Describing and Verifying FFT circuits using SharpHDL

Gordon J. Pace & Christine Vella

Department of Computer Science and AI,
University of Malta

Abstract. Fourier transforms are critical in a variety of fields but in the past, they were rarely used in applications because of the big processing power required. However, the Cooley's and Tukey's development of the Fast Fourier Transform (FFT) vastly simplified this. A large number of FFT algorithms have been developed, amongst which are the radix-2 and the radix-2^2. These are the ones that have been mostly used for practical applications due to their simple structure with constant butterfly geometry. Most of the research to date for the implementation and benchmarking of FFT algorithms have been performed using general purpose processors, Digital Signal Processors (DSPs) and dedicated FFT processor ICs but as FPGAs have developed they have become a viable solution for computing FFTs. In this paper, SharpHDL, an object oriented HDL, will be used to implement the two mentioned FFT algorithms and test their equivalence.

1 Introduction

Embedded domain specific languages have been shown to be useful in various domains. One particular domain in which this approach has been applied is hardware description languages. SharpHDL is an example of such a language. It is embedded in C#, a modern object-oriented programming language, enabling us to describe structurally large regular circuits in an intuitive way. Its structure makes it easily extendible and helps the user produce clear and elegant circuit descriptions. These descriptions can then be used by Verilog simulators and SMV model checkers.

This paper introduces us to SharpHDL and describes how it is used to build and verify FFT circuits. Section 2 gives a brief overview of the SharpHDL syntax and some of its features. General information about FFTs and derivation of two FFT algorithms are given in section 3 while section 4 describes how the circuits of these algorithms are implemented using SharpHDL. It also describes how their equivalence was verified. Section 5 gives an account of some works related to this paper while section 6 draws up some conclusions.

2 SharpHDL — An Objected-Oriented Hardware Description Language

SharpHDL [3] is an HDL embedded in C#, an objected-oriented language. By embedded we mean that SharpHDL uses the same syntax, semantics and other tools belonging to C#. By objected-oriented we mean that ideas are expressed in terms of objects, which are entities that have a set of values and perform a set of operations [8].

2.1 Syntax Overview

As an introduction to the syntax of SharpHDL we see how a half-adder circuit is built. A half-adder accepts two inputs x and y and outputs the sum $x\ XOR\ y$ and the carry $x\ OR\ y$. Below, is the SharpHDL code for this circuit followed by a quick explanation.

```
//<1>
using System;
using SharpHDL;

//<2>
public class HalfAdder: Logic {
    //<3>
    public HalfAdder(Logic parent):base(parent){}

    //<4>
    protected override void DefineInterface(){...}

    //<5>
    public void gate_o(LogicWire in0, LogicWire in1,
                ref LogicWire sum, ref LogicWire carryOut)
    {
        //<6>
        Wire[] input = {in0, in1};
        Wire[] output = {sum, carryOut};
        this.Connect(input, output);

        //<7>
        new And(this).gate_o(in0, in1, ref carryOut);
        new Xor(this).gate_o(in0, in1, ref sum);
    }
    //<8>
    public LogicWire gate(LogicWire in0, LogicWire in1,
                       out LogicWire sum){...}
}
```

$< 1 >$. To be able to use the main C# functionalities we call the `System` namespace, which contains classes that implement basic functionalities. The `SharpHDL` library is also invoked so as to be able to make use of its facilities. $< 2 >$ Since we are going to code a circuit, the `HalfAdder` class should inherit from the class `Logic`, an abstract class representing the general logic circuit. An instance of a class is initialized using a constructor which is defined in $< 3 >$. In SharpHDL, a user needs to specify the parent of the new circuit. By parent we mean the circuit to which the new circuit is a direct sub-circuit. Therefore, a half-adder circuit will be initialized as follows:

```
HalfAdder ha = new HalfAdder(parent_circuit)
```

