AN ASIC ENGINE COMPILER FOR DIGITAL SIGNAL PROCESSING

Julian G. Payne, Stephen M. Kruse, Stewart G. Smith*, Ralph W. Morgan**

VLSI Technology, Les Taissonières, Rte des Dolines
Sophia Antipolis, 06560 Valbonne, France
* now at VLSI Vision Ltd, Edinburgh, Scotland
** now at Mentor Graphics, Oregon, USA

ABSTRACT

Typical DSP applications, such as image processing and high-bandwidth communications, often demand dedicated, high-throughput machines. An ASIC DSP Engine Compiler has been developed specifically to address these types of problems. It is assumed that the problem can be partitioned into a series of passes through a fixed-function integer arithmetic processing core. Designs are synthesised by the compiler from a dataflow specification plus parameters to specify the desired computational throughput and accuracy. The user can override synthesised attributes and use interactive size and power estimators to make trade-offs between numerical accuracy and size or power consumption. To illustrate this approach we show the design flow and design iteration for an example of a Fast Fourier Transform.

INTRODUCTION

There is a category of DSP applications, such as image processing and high-bandwidth communications, where the problem can be decomposed into a series of passes through some central integer arithmetic processing core [1], [2]. Such a decomposition coupled with digit-serial hardware techniques [3], [4] makes an attractive solution for an ASIC implementation.

Work has now been underway for some time on a DSP Engine Compiler that uses these techniques combined with an integrated ASIC toolset to allow a simple but powerful route to silicon. The aim of the compiler is to provide a system that takes a high-level specification and to provide rapid interactive design exploration. In order to achieve this goal the tool contains knowledge about the implementation techniques used within the system. This allows the synthesis to provide solutions that meet specifications containing function, throughput and accuracy.

In general, synthesis techniques can often be broken down into three stages, namely selection of architectural template, scheduling and allocation. One of the problems in high-level synthesis is the fact that these three stages are often inter-related and the effects of a particular decision in one stage can often not be predicted in advance. An advantage of the digit-serial approach is that these three stages can be treated as separate stages where the effects of each can be predicted accurately. This makes this type of approach interesting for a high level synthesis because the compiler is able to encapsulate this information and hence allow the user to rapidly explore the possible design space.

ARCHITECTURE AND SYNTHESIS

The goal of the compiler is to take a high-level description of a DSP application and synthesise a circuit taking into account not only the required function, but the throughput and accuracy also. In order to do this the compiler has, built in to it, architectural knowledge as well as knowledge required to synthesise the design whilst meeting the users requirements. In this section we will discuss both the architectural ideas contained within the compiler as well as highlighting the mechanisms used in the synthesis process.

The architectural knowledge can be thought of as containing two levels; the architectural template of the resulting system and the architectural knowledge used to construct the circuit.

![Figure 1. Architectural template.](image)
The other part of the architectural knowledge lies in
the digit-serial architectures that the compiler uses to
generate the required function for the final circuit. As
will be seen later, the compiler provides a set of basic
functions (add, subtract, multiply, rotate etc) that are
used to specify the required function as a dataflow
network. It is at this point that more traditional
synthesis techniques converge from the technique used
by the compiler. The process of synthesis within any
compiler is to take a functional description (of some form or other) and map it (hopefully efficiently) into
a circuit. More conventional systems achieve this goal by
having arithmetic resources (such as add, multiply etc)
available and then the job of synthesis is to optimize
the use of the resources by scheduling their use in an
efficient manner. This approach is used because the
basic arithmetic resources that are available are
normally of a fixed throughput and size, such that if
the required function demands one multiply at one
tenth of the maximum frequency of the multiplier then
the multiplier will be idle most of the time (hence
wasting resources). The architectures used in the DSP
Engine Compiler differ in that different digit-serial
architectures result in different size and throughput.
Hence at the global level the job of synthesis within
the compiler is quite different because the arithmetic
resources that are available to it can be optimized for
the required throughput (whilst ensuring the required
function) such that the resulting hardware is 100%
utilized [6]. The picture however is not quite as rosy as
may appear due to the fact that extra pipelining will be
added wherever needed for synchronization of data
streams.

