

Object-Oriented Generation of VHDL Synthesizable Architectural Building Blocks

George S. Powley Jr.
powley@ee.eng.ohio-state.edu

Dr. Joanne E. DeGroat
degroat@ee.eng.ohio-state.edu

Department of Electrical Engineering
The Ohio State University
2015 Neil Avenue
Columbus, OH 43210-1272

Abstract

To allow the rapid prototyping of application specific signal processors, an extensive VHDL library of architectural building blocks is needed. In this paper, we describe an object-oriented approach to generating a library of VHDL synthesizable models. A hierarchical classification of digital component attributes is described, with examples for integer adders. A model generation methodology, using 'C' preprocessor directives, is also described. We conclude with an example of the VHDL code generated for a 4-bit ripple carry adder, and its synthesized schematic.

1. Introduction

To allow the rapid prototyping of application specific signal processors, an extensive VHDL library of digital architectural building blocks [1] is needed. The digital component library should include behavioral models of off-the-shelf parts, as well as VHDL synthesizable models. This library could be used by hardware designers to develop behavioral models of signal processing systems, and also by high level synthesis systems to verify behavioral synthesis results [2]. In this paper, an object-oriented approach to classifying and generating the VHDL synthesizable models of digital building blocks is described.

The inheritance concept from the object-oriented paradigm [3] is used to develop a hierarchical

description of all digital components. The hierarchical description is used to develop a database of digital component attributes, and these attributes are incorporated into synthesizable VHDL models for each component. The VHDL models are written using 'C' preprocessor directives [4] to reflect unique functionality for different attribute values. A graphical user interface for the attribute database has been developed using TCL/Tk [5]. The interface provides an easy method to maintain and upgrade the component attribute database. Another graphical user interface has been developed to allow the designer to bind attribute values to a component instance. The attribute values from the user interface and the preprocessed VHDL model are passed to the 'C' preprocessor, which generates the component instance model. The component instance model is targeted for synthesis, but it can be used for behavioral simulations as well.

2. Component Classification

The inheritance feature of the object-oriented paradigm provides an excellent method for describing the attributes of architectural building blocks. By organizing digital components in a hierarchical fashion, common attributes can be identified and placed in a higher level class.

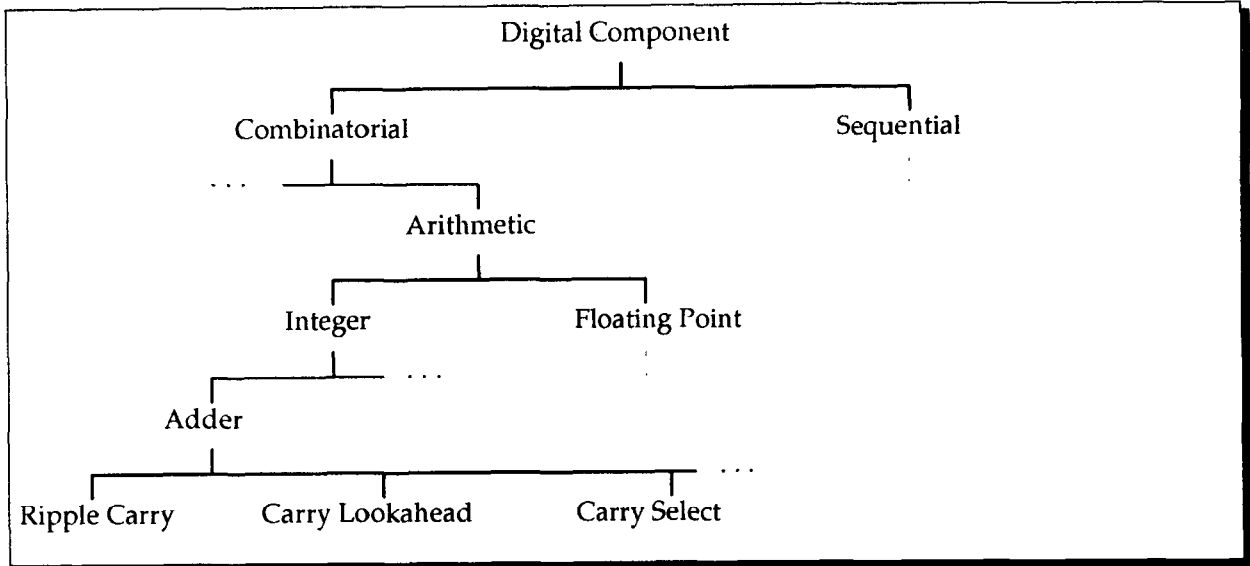


Figure 1: Component class hierarchy

Moving attributes to a higher level in the hierarchy reduces the amount of effort required to add new components to the description, since the new component will only need to add unique attributes.