$< 4 >$ The ports needed in the circuit are specified by overriding `DefineInterface`, which is a virtual method. Although this is an optional step it may be necessary when using the circuit as input to a generic circuit. $< 5 >$ The behavior of the circuit is defined in the method `gate_o`[1]. It accepts four wires, two of which are the inputs and the last two are the outputs. The keyword `ref` indicates that the parameter is a *Reference Parameter*. In other words, the values are supplied by reference and therefore can be read and modified. $< 6 >$ A compulsory step is to connect the wires to the respective ports. When the ports are not specified as explained in $< 4 >$, the ports are usually created automatically. $< 7 >$ As already mentioned, the circuit needs two other circuits, a `XOR` gate and an `AND` gate. An instance of each is created by calling their respective constructor

[1] In SharpHDL, the name `gate_o` is a standard method name for the method that specifies the structure and behavior of the given circuit, given the necessary input and output wires. Likewise is the method `gate` ($< 8 >$), which, although it has the same function, it creates new output wires.

and specifying that the `HalfAdder` is the parent circuit by using the keyword `this`. The `gate_o` method of each circuit is invoked, passing to it the required parameters.

2.2 Using Generic Circuits

Generic Circuits are circuits which accept another circuit as input and use it to construct a more complex circuit with a regular structure. Hence, such circuits are called higher-order circuits [10]. SharpHDL provides a library of various generic circuits. This section briefly describes some of these circuits that will be referred to later.

- The generic circuit *TwoN* accepts a list of wires and recursively divides it in two for n times, applying the input circuit to the resultant lists.
- *OneN* is similar to TwoN but instead of applying the circuit to each resultant half, it applies it to the most bottom half only.
- A *Butterfly circuit* is another circuit that is built recursively. The input circuit is a component having two inputs and two outputs. After riffling the list, it uses the `TwoN` generic circuit to divide a given list of wires for $\log_2 size_of_list$ times and applies the input circuit to the resultant subsets of wires. At the end, the list is unriffled. By riffling we mean shuffling the list such that the odd-indexed items are separated and followed by the even-indexed, while unriffling is the reverse operation.

3 Fast Fourier Transforms

The Fast Fourier Transform is one of the most important topics in Digital Signal Processing. It is a computationally efficient way to calculate the Discrete Fourier Transform (DFT) [2, 7]. A DFT's problem involves the computation of the sequence $\{X(k)\}$ of N complex numbers given a sequence of N $\{x(n)\}$, according to the formula:

$$X(k) = \sum_{n=0}^{N-1} x(n) W_N^{kn} \quad 0 \leq k \leq N-1$$

where

$$W_N^{kn} = e^{-j2\Pi/N} = \cos(\tfrac{2\Pi}{N} \cdot nk) - j\sin(\tfrac{2\Pi}{N} \cdot nk)$$

The latter is also known as the twiddle factor. When symmetric and periodicity properties of the twiddle factor are taken into consideration, an efficient computation of DFTs is carried out. The properties specify that

$$W_N^{k+\frac{N}{2}} = -W_N^k \text{ (Symmetry property)}$$
$$W_N^{k+N} = W_N^k \text{ (Periodicity property)}$$

FFTs use these properties, making them efficient algorithms for computing DFTs.

3.1 Radix-2 Decimation in Time FFT Algorithm

The Radix-2 DIT FFT algorithm [2] is the simplest and most written about form of the Cooley-Tukey algorithm. It works on an input sequence having length to the power of two. It splits the input into the odd-indexed numbers and the even-indexed numbers, hence making it a decimation-in-time algorithm. Therefore,

$$f_1(n) = x(2n)$$
$$f_2(n) = x(2n+1) \quad n = 0, 1, \ldots, \tfrac{N}{2} - 1$$

It follows that

$$X(k) = \sum_{n=0}^{N-1} x(n)W_N^{kn}$$
$$= \sum_{n\, even} x(n)W_N^{kn} + \sum_{n\, odd} x(n)W_N^{kn}$$
$$= \sum_{m=0}^{(\frac{N}{2})-1} x(2m)W_N^{2mk} + \sum_{m=0}^{(\frac{N}{2})-1} x(2m+1)W_N^{k(2m+1)}$$

But $W_N^2 = W_{\frac{N}{2}}$

Therefore,

$$X(k) = \sum_{m=0}^{(\frac{N}{2})-1} f_1(m)W_{\frac{N}{2}}^{km} + W_N^k \sum_{m=0}^{(\frac{N}{2})-1} f_2(m)W_{\frac{N}{2}}^{km}$$
$$= F_1(k) + W_N^k F_2(k) \quad k = 0, 1, \ldots N-1$$

where $F_1(k)$ and $F_2(k)$ are the $\frac{N}{2}$-point DFTs of the sequences $f_1 m$ and $f_2(m)$ respectively.

