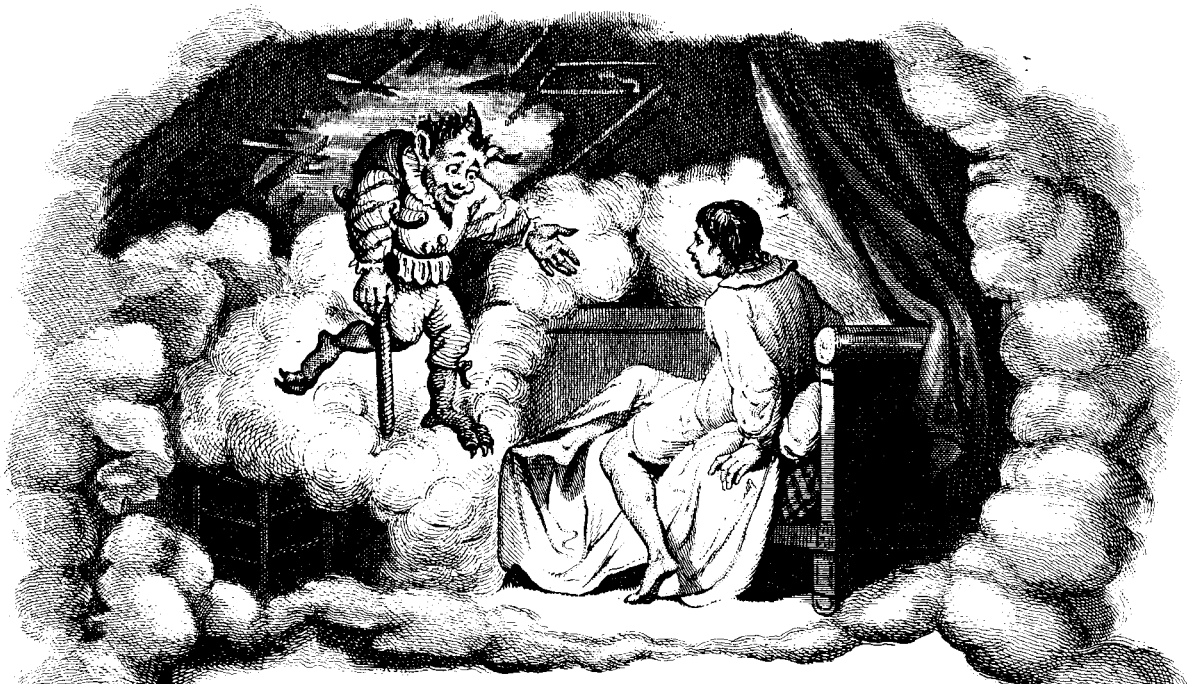

THE FHDL MACRO PROCESSOR

PETER M. MAURER

Technical Report DA-18, 1990, VCAPP Laboratory
Dept. of Computer Sci. & Eng.
University of South Florida, Tampa, Florida 33620

THE FHDL MACRO PROCESSOR

Peter M. Maurer
Department of Computer Science and Engineering
University of South Florida
Tampa, FL 33620

ABSTRACT

The FHDL (Florida Hardware Design Language) Macro processor provides a mechanism for extending the language features provided by the other components of the FHDL system (the ROM language, the PLA language, and the logic specification language). The primary use of the Macro processor is to provide flexible cells, such as ripple-carry adders, that can expand to match the size of the interface. The use of the Macro processor for this purpose is transparent with more standard hierarchical specification mechanisms. In addition, the Macro processor was designed to be an implementation vehicle for more sophisticated hardware specification and synthesis systems. The Macro processor provides most of the features found in other macro languages, and provides several new features that are found in few, if any, existing macro languages. The use of the Macro processor for high-level synthesis is the subject of much on-going research.

THE FHDL MACRO PROCESSOR

Peter M. Maurer
Department of Computer Science and Engineering
University of South Florida
Tampa, FL 33620

1. Overview

Macro-instructions have long been used to expand and extend the power of various programming languages[1,2,3]. Although the most extensive use of macro instructions has been in assembly languages, some high-level languages such as C also make extensive use of macros. In general, a macro is an instruction, a statement, or an expression that is expanded at compile time into a more complex expression or set of statements. Macros are specified in a language that is separate, but usually similar to the language that is being extended. The languages used to specify macro instructions usually provide features that are similar to existing high-level languages. It is generally possible to specify "if-then-else" structures, and usually possible to specify loops and variables. Sometimes data structures, function calls, and recursion are also provided. It is important to keep in mind that these are *compile time* features that are independent of the execution-time behavior of the program.