Hence the job of synthesis within the compiler is to
decide on an appropriate global serial decomposition for
the serial architectures such that the required throughput is met whilst minimizing a user-supplied
cost function. The default cost function is the
minimum size but this could be the minimum power
consumption for example.

The final part of the users specification is to specify the
required arithmetic accuracy to be used within the
engine. The compiler has growth models which are
used to avoid overflow within the growth model selected by the user. During synthesis the compiler
computes the maximum numerical resources required at
all points within the circuit. With this information the
compiler is able to minimize the hardware required at
all points in the circuit based on the required numerical
resources. all points in the circuit.

SPECIFYING A DSP ENGINE

A user of the compiler is required to be able to specify
the design as a central fixed-function task, which
operates on blocks of data on the inputs and produces
blocks of data on the outputs. This central task
identifies the required function of the design; however
the user is also required to specify the throughput of
these blocks of data as well as the numerical precision
within the core task, namely the maximum number of
bits used to store a data word. It should be noted that
serial decomposition of a data word results in a triplet
(we refer to it as a bds triplet, bits, digits, subwords) where the third parameter of this triplet is
the number of subwords that make up the data word.
The fact that the serial decomposition includes
subwords means that the compiler is free to use as
many subwords as are required (based on the growth
information) at any point within the circuit to
minimize the amount of hardware used.

To explain this better consider the following example.
The compiler has chosen a bds triplet of 3,4,2.

![Figure 2. Word structure for a bds triplet 3,4,2.](image)

The grey squares in Figure 2 represent bits within the
word that contain useful information, that is to say,
bits that are not considered to be sign extensions. If
the compiler finds that a subword contains sign
extensions only at any point within the circuit then the
compiler throws the subword away as it serves no
useful purpose.

The method of specification will be illustrated with an
easy of a 20 MHz 256-point complex FFT. Figure 3
shows the dataflow specification for the FFT butterfly
which is going to be used as the core processor.

![Figure 3. Radix-4 FFT butterfly core specification.](image)

In the case of our example the user has specified an
external data rate of 20MHz, a transform size of 256 at
radix-4 leads to an internal data rate of 20MHz
between buffer memory and the engine. Hence the task
rate is one-quarter of this, i.e. 5MHz. For the purposes
of this study we assume that the input and output data
are 20 bits wide and we use a minimum internal
precision of 24 bits within the engine. The users initial
specification of the FFT is now complete.
LOCAL OVERRIDES

The user, having scheduled the design, and hence having chosen the global attributes, is now able to look at the local attributes attached to the dataflow network during the scheduling phase and override them if appropriate. The compiler tracks headroom and noisefloor through the system and uses these parameters to control the synthesis as well as to optimize the resulting hardware. The user is able to make use of this information to make trade-offs between size and accuracy.

![Figure 4. Overriding of local attributes.](image)

Figure 4 shows the noisefloor at a rotor with the default synthesized attributes; from this figure it can be seen that there is a change of 3 dB across the rotor. The user is able to increase the number of bits used to represent the rotor coefficients from the default of 20 bits to 21 bits and look at the resulting noisefloor attributes. In this case the change in noisefloor across the rotor is reduced to 0.03 dB. The user can then look at the size estimate for the rotor and see that the size of the rotor has increased from 4201 μm² to 4550 μm², as shown in Figure 5.

![Figure 5. Local size estimation.](image)

Thus the user is able to override the local synthesized attributes in order to trade-off numerical performance against size of the processor or power consumption of the processor.

CIRCUIT GENERATION

Having completed the scheduling and synthesis stages the only remaining step is the circuit generation. Circuit generation can be broken into two steps. The first step takes each icon in the core specification, together with the appropriate global and local parameters, to produce an optimised functionally correct circuit. The hardware is minimized using data-recoding techniques as well as taking into account the numerical headroom present on the inputs of each operator. The second step takes the functionally correct netlist and includes the necessary internal buffering and control logic, so that all that is left for the user is to provide is the appropriately buffered serial clock.

