Synthesizing DSP Architectures from Behavioral Specifications:  
A Formal Approach

Khaled M. Elleithy and Magdy A. Bayoumi

The Center For Advanced Computer Studies  
University of Southwestern Louisiana  
Lafayette, LA 70504, U.S.A.

Abstract
This paper introduces a formal behavioral synthesis framework for specifying, simulating and synthesis of Digital Signal Processing (DSP) algorithms. The given algorithm is represented using a new language termed Algorithm Specification Language (ASL). The components and connectivity of the synthesized architecture can be represented in three different forms: a schematic captures and Prolog. Prolog is used as a user interface language between the user subsystem and the synthesis subsystem. Algorithms of linear time complexity to transfer between different representations are introduced.

1. Introduction
Synthesizing correct architectures is a major problem. Ad-hoc methods were proved to rarely result in a totally correct design specially for huge architectures. Testing is a popular method that is used to prove the correctness of systems for specific sets of inputs and outputs. With the current advances in VLSI circuits, there is no testing procedure that is capable of accomplishing an exhaustive examination of complex circuits. Chips that may be discovered to be faulty in future is a major consequence of testing procedures’ limitations. Verification is another trend to prove correctness of an architecture. Verification techniques have some implementation problems in a practical environment due to the large time required even to verify the simplest architectures and most techniques are tailored for a specific class of architectures.

Early works in synthesis field were basically require the input to have a structural description. CMU-DA [1-3] require the input to be described using Instruction Set Processor Specification (ISPS) [4]. The problem with such approaches is the requirement to know the target architecture in advance in order to have the structural description.

Synthesising a DSP algorithm from a behavioral description using a programming like language have been reported in [5-9]. Flame[5] uses Pascal as a behavioral description language. A number of restrictions are imposed on the input description: the description should be parameterless and nonrecursive, certain data types are only allowed, and certain arithmetic operations are not allowed. HARP[6] uses Fortran as the behavioral description language. The designer is responsible for determining the facility’s word length and the composition of units. Restrictions are imposed on the input specification such as: only certain data types are allowed, subroutine and function calls are not allowed, and certain functions are omitted. Communicating Sequential Processes (CSP) is used as a description language by Philips [7,8]. Using a CSP as a description language means that the designer is responsible for specifying the operations sequencing and communication among different units. Also, most of these languages have been used as a user behavior description of an algorithm. Johnson[9] has presented an approach for transforming a system described in the form of linear recursion equations to digital designs.

The scope of this paper is a formal synthesis framework for DSP architectures. Those systems have special features to be considered such as:

1. DSP architectures are based on fixed functions.
2. Algorithms are data-independent, so control can be hard-wired. The most efficient hardware solutions directly reflect the data-flow inherent in the algorithm.
3. Processor throughput is more important than processor latency. Throughput is achieved by pipelining and functional parallelism.

Several systems have been developed for high level synthesis of VLSI architectures from DSP algorithms[10-16]. These approaches have concentrated on resolving and optimizing several architectural issues such as using a single fixed architecture with a parameterizable data path and control unit, selecting one from a set of these units, or trying to use kind of multiprocessor systems. Also, most of these systems are only applicable to special class of algorithms.

Our new approach addresses two issues: Completeness and Correctness. This paper presents a formal high level synthesis framework which is applicable to a wide and general class of algorithms. The synthesis system is composed of two subsystems: synthesis subsystem and user-interface subsystem as shown in Figure 1.

2. Synthesis Subsystem

2.1 Algorithm Representation
The given algorithm is specified in a new language, termed ASL, which is based on recursive functions. ASL is capable of specifying any algorithm using a limited number of constructs. Several researches have been reported for specifying the input algorithm. Imperative languages have been used such as: Fortran[6], Pascal[5], ISPS[1,2], and VHDL[17]. Applicative languages have
been used such as: Recursion equation based specification[9] and Temporal Logic[8,10].

The design specification language must be simple and semantically well characterized. It must permit writing specifications using a few predefined objects and concepts as possible. Specifications must not be prematurely committed to particular implementations. Current hardware specification languages are far from this ideal.

The idea of representing an algorithm using a functional representation was introduced by Kleene[20] in which any function from numbers(strings) to numbers(strings) can be constructed using very primitive functions and combining them in a specific manner. Our approach for specifying an algorithm depends on using μ-recursive functions as a framework.