In addition to their ability to expand and extend programming languages, macro languages can also be used to simplify repetitive programming tasks and make source code more readable. On the other hand, macro languages sometimes present conceptual difficulties to programmers who are not used to them. Differentiating between statements that are executed at compile time and statements that are executed at run time tends to be difficult for programmers who are not used to thinking in those terms. For this reason, macros tend to be written by experts and encapsulated in libraries of macro definitions. It is precisely this form of usage that makes macro languages appealing for hardware design.

Hardware designers are already used to thinking in terms of libraries of objects that can be combined to create a more complex object. In particular, many microelectronic circuits are constructed using libraries of *standard cells*[4]. A standard cell is a predefined structure that defines some well known object such as a 2-input NAND gate. A collection of standard cells can be pasted together and wired to perform some arbitrarily complex function. The concept of standard cells can also be used at higher levels to combine simulation models of various objects into a larger design. To illustrate this, consider the

problem of constructing a four-bit ripple-carry adder out of full adders. A full adder computes the sum of three bits, and produces a "sum" and "carry" output. Figure 1 illustrates the block diagram of the four-bit adder, while Figure 2 illustrates the FHDL (Florida Hardware Design Language) description of the circuit.

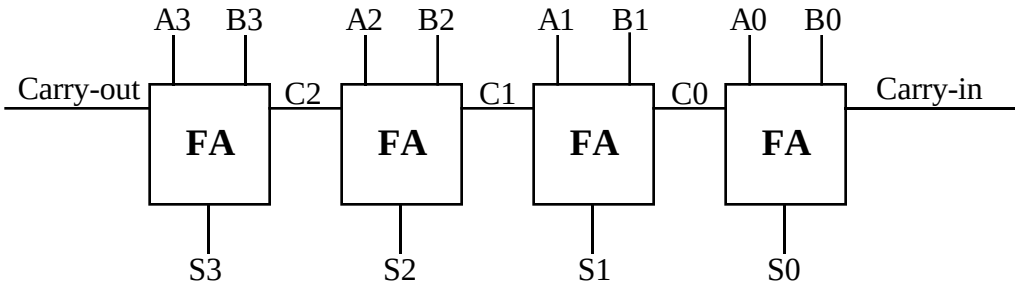

Figure 1. The Block Diagram of a Ripple-Carry Adder.

```
RIPPLE-ADDER:    circuit
                 inputs     A0,A1,A2,A3,B0,B1,B2,B3,Carry-in
                 outputs    S0,S1,S2,S3,Carry-Out
FA1:             FA         (A0,B0,Carry-in),(S0,C0)
FA2:             FA         (A1,B1,C0),(S1,C1)
FA3:             FA         (A2,B2,C1),(S2,C2)
FA3:             FA         (A3,B3,C2),(S2,Carry-Out)
                 endcircuit
```

Figure 2. The FHDL Description of a Ripple-Carry Adder.

In Figure 2, the full adders are named FA1 through FA4. The "FA" keyword defines the item as a full adder. Each full adder has two operands, the first of which is a list of input signals, and the second of which is a list of output signals. Connections between full adders are implied by signal-name usage. The "inputs" and "outputs" statements are used to identify signals that originate from and propagate to logic that is external to the circuit.

Once the FHDL description of the four-bit adder is completed, the adder can be used as a cell in other circuits by using a statement similar to the following, which is called a *subcircuit call*.

```
RIPPLE-ADDER    (X0,X1,X2,X3,Y0,Y1,Y2,Y3,CI),(T0,T1,T2,T3,CO)
```

When the FHDL circuit parser encounters a statement such as this, it replaces it with a copy of the subcircuit. Input and output signal names defined in the subcircuit are replaced by the names used in the subcircuit call, and other names (such as C0 in Figure 2) are

replaced by unique names to prevent unwanted "shorts" in case there is more than one subcircuit call to the same subcircuit.

