RTL Fast Convolution using the Mersenne Number Transform

Oscar N. Bria and Horacio A. Villagarcía

o.bria@ieee.org

CeTAD - UNLP - Argentina

Abstract

VHDL is a versatile high level language for the specification and simulation of hardware components. Here a functional VHDL model is presented for performing fast convolution based on Mersenne's number theoretic transform.

For filtering a rather long input sequence $x(n)$ we can decomposed it into a number of short segments, each of which can be processed individually. The output $y(n)$ then becomes a combination of partial convolutions. The superposition principle for linear operators is used here.

Each partial convolution can be solved using the Discrete Fourier Transform (DFT) implementing a fast FFT (Fast Fourier Transform) algorithm. This DFT approach is the most popular.

In this paper we use the Mersenne Number Transform (MNT) as an alternative for the DFT in the framework of a register transfer level (RTL) implementation of the filter operation. Even when the MNT does not have a fast algorithm it can be see that RTL in the natural level of abstraction for the implementation of the MNT.

This work is conceived as part of an academic exercise in the use of VHDL for modeling a DSP algorithm all the way from the mathematical specification to the circuit implementation.

Introduction

VHDL can be used in the first steps of circuit design as a model description tool [RTWG]. Here VHDL is used for functional modeling a method: the use of the Mersenne number transform for computing integer circular convolution (ICC).

Convolution, in particular ICC, is basic for many DSP applications [OpSc]. Besides the direct implementation, DFT based approaches (using FFT) are the most popular. The DFT is just one of the block transforms with the appealing convolution property.

But the FFT approach have the following drawbacks:

- The FFT is formulated for complex sequences. Some tricks are available for real sequences but lead to complications in procedure.
- The FFT uses computations involving weighting factor that are irrational numbers. There is an inherent need to tolerate approximation.
- Most of the multiplications needed in computing the FFT are not by ``easy'' numbers.

Number-theoretic transforms are based on computations in a finite ring, and can be used to compute convolutions and correlations. In particular, the MNT has the following ``good'' properties [PRFN]:

- MNT involves multiplications that are only by factors of the form 2^r. Then multiplication is solved by rotating the bits in the datum by r bit positions. This means a traditional rotate operation over a shift-register.
- MNT involves additions that are module M_p (a Mersenne number). It can be solved using one's complement arithmetic[1].
- Multiply two MNTs together, $C_k = A_k \times B_k$ is not more logically complex than ordinary multiplication.

Those properties suggest than the MNT is naturally well suited for ASIC/FPGA VLSI mapping.

Even when a functional description is presented here, a partitioning guideline has to be adopted. here an object based top-down (OBTD) approach is used for partitioning the model into sub-pieces.

Mersenne Number Transform

In this section we follow Proakis et. al. [PRFN]. Two sequences a_n and b_n can be convoluted by first ``transforming'' a_n and b_n, then multiplying the transforms together point to point, and finally inverse transforming the product. Many rectangular transform

[1] One's complement arithmetic can be implemented by the *end - around* carry technique.

can be chosen, leading to the so called BDA-type algorithm. The traditional transform used is the DFT, which is also in the BDA-type class. The DFT is popular because the transform can be computed quickly by FFT.

Let $M_p \equiv x^p - 1$ where p is a prime number. M_p is called a *Mersenne* number[2]. The order of x with respect to M_p is exactly p.

Let $x = 2$, then $2^p \equiv 1 \bmod 2^p - 1$. This is worth looking at the powers of 2, module M_p, in binary notation[3]. In fact, 2^m mod M_p, is a word with $p-1$ zeros and a 1 in bit position $\langle m \rangle_p$[4].

