.PMF(v);
    else
        return v;
    }
```

□

The partial instruction set of each value holder can be derived by Algorithm 8.4 following a topological order, that is, if value holder a is used in an operation of a unit to compute a value in the value holder b, then the partial instruction set of a is first computed.

Algorithm 8.4 *Derivation of partial instruction set.*

```
var pimap : Bus ∪ Store ↦ 2^PI;
procedure derivePIS(fsmd : FSMD) {
  forall v ∈ fsmd.B ∪ ⋃_{u∈fsmd.U} u.S
          in topological order {
      var PIS, PIS1 : 2^PI = ⊘;

      forall u ∈ fsmd.U
      forall op ∈ u.OP
      ∧getValueHolder(op.p.lhs) = v {
          PIS1 = PISPerOp(fsmd, u, op);
          PIS = PIS ∪ PIS1;
          }
      pimap = pimap ∪ ⟨v, PIS⟩;
      }
  }

function PISPerOp(
        fsmd : FSMD, u : Unit, op : Operation
        ) : 2^PI {
    var PIS : 2^PI = ⊘;
    var k : int = numOfOperands(op.p);

    forall kids ∈ PI^k {
        if (∀i, doesPIMatch(kids, op, i))
            var pi : PI;
            pi = composePI(fsmd, u, op, kids);
            PIS = PIS ∪ {pi};
            }
        }
    }

function doesPIMatch(
        kids : PI*, op : Operation, i : int)
        : boolean {
    var v1, v2 : Store ∪ Bus;
    v1 = kids[i].p.lhs;
    v2 = getValueHolder(operand(op.p, i));
    return v1 = v2;
    }
```

□

8.4 Deriving Spatial Constraints

Each instruction can be mapped to a partial instruction. Given a set of instructions, and its mapping to the partial instructions, the spatial constraints of the instruction set can be computed by examining if there are resource conflicts between any pair of instructions, as shown in Algorithm 8.5.

Algorithm 8.5 *Derivation of spatial constraints.*