Most hardware design languages provide a subcircuit facility similar to that provided by the FHDL circuit parser[5,6]. In a sense, this provides a macro-like facility, because a single statement can be expanded into a collection of statements. However this facility is strictly limited. If it is necessary to define both a four-bit adder and a six-bit adder, two separate subcircuits must be defined. The subcircuit facility does not have the power to expand or shrink to fit the user's needs. The FHDL Macro language was designed to rectify this problem. (Although macro facilities are provided by other hardware design languages, the FHDL Macro processor provides a much more extensive set of features than is typically provided in by such a facility. In particular, the output structuring facility described in Section 5, and the expression handling features described in Section 6 are, as far as I know, unique to FHDL.)

The FHDL Macro language was designed to provide "dynamic" cells that could adjust their size, function, and interface to meet the needs of the user. In addition, the macro language was designed to be used at many different levels of the design, from the RTL level down to the layout level. However, the FHDL Macro processor was envisioned to be more than simply a replacement for the subcircuit facility already present in the FHDL circuit parser. In addition to providing flexible cells, the Macro processor was designed to be an implementation vehicle for high-level synthesis systems. An example of such a system is a microprocessor synthesis system that accepts a set of parameters describing instruction formats and RTL descriptions of the instructions, and synthesizes a microprocessor capable of executing these instructions. The statements of such a high-level synthesis system can be defined as FHDL Macros which automatically generate the low-level logic to implement the required operations. Figure 3 gives an example of such a system.

```
A:      REGISTER            width=32
B:      REGISTER            width=32
R:      REGISTER            width=32
ACC:    REGISTER            width=32
PSW:    REGISTER            width=32
        INSTRUCTION_FORMAT  opcode=8,operand=(address,24,direct)
ADD:    BEGIN_INSTRUCTION
        OPCODE              0x01
        RTL                 MEM(OPERAND[1])->A
        RTL                 ACC->B
        RTL                 A+B->R,OVFL->PSW[0],CARRY->PSW[1]
        RTL                 R->ACC
        END_INSTRUCTION
```

Figure 3. An Example of a Complex System.

Features to support systems such as that illustrated in Figure 3 are the subject of much on-going research on the FHDL Macro processor.

2. The Language.

To put the FHDL Macro processor into perspective, it is necessary to show its relationship to the rest of the FHDL tools. The basic framework of the FHDL system is pictured in Figure 4.

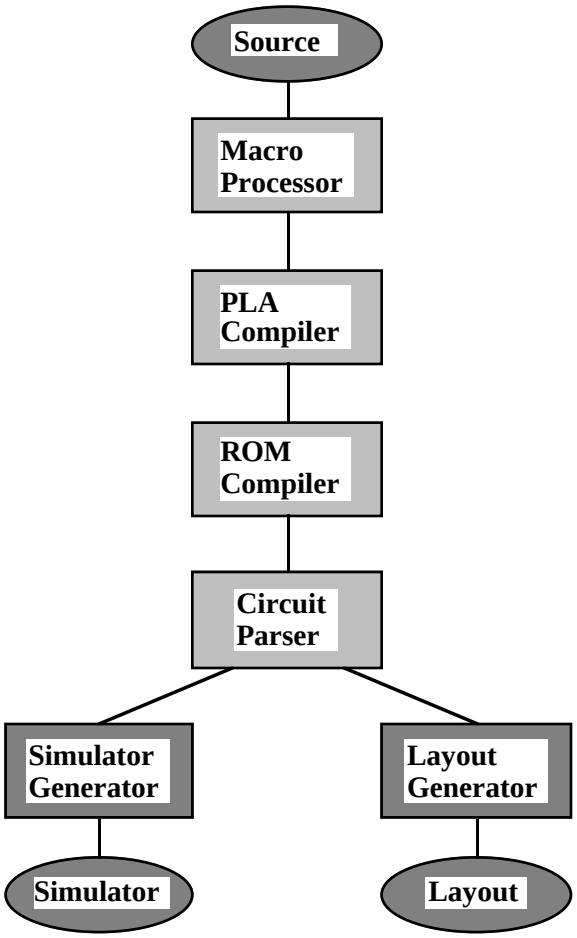

Figure 4. The Structure of the FHDL System.