Using the symmetry property, we know that $W_N^{k+\frac{N}{2}} = -W_N^k$. We also know that $F_1(k)$ and $F_2(k)$ are periodic, having period $\frac{N}{2}$. Therefore,

$$F_1(k + \tfrac{N}{2}) = F_1(k) \quad \text{and}$$
$$F_2(k + \tfrac{N}{2}) = F_2(k)$$

Hence,

$$X(k) = F_1(k) + W_n^k F_2(k) \;\; k = 0, 1, \ldots, \tfrac{N}{2} - 1$$
$$X(k + \tfrac{N}{2}) = F_1(k) - W_n^k F_2(k) \;\; k = 0, 1, \ldots, \tfrac{N}{2} - 1$$

This is known as the Radix-2 FFT Butterfly, better illustrated in Figure 1. In this diagram, multiplications are represented by numbers on the wires and crossing arrows are additions.

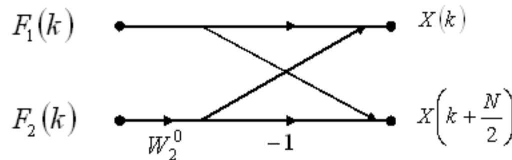

Fig. 1. Radix-2 FFT Butterfly

The decimation of the data is repeated recursively until the resulting sequences are of length two. Thus, each $\frac{N}{2}$-point DFT is computed by combining two $\frac{N}{4}$-point DFTs, each of which is computed using two $\frac{N}{8}$-point DFTs and so on. The network for this algorithm can be seen in Figure 2.

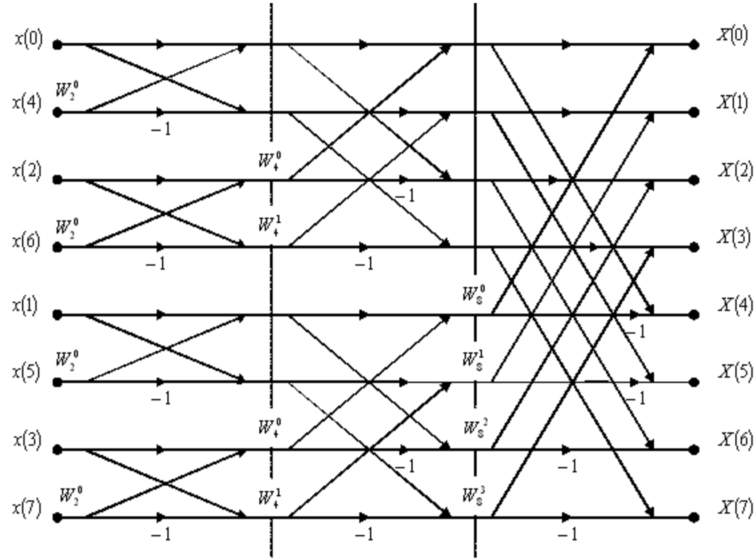

Fig. 2. Size 8 Radix-2 DIT FFT

One should note that the indexes of the input sequence are re-ordered. The technique used is called *Bit Reversal* [1, 2, 9]. As it suggests, it basically involves the reversing of the bits such that the MSB becomes the LSB and vice-versa. Therefore, bit-reversed indexes are used to combine FFT stages.

3.2 Radix-2^2 Decimation in Time FFT Algorithm

The Radix-2^2 FFT Algorithm [4] is a less popular algorithm than the one described above. It is used by an input sequence of length to the power of 4, so that the N-point DFT formula can be broken down into four smaller DFTs.

