ABSTRACT

In this paper, we introduce a novel approach for high level synthesis for DSP algorithms. Two features are provided by the approach: completeness and correctness. A given algorithm will be represented in a new developed language termed Algorithm Specification Language (ASL). ASL has the ability to describe any general algorithm. An automatic procedure is used to transform an ASL representation into a specific realization specification using a correctness preserving set of transformations. The realization format is based on representing the digital architectures by another developed language called Realization Specification Language (RSL). Logic Programming is used as a user interface for the synthesis procedure.

1. Introduction

Although Digital Signal Processing (DSP) has emerged as a major technological application area, DSP design tools has not progressed enough to meet the challenges of the new applications requirements. The difficulty of developing such tools is that the chip design involves several levels of abstraction; starting from system level specification to the geometrical layout. The lowest level of this spectrum is understood sufficiently well to be fairly automated, while, developing design methods for high level synthesis is still in its infancy stage.

Synthesising a VLSI architecture for a DSP algorithm starting from a behavioral description have been reported in a number of researches. Works reported in [1-4] depend on using only a single fixed architecture. Works reported in [5-10] do not produce efficient architectures as a result of huge underutilised communications. Haroun and Elmasry [11] introduced an approach to overcome the previous problems. Generalised Signal flow graphs (GSFG) are used to describe the input algorithm. A number of restrictions should be satisfied by the used GSFG. High level synthesis approaches for DSP algorithms don't only suffer from the previous drawbacks, but also they don't address two focal points: Completeness and Correctness. Reported approaches aren't capable of transforming a general DSP algorithm to a digital Architecture. There are approaches to transform the class of linear recursive algorithms to digital circuit, approaches to transform regular iterative algorithms to systolic arrays, and approaches for synthesising a specific algorithm. For the correctness aspect a form approach is required for the synthesis process to assert the correctness of the high level synthesis techniques.

In this paper, we introduce a novel approach for high level synthesis that provides two main features: completeness and correctness. Completeness means the ability to use the approach for any general algorithm. Correctness is achieved by using a set of transformations that are proved to be correct. A formal framework for the synthesis procedure will be developed which can be easily automated.

2. System Overview

The proposed framework for synthesising any DSP algorithm is shown in Figure 1. The system is composed from two subsystems: synthesis subsystem and user-interface subsystem.

2.1 Synthesis Subsystem

(1) The given algorithm is specified in a new language, termed ASL, which is based on μ-recursive functions. ASL is capable of specifying any algorithm using a limited number of constructs. 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.

(2) 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 basis of the automated transformation. That transformation algorithm is 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.

(3) A library of the basic functions is defined starting from the initial functions to be used in the definition of larger functions. This approach is useful for building a cell library to support the automated synthesis system. All the basic functions that have been designed using the proposed approach can be used for specifying other functions. This technique supports a hierarchical design methodology in the sense that the specification can be stopped at any level as long as the lower levels were previously defined. When Basic functions are used to represent another functions, each Basic function is written in a box.

2.2 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. 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. It can be employed for many DSP algorithms and applications development.

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:
3. Algorithm Specification Language (ASL)

Several researches have been reported for specifying the input algorithm. Imperative languages have been used such as: Fortran[10], Pascal[13], ISPS[14,15], and VHDL[16]. Applicative languages have been used such as: Recursion equation based specification[17] and Temporal Logic[18,19].

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 primitive functions as a framework.

The Language consists of Initial Functions and Operations. Initial functions are very primitive functions. Three initial functions are used:

1. The zero function which returns the value zero.
2. The projection function has a K argument and we use it to choose argument i.
3. The successor function which increments its input by one.

Three operations are used to construct larger functions from smaller functions:

1. Composition: if we have an m functions (x_1, x_2, ..., x_m each is an n-argument function), and if y is an m-argument function, then we can define a new function z as follows:
   \[ z = y(x_1, x_2, ..., x_m) \]

2. Recursion: if z is an n-argument function, y is an n+1-argument function and y is n+2-argument function, then z is defined as follows:
   \[ r(y_1, ..., y_n, m) = \begin{cases} \text{base case} & \text{if } m \leq n \\ r(y_1, ..., y_n, m+1) & \text{if } m > n \end{cases} \]

3. Unbounded minimisation: Let z be an n-argument function, then the unbounded minimization of z is an n-function which defined as follows:
   \[ f(y_1, ..., y_n, m) = \min_m \{ z(y_1, ..., y_n, m) \} \]

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 permits formal verification of the initial specification.
6. It is complete.

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

4. Realization Specification Language (RSL)

4.1 Realization Specification Language

The realization introduced here is described at the architectural level. The specification describes two elements of a circuit: the components and the connectivity. 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 r is used to represent a register. If it is required to initialize the register to a certain value \( \delta \), then this is represented as \( r = \delta \).

1. If \( \text{unit} \) is a certain component that is used in the system, and \( \text{in} \) is any input for \( \text{unit} \), then we can use this input explicitly using the following syntax:
   \[ \text{unit}(\text{in}) \]
   Also an output(out) of \( \text{unit} \) can be used as follows:
   \[ \text{out}(\text{out}) \]

2. Identifiers starting with upper case letters are used 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 \( a \) that is initialized to a value \( \delta \), then this is expressed as \( r = \delta \).

An example the expression \( \text{norm}(1) \) means that see unit has the register 1 connected to its input with a value 2 as the register initial value. Initial statement is a RSL expression that can be used separately to express that register \( \text{reg} \) is initialized to value \( \delta \) as follows:

\[ \text{Init}(\text{reg}) \]

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:

\[ \text{Init}(a_1, a_2, \ldots, a_n) \]

The previous statement means that \( a_1 \) registers are initialized in parallel such that register \( a_1 \) is initialized with the value \( \delta_1 \), register \( a_2 \) is initialized with the value \( \delta_2 \), 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.
4.2 Transformation Algorithm

The transformation algorithm is based on using a one-to-one mapping procedure. The algorithm takes an ASL representation and transforms each ASL's construct to an equivalent RSL representation and an equivalent architecture implementation. The complete algorithm is given in [21].