In addition to generating the circuitry to perform the core process, the compiler also generates register banks that take the parallel data and transform it into the appropriate serial format based on the value of the design space bds triplet. The register bank consists of parallel and serial storage buffer areas. The conceptual organization for one half of a (1,3,4) structure (where there are two parallel words per tick) is shown in Figure 6. Alternate data words are loaded into the parallel buffer on the first tick of the parallel clock. Both words of the input data block are copied into the leftmost column of the serial buffer on each tick of the task clock, which always coincides with the last pulse of the parallel clock. This is known as corner-turning the input data from the parallel into the serial domain.

![Figure 8. Corner-turning within the Registers](image)

Data on the diagonal of the serial buffer is shifted into the front end of the arithmetic processor pipeline at the serial clock rate. As the results emerge from the back end of the processor they "chase" the input data out of the diagonal storage locations. With subsequent ticks of the task clock data on the lower diagonal of the serial buffer is shifted horizontally until eventually it enters the processor via the diagonal storage elements. Similarly, results from the processor shift across the upper diagonal at the task rate until they reach the rightmost column, at which point they "turn the corner" back into the parallel buffer.
It should be emphasised that, as well as generating the

**SYSTEM SIMULATION**

The compiler allows very rapid specification of the core
task for an application and provides the necessary
features to allow the user to verify that his engine is
functionally and electrically correct.

However the philosophy of the compiler is that a
typical DSP application can be broken down to a
central core task which is executed many times within
the DSP transform. Hence it is important that the user
can verify the system over such a transform. In order
to carry out such verification the user needs to model
his system within an appropriate environment (such as
a VHDL simulator). Figure 7 shows pseudo-code for
the operational template for the FFT machine (much
simplified) using the butterfly as core processor.

```plaintext
# swinging buffer memories
for 13 = 0 upto 1 { # row
  for 12 = 0 upto columns-1 { # columns
    for 11 = 0 upto rows-1 { # engine terminals
      for 10 = 0 upto blocksize-1 { #
        form_addresses
      }
    }
  }
}
```

**Figure 7. Operational Template**

It should be emphasised that the DSP Engine Compiler
does not yet synthesise the hardware in the operational
template. The operational template is merely a
mechanism to allow the user to verify the design not
just over a single computational pass, but over the
entire sequence of tasks which constitute a DSP
transform. To make this possible the DSP Engine
Compiler provides the appropriate model of the task
that is used within the central loop of the operational
template and the user has to create the models for the
rest of the system.

**CONCLUSIONS**

A DSP Engine Compiler has been described which
allows a high-level description of the required processor
as well as the ability to separate the functional
requirement from the speed and accuracy of the
processor. The synthesis mechanism has been outlined,
illustrating the advantage for synthesis that is gained
by the ability to build functional units that are
parameterized by throughput and precision. The
approach has been illustrated with an FFT example
and it has been shown that the design flow is
straightforward and fast whilst leaving the user the
flexibility to override the synthesis during all stages of
the design.

**REFERENCES**

1. S. G. Smith and J. M. Rischard, "20 MHz 16x16
Discrete Cosine Transform IC: CAD and Architectural
Methodology", pp. 359 - 378 in VLSI'89, ed. G.
Mesgrave and U. Lauther, Elsevier Science Publishers
(1990)
2. R. Jain et al., "Custom Design of a VLSI PCM-FDM
Transmultiplexer from System Specifications to Circuit
Layout Using a Computer-Aided Design System", J.
IEEE SC-21 pp. 73 - 85 (February 1986)
3. S. G. Smith and P. B. Denyer, Serial-Data
4. R. I. Hartley and P. F. Corbett, "Digit-Serial
Processing Techniques", Trans. IEEE CAS-37 pp. 707
- 719 (June 1990)
5. S. G. Smith, R. W. Morgan, and J. G. Payne,
"Generic ASIC Architecture for Digital Signal
Processing", Proc. IEEE ICCD'89 pp. 82 - 85
(Cambridge MA, October 1989)