The Language consists of Initial Functions and Operations. Initial functions are zero, projection, and successor functions. The zero function returns the value zero. The projection function is used to choose argument i. The successor function which increments its input by one. The three operations are Composition, Recursion, and Unbounded minimization. Although the constructs are very primitive, complex constructs can be used through a developed cell library. Units that have been designed through this design methodology can be added to the cell library. ASL has the following characteristics:

(1) Simple and semantically well characterized.
(2) It permits writing specifications using a few predefined objects and concepts.
(3) It is a suitable tool for formal synthesis.
(4) It supports a hierarchical design methodology.
(5) It supports formal verification of the initial specification.
(6) It is complete.

A complete description of the language syntax can be found in[21].

2.2 Design Representation

A transformation technique is developed to transform an algorithm represented in ASL to a specific realization language, termed RSL. The RSL version specifies the components and connectivity of the digital architecture that realizes the algorithm. Every construct in ASL has an isomorphic representation in RSL, which is the automated transformation technique developed to transform an algorithm represented in ASL to a specific realization language, termed RSL. The RSL version specifies the components and connectivity of the digital architecture that realizes the algorithm. Every construct in ASL has an isomorphic representation in RSL, which is the automated transformation technique proved to be correct. Everything in the system is built from certain primitives. The proofs of correctness are applied to these primitives. A hierarchical proof is used for ensuring correctness at different levels. The realization introduced here is described at the architectural level. The physical parameters of the circuit are not addressed at this level of specification. The timing is not expressed explicitly, but implicitly in terms of registers. The symbol ρ is used to represent a register. If it is required to initialize the register to a certain value β, then this is represented as ρβ.

(1) If unit is a certain component that is used in the system, and inp is any input for unit, then we can use this input explicitly in the following syntax:

\[ \text{unit}(\text{inp}) \]

Also an output(out) of unit can be used as follows:

\[ \text{unit}^\text{out} \]

(2) Identifiers starting with lower case letters are used also to represent units that will be further expressed at lower levels. Identifiers starting with lower case letters will be used to express Basic functions. Identifiers starting with lower case letters are used also to represent constants, while those starting with upper case letters are used to represent temporary variables used in computation.

(3) Register: If it is required to use a register α that is initialized to a value β, then this is expressed as ραβ. As an example the expression \( \text{register}() \) means that unit has the register connected to its input with a value 2 as the register initial value. \( \text{Init} \) statement is a RSL expression that can be used separately to express that register \( \text{alpha} \) is initialized to \( \text{value} \) β as follows:

\[ \text{Init}(\text{alpha}, \beta) \]

Another version of \( \text{Init} \) can be used to denote the initialization of more than one register at the same time, this is represented as follows:

\[ \text{Init}(\alpha_1, \beta_1 ; \alpha_2, \beta_2; \ldots ; \alpha_n, \beta_n) \]

The previous statement means that n registers are initialized in parallel such that register α1 is initialized with the value β1, register α2 is initialized with the value β2, etc... Notice that using registers is the only way to represent time in our circuits. It is assumed that all registers are synchronized using a global clock.

(4) Constant values are expressed using registers. If it is required to have a constant value \( c \) in the circuit, then this is represented as \( \rho_c \). If it is required to represent the zero function directly the following specification is used:

\[ \text{Result} = \text{zero} \]

2.3 Transformation Algorithm

A transformation algorithm is based on using a one-to-one mapping procedure. The algorithm takes an ASL representation and transform each of ASL constructs to an equivalent RSL representation and an equivalent architecture implementation. The algorithm has a \( \Theta(n) \) time complexity where \( n \) is the number of ASL constructs used to represent the algorithm. Table I shows the relationship between each ASL construct and equivalent RSL constructs. The complete algorithm is given in[21].

Example

If we consider that pro is a basic function, then the factorial function \( \text{pow}(n, m) \) is defined as follows:

\[ \text{pow}(n, m) = \Lambda((\text{pro}(n, m), \text{pow}(n, m-1))) \]

From the previous definition we see that we do not express the function \( \text{pow} \) in terms of \( \text{init} \) functions, but we use \( \text{pow} \) as a basic function, which may be expressed explicitly in a library of basic functions.

3. User Interface Subsystem

The user interface environment that is used for the synthesis process will be implemented as a logic programming environment. The logic programming environment supports specifying, simulating, and testing Digital Signal Processing (DSP) systems[22]. Backtracking and pattern matching of Prolog are employed for simulation and testing, respectively. Prolog provides homogeneity to the developed system as it supports hierarchical development and mixing of description at various hierarchical levels. Prolog has powerful facilities for simulation. Using the same code without any modifications we can perform different types of simulation: forward, backward or bidirectional. Simulation can be performed on different levels and it can be mixed. Circuits specified in Prolog can be specified at different levels and any degree of details without too much complexity.