Therefore, the DFT can be calculated as follows:

$$X(k) = \sum_{n=0}^{N-1} x(n)W_N^{kn}$$
$$= \sum_{n=0}^{\frac{N}{4}-1} x(n)W_N^{kn} + \sum_{n=\frac{N}{4}}^{\frac{N}{2}-1} x(n)W_N^{kn} + \sum_{n=\frac{N}{2}}^{\frac{3N}{4}-1} x(n)W_N^{kn} + \sum_{n=\frac{3N}{4}}^{N-1} x(n)W_N^{kn}$$
$$= \sum_{n=0}^{\frac{N}{4}-1} x(n)W_N^{kn} + W_N^{\frac{Nk}{4}} \sum_{n=0}^{\frac{N}{4}-1} x(n+\tfrac{N}{4})W_N^{kn}$$
$$+W_N^{\frac{Nk}{2}} \sum_{n=0}^{\frac{N}{4}-1} x(n+\tfrac{N}{2})W_N^{kn} + W_N^{\frac{3Nk}{4}} \sum_{n=0}^{\frac{N}{4}-1} x(n+\tfrac{3N}{4})W_N^{kn}$$

We know that

$$W_N^{\frac{kN}{4}} = (-j)^k, \quad W_N^{\frac{kN}{2}} = (-1)^k, \quad W_N^{\frac{3kN}{4}} = (j)^k$$

Hence,

$$X(k) = \sum_{n=0}^{\frac{N}{4}-1} [x(n) + (-j)^k x(n+\tfrac{N}{4}) + (-1)^k x(n+\tfrac{N}{2}) + (j)^k x(n+\tfrac{3N}{4})]W_N^{nk}$$

To get the radix-2^2 decimation-in-frequency DFT, we subdivide the DFT sequence into four $\frac{N}{4}$-point sub-sequences:

$$X(4k) = \sum_{n=0}^{\frac{N}{4}-1} [x(n) + x(n+\tfrac{N}{4}) + x(n+\tfrac{N}{2}) + x(n+\tfrac{3N}{4})]W_N^0 W_{\frac{N}{4}}^{kn}$$

$$X(4k+1) = \sum_{n=0}^{\frac{N}{4}-1}[x(n) - jx(n+\tfrac{N}{4}) - x(n+\tfrac{N}{2}) + jx(n+\tfrac{3N}{4})]W_N^n W_{\frac{N}{4}}^{kn}$$

$$X(4k+2) = \sum_{n=0}^{\frac{N}{4}-1}[x(n) - x(n+\tfrac{N}{4}) + x(n+\tfrac{N}{2}) - x(n+\tfrac{3N}{4})]W_N^{2n} W_{\frac{N}{4}}^{kn}$$

$$X(4k+3) = \sum_{n=0}^{\frac{N}{4}-1}[x(n) + jx(n+\tfrac{N}{4}) - x(n+\tfrac{N}{2}) - jx(n+\tfrac{3N}{4})]W_N^{3n} W_{\frac{N}{4}}^{kn}$$

Note that the property $W_N^{4kn} = W_{\frac{N}{4}}^{kn}$ is used. This procedure, illustrated in Figure 3, is repeated for $\log_4 N$ times.

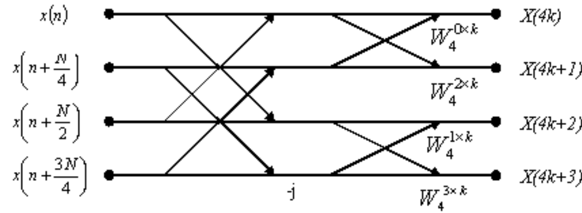

Fig. 3. A Radix-2^2 procedure

The corresponding network for this algorithm can be seen in Figure 4. One can note that the output needs to be re-ordered using a bit-reversal permutation, as explained in the previous section.

4 FFTs in SharpHDL

This section explains how SharpHDL is used to implement FFT circuits.

FFTs work with complex numbers, therefore, a new type of wire `ComplexNumber` was created in SharpHDL which represents such numbers. `ComplexList` is a list object that accepts `ComplexNumber` objects.