As illustrated in Figure 4, the FHDL system provides a set of interrelated languages that can be used either to simulate integrated circuits, or to lay them out automatically. The specific languages (the ROM language, the PLA language, and the gate-level language) are all based on a "name/command/operands" syntax that resembles a typical assembly language. This syntax was chosen because of its simplicity and its adaptability. Even though individual statements resemble assembly language statements, it is quite easy to incorporate high-level-language features into this syntax. Providing all languages with a common syntax allows the macro processor to be used with any or all of the languages. Examples of the three different types of FHDL statements are illustrated in Figure 5.

```
(Logic)
Gate1:      AND                 (in1,in2,in3),out1
Mux1:       MULTIPLEXER         (in1,in2,in3,in4,control),out1

(Rom Code)
L1:         jump                OVERFLOW

(PLA Expressions)
(a&b)|(c&d):        word        timeout
```

Figure 5. Samples of FHDL statements.

The FHDL Macro processor provides many of the features of a high level language, including variables, loops, conditionals, recursive functions, and data structures. Furthermore, the Macro processor provides several special features to simplify the process of generating circuit descriptions. To illustrate the basic features of the macro processor, consider the problem of replacing the subcircuit of Figure 2 with a flexible cell that will expand to meet the user's needs. The definition of such a cell is illustrated in Figure 6.

```
RIPPLE-ADDER:  'macro
    'int     'i,'incount
    'str     'CI,'CO
    'incount<-('inputs-1)/2
    'CI<-'ilist('inputs-1)
    'CO<-'olist('outputs-1)
    'for     'i<-0,'i<'incount,'i<-'i+1
             'if      'i=0
                      FA#'i: FA     ('ilist('i),'ilist('i+'incount),'CI),('olist('i),C#'i)
             'elif    'i='incount-1
                      FA#'i: FA     ('ilist('i),'ilist('i+'incount),C#('i-1)),('olist('i),'CO)
             'else
                      FA#'i: FA     ('ilist('i),'ilist('i+'incount),C#('i-1)),('olist('i),C#'i)
             'endif
    'endfor
RIPPLE-ADDER:  'endmacro
```

Figure 6. A Ripple-Carry Macro.

Although the macro of Figure 6 may appear cryptic, the principles governing is construction are quite simple. Every macro keyword and variable begins with an apostrophe to distinguish it from ordinary text. The operator "<-" is used for assignments, and the operator "#" is used for concatenation. The built-in function 'inputs returns the number of input arguments on the macro-call statement, while the function 'ilist is used to access the individual input arguments (the *n* input arguments are numbered from 0 to *n*-1). The function 'olist provides access to the individual output arguments. The 'for statement is similar to the "for" statement of the C language. The first expression is used for

initialization, the second is a test that causes the loop to terminate when false, and the third is executed after each iteration to increment the loop counter. The 'if and 'else statements work as expected, while the 'elif statement is shorthand for "else if." Every 'for statement must be terminated by an 'endfor statement, and every 'if statement must be terminated by an 'endif statement. There are two types of variables, integers declared with an 'int statement, and strings declared with a 'str statement.

The function of the macro illustrated in Figure 6 should now be clear. The macro assumes that the call statement contains $2n+1$ input arguments, and $n+1$ output arguments. (In practice, error checking and reporting would be done to verify this.) The macro first assigns the value of *n* to the variable 'incount and then assigns the name of the carry-in and carry-out signals to 'CI and 'CO respectively. The 'for loop is used to generate the individual stages of the adder. The first stage ('i=0) uses the carry-in signal instead of the carry from the previous stage. The last stage ('i='incount-1) generates the carry-out signal rather than an intermediate carry. All other stages generate an intermediate carry and use the carry from the previous stage.

In addition to the lack of error checking, the macro of Figure 4 has one other problem. The names generated for the full adders and for the intermediate carries will not be unique if the macro is used more than once in a single circuit. To get around this problem, the macro processor provides the built-in function 'calln which is guaranteed to return a unique value for each macro call. This function can be concatenated with the generated names to make them unique.

Once this macro has been defined and placed in a macro library, the circuit designer can use it just as if it were an ordinary subcircuit definition. Figure 7 gives examples of macro calls that generate different sized adders.

```
ADD4:   RIPPLE-ADDER   (X1,X2,X3,X4,Y1,Y2,Y3,Y4,CI),(Z1,Z2,Z3,Z4,CO)
ADD3:   RIPPLE-ADDER   (P0,P1,P2,Q0,Q1,Q2,CI2),(K1,K2,K3,CO2)
```

Figure 7. Examples of Macro Calls.

