TREIZIÈME COLLOQUE GRETSI - JUAN-LES-PINS DU 16 AU 20 SEPTEMBRE 1991

Design Automation for Digital Signal Processing Applications

Julian G. Payne

Compass Design Automation, Sophia Antipolis, France

RÉSUMÉ

Cet article fait un bilan sur les derniers résultats concernant un projet de développement d'un compilateur de traitement numérique du signal, orienté ASIC. Le compilateur utilise des architectures de traitement en série comme solution pour certains types de problèmes. La spécification est donnée en termes de fonctionnement, flot et précision. Des attributs locaux permettent le contrôle numerique des grandeurs telles que le niveau de bruit et le taux de variation.

ABSTRACT

This paper reports the latest progress on a project to develop a DSP orientated ASIC compiler. The compiler uses digit-serial architectures to provide a solution for a certain class of applications. The specification is given in terms of the function, throughput and accuracy. Local attributes allow for inspection of numerical properties including noisefloor and growth.

1. Introduction

Digital Signal Processing as a market segment has been, and still is, rapidly expanding. However this rapid growth has not been matched by a similar growth in applications for ASIC design. This paper covers an ASIC compiler which is targetted at DSP applications that require intensive computational power without the need for reprogrammability. These applications can be characterized by the following assumptions:

- Tasks are computation intensive
- Tasks are fixed-function
- Computation is forward-flow additive (X,+)
- Control is data-independent
- Algorithms should exhibit functional parallelism
- Throughput is more important than latency

An example of such an application is a Discrete Cosine Transform [1] or a complex FFT, the complex FFT will be used as an example later on in the paper. The approach outlined below has been developed to provide a competitive solution for this class of applications.

2. Architectural Model

The particular approach that is being outlined in this paper uses digit-serial techniques that allow almost arbitrary mixtures of serialism, parallelism and pipelining. This enables us to achieve near-optimal circuit solutions for the class of applications detailed above. Some of the details of the digit-serial architectures used are explained in the later sections, however it is important to understand that the central processing core of the circuit uses serial techniques, whereas the external interface to the circuit is in the form of parallel busses (mainly because the digit-serial decomposition of the parallel word is not normally what the user wants to interface to). Hence the physical template for the resulting circuit contains three main parts; the central processing core, the register banks that convert between the parallel busses and the serial connections to the processor, and the control circuitry which is automatically added by the compiler during the synthesis phase. Thus the overall scheme within the compiler follows the physical template illustrated in Figure 1. The compiled machine consists of a hardwired, pipelined processor cell, and a set of register banks for communication. Input data arrive in sequence on a bit-parallel bus, and load up the register bank.

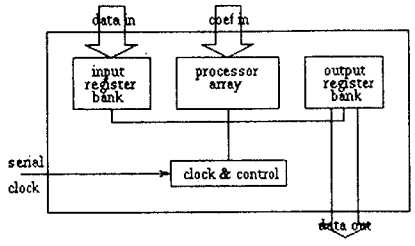

Figure 1. Physical template.

When this process is complete and the register bank is full, the data block is transferred into the *serial domain* and transmitted in pipelined fashion (according to the *word-structure)* through the processor. The register bank immediately starts filling up with operands for the next computation. Results are captured in the output register bank, are transferred back into the *parallel domain,* and depart the register bank in sequential fashion. For reasons of storage efficiency, the input register bank and the output register bank of Figure 1 are usually merged.

3. Digit-Serial Architectures

The compiler makes extensive use of digit-serial architectures [2],[3] to generate the final circuit. The power of this approach results from the fact that the compiler is able to generate arithmetic functions (namely add, subtract, multiply etc) that are parameterized both by speed as well as accuracy. The compiler chooses a global decomposition from the parallel domain to the serial domain, by using the information it has built-in, to determine factors such as the maximum clock-rate obtainable for a given serial decomposition. Each unique serial decomposition, and there are many of them due to the fact that the design space has three parameters (`bits`, `digits` and `subwords`), results in a different circuit that performs the same function but has different characteristics with respect to speed, power consumption and area. The effect of this decomposition is shown on a simple example, namely an adder for a six bit word, showing the resulting hardware for three possible serial decompositions. These decompositions illustrate some of the possibilities in terms of the amount of serialism/parallelism and pipelining, see Figure 2 and Figure 3.