Although being two different approaches to solving FFTs, the algorithms described above have common components:

- From the network designs one can easily notice the common *butterfly operation*, known as the `FFTComponent` in SharpHDL. This takes two inputs x_1 and x_2, and returns $x_1 + x_2$ and $x_1 - x_2$. This is the main operator of the algorithms as it calculates the 2-point DFT.
- Another important operation is the *twiddle factor multiplication*. This component takes a complex number x and multiplies it with the W_N^k constant. The Radix-2^2 FFT algorithm also uses multiplication with $-j$ which is equal to the constant W_4^1.
- Both algorithms use the *bit reversal permutation*, though in different times as described previously. This operation can be carried out by riffling the sequence recursively [10].

Using the above components and the generic circuits described in a previous section we can construct the circuits for the two FFT algorithms using SharpHDL. A Radix-2 FFT algorithm is described as follows:

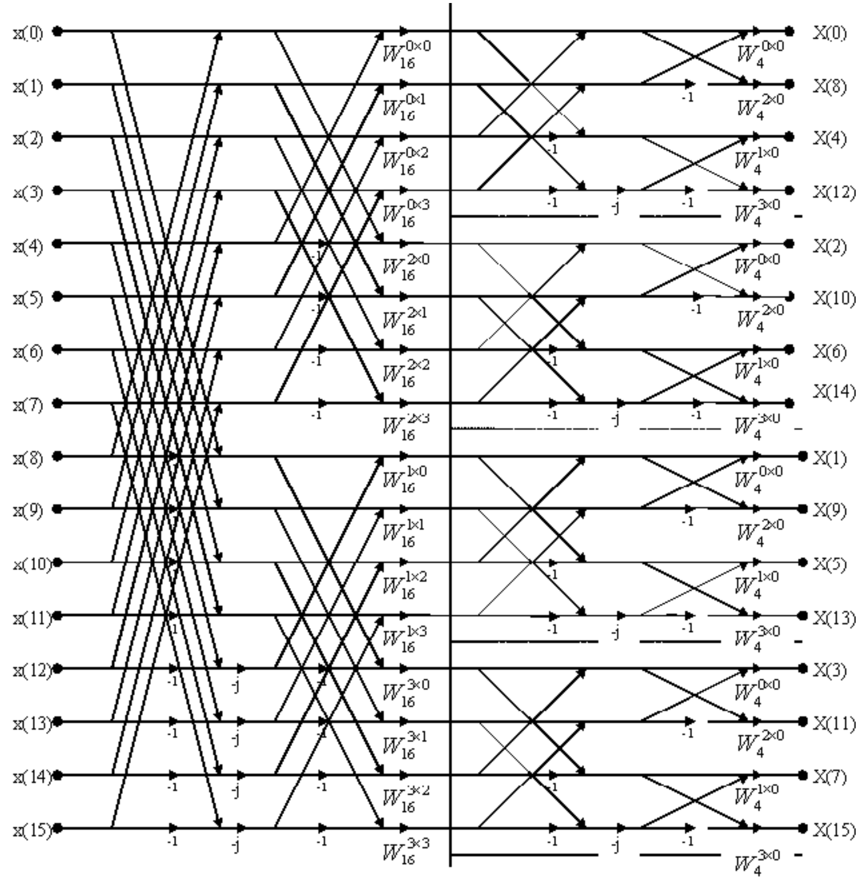

Fig. 4. A SIZE 16 RADIX-2^2 DIF FFT ALGORITHM

```
//BitReversal
in0 = (ComplexList)new BitReversal(this).gate(in0);

int exponent = (int) Math.Log(in0.Count, 2);

for (int i = 0; i < exponent; i++)
{
    int n = (exponent-1) - i;

    //Use TwoN to divide the list for n times and apply
    //the Radix2FFTStage to each resultant set
    ComplexList output0 = ComplexList)new TwoN(this,
                    new Radix2FFTStage(null),n).gate(in0);
    in0 = output0;
}
```

where a `Radix2FFTStage` circuit is described as follows:

```
...
//Apply twiddle using OneN generic circuit
BusWire one = new OneN(this, new FFT2Twiddle(null), 1).gate(in0);

//Call Butterfly using the FFTComponent
new Butterfly(this, new FFTComponent(null)).gate_o(one, ref outBus);
```

