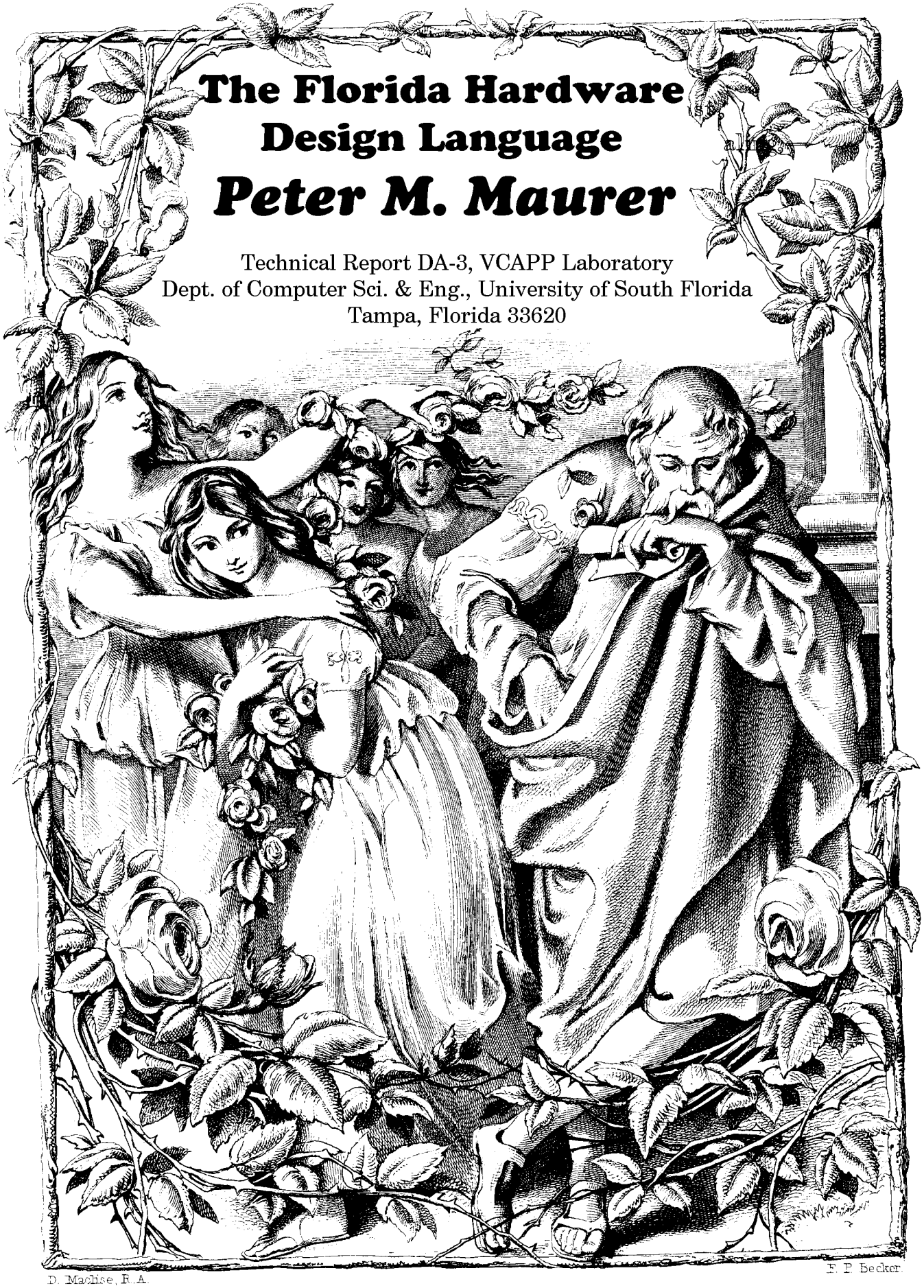
The Florida Hardware Design Language

Peter M. Maurer

Technical Report DA-3, VCAPP Laboratory
Dept. of Computer Sci. & Eng., University of South Florida
Tampa, Florida 33620



THE FLORIDA HARDWARE DESIGN LANGUAGE

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

ABSTRACT

The Florida Hardware Design Language is an expandable language that was originally developed to support a Wafer Scale Integration project under way an the University of South Florida. FHDL currently supports the uniform specification of gates, high-level functional blocks, state machines, ROM and PLA contents, and parameterized functional blocks. The language is easily extendable and a number of enhancements are planned or currently under way.

THE FLORIDA HARDWARE DESIGN LANGUAGE*

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

1. Introduction

FHDL, the Florida Hardware Design Language, is an expandable language that provides a uniform method for specifying all portions of an integrated circuit's design. FHDL was designed to support the development and testing of Wafer Scale Integrated circuits, especially at the functional level, but can be used to design other integrated circuits at many different levels. In a sense, FHDL is not just a design language, but a framework for developing a wide range of CAD tools. This point will become apparent as the details of the FHDL system are presented.