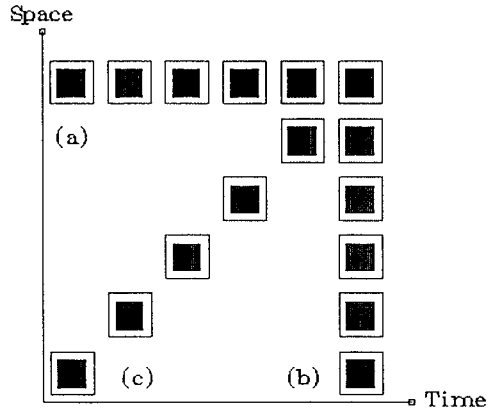

Figure 2. Three serial decompostitions for a six bit word

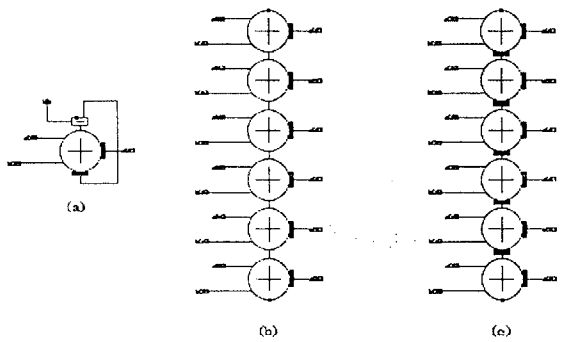

Figure 3. Adder hardware for the three serial decompositions

An advantage with digit-serial architectures comes from the ability to be able to implement a multiply in a very small area. The hardware techniques used in this system further enhance this capability for the case where a mutliplier coefficient is known in advance, we call these *fixed multipliers*. Fixed multipliers have two major gains in terms of silicon area, firstly because the coefficient is known we do not need a bus to drop the coefficient into the multiplier, secondly we can recode the coefficient (using canonic signed-digit recoding techniques) to reduce the hardware in the multiplier by at least 50%. An example of this is shown in Figure 4 and Figure 5. The digit-serial architectures have been described in more detail elsewhere [4].

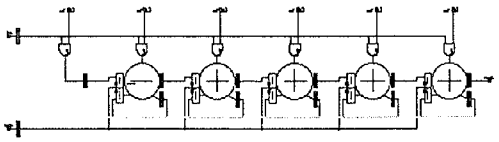

Figure 4. Bit-serial programmable multiplier

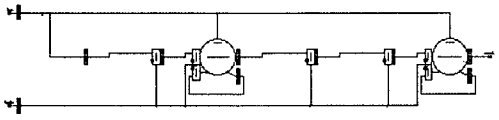

Figure 5. Bit-serial fixed multiplier

4. Design Specification

Design specification requires the identification, in terms of *function*, *throughput* and *accuracy*, of one or more fixed core processing tasks in each application. The compiler places equal importance on these three aspects of the specification. The *function* is specified schematically by a dataflow network, the *throughput* and *accuracy* are global parameters provided by the user. The method of specification will be illustrated with an example of a 20 MHz 256-point complex FFT. Figure 6 shows the dataflow specification for a FFT butterfly, which is going to be used as the core processor.

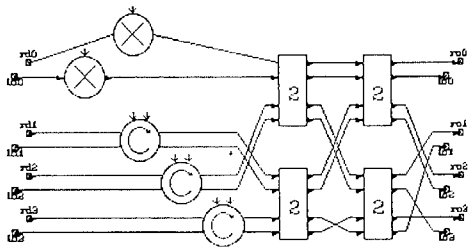

Figure 6. Radix-4 FFT butterfly core specification.

In the case of our example the user has specified an external data rate of 20MHz and 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.

5. Synthesis

The mechanism of synthesis involves choosing a *design space triplet* that represents a serial decomposition. A triplet, e.g. `(2,16,1)` has three parameters that we refer to as `bits`, `digits` and `subwords`. The parameter `bits` is one measure of the amount of parallelism within the processor, but more importantly it is a measure of the maximum logic depth within the processor and hence controls the maximum allowable serial clock speed. For a given technology and library it is possible to characterize the relationship between `bits` and serial clock speed such that the compiler is able to predict accurately the maximum achievable clock speed for any point within the design space. The parameter `digits` controls the amount of serialism within the processor, but also affects the throughput of the processor because it takes `digits` serial clock cycles to process one task. The last parameter `subwords` affects the amount of pipelining within the processor but is independent of throughput. The job of synthesis in the compiler is to choose the appropriate values for the triplet depending on the users specification in order to minimize a cost-function supplied by the user. In order to simplify this process the compiler provides a graphical mechanism to show the possible choices for the design space.

6. Local Overrides

The user having scheduled the design, and hence having chosen the global attributes, is 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.

