

SharpHDL: A Hardware Description Language Embedded in C#

Christine Vella¹

Department of Computer Science and AI,
University of Malta

Abstract. 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. In this paper, we present such a language embedded in C#, enabling us to describe structurally large regular circuits in an intuitive way. These descriptions can then be automatically used in simulators and verification tools. We show the versatility of the approach by writing a hardware compiler for regular expressions in our language.

1 Introduction

Hardware Description Languages (HDLs) are programming languages used to describe a circuit behaviorally, structurally or both. They usually work hand-in-hand with other tools like simulators, which help a designer check that the description of a circuit works correctly. Although powerful HDLs exist, most of them lack to provide tools for verifying properties of circuits.

SharpHDL is an HDL that allows circuit verification by providing a means for model-checking *safety properties*. Safety properties can be defined in such a way that they state that some condition is always true no matter what the situation is [6]. Being an embedded language, SharpHDL is a meta language which allows a hardware designer to generate regular circuits. It is hosted in the C# language, an object-oriented programming language (OOL) which is easy to use and many documentation exist about it. The strong structure it is based upon makes SharpHDL easily extendible and helps the user produce neat circuit descriptions.

In a nutshell, SharpHDL is an elegant and simple structural HDL embedded in C# which incorporates various tools including the possibility of communicating with SMV² and Verilog³ applications. This paper gives a brief introduction to SharpHDL syntax and highlights the various concepts combined to develop this HDL.

2 SharpHDL

SharpHDL is an HDL embedded in C# so one can describe a circuit by writing a C# program. Presently, it consists of three libraries:

¹ This work was presented in June 2004 to the Board of Studies of Information Technology at the University of Malta as a Final Year Project for the B.Sc.(Hons.)I.T. degree programme, supervised by Dr. Gordon Pace.

² A standard model checker.

³ A standard HDL.

1. **SharpHDL**, which is the core library. It provides functions to control the structure of circuits as well as the necessary tools to build them, like wires and logic gates. Besides, it provides functions for a circuit designed in SharpHDL to be converted to two formats:
 - *Verilog* format which, given to a Verilog application, can be used to analyze and simulate a particular circuit, produce circuit layout etc.
 - *SMV* format which, given to the SMV application, checks a particular safety property for a wire in the circuit.
2. **LogicGates** library where composite logic gates, such as **Nand** and **Nor** amongst others are implemented;
3. **GenericCircuits** library, which caters for special structures that build a circuit in a regular format from a relatively simple gate.

2.1 Syntax Overview

As an introduction to basic SharpHDL syntax let us consider the construction of a half-adder circuit shown in figure 1.

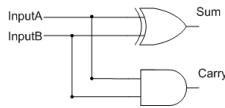


Fig. 1. A HALF-ADDER CIRCUIT

A half-adder consists of two input wires and it outputs two wires: **sum** and **carry**. The **sum** result is produced by applying a **Xor** operation over the two inputs; the **carry** result is produced by applying an **And** operation. Following is the code needed to build this circuit.

```
using System;
using SharpHDL;
using LogicGates;

namespace Arithmetic {

    //<1>
    public class HalfAdder : Logic
    {
        //<2>
        public static CellInterface[] cell_interface = {
            inPort("HALF_ADDER_A", 1),
            inPort("HALF_ADDER_B", 1),
            outPort("HALF_ADDER_SUM", 1),
            outPort("HALF_ADDER_CARRY", 1)
        };
        //<3>
        Xor xor;
        And and2;

        //<4>
        public HalfAdder()
        {
            xor = new Xor();
            and2 = new And();
        }

        //<5>
        public void gate_o(Wire inputA, Wire inputB,
            Wire sum, Wire carry)
    }
}
```

```

    {
        //<6>
        connect(this, "HALF_ADDER_A", inputA);
        connect(this, "HALF_ADDER_B", inputB);
        connect(this, "HALF_ADDER_SUM", sum);
        connect(this, "HALF_ADDER_CARRY", carry);

        //<7>
        xor.gate_o(inputA, inputB, sum);
        and2.gate_o(inputA, inputB, carry);
    }
}
}

```

The `HalfAdder` class is created by subclassing the SharpHDL `Logic` class < 1 >. The interface to the circuit, which consist of input and output ports are declared using the `CellInterface` mechanism < 2 >. Input ports are declared using the `inPort` method, which accepts a string for the name of the port and an integer for its width. In this case, both inputs are one-bit so the width of each port is 1. The output ports are declared using the method `outPort` which works the same as `inPort`. Next, is the declaration of the two instances representing the gates that are needed to build the half-adder: a `Xor` instance and an `And` instance < 3 >. Like the majority of C# classes, the constructor of the `HalfAdder` is declared so that it can be invoked by the designer needing a half-adder in his circuit < 4 >. Here we create a `Xor` and `And` instance by calling their respective constructors.