We have started an effort to develop a hierarchical description of all basic architectural building blocks. Currently, we are concentrating on determining the attributes for integer adders, so that we can develop a collection of VHDL synthesizable adder models.

Figure 1 shows the hierarchy for the adder classification. Each level of the hierarchy can specify zero or more attributes that will be inherited by descendants of that class. The attributes identified for the adder class are shown in Figure 2. The `NumInputs` and `NumOutputs` attributes are used with the `WordLength` attribute to specify the data I/O for a component. Although specifying the word length at this level of the hierarchy may seem confining, this is the most generic representation, since different word lengths do not make sense for some components (i.e. bit-wise logic gates). Cases where the word length differs for a data input or output can be handled by classes at a lower level of the hierarchy. The `Enable` attribute gives the option of using an enable signal for any digital component.

```

DigitalComponent {
  NumInputs : integer;
  NumOutputs : integer;
  WordLength : integer;
  Enable : Boolean; }

Integer {
  Zero : Boolean;
  Overflow : Boolean;
  Negative : Boolean; }

Adder {
  DigitalComponent::NumInputs = 2;
  DigitalComponent::NumOutputs = 1;
  Cin : Boolean;
  Cout : Boolean;
  Add : Boolean;
  Sub : Boolean; }
  
```

Figure 2: Class attributes

The next level of the hierarchy that contains attributes is the `Integer` class. The `Zero`, `Overflow`, and `Negative` attributes are used to determine if optional integer arithmetic flags should be included.

Continuing down the hierarchy, the `Adder` class contains the attributes `CIN` and `COUT`. These attributes are used to determine if carry in and carry out signals should be included in the VHDL

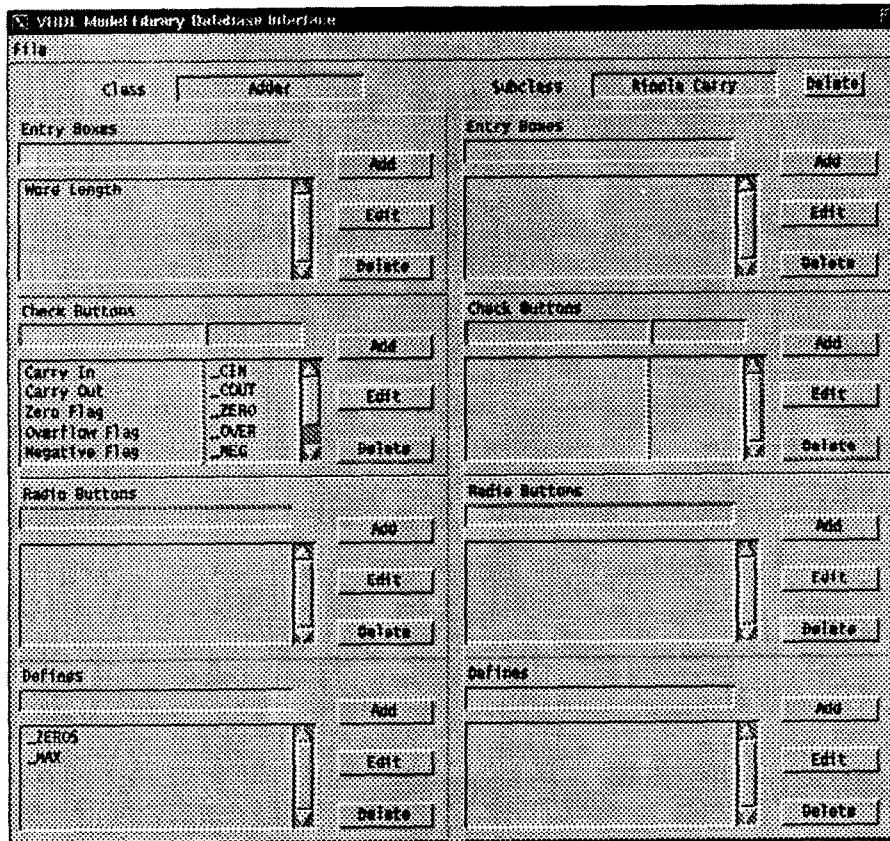


Figure 3: Attribute database Interface

model. The Add and Sub attributes are used to determine which operations should be supported by the adder. A component can be generated that will add only, subtract only, or perform either add or subtract. If the component is specified to perform both adds and subtracts, the model will specify the operation selection scheme.