As Figure 7 illustrates, macro calls are identical to subcircuit calls (which are in turn identical to gate declarations). This transparency of syntax allows the circuit designer to use objects without regard to whether they are gates, subcircuits, or macros.

3. Function Calls.

As in any macro language, one macro may call another. For example, the "FA" statements generated by the macro of Figure 6 may themselves be calls to a macro that

generates the logic of the full adder. Although this is, in a sense, a subroutine facility, it is sometimes necessary to construct a function that returns a value and can be used in an expression. The FHDL macro processor allows any macro to be treated as a function. To use this facility, the macro programmer codes expressions of the form "macro_name(arguments)." The macro definition returns a value by assigning a value to its own name. When a macro is used as a function, it may be recursive, as Figure 8 illustrates.

```
Factorial:    'macro
      'if     '0<=1
              'Factorial<-1
      'else
              'Factorial<-Factorial('x-1)*'x
      'endif
Factorial:    'endmacro
```

Figure 8. A Recursive Function Macro.

Figure 8 introduces the new built-in function '0 Functions of the form '<number> are used to access the arguments of function macros. (These functions can also be used with ordinary calls. For the macro of Figure 6, '0 represents the entire input list, while '1 represents the entire output list.) Note the difference between the return-value variable 'Factorial and the function name "Factorial."

Function macros normally do not generate output other than the returned value, but in other respects they are identical to ordinary macros.

4. Systems of Macros.

Although the primary use of the Macro processor is to create dynamic cells, it can also be used to construct more general synthesis systems, such as that illustrated in Figure 3. Such systems generally require coordinated interaction between several different macros. For example, in the system pictured in Figure 3 the output generated by the RTL macros may be different depending on the declared width of the operand registers.

Global variables are the primary mechanism for coordinating the actions of several macros. When a macro declares a variable to be global, the variable is shared between all macros that declare the same global variable. (The declarations can be placed in an "include" file to reduce coding.) Furthermore, the value of a global variable persists from one macro call to the next. To illustrate the use of global variables, consider the macro of Figure 9, which should not be called more than once.

```
Once:   'macro
        'gblint    'Once_has_been_called
        'if        'Once_has_been_called=1
                   'error     severe,"Once is called more than once"
                   'exit
        'endif
        'Once_has_been_called<-1
        ...
Once:   'endmacro
```

Figure 9. A Macro That Can Be Called Only Once

The macro of Figure 9 takes advantage of the fact that global variables are initialized to zero. If the value is equal to 1, the macro has already been called, so a severe error is reported using the 'error statement, and processing of the macro is terminated using the 'exit statement.

By default, global integers are initialized to zero while global strings are initialized to the null string. For some macro systems it may be desirable to use different initialization values. Furthermore, after all input has been processed it may be necessary to examine the state of various global variables to determine whether the proper macros have been called and whether a consistent set of options has been chosen. One mechanism for doing this is to force the user to code "START" and "END" macro calls at the beginning and the end of all specifications. However, the Macro processor provides an invisible (and unavoidable) mechanism for doing the same thing.

To use this mechanism, it is first necessary to place all macros for the system into a common macro library. If the library contains a macro named "$START" it will be processed before any other input, and if the library contains a macro named "$END" it will be processed after all other input. These two macro names are illegal except in a macro library, which prevents the user from overriding them. These two macros provide a convenient place to put initialization and cleanup operations. Since the user has no control over the processing of these macros, the initialization and cleanup operations cannot be accidentally omitted.

To illustrate the use of the $START and $END macros, consider the example of Figure 10. This example uses the $START and $END macros along with global variables to define a pair of block-structuring macros.

```
$START:  'macro
         'gblint  'DO_nesting
         'DO_nesting<-0
$START:  'endmacro

DO:      'macro
         'gblint  'DO_nesting
         'DO_nesting<-'DO_nesting+1
DO:      'endmacro

ENDDO:   'macro
         'gblint  'DO_nesting
         'if      'DO_nesting>0
                  'DO_nesting<-'DO_nesting-1
         'else
                  'error    severe,"ENDDO without DO"
         'endif
DO:      'endmacro

$END:    'macro
         'gblint  'DO_nesting
         'if      'DO_nesting>0
                  'error    severe,"DO without ENDDO"
         'endif
$END:    'endmacro
```

Figure 10. A Block-Structured Set of Macros.