Finally, we need to define the structure of the circuit we are designing. We do this in a function accepting four wires: two representing the input wires and the other two representing the output wires. We call this function `gate_o`⁴ < 5 >. The first thing to do in this method is to connect the wires to the ports using method `connect` < 6 > which accepts three parameters: the gate the port belongs to, the name of the port to which the wire is going to be connected to and the wire variable itself. Finally, we define the structure of the circuit by calling the respective methods that structurally define the `Xor` and `And` gate < 7 >. The first line invokes the method `gate_o` belonging to the `Xor` class. This accepts three wires: the first two being the input wires and the third being the output wire. The two input wires to the half adder and the wire `sum` are passed to this method. The same is done for the `And` instance, but this time the two input wires and the `carry` wire is passed to its `gate_o` method.

It is important to note the call to the SharpHDL and `LogicGates` library at the beginning. This is necessary to be able to use the classes implemented in them. It is also a good idea to define a `gate` method which although does the same operations as the `gate_o` method described, it accepts three `Wire` objects and creates a new `Wire` object, calls the `gate_o` method and returns the newly created `Wire` object. The method looks as follows:

```

public Wire gate(Wire inputA, Wire inputB, Wire sum)
{
    Wire carry = new LogicWire();
    this.gate_o(inputA, inputB, sum, carry);
    return carry;
}

```

⁴ It is a convention of the SharpHDL libraries that every class representing a gate has a `gate_o` method which use the provided input and output wires to describe the particular circuit structurally. SharpHDL also provides the same method without the `_o` suffix. Such methods instantiate an output wire and returns it. Nevertheless they perform the same operation.

3 Generic Circuits

In this section we will discuss the various tools offered by the `GenericCircuits` library, which caters for *generic circuits*: circuits with a special regular format built from a relatively simple gate, and which we therefore refer to as *higher-order circuits* [6]. SharpHDL provides four different generic circuits:

1. **Map** — Sometimes, we want to perform a particular operation on each element of a bunch of signals. Instead of defining the operation for each signal, the generic circuit `Map` can be used. `Map` accepts a component having one input port and one output port, and a list of wires. It then constructs the circuit as shown in figure 2.

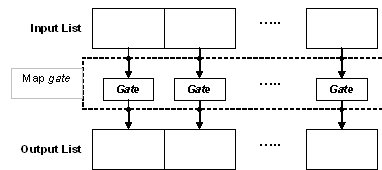


Fig. 2. MAP USING LOGIC COMPONENT GATE AND LIST OF WIRES INPUT.

SharpHDL also implements two other variations of `Map`. One variation accepts a two-input port gate whilst another variation of `Map` accepts a three-input port gate.

2. **Sequence** — As its name suggests, a `Sequence` performs an operation sequentially over a set of input wires. Therefore, it accepts a component having an equal number of input and output ports, and a set of wires which can be accepted by such a component. Figure 3 illustrates the general structure of a `Sequence` generic circuit.

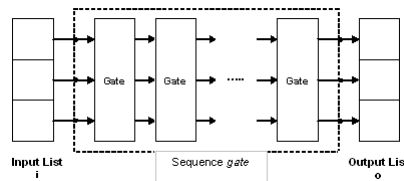


Fig. 3. A SEQUENCE USING COMPONENT GATE AND THE LIST OF WIRES I , REPEATED FOR n TIMES. THE OUTPUT IS THE LIST OF WIRES O .

3. **Row** — The `Row` Generic Circuit resembles the combination of the `Map` and `Sequence` structures. It accepts a gate having two input ports and two output ports. Besides, it accepts a separate wire and a list of wires. The circuit formed is illustrated in figure 4.

SharpHDL also implements two other variations of `Row`. One variation accepts a three-input port gate whilst another variation of `Row` accepts a four-input port gate.

4. **Tree** — There are circuits that can be built recursively, and an example of such is the `Tree` generic structure. The input to this structure is a logic component which takes two wires as inputs and outputs a single wire. It also accepts a list of wire inputs. Basically, the `Tree` structure builds a binary tree of the inputted component, as shown in Figure 5.