In order to simplify the effort to generate VHDL models, classes at a lower levels in the hierarchy may restrict the attribute values of an ancestor class. For example, the adder class will set NumInputs to two and NumOutputs to one. This will reduce the amount of code required to generate an adder model, since no error checking code will be required for these attributes.

3. Attribute Database

The attribute database is used to store the attributes identified by the digital component

classification. Therefore, the database holds the attribute representation, not the attribute values. The database is designed to hold attributes from the lowest two levels of the component hierarchy, referenced as the class and subclass. This greatly reduces the complexity of the database and the model generation interface. In this section, we describe the data types used to represent the attribute information. We also describe a graphical user interface designed to create and maintain the attribute database.

3.1. Attribute Types

While identifying the attributes of the digital components, it was determined that the attributes could be represented by three types: integers, Booleans, and enumerated types. Integers are used to represent attributes that take on a numeric value, such as the word length of an adder. Boolean values are used to determine the

```

BEGIN
  all_bits : FOR i IN 0 TO _MAXOUT GENERATE
    #ifndef _CIN
      lsb : IF (i = 0) GENERATE
        carry(i+1) <= _IN1(i) AND _IN2(i);
        sum(i) <= _IN1(i) XOR _IN2(i);
      END GENERATE lsb;
    #else
      lsb : IF (i = 0) GENERATE
        carry(i+1) <= (_IN1(i) AND _IN2(i)) OR (_IN1(i) AND Cin) OR
          (_IN2(i) AND Cin);
        sum(i) <= _IN1(i) XOR _IN2(i) XOR Cin;
      END GENERATE lsb;
    #endif
  ...

```

Figure 4: Preprocessed VHDL code segment

presence or absence of some attribute, such as the carry out signal from an adder. Enumerated types are used to set an attribute to one choice from a predefined set of values. An example of an enumerated type would be the lookahead length in a carry lookahead adder. Even though the lookahead length could be represented by an integer, the VHDL synthesizable model will only support a predefined ranges of values. By using an enumerated type, the need for error checking code in the model generation interface is eliminated.

3.2. Database Interface

A database interface has been developed to aid the creation and maintenance of the attribute database. The interface, shown in Figure 3, gives the model designer an easy method to add new classes and subclasses to the database, or to change the attributes of a class already in the database. The attributes of all classes and subclasses are divided into four categories: entry boxes, check boxes, radio buttons, and define scripts.

The entry boxes are used to hold integer attributes. The integers are not passed directly to the model generator, but they can be used in define scripts, described below.

Check buttons are used to pass information directly to the model generator. For each check button, a `#define` directive corresponding to the specified define name is passed to the model generator, if the check button is selected in the model generation interface. Check buttons may also be used in define scripts.

Radio buttons are also used to pass information directly to the model generator. For each radio button, a `#define` directive corresponding to the specified define name is passed to the model generator. Radio buttons also may be used in define scripts.

Define scripts are Tcl scripts used to pass complex information to the model generator. The define script may use information from entry boxes, check boxes, and radio buttons. For each define script, a `#define` directive corresponding to the specified define name is passed to the model generator. The value of the define is the result of executing the Tcl script. The define script provides the model designer with a very powerful method of passing information to the model generator, thereby reducing the complexity of the VHDL models.

4. Model Generation

Model generation is required in order to bind attribute values to a generic component model.

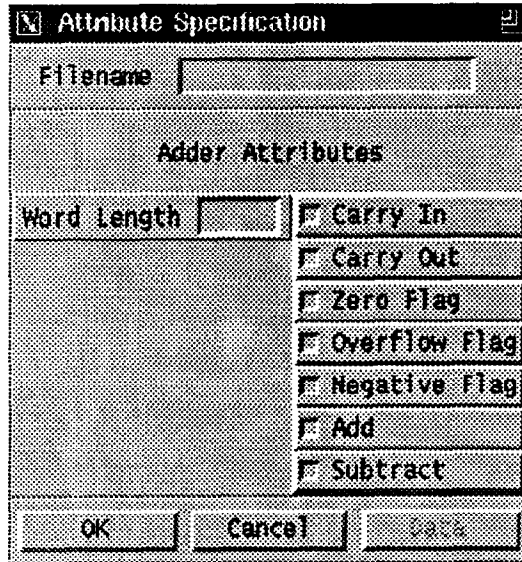


Figure 5: Model generation Interface

By fixing the attribute values in the VHDL code, the models should be portable across all synthesis tools. A model generation interface has been developed, which allows the user to specify attributes for a component model. After the attributes are specified, the 'C' preprocessor is used to substitute the specified attributes into the generic VHDL model. In this section, we will discuss the format of the preprocessed VHDL code, and describe the model generation interface.