```
function deriveSC(
```

$I : 2^{Instrn}, imap : Instrn \mapsto PI$
$) : 2^{I \times I \times int}$ {
var $SC : 2^{I \times I \times int} = \oslash$;
forall $i, j \in I, i \neq j$ {
if$(imap(i).R \cap imap(j).R \neq \oslash)$ {
$SC = SC \cup \langle i, j, 0 \rangle$;
}
}
return SC;
}
□

8.5 Deriving Instruction Set

The instruction set can be derived by enumerating the partial instruction set associated with the stores of the FSMD. Since our current FSMD model does not involve pipelined control, it is easy to conclude that $m = 1$ and $TC = \oslash$. The communication architecture can be directly inherited. The other information needed for an ISA model can be derived by the methods discussed.

Algorithm 8.6 *Derivation of ISA from FSMD.*
function $deriveISA(fsmd : FSMD) : ISA$ {
var $m : int$;
var $S : 2^{Store}$;
var $I : 2^{Instrn} = \oslash$;
var $SC : 2^{I \times I \times int} = \oslash$;
var $imap : Instrn \mapsto PI$;

$m = deriveInstrnLayout(fsmd)$;
$S = deriveStores(fsmd)$;
$derivePIS(fsmd)$;
forall $s \in S$ {
var $PIS : 2^{PI} = pimap(s)$;
forall $pi \in PIS$ {
var $instrn : Instrn$;

$instrn = \langle pi.p, pi.opcode, pi.dest, pi.srs \rangle$;
$imap = imap \cup \langle instrn, pi \rangle$;
$I = I \cup instrn$;
}
}
$SC = deriveSC(I, imap)$;
return $\langle m, 1, S, I, SC, \oslash, CA \rangle$;
}
□

9 Experimental Results

In this section, we present experimental results to demonstrate the use of the architectural model developed to characterize real life architectures, and the effectiveness of the unifying algorithm to automatically derive instruction set from the ASIC architectures.

9.1 Unit Library

For experimental purposes, we construct a unit library with relevant information outlined in Table 1. This library includes arithmetic units such as adder, subtracter, and ALU. It also includes storage elements such as register file, dual-port register and register. Steering units, such as multiplexers and bus drivers, are also included. While the above mentioned units correspond very well to real components, abstract components are also allowed. One example is the point-to-point register file. While it appears as a normal register file, its size is on demand and determined by the synthesis tool. In addition, the registers in the register file are not grouped to share reading and writing ports. Instead, they will have point to point connections with other components in the architecture, which is in turn determined by the synthesis tool when the final implementation is resolved.

The unit library is implemented as a class library in OpenJ. The class library is designed to follow the model in Definition 7.2, while slight changes are made for implementation efficiency.

9.2 Fully Constrained Architecture

The FSMD model is also implemented as a class library, where the capture of a custom ASIC architecture involves only subclassing the appropriate classes and instantiating the desired units and buses, as well as their connection.

We first captured an architecture which is originally designed for an elliptic filter ASIC. The data-path of the architecture is shown in Figure 7, which include two adders, two multipliers, two roms and eleven registers. Note that this architecture is fully constrained in the sense that the connection in the data-path is fully determined and cannot be configured with the application to be synthesized with this architecture.

The description of the architecture takes 45 lines of OpenJ code. When the unifying algorithm is applied, it takes less than 15 milliseconds, and generates an instruction set with 190 instructions.

9.3 Partially Constrained Architecture

The architecture discussed in the previous section can be refined in such a way that it can be adapted

Name	Function	Operations	Ports	Stores	Controls
Adder	$+$	1	3	0	0
Sub	$-$	1	3	0	0
Mult	$*$	1	3	0	0
Div	$/$	1	3	0	0
Cmp	$< > ==$	3	3	0	1
Sft	$<< >>$	2	3	0	1
Alu	$+, -, \&, \vert, \tilde{}, \hat{}$	6	3	0	1
Mux2	multiplex	1	3	0	1
Reg	register	2	2	1	1
Cnt	counter	3	2	1	1
Rom	read only memory	1	1	1	1
RF1r1w	register file	2	2	1	2
RF2r1w	dual-port register file	3	3	1	3
P2PRF	point-to-point register file	3	3	1	vary
Drv	bus driver	2	2	0	1

Table 1. Unit library.

to the application. Figure 8 shows such as a refinement where the interconnection between storage units and functional units are configurable by using the "on demand" type bus (Definition 7.4).

