A Unified Formal Model of ISA and FSMD

Jianwen Zhu, Daniel D. Gajski
CECS, Information and Computer Science
University of California, Irvine, CA 92717-3425, USA

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.

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 [1], 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 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 information one need to build the actual hardware, for example, the detailed netlist; nor repeating every information one can find on the traditional architectural manual. Instead, the architecture model should serve as guidance to the corresponding synthesis tools.

In this paper, we demonstrate the first step toward our goal of developing formal architectural model for system-on-a-chip by a unified treatment of its most important components: the instruction set architecture (ISA) for processor and finite state machine with a datapath architecture (FSMD) [14] for ASIC. The rest of the paper is organized as follows. Section 2 reviews the related works and highlight our contributions. Section 3 discusses the behavioral models that are relevant to the subsequent discussion of architecture models. Section 4 discusses the instruction set architecture. Section 5 discusses FSMD architecture. Section 6 describes the algorithms which unite the FSMD and ISA architecture. Due to space reasons, detailed illustrations and algorithms are omitted. Interested readers are referred to [15].

2 Related Works and Contributions

The earliest forms of explicit architecture model descriptions are various code generator's generator (CGG) [2] [3] [4]. 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 Zephra project, where [5] is devoted to the description of binary encoding of instruction set, [6] focuses on the semantics of instruction sets, and [7] 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 and application specific instruction set processors (ASIPs), also receive intensive interest in the recent years. Representative works are MIMOLA [9], where a hardware description language is used to describe the structural model of the processor; as well as nML [12] and ISDL [11], which take a more traditional approach. Representing a more recent effort, the Expression [13] 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 [14]. However, the simple model given in [14] does not handle advanced features such as procedure call. How the ASIC interacts with its envioronment 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 are limited to allocation table, in the form of graphical user interface or compiler pragmas. Little work has been done on defining architectural model flexible enough to describe a partially constrained architecture.

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, under the retargetable compiler infrastructure of our experimental codesign platform, behavioral synthesizer appears just as yet another target in the backend.
- **formality:** Our model is the first to formally define the essential elements as well as their relationship in the architecture. Without a formal model, the architecture specifier tends to be overwhelmed by the language syntax and the amount of information one has to capture in a typical architecture, and a clean interface between the architectural model and the synthesis tool is difficult to define.

Despite its uniqueness, our work is in many ways 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 [9] and detailed in [10]. Expression [13] 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].

3 Behavioral Model

As we discussed before, it is important in the architectural model to associate architecture resource with the behavior piece that it can implement. We define our model of behavior piece in terms of trees.

Definition 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 2 *A **behavior pattern** $lfs \leftarrow rhs$ over the **terminal** set Σ_t and **nonterminal** set Σ_n is a member of*

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

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

Definition 2 is adopted by many tree-based CGGs such as BURG. Typically, the nonterminal set Σ_t is taken as the **value holders**, such as storages, in the architecture model.

4 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 3 gives our formal model of an ISA architecture.

Definition 3 *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 simply 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 maybe 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.

4.1 Stores

Much the same way as the addressing modes one can find in the architectural manual, The stores (Definition 4) model processor storage resources such as register files and memories. The information of interest is the set of cells, which is in turn defined in Definition 5, that it contains and the way they can be *accessed* and *allocated*.

Definition 4 *A store is a member of set*

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

*, 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 5 *A cell is a member of*

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

, where s is the store to which it belongs, 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, they are conceptually simpler to be considered as 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. The allocation state space of a memory store is usually modeled after the alignment status of the current available memory location.

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.

4.2 Instructions

The instruction (Definition 6) models processor computational resources. The information of concern is its semantics and binary encoding. Modeled as a behavioral pattern, the instruction semantics helps to identify the behavior piece that the instruction can implement. When a behavior piece is identified to be implemented by a particular instruction, the storage allocation has to 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 pair of fields (Definition 7) for the base and offset of the instruction destination respectively; and $srcs$, which are field pairs for the bases of offsets of instruction sources.

Definition 6 *An instruction is a member of set*

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

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

The field is characterized by its offset and width.

Definition 7 *A **field** is a member of*

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

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

4.3 Constraints

The temporal constraint 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 capability of bypassing. To make things even worse, 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 spatial parallelism between the instructions. Modern microprocessors, whether a superscalar or VLIW architecture, contains 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 c 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 schedule with a delay of any cycles, or, can be executed in parallel. Note that it is always the case that $d \geq 0 \wedge d < n$.

5 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 data path performs the computational tasks specified by the control signals by transforming data values in its storages.

Our formal model of FSMD is given in Definition 8, which specifies the control path implementation style, the set of units and buses in the data path, as well as their interconnection. The implementation style can be random logic based, PLA based, or microcode based.

Definition 8 *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 data path 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 9 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 4. The functions that the unit is able to perform are specified by its operations.