A Radix-2^2 FFT algorithm is described as follows:

```
...
int exponent = (int)Math.Log(in0.Count, 4);
int n;
for (int i = exponent; i > 0; i--)
{
    n = (exponent - i)*2;

    //Use TwoN to divide the list for n times and apply
    //the Radix4FFTStage to each resultant set
    ComplexList output0 = (ComplexList)new TwoN(this,
                new Radix4FFTStage(null), n).gate(in0);
    in0 = output0;
}

//BitReversal
new BitReversal(this).gate_o(twoN, ref output0);
```

where the `Radix4FFTStage` circuit is described as follows:

```
 ...
//Use Butterfly generic circuit using the FFTComponent
BusWire bflys = new Butterfly(this, new FFTComponent(null)).gate(in0);

//Use OneN to multiply with -j
BusWire ones = new OneN(this, new MinusJ(null), 2).gate(bflys);

// Use TwoN to divide the list of wires and apply a Butterfly to it
// The Butterfly circuit uses the FFTComponent
ComplexList twoBflys = (ComplexList)new TwoN(this,
        new Butterfly(null, new FFTComponent(null)), 1).gate(ones);

//Multiply with twiddle constant
new FFT4Twiddle(this).gate_o(twoBflys, ref out0);
```

4.1 Verifying the equivalence of the FFT circuits using SMV

One of the tools offered by SharpHDL is that of converting a circuit description to SMV to verify safety properties. SMV [11] is a tool used for verifying that a finite system conforms to a set of CTL specifications using symbolic model checking. Safety properties state that a condition is always true or never false. Using this tool, we can verify that two circuits are equivalent, i.e. for any sequence of input they generate the same output. Therefore, the equivalence relation R can be defined as

$$R = \lambda x, y.\forall z.(\gamma(x, z) = \gamma(y, z))$$

where $\gamma(x, z)$ is the function that determines the output from circuit x given input z [11]. Using SharpHDL, this can be done by constructing a *Comparison Circuit* which calls two circuits using the same input and compares the outputs. The circuit outputs `true` if all the outputs are equal [13].

The equivalence of the two FFT circuits can be verified using this methodology. The circuits are given the same input list of complex numbers and the two outputs $output_{radix-2}$ and $output_{radix-2^2}$ are compared. Iff they are all equal the FFT circuits are equivalent. On giving the generated SMV code of this circuit to the model checker, it proved that the FFT circuits of size 4 are equal.

4.2 Related Works on FFT circuits description and verification

The equivalence of two FFT algorithms using a Hardware Description Language has been shown using Lava in [9]. Lava is an HDL embedded in Haskell, a functional programming language. Although the programming paradigm used is different, the two algorithms and the verification approach are more or less the same as used in this paper.

5 Related Work

In this section we discuss some works related to other HDLs.

Synchronous programming languages provide tools that easily construct automatons that describe reactive systems, where the main tool they provide is the concept of synchronous concurrency. **Lustre** is one such language. Besides programming reactive systems, Lustre is used to describe hardware and express and verify their properties which makes it similar in concept to SharpHDL.

Lustre programs define their outputs as functions of their inputs. Each variable is defined using equations such that `A=B` means that the variable `A` is always equal to expression `B`. Expressions consist of variable identifiers and constants, arithmetic, boolean and conditional operators and another two special operators: *previous* and *followed-by* operators. Taking E_n and F_n to denote the values of `E` and `F` at instant n, the *previous* operator `pre(E)` returns the previous value of `E`, such that $(pre(E))_n = E_{n-1}$. The *followed by* operator is used to define initial values, such that $(E- > F)$ is defined as follows:

- $(E- > F)_0 = E_0$
- $(E- > F)_n = F_n$ for $n > 0$

On the other hand, SharpHDL defines a circuit using instances of objects representing wires and circuits thus making the language a structural hardware description language. One must point out that the difference in concept and paradigm does not effect SharpHDL's power or efficiency. Nevertheless, it should be noted that the various studies [12–14] conducted by Halbwachs et al concerning synchronous observers and the use of Lustre were of great help in understanding and using verifications techniques in SharpHDL. Further use of such techniques in SharpHDL is already in the pipeline.