The Mersenne number transform (MNT) of the sequence $\{a_0, a_1, \ldots, a_{p-1}\}$ is defined as,

$$A_k = \left\langle \sum_{n=0}^{p-2} a_n 2^{nk} \right\rangle_{M_p}$$

The inverse Mersenne number transform can be defined given a number $\hat{N}$ (guaranteed to be integer by Euler's theorem) defined as,

$$\hat{N} = \frac{1 - M_p}{p} = \frac{2 - 2^p}{p}$$

It can be proved that the above defined number transform has the convolution property, but there is no FFT-like algorithm for computing the Mersenne transform. But the arithmetic is so simple that in fact can be solved by shift operations and addition including *end-around carry* (i.e., *one's complement* number representation).

The ability to replace multiplications by complex transcendental quantities by multiplication by powers of 2 (simply implemented by shift operations) is the principal advantage of MNT over DFT (even when a FFT algorithm is used). Besides, the result of the computed convolution via MNT is always exact without round-off; but this can be a disadvantage because there is not any possibility for round-off[5] and this can force the use of a an inconveniently large module (i.e., word length).

Integer Circular Convolution with MNT

The procedure for computing any Υ-transform convolution has always the same three steps: a direct transform, a two-operand multiply step, and finally an inverse transform.

---

[2] If M_p is prime is called a Mersenne prime, but in this discussion M_p need not to be prime.
[3] Note the fact that the word length is always a prime number.
[4] The notation $\langle m \rangle_p$ means m mod p.
[5] Because exact computation is not always required.

$$\mathrm{Y}^{-1}\left(\mathrm{Y}\left(a_n\right) \otimes \mathrm{Y}\left(b_n\right)\right)$$

Therefore, for computing the ICC it is necessary to multiply point to point two transforms in that mod M_p arithmetic ($\otimes$). When two Mersenne number transform are multiplied together $C_k = A_k \times B_k$, the product C_k has $2p$ bits. Calling C_L the p least significant bits, and C_H the remaining bits of C_k, then,

$$C_k \equiv C_L + C_H \bmod M_p$$

where the add operation is made in one's complement arithmetic.

Let $M_p = 127$ (or $p = 7$). Suppose $a_n = \{1,2,3,4,3,2,1\}$ [6] and $b_n = \{1,1,0,0,0,0,0\}$.

1) Transforming, we get $A_k = \{16,-29,-14,-40,61,-56,-58\}$ and $B_k = \{2,3,4,9,17,33,-62\}$.

2) Multiplying A_k by B_k we get $C_k = \{32,40,57,21,21,57,40\}$.

3) Inverse-transforming with $\hat{N}$ we obtain the expected result $c_n = \{2,3,5,7,7,5,3\}$.

Object Based Top Down Structure for ICC with MNT

Even in a functional description, a partitioning guideline has to be adopted due to practical reasons. The partitioning guideline is object based top down (OBTD) [VhdT]. This approach is particularly useful for structuring the future implementation. The key in OBTD is the definition of objects (eventual final components) that can be reused by other projects. An atomic object has two important characteristics: it is simple to implement in a reasonable sized architecture and it is sufficient independent that it can be exhaustively tested after every modification. In VHDL, objects can be grouped into practical packages.

The so called ICC is the top object. Down from it the following objects and packages can be identified:

- ICC object
    - MNT package: Direct and Inverse MNT
    - OPS package: These are common MNT operators
        - Rotate object
        - Crush object
    - CA1 package: 1's complement arithmetic operators
        - Sum object
        - Product object
        - Cproduct object

[6] In this example we are using decimal instead of one's complement notation for the seek of simplicity.

The names of the packages and functionality of the object is almost self-explanatory:

- OPS is the package of the basic operations for the implementation of the direct as well as the inverse MNT. Rotate is a classical rotate operation over a register, and Crush is the structure for a multiple operation. This package can be reused.

- C1A implements one's complement arithmetics. Sum is a two-operand sum, Product is a two-operand multiply, and Cproduct implement the multiplication of a variable number by a constant number (this is useful for the inverse transform). This package can be reused.