Definition 9 *A* ***unit*** *is a member of*

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

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

A control field (Definition 7) 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 operations that a unit can perform are characterized by the behavior patterns, and the corresponding control configurations. Note that the nonterminal 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 10 *An* ***operation*** *of unit u is a member of*

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

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

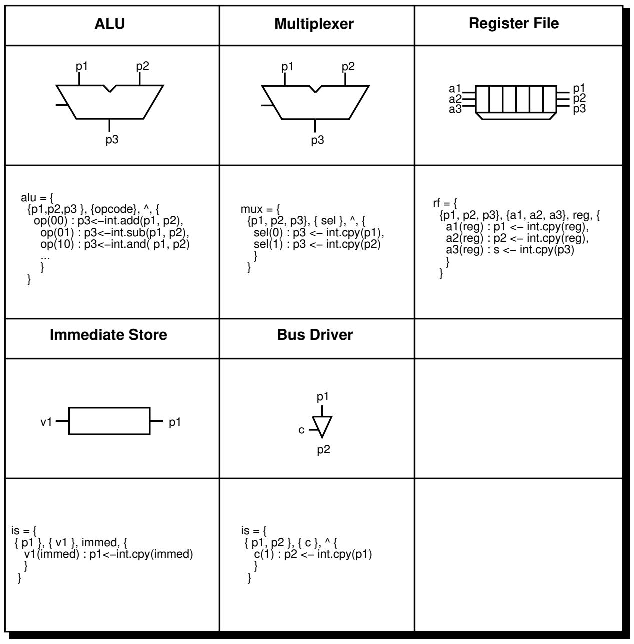

Figure 1. Units.

Example 1 *Figure 1 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 physical wires. They are used to indicate the possible data transfers between the units. A shared bus does imply one physical interconnection between the set of units, but the corresponding steering units such as bus drives and keepers can be automatically generated. On the other hand, an on-demand bus implies as many point-to-point connection as needed by the behavior.

Definition 11 *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. □

6 Bridging the Gap between FSMD and ISA

The FSMD model defined in Section 5 has an apparently different structure than the ISA model defined in Section 4. 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 optimzations, 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 simply the user interface of the synthesis tool.

This goal is achieved by deriving an ISA model $\langle m, n, S, I, IE, SC, TC, CA\rangle$ from the FSMD model $\langle c, U, B, PMF, CA\rangle$.

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, whose side effect is to determine the offset of each control field of the FSMD units [15].

The set of stores S can be simply computed by enumerating all the stores associated with the FSMD units [15].

The problem left is the derivation of the set of instructions as well as the spatial constraints. The problem 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 12 *A **partial instruction** in an FSMD architecture* $\langle c, U, B, PMP, COMM\rangle$ *is a member of*

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

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

The partial instruction set of each value holder can be derived 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 [15].

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 [15].

The instruction set can be derived by enumerating the partial instruction set associated with the stores of the FSMD. Since FSMD 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 [15].

7 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.

References

[1] D.D. Gajski and R.H.Kuhn. *New VLSI Tools*. IEEE Computer, vol. 16, no. 12, 1983, pp. 11-14.

[2] R.G.G. Cattell. *Code Generation and Machine Descriptions*. Technical Report. CSL-79-8, Xerox Palo Alto Research Center, Oct. 1979.

[3] A.V. Aho, M. Ganapathi, and S.W.K. Tjiang. *Code Generation Using Tree Pattern Matching and Dynamic Progragming*. ACM TOPLAS, Vol 11, No. 4, Oct. 1989, pp.491-516.

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

[5] N. Ramsey and M. Fernandez. *The New Jersey Machine-Code Toolkit*. Proceedings of the 1995 USENIX Technical Conference, New Orleans, LA, January 1995, pp 289-302.

[6] N. Ramsey and J.W. Davidson. *Specifying Instructions' Semantics Using CSDL*. Prelimimary Report, Department of Computer Science, University of Virginia.

[7] M.W. Bailey and J.W. Davidson. *A Formal Model and Specification Language for Procedure Calling Conventions*. Computer Science Report No. CS-94-39, University of Virginia.

[8] J. Gyllenhaal. *A Machine Description Language for Compilation*. MS thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana IL, Sept. 1994.

[9] P. Marwedel. *The MIMOLA Design System: Tools for the Design of Digital Processors*. Proceedings of the 21th Design automation Conference, pages 587-593, 1984.

[10] R. Leupers, P. Marwedel. *Retargetable Code Generation Based on Structural Processor Descriptions*. Design Automation for Embedded Systems, 3(1), 1998.

[11] G. Hadjiyiannis, S. Hanono, and S. Devadas. *ISDL: An Instruction Set Description Language for Retargetability*. In Proceedings of 1997, ACM/IEEE Design Automation Conference.

[12] A. Fauth,J.V. Praet, and M. Freericks. *Describing Instruction Sets using nML* (Extended Version). Technical Report. Technische Universität Berlin and IMEC, Berlin(Germany)/Leuven(Belgium), 1995.

[13] A. Halambi, P. Grun, V. Ganesh, A. Khare, N. Dutt, A. Nicolau. *Expression: A Language for Architecture Exploration through Compiler/Simulator Retargetability*. Proceedings of Design Automation and Test in Europe, Munich, Germany, 1999.

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

[15] J. Zhu, D.D. Gajski. *A Unified Formal Model of ISA and FSMD*. Technical Report ICS-98-44, University of California, Irvine.