We elected to create a new language rather than use an existing one for several reasons. The primary reason for creating a new language is that this gives us the freedom to adapt the language to precisely fit our needs. Because we anticipated *some* automatic generation of FHDL specifications, we wanted the syntax to be lean and reasonably rigid. We wanted a language that was extendible in many different directions without requiring modifications to the basic core of the language. Finally, we wanted a syntax that would be be universal in the sense that it could be used to describe all portions of a circuit, including gates, large scale functional blocks, ROM and PLA contents, and state machines.

2. The Syntactic Structure of FHDL

We settled on an "assembly language" format for FHDL because the "name," "operator," "operands" format is simple and is readily adaptable to many different uses. The basic form of an FHDL statement is given below. All fields except the opcode are optional.

<name>: <opcode> <operand list>,<attribute list>

Although it is not readily apparent, this format has several features that enhance the extensibility of the language. First, except for a small handful of statements, no specific meaning is attached to any of the fields by the language itself. The semantics of most statements are defined by the several different tools that parse and process the code. This is particularly true for the attribute list which was design to provide tool-specific information. An attribute, which has the form "<name>=<value>" is parsed in such a was as to allow various tools to pick out the attributes in which they are interested, and to ignore others.

Although the free-form nature of FHDL makes it adaptable, most tools need to attach *some* meaning to fields early in the parsing process. In order to permit this to be done without sacrificing extensibility, FHDL uses the concept of "typed blocks" of code. A typed block begins with the following statement, which may have operands and attributes.

<name>: <type>

* This work was supported by DARPA and the University of South Florida Center for Microelectronics Design and Test.

The typed block ends with the following statement which may have a name, operands and attributes.

```
end<type>
```

Figure 1 contains an example of a block containing gate-level information. This block represents a half-adder.

```
abc:    circuit
        inputs      a,b
        outputs     s,c
        xor         (a,b),s
        and         (a,b),c
        endcircuit
```

Figure 1. An Example of a Typed Block.

In every case, the name field of the <type> statement identifies the block. Between the <type> and end<type> statements, specific meanings can be attached to names, opcodes, attributes and operands without affecting the meanings of the same elements in other types of blocks. As will be seen below, even within a typed block the assignment of meanings to fields is delayed as long as possible.

It is also possible for the syntax of statements within a typed block to depart radically from the standard FHDL format. This, in a sense, makes FHDL a superset of all other hardware design languages. For example, code written in a high-level language, such as C, could be included by enclosing it in the following two statements.

```
abc:    C_code
        endC_code
```

FHDL is parsed by several different translators, each one of which is responsible for blocks of one or more different types. In most existing cases, the translators take the form of preprocessors that transform their types of blocks into blocks of type "circuit." There are exceptions to this rule, and there will be further departures in the future.

Currently there are four FHDL translators, a macro processor that can be used to process any statement in the standard FHDL format, regardless of block type; a ROM preprocessor that translates blocks of type "rom" into blocks of type "circuit;" a PLA preprocessor that translates blocks of type "pla" into blocks of type "circuit;" and a functional simulator generator that translates a collection of blocks of type "circuit" into a C program that provides a logic-level simulation of the circuit. The code-generation phase can also be used to generate layout for ROMs, PLAs, and standard-cell circuits. The remainder of this paper will describe the existing translators and give some indication of the direction in which FHDL is growing.

3. The Functional Simulator Generator.

As stated above, the functional simulator generator translates blocks of type "circuit" into C programs capable of simulating the circuit. In addition to the extensibility features of the language itself, the generator also contains a number of its own extensibility features. The generator's parser is a general purpose front-end than can be used by any tool requiring logic-level input. (Several experimental simulators have been developed that use this this front end, and a number of other tools in the planning stages.) The independence of the parser is due to the generic data structures it produces. The parser translates FHDL blocks of type "circuit" into a collection of internal data structures that are passed to an

independent code generator. To enhance the utility of the parser there is a library of subroutines that can be used to manipulate the data structures produced by the parser. Included in the library are circuit flatteners, storage management routines, auditors, cyclic reference checkers, and routines to extract the strongly connected components of cyclic circuits. These subroutines can be used by any tool.

Another extensibility feature of the functional simulator generator is the fact that with a handful of exceptions the parser assigns no meaning to the opcode field of a statement. The text of the opcode is recorded in the data structure, and interpreted by the code generator. Although it is anticipated that most opcodes will have identical meanings for all tools, this feature allows certain opcodes to be interpreted differently by different tools. For example, while one tool might treat an opcode of "register" as a basic block and simulate it directly, another may may expand it into a collection of flip-flops.