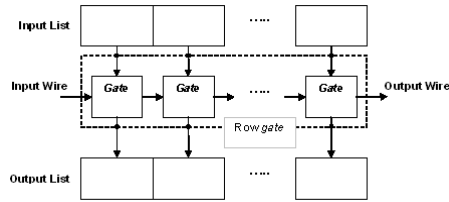


Fig. 4. A TYPICAL ROW GENERIC CIRCUIT STRUCTURE

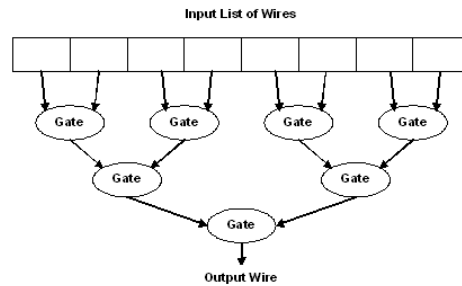


Fig. 5. A TREE STRUCTURE USING 8 INPUT WIRES.

Organizing a set of gates of the same type in a tree structure instead of leaving them in linear format, reduces the length of the circuit, such that applying an operator on n wires using a **Tree** structure will have path of approximately $\log_2 n$. On the other hand, if a circuit is generated linearly, the length of the longest path is $n - 1$.

The major benefit of these type of circuits is that they reduce the number of lines of code needed to construct a complex circuit, thus helping the user to generate more elegant code, making it more understandable and in some cases building more efficient circuits as we have seen when using the **Tree** structure. Generic circuits also increase code usability since a particular pattern can be used for several components without having to rewrite any code.

4 Hardware Compilation

SharpHDL is a structural language and therefore it provides tools for specifying a circuit by specifying the components to use and the connections between them. Another way of describing hardware is by specifying the behavior of a circuit using *synthesizable behavioral description languages*. These type of languages can be used to write a program and then transform the program to a circuit with the same behavior. Such a process is better known as *Hardware Compilation*. SharpHDL was used to embed a behavioral language which allows a user to build circuits describing Regular Expressions⁵.

4.1 Regular Expressions Compilation

The language consists of five regular expression constructs:

⁵ This example is totally taken from [5].

Empty String, which is a string whose length is zero.

Signal Input, which is the input signal provided by the programmer of the regular expression. It can either be an output from another existing circuit or an extra parameter to the definition of a particular regular expression.

Sequential Composition, which provides for concatenation of two expressions.

Iteration, which provides for repetition of an expression for zero or more times. Repeating an expression for zero times produces **Empty String**.

Non-Deterministic Choice, which provides for alternation.

4.2 Regular Expression Circuits

The circuits that are generated for these constructs after compilation have one input signal **start** and two output signals, **match** and **prefix** as shown in figure 6. The circuit starts sampling the signals when **start** is set to **True**. When the sequence of signals are part of the language being represented by the expression, **match** outputs **True**. The output **prefix** indicates whether the circuit is still active and the parsing of the regular expression has not yet failed. The different constructs compile to a circuit with this structure which implements the necessary operation.

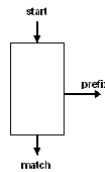


Fig. 6. THE GENERAL STRUCTURE OF A REGULAR EXPRESSION CIRCUIT

Implementationwise, each construct is represented by a class which implements a compilation method according to the circuit it has to produce. To build a regular expression circuit, we must call the classes representing each part of the expression and then call the **compile** method. So, for example, $(\epsilon + C)^* + (A.B)$ is expressed as:

```

RegularExpression example = new Choice(
    new Loop(
        new Choice(
            new EmptyString(),
            new SignalInput(new LogicWire("C"))
        )
    ),
    new SequentialComposition(
        new SignalInput(new LogicWire("A")),
        new SignalInput(new LogicWire("B"))
    )
);
\\Output wires: prefix and match
OutputSignals output = new OutputSignals();
example.compile(new LogicWire(), output);
  
```

5 Model Checking

SharpHDL can also be used in circuit verification by allowing the user to produce an SMV description of a SharpHDL circuit so that, given to an SMV system, he could verify *safety properties* of

the particular circuit. The safety property of a given circuit can be verified by using *synchronous observers*. These are circuits which take the inputs and outputs of the circuit which needs to be verified and decides whether at any time the stipulated property that the given circuit must hold is violated [7].