4.1. Preprocessed VHDL Format

The most straightforward method to write VHDL synthesizable code for varying word sizes, is to use loops with the length defined by a variable or a generic. Unfortunately, most VHDL synthesis tools require constant values where the variable values would be used, and some tools do not support generic values. In order to keep our VHDL models portable across synthesis systems, we use a model generation methodology that allows us to bind variable values to a fixed value at model generation time, using 'C' preprocessor directives. The #define directive is used to replace "generic" values with integers or constants, thereby meeting the requirements of synthesis tools. This methodology also provides more powerful features when used in conjunction with the define scripts described in Section 3.2. In the adder models, a define script is used to

produce a binary string of zeros of length WordLength. This value is compared to the output of the adder, and the result is reported on the Zero signal. Although there are other methods to achieve the same result, this method yields straightforward VHDL code, as will be shown in Section 5.

4.2. Model Generation Interface

The model generation interface is used to bind attribute values to a VHDL model. The model generation interface presents the designer with a list of component subclasses for a chosen class. When the component subclass is selected, a dialog box containing the attributes for that class and subclass is opened, as shown in Figure 5. After the attributes are designated, the model generator sends a file of define directives together with the preprocessed VHDL model to the 'C' preprocessor. The output of the 'C' preprocessor is a VHDL synthesizable model of the desired component.

The attribute dialog box used by the model generation interface is created based on the values in the attribute database. Employing the concept of creating a dialog box from entries in a dynamic database is a very powerful technique. Now that the code to produce the dialog box is completed, new components can be added to the database

```

ENTITY adder IS
  PORT( Input1 : IN bit_vector(3 DOWNTO 0);
        Input2 : IN bit_vector(3 DOWNTO 0);
        Cin : IN bit;
        Cout : OUT bit;
        Zero : OUT bit;
        Output : OUT bit_vector(3 DOWNTO 0) );
END adder;

ARCHITECTURE ripple_carry OF adder IS
  SIGNAL sum : bit_vector(3 DOWNTO 0);
  SIGNAL carry : bit_vector(3+1 DOWNTO 1);
BEGIN
  all_bits : FOR i IN 0 TO 3 GENERATE
    lsb : IF (i = 0) GENERATE
      carry(i+1) <= (Input1(i) AND Input2(i)) OR (Input1(i) AND Cin) OR
        (Input2(i) AND Cin);
      sum(i) <= Input1(i) XOR Input2(i) XOR Cin;
    END GENERATE lsb;

    rest : IF (i > 0) GENERATE
      carry(i+1) <= (Input1(i) AND Input2(i)) OR (Input1(i) AND carry(i))
        OR (Input2(i) AND carry(i));
      sum(i) <= Input1(i) XOR Input2(i) XOR carry(i);
    END GENERATE;
  END GENERATE all_bits;

  Output <= sum;
  Cout <= carry(3+1);
  Zero <= '1' WHEN sum = "0000" ELSE '0';
END ripple_carry;

```

Figure 6: VHDL code generated for 4-bit ripple carry adder

without changing the model generation interface code.

5. Example

In this section, we will show the results of generating and synthesizing a 4-bit ripple carry adder. The first step of the process is to select the attributes for the adder. The attributes shown in Figure 5 will be used for this example. The VHDL code produced by the model generator is shown in Figure 6. This is a behavioral VHDL model of a 4-bit adder, which is targeted for logic synthesis. Synthesizing the model using the Compass ASIC Compiler generates the schematic shown in Figure 7.

6. Conclusions

In this paper, we have described an object-oriented approach to generating VHDL synthesizable models of architectural building blocks. To achieve this goal, a hierarchical classification of all digital component attributes has been started, with the current effort concentrated on integer adders. An attribute database has been designed to store the attribute definitions, and a database interface has been developed to aid the creation and maintenance of the database. The 'C' preprocessor is used to generate the final VHDL model, by binding the component attributes to a "generic" VHDL model of the component. This method proves to be powerful, since it allows the models to be portable across synthesis tools.