In Figure 10 note how the requirement of matching every DO statement with an ENDDO statement is enforced. A similar mechanism can be used to implement the "START_INSTRUCTION" and "END_INSTRUCTION" macros of Figure 3..

5. Generation of Hierarchical Circuits.

Generation of hierarchical circuit descriptions presents a difficult problem for most macro processors. To see why this is so, consider the system illustrated in Figure 3. Suppose that the target microprocessor is organized as four subcircuits, a ROM, a microcode sequencer, an ALU, and a block containing control and status logic. The four subcircuits are tied together using a parent circuit which includes each as a subcircuit. Conceivably, each instruction description could contribute to both the body and the interface of each of the four subcircuits. The output would be generated in the order given in Figure 11, but the circuit parser will expect the data to be organized as illustrated in Figure 12.

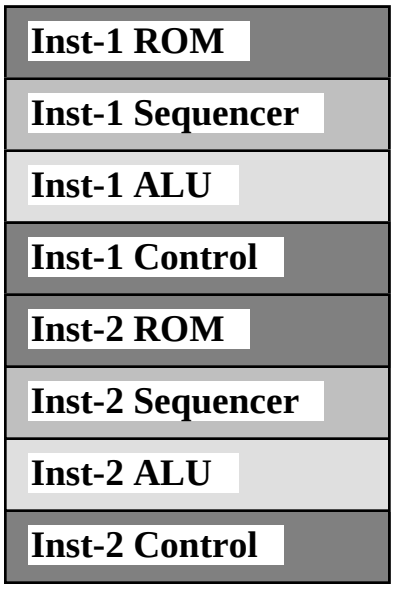

Figure 11. Output Generation Sequence.

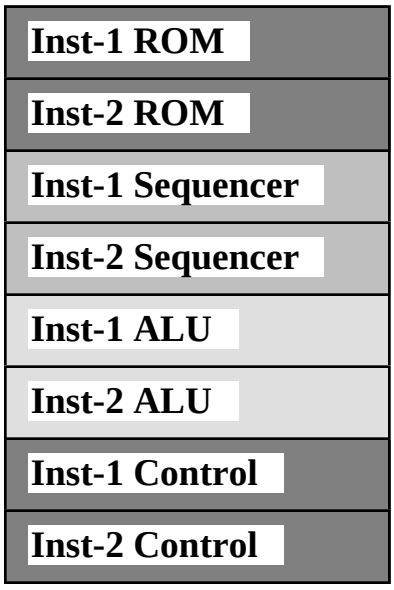

Figure 12. Required Output Sequence.

Using the output-partitioning feature of the FHDL Macro processor one can generate output in the order pictured in Figure 11, and obtain the output in the order pictured in Figure 12. One begins by declaring several output subsections as illustrated in Figure 13.

```
'subnetwork     ROM
'subnetwork     Sequencer
'subnetwork     ALU
'subnetwork     Control
```

Figure 13. Defining an Output Partition.

The four statements of Figure 13, which would normally be included in the $START macro, define four output partitions named ROM, Sequencer, ALU, and Control. These partitions will appear in the output in the order in which they were defined. Once the partitions have been defined, the 'subnetwork statement is used to switch from one partition to another, as illustrated in Figure 14.

```
'subnetwork     ROM
<output for inst-1>
'subnetwork     Sequencer
<output for inst-1>
'subnetwork     ALU
<output for inst-1>
'subnetwork     Control
<output for inst-1>
'subnetwork     ROM
<output for inst-2> ...
```

Figure 14. Generating Output Into Several Partitions.

Because each instruction may affect both the interface and the body of each subcircuit, the four partitions illustrated in Figure 13 will probably be insufficient. Separate partitions for the interface and the body will probably be required. In fact, the interfaces would probably be generated by the $END macro, at the same time the parent circuit is generated.

Every interface description consists of two lists of signal names, one for inputs and one for outputs. It is necessary to permit the processing routines for the instructions to add signal names of both types to the interface of any subcircuit, however postponing the generation of the interface until the $END macro is processed requires some mechanism for storing lists of names. The Macro processor provides a "list" data structure for this purpose. (Because lists are extremely versatile, they are the *only* data structure provided by the Macro processor.) The 'list statement is used to declare list variables, and there are a number of built-in list variables such as 'args which contains the argument list of the macro call. The usual list operations 'first, 'rest, and 'cons are provided, along with additional operations 'consr, which adds a new element to the right, and 'select(<list>,<number>), which selects the *n*th element from a list. The usual test functions 'atom and 'nil are also provided. One may also use list constants of the form (a,b,c, ...).