5.1 Case Study: Verifying Adder Implementations

There are many applications where safety properties can be used. One such application is to check that a particular implementation of a circuit is correctly built. This can be done if its output is compared to the output of a text-book example implementation of the circuit, given that both circuits are given the same input. An example is verifying that a *carry-select adder*, which is a special type of adder, is correctly implemented and this is done by comparing it to the classical *ripple-carry adder*. To verify that the former adder produces the correct result, the `sum` and `carry` wires of both adders are passed to an observer which compares the outputs and produces `True` if they are the same. The SMV definition of the whole circuit is given to an SMV application which verifies that for any input, the circuit outputs `True`, implying that the implementation is correct.

5.2 Another Case Study: Comparing Two Regular Expressions

In section 4 the Regular Expression Language was embedded in SharpHDL. A clear advantage of embedding languages is that the tools offered by the host language can be accessed by the embedded language without any difficulty. Therefore, circuits created using the Regular Expression Language can be verified. This can be illustrated by verifying that two expressions are equivalent.

An observing circuit *Comparer* can be constructed, which outputs `True` if the expressions are equal, given that both expressions are fed the same `start` signal. Figure 7 displays the framework of this circuit, which accepts two input expressions and outputs a signal wire. This circuit uses an `Equal` gate to check the `prefixes` of the two circuits, and another one to check their `matches`. The outputs from this gate are used as inputs to an `And` gate, since the equivalence property is followed iff both outputs are `True`.

Observing the output wire of this circuit will produce an SMV report, which can be fed to an SMV application for verification. So, given the described circuit, we can check the equivalence property of the two expressions $(ab+a)^*a$ and $a(ba+a)^*$.

Feeding these expressions to the `Comparer` object, we can observe and produce the equivalent SMV for the whole circuit. When this code is verified by an SMV application, it confirms that the output of the circuit is `True` for any situation, meaning that the expressions are equivalent.

6 Producing Verilog code using SharpHDL

The two standard HDLs are Verilog and VHDL. Both provide efficient tools for hardware description and simulation [2, 4]. SharpHDL allows the user to generate a Verilog description of a circuit in design.

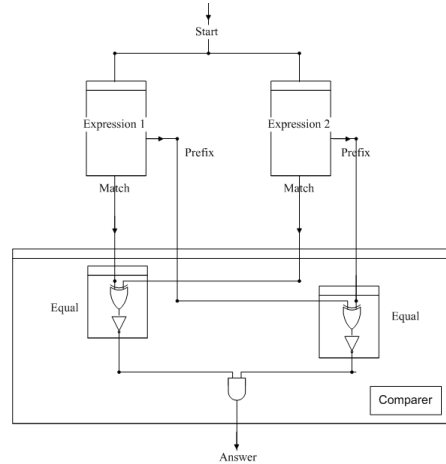


Fig. 7. CIRCUIT FOR COMPARING TWO EXPRESSIONS

A user may use the Verilog code for various reasons including that of simulating the circuit in design using a Verilog simulator. SharpHDL does not include its own simulator since well-tested simulators already exist. The Verilog code generation tool provided by SharpHDL gives the user the chance to use the wide range of tools which exist for Verilog and therefore it is still offering the necessary tools for simulation and analysis of circuit.

7 Related Works — Comparing SharpHDL

Besides Verilog and VHDL other powerful HDLs exist, amongst which there is Lava and JHDL, both very similar to SharpHDL. This section will briefly describe these two languages and compare them to SharpHDL.

7.1 Lava & SharpHDL

Lava [6] consists of an HDL embedded in the functional programming language Haskell. Although, like SharpHDL, it is not possible to describe circuits in such a depth as the standard HDLs, the descriptions are short and sweet. It also offers the facilities of *connection patterns*, which basically are the generic circuits found in SharpHDL.

Lava is very elegant and simple and circuit descriptions are made using Haskell modules. Besides, it provides tools to analyze and test circuits, including simulation and verification. Unlike SharpHDL, Lava links and controls all the tools and therefore the user sees only one language.

7.2 JHDL & SharpHDL

JHDL [1] is an HDL embedded in the object-oriented programming language Java. It consists of a set of necessary tools needed to design a circuit in an efficient and user-friendly manner.

Besides providing a set of classes which help a user build an electronic circuit, it allows the user to simulate and test a circuit in design. It also provides a rich CAD suite. Such tools enhance the user-friendliness of the language, a lacking characteristic in SharpHDL. Nevertheless, one must point out that given the strong structure SharpHDL is based on, these can be easily implemented.

Another important difference is that unlike SharpHDL, JHDL does not provide the possibility for circuit verification. On the other hand, SharpHDL does not implement a simulator of its own. Nevertheless, one must point out that a user can still simulate and analyze a circuit by generating Verilog code for a particular SharpHDL circuit description and input it to a Verilog tool.