Another HDL with verification capabilities is the already-mentioned Haskell-based **Lava** [9, 10]. One of Lava's high points is its simplicity and elegance, much aided by its functional-oriented host language. It also offers the facilities of *connection patterns*, which basically are the generic circuits found in SharpHDL. Another similar tool to SharpHDL is that a user can analyze circuits using both simulation and model checking.

Lava is used as an experimental tool for formal verification of hardware at Chalmers. It is also being successfully used in the development of FPGAs such as filter, Bezier curve drawing circuits and digital signal processing [10].

6 Conclusion

In this paper, we have described and made use of SharpHDL, an object-oriented HDL. We have seen how the object-oriented paradigm can be used to create a powerful HDL. This success can especially be seen in the easy implementation of generic circuits. The inheritance and polymorphism properties of OOP makes it possible for any generic circuit to accept any other circuit of the right structure, even itself, and build a bigger circuit, without the need to know what it actually does.

Another powerful tool discussed is the model-checker language SMV. Being able to convert SharpHDL descriptions into SMV allows the verification of circuit

properties. This tool was used to verify that the circuits of two FFT algorithms are equivalent. The mathematical derivation of these two algorithms — the Radix-2 FFT algorithm and the Radix-2^2 FFT algorithm — were explained and then used to build their circuits using SharpHDL. The results proved that the algorithms produce the same results.

References

1. Alan K. Karp *Bit Reversal On Uniprocessors,* Hewlett Packard Laboratories.
2. Communication and Multimedia Laboratory, Department of Computer Science and Information Engineering, NTU *Fast Fourier Transform (FFT),* http://www.cmlab.csie.ntu.edu.tw/cml/dsp/training/coding/transform/fft.html. Last viewed on January 11, 2005.
3. Christine Vella, *SharpHDL: A Hardware Description Language Embedded in C#,* University of Malta, Final Year Project. June 2004
4. Dag Björklund and Henrik Enqvist, *Parallel Radix-4 DIF FFT HArdware - Implementation using VHDL,* 24th October 2002.
5. Daniel C. Hyde, *CSCI 320 Computer Architecture Handbook on Verilog HDL,* Computer Science Department, Bucknell University, Lewisburg, PA 17837, August 23, 1997.
6. Gordon J. Pace, *Hardware Design Based on Verilog HDL,* Oxford University Computing Laboratory, Programming Research Group. Trinity Term 1998.
7. I. S. Uzun, A. Amira & A. Bouridane, *FPGA Implementations of FFT for real-time signal and image processing*, School of Computer Science, The Queen's University of Belfast, BT7 1NN Belfast, United Kingdom.
8. Jeff Ferguson, Brian Patterson, Jason Beres, Pierre Boutquin and Meeta Gupta, *C# Bible,* Wiley Publishing, Inc. ISBN 0-7645-4834-4, 2002
9. Koen Claessen, *Embedded Languages for Describing and Verifying Hardware,* Department of Computing Science, Chalmers University of Technology and Göteborg University, Sweden. April 2001.
10. Koen Claessen & Mary Sheeran, *A Tutorial on Lava: A Hardware Description and Verification System,* August 15, 2000.
11. K. McMillan, *Symbolic model checking: An approach to the state explosion problem,* Kluwer Academic Publishers, 1993.
12. Nicolas Halbwachs, *Synchronous Programming of Reactive Systems - A tutorial and Commented Bibliography,* Verimag, Grenoble - France. Partially supported by ESPRIT-LTR project "SYRF". Computer Aided Verification 1998
13. Nicolas Halbwachs, P.Caspi, P. Raymond, D.Pilaud *The Synchronous Dataflow Programming Language Lustre,* IMAG/LGI - Grenoble, Verimag - Grenoble. Partially supported by French Ministrere de la Recherche within contract "Informatique 88", and by PRC C^3 (CNRS).
14. N. Halbwachs and F. Lagnier and P. Raymond *Synchronous observers and the verification of reactive systems,* Third Int. Conf. on Algebraic Methodology and Software Technology, AMAST'93, M. Nivat and C. Rattray and T. Rus and G. Scollo, Workshops in Computing, Springer Verlag. June 1993.