List variables may be either local or global. Global list variables may be used to hold the interface descriptions of various subcircuits until the $END macro is processed. These variables may then be used by the $END macro to generate both the interfaces of the various subcircuits, and the subcircuit calls of the parent circuit.

6. Current Research.

Much of the current work on the Macro processor involves features to analyze the structure of expressions. In the example of Figure 3 above, it is assumed that the "RTL" macros have the ability to analyze the structure of the expressions supplied as operands. Ordinarily, one assumes that only the *value* of an operand expression is of interest to a macro. To support macros such as the "RTL" macro, an "expression" data type and four new functions were added to the Macro processor. When an expression is assigned to a variable of type "expression," the expression is assigned *unevaluated*. The new function 'argexp returns the argument list of the macro call as a list of unevaluated expressions. The functions 'right and 'left return the right and left operands of an expression, while the function 'operator returns the operator of an expression.

The main difficulty in providing such a feature is determining the precise meaning of "unevaluated expression." Suppose x and y are two variables of type "expression." When the assignment statement 'x<-'y is executed, what should be assigned to 'x? If the "unevaluated expression" rule is followed strictly, then the *name* of 'y will be assigned to 'x, however the macro programmer probably intends for such a statement to copy the *contents* of 'y into 'x. It is not yet clear exactly where the line should be drawn, but the rule currently used by the Macro processor is to evaluate all operations that are meaningful only to the Macro processor, and leave unevaluated all operations (such as "+") that have meaning to one of the other FHDL languages. Furthermore, when an unevaluated expression is used as part of the generated output, it is completely evaluated at that point. The function 'eval can be used to force the evaluation of an expression regardless of where it is used, while the function 'exp can be used to prevent *any* evaluation of an expression. However, these two functions do not completely solve the problem of when to evaluate and when not to evaluate. For example, if an assignment of the form 'v<-'exp(...) is performed, and later the expression 'right('v) is evaluated, should the right side of the expression be partially evaluated at this point, or should it be left completely unevaluated? The 'exp function cannot be used to select one or the other since 'exp('right('v)) would prevent the 'right function from being evaluated.

Another problem that needs to be addressed is whether macros should be able to create their own expressions. At the present time macros can use expression constants, and can

decompose expressions, but there is no mechanism for combining two expressions together to create a new expression. It is not yet clear that such a feature would be useful.

Although the expression handling features of the Macro processor are still the subject of on-going research, the existing features are quite powerful. For example, a library macro has been provided that allows circuit designers to specify control logic in the form of logic equations rather than as networks of gates. Figure 15 contains an example of a call to this macro, and the logic generated by the macro definition.

```
(call)
equation    a=~((b&c)|~(c&d))

(Generated Logic)
AND         (b,c),UNIQ_11
AND         (d,e),UNIQ_13
NOT         UNIQ_13,UNIQ_12
OR          (UNIQ_11,UNIQ_12),UNIQ_10
NOT         UNIQ_10,a
```

Figure 15. An Equation Macro Call.

In Figure 15, the operators "~," "&," and "|" are used to represent the NOT, AND, and OR operations respectively. Figure 16 illustrates the "equation" macro and a sub-macro "equgate" used to generate the individual gates. Note that "equgate" is recursive.

```
equation:   'macro
            'exp        'parm

            'parm<-'argexp
            'if         'operator('parm)'!="="
                        'error      s,"The equation is in the wrong format"
                        'exit
            'endif
            equgate     'right('parm),'left('parm)
            'endmacro

equgate:    'macro
            'exp        'express,'osig,'le,'re
            'str        'ln,'rn
            'gblint     'uval

            'osig<-'right('argexp)
            'express<-'left('argexp)
            'if         'operator('express)'=="&"
                        'le<-'left('express)
                        're<-'right('express)
                        'if  'operator('le)'!='null
                                    'ln<-("UNIQ_"#'uval)
                                    'uval<-'uval+1
                                    equgate      'le,'ln
                        'else
                                    'ln<-'le
                        'endif
                        'if  'operator('re)'!='null
                                    'rn<-"UNIQ_"#'uval
                                    'uval<-'uval+1
                                    equgate      're,'rn
                        'else
                                    'rn<-'re
                        'endif
                        AND         ('ln,'rn),'osig
            'elif       'operator('express)'=="|"
            ...
            'elif       'operator('express)'=="~"
            ...
            'else
                        'error      severe,...
            'endif
            'endmacro
```

Figure 16. A Macro To Generate Logic From Equations.