Top Level VHDL Model

VHDL functional model refers to writing VHDL code for the objects. The goal is to write some code and get a simulation running, but not to get some synthesized gates. In this steps it is possible to use any other suitable language or system, C, Eiffel, Signal, Matlab[7], Khoros, etc. Nevertheless, we choose to use VHDL because: It is a standard language from which, if desired, it is possible automatic gate synthesis (in general from a given subset of the language); and it is, at a time, a description, behavioral, and simulation language. Hence, SCOUT [Com2], the companion VHDL simulation environment for Compass Tools [Com1][8], is used for the functional and structural description.

The main characteristics of this step are summarized here:

1. The VHDL model is writing following a down top schedule, based on the objects defined in the OBTD partitioning.
2. Almost every object is quick implemented as a VHDL function or a VHDL procedure. No concern about interfaces is taken but the data flow signals.
3. Sizing and timing are out of concern in the functional model. No clock is necessary for a functional description, but it is convenient to think about its future insertion in the system.
4. when defining a package it is natural to extend then to a more complete set. This is the case for the ca1 arithmetic package: not only Sum, Product, and Cproduct are implemented but also a complete set of ca1 operators and related conversion functions for friendly interface.
5. Concurrence among objects, if any, is easily detected and described at this level.
6. Synchronization issues can be tested and the occurrence of deadlock can be prevented[9]. SCOUT has not automatic detection of those problems [Tine]. In this case synchronization among the objects is quite simple, but synchronization inter object may be a more complex subject particularly in C1A's ones.
7. All the objects are functionally tested with their own test bench.
8. ICC and its test bench are the final objects implemented, and exhaustive behavioral verifications are performed.

[7] Matlab is a trademark of MathWorks Inc.
[8] SCOUT and the Compass Tools here mentioned are trademarks of Compass Design Automation Inc.
[9] Note that abstract synchronization and deadlocks do not depend on timing.

Detailed VHDL Design

For the detailed VHDL design (logic or even circuit design) prior to synthesis there are many architectural alternatives. We present in the appendix an example with the Sum object. The functional definition of the Sum object as part of the C1A package can be see in the appendix as well as an alternative for the detailed VHDL design for the Sum object, using a FullAdder as a basic component.

Actually, this step is just an explanation[10] with constrains from a top level of functional description to a down level of structural detail. This procedure can be performed more or less automatically depending of the availability of software and the degree and confidence of its mapping performance for the target technology.

Linear Convolution and LIT system implementation using the MNT

Given an algorithm that implement ICC it is computationally possible to implement an integer linear convolution (ILC) algorithm ensuring that the circular convolution has the effect of linear convolution. Whether a circular convolution corresponding to the product o two N-point MNTs is the linear convolution of the corresponding finite-length sequences depends on the length of the MNT in relation to the length of the finite-length sequences. We can interpret these in terms of time aliasing (see [OpSc]).

Two basic approaches are possible:

1) The overlap-add method.
2) The overlap-save method.

In many application the input signal for practical purposes is of infinite duration. In both methods the solution to that problem is to split the signal into section or blocks. In the first approach the filtered blocks are overlapped and added to construct the output. In the second approach the part of the circular convolution that correspond to exact linear convolution are identified and the rest discarded. The resulting output segments are then concatenated together to form the output.

Both approaches have been implemented in VHDL conforming a new top object called Linear Time-Invariant System using Mersenne,

- LITMa object or,
- LITMs object.

Actually this two object are independent of the Mersenne transform, and can be used in a DFT or any alternative context.

Concluding Remarks

We are convinced that designers will rely increasingly often on reusing previous designs, either created within their working place or purchased as intellectual property from other sources. Another trend is the increasing number of complex digital signal processing applications (e.g. interactive natural language processing). Then, the development of a DSP VHDL library with an OBTD structure is becoming a precious

[10] Within the same language.

and strategic commodity. We presented here just a specimen of such a library, and defined at a top functional level.

Besides the development of a concrete library component for DSP design, this effort is an academic exercise using the VHDL language as a powerful tool for specification, verification and reusability. We think it is an excellent case of study in which it is necessary a sound background in numerical analysis as well as in DSP.