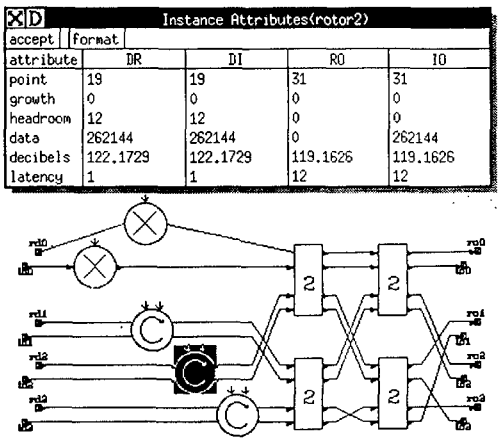

Figure 7. Overriding of local attributes.

Figure 7 shows the noisefloor at a rotor with the default synthesised attributes. The user is able to increase the number of bits used to represent the rotor coefficients and look at the resulting noisefloor attributes. The user can then look at the size estimate for the rotor in order to make rapid tradeoffs between size and accuracy, this is shown in Figure 8.

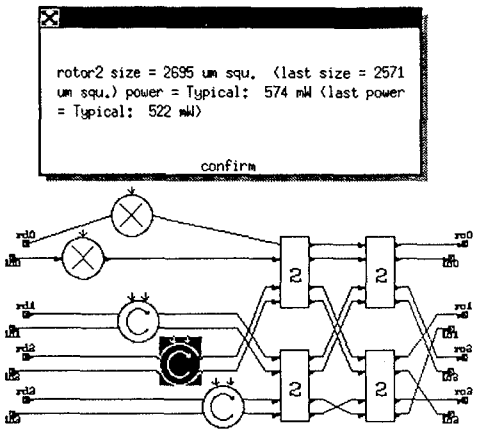

Figure 8. Local size estimation.

7. 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 optimized functionally correct circuit. The hardware is minimized using data-recoding techniques according to 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 serial clock appropriately buffered.

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 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 task) is shown in Figure 9. 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.

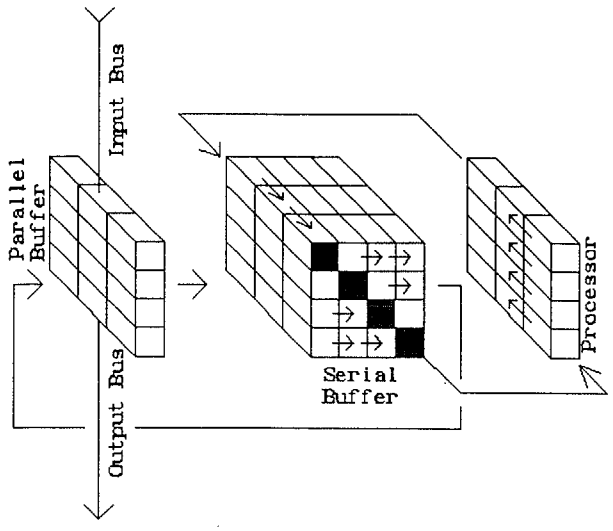

Figure 9. Corner-turning within the Registers

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 circuitry for the core and registers, the compiler automatically generates all control logic and all the logic required to ensure data consistency at the inputs of every operator.

8. 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.

```
# swinging buffer memories
For i3 = 0 upto 1 { # rows
  For i2 = 0 upto columns-1 { # columns
    For i1 = 0 upto rows-1 { # engine terminals
      For i0 = 0 upto blocksize-1 {
        form_addresses
      }
      task
    }
  }
}
```

Figure 10. Operational Template

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 10 shows pseudo-code for the operational template for the FFT machine (much simplified) using the butterfly as core processor.

It should be emphasised that the DSP Engine Compiler does not yet synthesize 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.

9. Conclusions

The paper has described an ASIC compiler that uses digit-serial architectures in order to address certain classes of DSP applications. The paper has outlined the architectural methods used by the compiler showing some examples of the efficiency of the solution in terms of silicon area. The method for design specification has been illustrated with an example which illustrates the simplicity of the specification as well as the ability to track specific information, such as the noisefloor and word growth, that is of interest to a DSP designer.

10. References

1. S. G. Smith and J. M. Rischard, "20 MHz 16x16 Discrete Cosine Transform IC: CAD and Architectural Methodology", pp. 369 - 378 in VLSI'89, ed. G. Musgrave and U. Lauther, Elsevier Science Publishers (1990)
2. R. I. Hartley and P. F. Corbett, "Digit-Serial Processing Techniques", Trans. IEEE CAS-37 pp. 707 - 719 (June 1990)
3. S. G. Smith and P. B. Denyer, Serial-Data Computation, Kluwer Academic Publishers (1988)
4. 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)