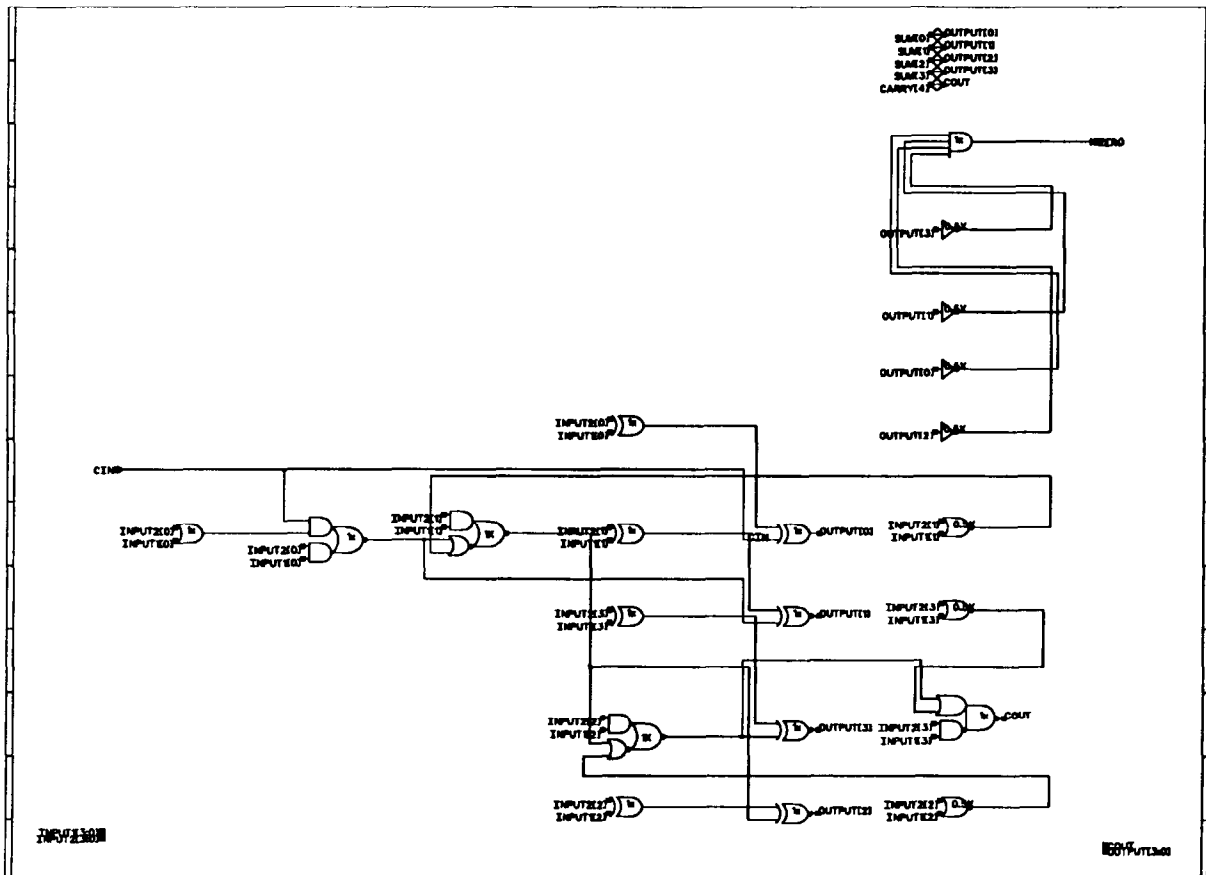


Figure 7: Schematic of synthesized 4-bit ripple carry adder

Further research will involve writing and testing models for integer adders, shifters, and multipliers. Models for floating point adders and multipliers will be developed when the integer models are completed.

Acknowledgements

This work has been funded in part by U.S. Air Force Contract F33615-94-C-1496.

References

[1] Hayes, J. P., *Computer Architecture and Organization*, McGraw-Hill, 1988.

- [2] Dutt, N.D., "GENUS: A Generic Component Library for High Level Design", *Technical Report #88-22*, University of California at Irvine, Sept 1988.
- [3] Korsen, T. and McGregor, J. D., "Understanding Object-Oriented: A Unifying Paradigm", *Comm of the ACM* 33, 9 (Sept 1990), pp. 40-60.
- [4] Kernighan, B. W. and Ritchie, D. M., *The C Programming Language*, Prentice Hall, 1988.
- [5] Ousterhout, J. K., *Tcl and the Tk Toolkit*, Addison-Wesley, 1994.