It remains, as a task for the final user, to compare the performance and cost of this MNT approach to traditional solutions for implementing ICC when they are embedded in the framework of a selected IC technology for a particular DSP application, and when final architectures are fixed.

VHDL is a very flexible language, well suited for the description of algorithms from the top functional level, that correctly models concurrence, and that can be eventually used to generate scaled nets to synthesize circuits and fully integrate them on silicon.

References

[Com1] "ASIC Synthesizer for VHDL Design. "
Compass Design Automation, Inc., 1991.

[DePH] "Discrete-Time Processing of Speech Signals."
John Deller, John Proakis, and John Hansen.
MacMillan, 1993.

[StSh] "ASIC System Design with VHDL: A Paradigm."
Steven Leung and Michel Shanblatt.
Kluwer, Academic Publishers, 1990.

[LiSU] "VHDL: Hardware Description and Design."
Roger Lipsett, Carl Schaefer, and Cary Ussery.
Kluwer Academic Press, 1991.

[OpSc] "Discrete-Time Signal Processing."
Alan Oppenheim and Roland Schafer.
Prentice Hall, 1989.

[PRFN] "Advance Digital Signal Processing."
John Proakis, Charles Rader, Fuyun Ling, and Chrysostomos Nikias.
MacMillan, 1992.

[RTWG] "RASSP VHDL Modeling Terminology and Taxonomy."
RASSP Taxonomy Working Group (RTWG).
URL:http://rassp.scra.org, 1996.

[Com2] "The VHDL Scout Handbook, " Third Edition.
Compass Design Automation, Inc., 1994.

[Tine] ``Signal as a HDL for Image Processing.''
Fernando Tinetti.
Anales de las 3ras Jornadas de la AUGM, 1995.

[VhdT] ``VHDL Designer.''
VHDL Times, Vol.4 No.2.
URL:\verb"http://vhdl.org", 1995.

Appendix A:

- VHDL functional definition of the Sum object as part of the C1A package:

```
package C1A is

     -- overloaded functions for bit_vector operations
     function "+" (a,b: bit_vector) return bit_vector;
     --
     --
     --
    end C1A;

    package body C1A is

     function "+" (a,b: bit_vector) return bit_vector is
      variable av,bv: bit_vector (1 to a'length);
      variable carry: bit;
      variable sum: bit_vector (1 to a'length);

     begin
      assert a'length=b'length
         report "Operands to overloaded '+' operator with different lengths"
         severity failure;

      carry:='0';
      av:=a;
      bv:=b;

      for j in 1 to 2 loop

        for i in a'length downto 1 loop
          sum(i):=av(i) xor bv(i) xor carry;
          carry:=(av(i) and bv(i)) or (av(i) and carry) or (bv(i) and carry);
        end loop;

        av:=sum;
        bv(a'length):=carry;
        carry:= '0';
        j:=j+1;

      end loop;
      return sum;
     end;

     --
     --
     --

    end C1A;
```

- VHDL design for the Sum object, using a FullAdder as a basic component:

```
entity Sum is
     port (A, B:  in   Bit_Vector(7-1 downto 0);
          Sum:   out  Bit_Vector(7-1 downto 0));
    end Sum;

    architecture Structure of Sum is

     component FullAdder
      port (X, Y:  in   Bit;
           Cin:   in   Bit;
           Cout:  out  Bit;
           Sum:   out  Bit);
     end component;
```

```
  signal C, H: Bit_Vector(7-1 downto 0);
  signal Z:    Bit := '0';

begin
  Stages:
    for i in 7-1 downto 0 generate

      LowBit:
        if i = 0 generate
          FA1: FullAdder port map
                (A(0), B(0), Z, C(0), H(0));
          FA2: FullAdder port map
                (H(0), Z, C(7-1), C(0), Sum(0));
        end generate;

      OtherBits:
        if i /= 0 generate
          FA1: FullAdder port map
                (A(i), B(i), C(i-1), C(i), H(i));
          FA2: FullAdder port map
                (H(i), Z, C(i-1), C(i), Sum(i));
        end generate;

    end generate;

end;
```