8 Further Work and Enhancements

In the preceding sections, we have illustrated how circuits can be described and used in SharpHDL. Future work on SharpHDL will mainly focus on the following points:

- *Placement Extensions and Description of Other Non-functional Circuit Properties.* SharpHDL allows the structural description of any circuit, but it lacks to provide the user the ability to specify placement information. Although automatic placement tools exist, study has shown that user-specified placement information often improves the performance of Field Programmable Gate Arrays (FPGAs). Therefore, it is important that an HDL should include placement information which guides design tools to produce efficient designs. Providing explicit coordinate information for every component in a large circuit, nevertheless, can become very tedious. It is therefore ideal to provide the user with the ability to use relative placement information, with high-level descriptions like `Beside` and `Below`. These descriptions allow blocks to be placed relative to each other without the user providing the actual coordinates [8]. During the design phase, designers need to estimate other non-functional properties like area and timing. Since most of these properties are controlled by information found in wires, detailed information can be kept about wires. SharpHDL could be extended to accommodate such information [3].
- *Developing a Framework for Hardware Compilers.* In [5], a uniform framework was proposed within which behavioral languages can be developed, allowing them to be combined together for simulation, synthesis and verification. This is done by embedding the languages in Lava — a HDL embedded in Haskell.

This concept can also be applied in SharpHDL. We saw how a behavioral language that allows the user to build circuits describing regular expressions was embedded in this language. To allow more behavioral languages to be embedded and combined in this language, a framework similar to the one proposed in [5] can be constructed. Such a framework will impose the functional and non-functional rules, syntax and other methods that hardware compilers should implement.

9 Conclusions

Throughout this document, we have described how we managed to merge various concepts and form a strong hardware description language. We have explored the idea of embedding an HDL in an object-oriented programming language by not only embedding SharpHDL in C#, but we

also used SharpHDL to embed another language which allows the user to build circuits describing regular expressions.

To develop the latter language we used standard hardware compilation techniques, where the idea was to write a program in a language and then transform the program to a circuit with the same behavior such that hardware is directly generated from a program.

SharpHDL provides other tools than just a set of classes to design simple circuits. It also includes a special library that constructs different complex circuits with a particular structure, called generic circuits. This library is useful especially to write elegant and short descriptions of large circuits.

An important tool provided by SharpHDL is that of generating input to a model checking application, which could allow verification of safety properties of a circuit. One useful application of this tool is to verify implementations of circuits. It was also used in conjunction with Regular Expression circuits and we were able to check the equivalence of two expressions.

Although SharpHDL does not implement simulation and analysis tools, it provides the user the tool to generate Verilog descriptions which allows the user to make use of the standard well-tested Verilog tools. At the end we managed to compare SharpHDL with Lava and JHDL and we saw that it compares well with such strong HDLs.

The main motivation was to explore an HDL in an OOL that is elegant and simple. By combining various concepts together we managed to develop a strongly-based HDL. We will be exploring the extension of the language for more advanced techniques in the future.

References

1. Brad Hutchings, Peter Bellows, Joseph Hawkins, Scott Hemmert, Brent Nelson, Mike Rytting, *A CAD Suite for High-Performance FPGA Design (1999)*, IEEE Symposium on FPGAs for Custom Computing Machines 1999.
2. Daniel C. Hyde, *CSCI 320 Computer Architecture Handbook on Verilog HDL*, Computer Science Department, Bucknell University, Lewisburg, PA 17837, August 23, 1997.
3. Emil Axelsson, Keon Claessen & Mary Sheeran, *Wired - A Language for Describing Non-Functional Properties of Digital Circuits*, Chalmers University of Technology. Designing Correct Circuits 2004, Barcelona. March 2004.
4. Gordon J. Pace, *Hardware Design Based on Verilog HDL*, Oxford University Computing Laboratory, Programming Research Group. Trinity Term 1998.
5. Koen Claessen & Gordon Pace, *An Embedded Language Framework for Hardware Compilation*, European Research Consortium in Informatics and Mathematics. Grenoble, France, 2002.
6. Koen Claessen & Mary Sheeran, *A Tutorial on Lava: A Hardware Description and Verification System*, August 15, 2000.
7. 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
8. Steve McKeever, Wayne Luk & Arran Derbyshire, *Towards Verifying Parameterised Hardware Libraries with Relative Placement Information*, Department of Computing, Imperial College, 180 Queen’s Gate, London UK. Proceedings of the 36th Hawaii International Conference on System Sciences (HICSS’03).