To simplify the presentation of Figure 16, only the AND processing section of the "equgate" macro is shown in detail, the processing for the OR and NOT sections is similar. The 'exp statement is used to define variables of type "expression." The function 'operator returns the value 'null if its operand is not an expression. The two operands of the

"equgate" macro are the expression defining the gate, and the signal name to which the value of the expression must be assigned. The operand list of the macro is treated as a single expression with the operator ",", which explains why the 'left and 'right functions are used to extract the two operands from the operand list.

The AND processing section first determines whether the operands of the AND are expressions. If so, a recursive call to "equgate" is done to generate logic for the expressions. Note how the global variable is used to generate unique signal names. Once logic has been generated for both sides of the expression, an AND gate is generated for the current operator.

This example suggests another avenue for research for the Macro processor. Logic synthesis and minimization has long been the subject of intense research[7,8,9]. Although the example of Figure 16 performs elementary logic synthesis, most logic synthesis algorithms are too complex to be implemented as FHDL Macros. Nevertheless, the Macro processor provides a natural place for performing such operations. One possibility for future research is to extend the Macro processor to allow calls to external subroutines. Mechanisms are needed for supplying input data to external subroutines, and incorporating the output of these subroutines into the output of the Macro processor.

Another avenue of research is suggested by the "equation" macro. Rather than forcing the user to use the "equation" keyword for each equation, it may be more convenient to be able to define a new infix operator ":=", or overload an old operator. This would allow the macro name to be omitted when coding logic equations, but still allow the equations to be processed by a library macro.

7. Conclusion.

The FHDL Macro processor is a powerful tool that can be used to extend the power of the FHDL hardware specification system. Its primary use is in constructing flexible cells that can expand or contract to meet the needs of the hardware designer. To support such cells, the Macro processor provides many of the features of high-level languages, including loops, conditionals, recursive functions and data structures. The Macro processor also provides output structuring facilities for generating hierarchical circuits and expression handling features for processing complex expressions. In addition to its use in constructing flexible cells, the Macro processor also provides an implementation mechanism for more sophisticated synthesis systems. The FHDL Macro processor is the subject of much on-going research at the University of South Florida. As time goes on, the Macro processor will be adapted to support ever more complex synthesis systems. The

Macro processor is currently a component of the FHDL design system which is being used in several VLSI projects at the University of South Florida.

REFERENCES

1. J. J. Donovan, *Systems Programming*, McGraw-Hill, New York, 1972.

2. B. W. Kernighan, D. M. Ritchie, *The C Programming Language*, Prentice-Hall, Englewood Cliffs, NJ, 1978.

3. B. W. Kernighan, D. M. Ritchie, "The M4 Macro Processor," in *Unix Programmer's Manual*, Seventh Edition, Volume 2B, AT&T Bell Laboratories, Murray Hill, NJ, 1979.

4. Preas, B. and M. Lorenzetti (eds.), *Physical Design Automation of VLSI Systems*, The Benjamin/Cummings Publishing Co. Inc., Menlo Park, CA, 1988.

5. *VHDL Language Reference Manual*, IEEE Standard 1076, IEEE Computer Society Press, Los Amitos, CA, 1987.

6. *Electronic Design Interchange Format Version 2 0 0*, Recommended ANSI Standard EIA-548, Electronic Industries Association, Washington D.C., 1988.

7. J. A. Darringer, W. H. Joyner, Jr., C. L. Berman, L. Trevillyan, "Logic Synthesis Through Local Transformations," *IBM Journal of Research and Development*, Vol. 25, No. 4, pp. 272-280, Jul. 1981.

8. R. K. Brayton, C. McMullen, "Synthesis and Optimization of Multistage Logic," *ICCD-84*, pp. 23-28, Oct. 1984.

9. R. L. Rudell, A. L. M. Sangivanni-Vincentelli, "Espresso-MV: Algorithms for Multiple-Valued Logic Minimization," *IEEE CICC-85*, pp. 230-234, May 1985.