5. Implementation of an Inner-product Cell

An example of an Inner-product cell is introduced in this section. The cell has three inputs \( x, y, \) and \( c, \) and an output \( d. \) The function of the cell is specified by:

\[
d = x \cdot y + c
\]

The implementation of this equation is done in two levels. At the first level we need to multiply \( A \) and \( B. \) The multiplication is done in a recursive way using repetitive addition. This can be simply described by the following high level subroutine:

```
function multiplication(n, ni)
begin
  if (n = 0) then result = ni
  else
    temp = multiplication((n-1), ni)
    result = addition(temp, result)
  end
end
```

The second level of the process is an implementation of the addition operation. The addition is done also recursively and can be described as follows:

```
function addition(n, m)
begin
  if (n = 1) then result = m
  else
    temp = addition((n-1), m)
    result = increment(temp)
  end
end
```

The operation of the cell can be described as follows:

```
inner_product(a, b, c)
begin
  if (c = 0) then result = multiplication(a, b)
  else
    temp = inner_product(a, b, c-1)
    result = increment(temp)
  end
end
```

Other representations may be found for the inner_product cell. We are not trying to find all representations at this level, but we want to mention that different representations can affect the final design. Figure 2 shows the implementation of the inner_product cell, and Appendix I shows the complete ASL and RSL description of this example.

6. Logic Programming Interface

In this section we will show how the algorithm specified in ASL can be represented in the logic programming environment. The following steps represent the transformation algorithm:

1. **Zero function**: is represented by the following fact:

2. **Projection function**: is represented by the following code:

3. **Successor function**: is implemented as follows:

4. **Composition**: is implemented as follows:

5. **Primitive recursion**: is expressed in the following manner:

6. **Unbounded minimization**: is represented in the following way:

Conclusions

In this paper a procedure for transforming general DSP algorithm to a digital architecture is introduced. The proposed framework has the following advantages:

1. It is suitable for large problems since it does not require solving any NP-C problems.
2. The transformation algorithm is linear.
3. It does not require to know the target architecture in advance.
4. There are no restrictions imposed on the input description.
Appendix I

ASL and RSL for the Inner-product Cell Example

**ASL Representation**

Multiplication unit code:

\[ \text{pro}(x_0) = \xi(1) \]
\[ \text{add}((y_0, y_1, y_2, y_3)) = \text{add}(\xi(y_0, y_1, y_2, y_3), \xi(y_0, y_1, y_2, y_3)) \]
\[ \text{pro}(x, m + 1) = \text{add}(x, m, \text{pro}(x, m)) \]

Addition unit code:

\[ \text{add}(n_0) = \eta(1) \]
\[ Q(y_0, y_1, y_2, y_3) = \text{add}(\xi(y_0, y_1, y_2, y_3)) \]
\[ \text{add}(n, m + 1) = Q(u, m, \text{add}(n, m)) \]

Inner-product cell code:

\[ \text{inner} \_\text{product}(x, y, z) = \text{pro}(\xi(x, y, z, z)) \]
\[ \text{inner} \_\text{product}(x, y, z, z) = \text{add} \_\text{one}(x, z, \text{inner} \_\text{product}(x, y, z)) \]

**RSL Representation**

The multiplication unit code:

\[ \text{Result} = \text{zero} \] (1)
\[ \text{Result}_1 = \text{mul}(x, y_1, y_2, y_3) \] (2)
\[ \text{Result}_2 = \text{mul}(x, y_1, y_2, y_3) \] (3)
\[ \text{Result}_3 = \text{add} \_\text{one}(\text{Result}_1, \text{Result}_2) \] (4)
\[ \text{Init}_1(0, m + 1) \] (5)
\[ \text{Init}_2(1, 0) \] (6)
\[ I = \eta(1) \] (7)
\[ \text{Ready} = \text{eq}(I, m) \] (8)
\[ \text{Result} = \text{pro}(x, \text{Init}_1, \text{Init}_2) \] (9)

The addition unit code:

\[ \text{Result}_1 = \text{mul}(x, y_1, y_2, y_3) \] (10)
\[ \text{Result}_2 = \text{mul}(x, y_1, y_2, y_3) \] (11)
\[ \text{Result}_3 = Q(\text{Result}_1, \text{Init}_2) \] (12)
\[ \text{Init}_1(0, m + 1) \] (13)
\[ \text{Init}_2(1, 0) \] (14)
\[ I = \eta(1) \] (15)
\[ \text{Ready} = \text{eq}(I, m) \] (16)
\[ \text{Result} = \text{add} \_\text{one}(x, \text{Init}_1, \text{Init}_2) \] (17)

The inner-product cell code:

\[ \text{Result}_1 = \text{mul}(x, y_1, y_2, y_3) \] (18)
\[ \text{Result}_2 = \text{mul}(x, y_1, y_2, y_3) \] (19)
\[ \text{Result}_3 = \text{temp}(\text{Result}_1, \text{Result}_2) \] (20)
\[ \text{Result}_4 = \text{mul}(x, y_1, y_2, y_3) \] (21)
\[ \text{Result}_5 = \text{add} \_\text{one}(\text{Result}_4, \text{Result}_3) \] (22)
\[ \text{Init}_1(0, m + 1) \] (23)
\[ \text{Init}_2(1, 0) \] (24)

References