Two transformation algorithms are used to link the synthesis subsystem and the user interface subsystem. The purpose of these two algorithms is to allow the user to use the system through the logic programming environment without the need to know the details of ASL and RSL. The two algorithms are as follows:

(1) The first algorithm is used to transform between Prolog and ASL. This transformation algorithm allows the user to specify his algorithm in terms of Prolog. Table II shows the transformation Algorithm.

(2) The second algorithm is used to transform between
Name | ASL | RSL | Steps
--- | --- | --- | ---
zero | $\xi() = 0$ | Result = zero | 2
Projection | $n_i([arg_1, arg_2, \ldots, arg_n]) = arg_i$ | $\text{Result} = \mu x([arg_1, \ldots, arg_n] \neq i)$ | 2
Successor | $\lambda(n) = n + 1$ | $\text{Result} = \text{suc}(n)$ | 2
Composition | $z = y(x_1, x_2, \ldots, x_m)$ | $\text{Result} = \text{Comp}(x_1, \ldots, x_m \# y)$ | 2
Recursion | $f(arg_1, \ldots, arg_n) = x(arg_1, \ldots, arg_n, 0)$ | $\text{Initp}(0, m \; ; 1, arg_1 \; ; 2, arg_2 \; ; \ldots \; ; n, arg_n)$ | 5
Unbounded Minimization | $\mu x(\alpha x_1 x_2 \ldots x_m)$ | $\text{Ready} = \text{eq}([\alpha, x_1 x_2 \ldots x_m])$ | 2
| $\mu x(m \; ; 1, arg_1 \; ; 2, arg_2 \; ; \ldots \; ; n, arg_n)$ | $\text{Result} = \text{suc}(\text{Result})$ | 3
| $f[arg_1, arg_2, \ldots, arg_n] = \text{min } m \{ \text{set } z(arg_1, arg_2, \ldots, arg_n, m) = 0 \}$ | $\text{Initp}(0, m \; ; 1, arg_1 \; ; 2, arg_2 \; ; \ldots \; ; n, arg_n)$ | 3
| $= 0$ if $m$ does not exist | $\text{Result} = \text{suc}(\text{Result})$ | 3

| | | | 3

Table I. Transformation Algorithm between ASL and RSL.

the RSL and Prolog. This algorithm allows us to have the output circuit specified in Prolog.

4. Conclusions

In this paper a framework for synthesising DSP systems from behavioral descriptions has been introduced. The proposed framework has the following advantages: It is suitable for large problems, the transformation algorithms are linear, it does not require to know the target architecture in advance, there are no restrictions imposed on the input description, the technique can be fully automated, the designer is not responsible for specifying the operations sequencing and communications among different units, and the approach is applicable to any general algorithm.

References

<table>
<thead>
<tr>
<th>Name</th>
<th>Prolog</th>
<th>Number Of Steps</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zero</td>
<td><code>zero(0).</code></td>
<td>1</td>
</tr>
<tr>
<td>Projection</td>
<td><code>project(0,L,[ ]).</code></td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>`project(1,[X</td>
<td>Y],X).`</td>
</tr>
<tr>
<td></td>
<td>`project([M],X</td>
<td>Y,Result)</td>
</tr>
<tr>
<td>Successor</td>
<td>`suc(N,M)</td>
<td>M is N+1. `</td>
</tr>
<tr>
<td>Composition</td>
<td><code>z1 code</code></td>
<td>3</td>
</tr>
<tr>
<td></td>
<td><code>z code</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>y code</code></td>
<td></td>
</tr>
<tr>
<td>Recursion</td>
<td><code>code for z</code></td>
<td>4</td>
</tr>
<tr>
<td></td>
<td><code>code for y</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td>`z(Arg1, · · · ,Argn,0,Result)</td>
<td>z(Arg1, · · · ,Argn,Result).`</td>
</tr>
<tr>
<td></td>
<td>`z(Arg1, · · · ,Argn,M,Result)</td>
<td>M is M-1, z(Arg1, · · · ,Argn,1,M,Temp),`</td>
</tr>
<tr>
<td></td>
<td><code>y(Arg1, · · · ,Argn,M,Temp)</code>.`</td>
<td></td>
</tr>
<tr>
<td>Unbounded Minimization</td>
<td><code>code for g</code></td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>`f([Arg1,Arg2, · · · ,Argn,M])</td>
<td>g([Arg1, · · · ,Argn,M,0]).`</td>
</tr>
<tr>
<td></td>
<td>`f([Arg1,Arg2, · · · ,Argn,M])</td>
<td>M is M+1, f([Arg1,Arg2, · · · ,Argn,M]).`</td>
</tr>
</tbody>
</table>

Table II. Transformation Algorithm between ASL and Prolog.