The description of the architecture in Figure 8 takes 35 lines of OpenJ code, which is shorter than before, as expected. The unifying algorithm generates an instruction set with 7436 instructions in 370 milliseconds. Note that this architecture allows much more possible register transfers and their enumeration takes longer.

A even less constrained architecture is shown in Figure 9, where the set of registers in Figure 8 is replaced with a point-to-point abstract register file.

The description of the architecture in Figure 9 takes only 12 lines of code. By applying the unifying algorithm, we generate an ISA architectural model with 36 instructions in 4 milliseconds.

9.4 Further Experiments

We conducted similar experiments with real architectures extracted from real ASIC designs. The results are summarized in Table 2. Here Atm1 and Atm2 are abstracted from design for an ATM transceiver. JPEG is abstracted from an JPEG ASIC. And Vocoder is abstracted from an ASIC designed for GSM speech codec. We showed the complexity of the architecture in terms of number of lines of OpenJ code used. The number of instructions for the generated instruction set as well as the unifying algorithm runtime in milliseconds are also shown.

10 Conclusion

We have presented the formal models for the ISA and FSMD architectures. These formal models can serve as the basis of architecture description for an retargetable compiler environment, and represents the first step towards unifying code generation and behavioral synthesis. Future work on architecture modeling needs to include calling convention for both the FSMD and ISA architecture, as well as pipelined control for the FSMD architecture.

11 References

[1] A. Aho, M. Ganapathi, and S. Tjiang. Code generation using tree pattern matching and dynamic programming. *ACM TOPLAS*, 11(4):491–516, October 1989.

[2] M. Bailey and J. Davidson. A formal model and specification language for procedure calling conventions. Technical Report CS-94-39, Computer Science, University of Virginia, 1994.

[3] R. Cattell. Code generation and machine descriptions. Technical Report CSL-79-8, Xerox Palo Alto Research Center, October 1979.

[4] A. Fauth, J. Praet, and M. Freericks. Describing instruction sets using nML. Technical report, Technische Universiteat Berlin and IMEC, Berlin(Germany)/Leuven(Belgium), 1995.

[5] C. W. Fraser, R. R. Henry, and T. A. Proebsting. BURG—fast optimal instruction selection and tree parsing. *SIGPLAN Notices*, 27(4):68–76, April 1992.

Architecture	complexity (#lines)	#instructions	runtime (ms)
Ellip1	45	190	15
Ellip2	35	7436	370
Ellip3	12	36	4
Atm1	12	7	2
Atm2	20	12	3
JPEG	20	3468	199
Vocoder	25	438	34

Table 2. Results.

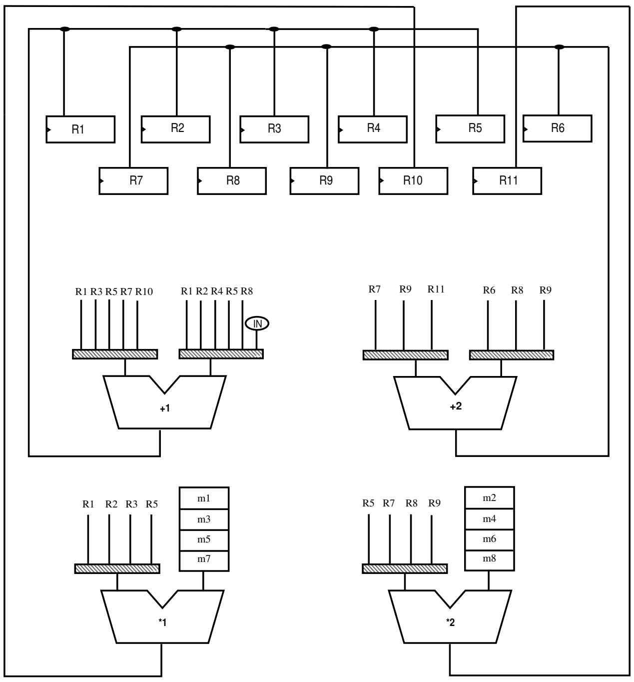

Figure 7. A fully constrained architecture.

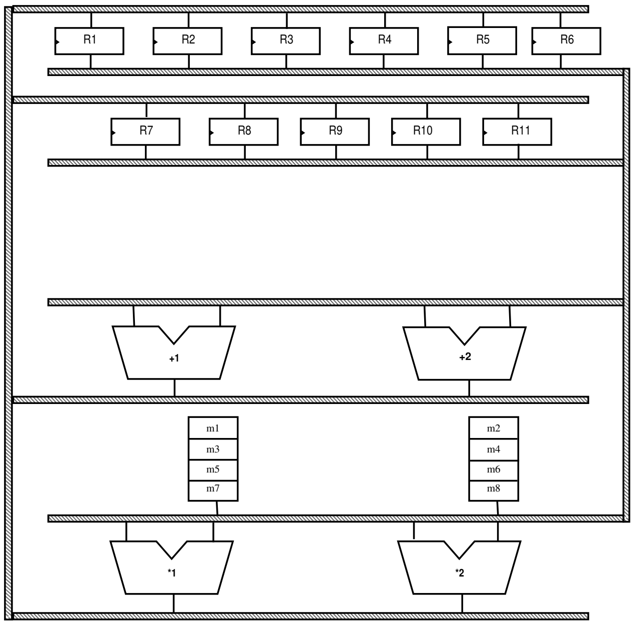

Figure 8. A partially constrained architecture.

[6] D. Gajski, N. Dutt, A. Wu, and S. Lin. *High Level Synthesis: Introduction to Chip and System Design.* Kluwer Academic Publishers, 1992.

[7] D. Gajski and R. Kuhn. New VLSI tools. *IEEE Computer*, 16(12):11–14, June 1983.

[8] J. Gyllenhaal. A machine description language for compilation. Technical report, Department of Electrical and Computer Engineering, University of Illinois, Urbana IL, September 1994.

[9] G. Hadjiyiannis, S. Hanono, and S. Devadas. ISDL: An instruction set description language for retargetability. In *Proceeding of the 34th Design Automation Conference*, June 1997.

[10] 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 the Design Automation and Test Conference in Europe*, March 1999.

[11] R. Leupers and P. Marwedel. Retargetable code generation based on structural processor descriptions. *Design Automation for Embedded Systems*, 3(1), 1998.

[12] P. Marwedel. The MIMOLA design system: Tools for the design of digital processors. In *Proceeding of the 21st Design Automation Conference*, pages 587–593, June 1984.

[13] N. Ramsey and J. Davidson. Specifying instructions' semantics using CSDL. Technical report, Department of Computer Science, University of Virginia, 1998.

[14] N. Ramsey and M. Fernandez. The New Jersey machine-code toolkit. In *Proceedings of the 1995 USENIX Technical Conference*, pages 289–302, January 1995.

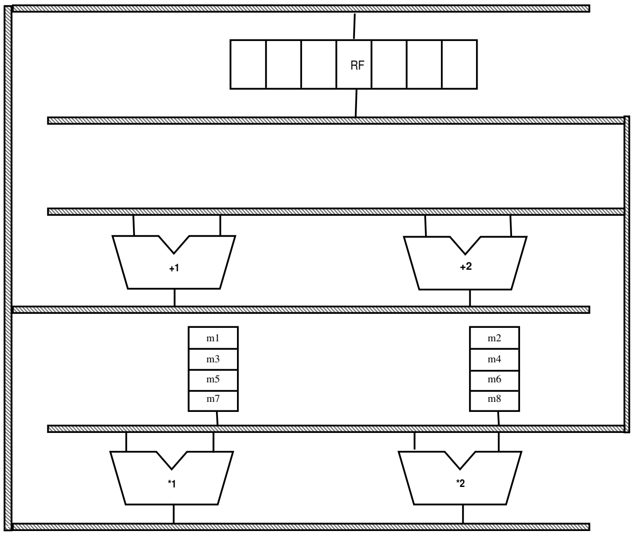

Figure 9. Another partially constrained architecture.