The code generator recognizes four different classes of blocks, logic blocks, ROM blocks, PLA blocks, and Algorithmic State Machine (ASM) blocks. Logic blocks contain gates, flip-flops, large-scale functional blocks such as registers and ALUs, and hierarchical references to other blocks. An example of a hierarchical reference is given in Figure 2. Hierarchical references in FHDL are equivalent to those found in many other hardware design languages[1,2].

```
fadd:   circuit
        input     a,b,c
        output    s,c
ha1:    hadd      (a,b),(s1,c1)
ha2:    hadd      (s1,c),(s,c2)
or1:    or        (c1,c2),c
        endcircuit

hadd:   circuit
        input     a,b
        output    s,c
        xor       (a,b),s
        and       (a,b),c
        endcircuit
```

Figure 2. An Example of a Hierarchical Circuit.

The input and output statements identify the primary inputs and outputs of a logic block. Each statement in a logic block, (with the exception of input and output) is of the following form.

<name>: <gate type> (<input list>),(<output list>),<attributes>

In addition to the usual gate types of "and," "or," and so forth, the designer may also use types of "register," "adder," "ram," "alu," "decoder," and so forth. A complete set of high-level blocks is provided. For gates such as "register" and "counter" the <attributes> field is used to specify various additional options, such as shift capability, the presence of serial and parallel inputs, and the presence of status outputs.

A connection between gates is implicitly declared by using the name of a connection in a gate. A connection can also be explicitly declared using a "wire" statement. The primary function of "wire" statements is to assign attributes to nets (primary input, primary output or internal connection). Net attributes include active-high, active-low, and no-connect status, as well as a width declaration for buses. When code is generated for gates, the active-high/active-low status and the width is taken into account. For example, the following code fragment is equivalent to declaring 32 separate "xor" gates.

```
xor     (a,b),c
wire    a,width=32
wire    b,width=32
wire    c,width=32
```

An explicit data structure is created for each net regardless of whether it is explicitly or implicitly declared. Similarly an explicit data structure is created for each gate. A gate's data structure contains pointers to the net data structures that represent the gate's input and output nets. Similarly the data structure for a net contains pointers to the gates that use the net as an input or an output.

The data structures for ASM blocks are quite similar to those for logic blocks. Each ASM block contains three types of statements, state declarations, test declarations and conditional output declarations. These statements have the following form.

```
asm_state  <name>,<next statement>,o=(<unconditional outputs>)
asm_test   <name>,(<target1>,<target2>,...,<target2^n>),
        c=(<cond1>,...,<condn>)
asm_cond   <name>,<next statement>,o=(<conditional outputs>)
```

Modeling an ASM with these statements is a trivial exercise. Modeling a conventional state machine is only slightly more difficult. Nevertheless, because the format is somewhat confusing, we may decide to add an ASM preprocessor at some time in the future.

ROM blocks contain one statement for each word in the rom. The opcode of each statement is "romword" and the single operand specifies the contents of the word in hexadecimal format. Although the contents of any ROM may be specified in this format, the ROM preprocessor format is much easier to use. Similarly, each statement in a PLA block represents one wordline and has the opcode "plaword." Every statement has two operands, one that specifies the "and" plane connections and one that specifies the "or" plane connections. Again, the format is universal, but the PLA preprocessor language is much easier to use.

After parsing is complete, the code generator breaks categorizes all blocks according to type, performs some audits, and flattens all hierarchical circuits. Subcircuit definitions can appear "in line" with the rest of the code or be taken from one or more libraries of "standard cells". These steps are performed using library routines that are available to all tools. Code is generated by marking all primary inputs and all outputs of sequential elements as known. Gates whose inputs are all known are placed on a processing stack. An element is popped from the stack, code is generated for it, and its outputs are marked as known. This marking may cause other gates to be placed on the processing stack. When the processing stack is empty, code generation ceases. If there are any gates that were never placed on the processing stack, an error is reported. This procedure is essentially identical to that described in [3]. This processing will change once the work described in [4] is fully implemented.

Code for sequential elements is broken into two phases. Each sequential element is represented by two variables, a current-state variable and a next-state variable. Phase 1 computes the value of the next-state variable using the values of the current-state variables. After the values of the next-state variables and the primary outputs have been computed, phase 2 is executed which copies the next-state variables into the current-state variables. Sequential elements may change state at most once per clock period.

After code generation for all gates has been completed, the code generator adds a main routine and an elementary vector processor for asserting inputs and displaying outputs. In the future we plan to add a sophisticated vector generation system along the lines of that presented in [5].

In addition to textual input, a graphical schematic interface has been designed for entering gate-level information. This interface is described elsewhere in these proceedings[6].

4. The ROM Programming Language

Although the "romword" statements of the functional simulator generator can be used to construct any ROM, it is well recognized that a more powerful programming language is needed for microcoded ROMs. The ROM preprocessor, which translates blocks of type "rom" into blocks of type "circuit," provides just such a language. Unlike many other ROM programming languages, the FHDL ROM language is not dedicated to a particular ROM sequencer, or to a particular word format. Statements are provided that allow the user to specify word formats, address formats, and command formats. Although a large number of statements may be required to establish the programming environment, these statements can be placed in an "include" file and used for many different applications. The structure of the ROM language is illustrated in Figure 3.

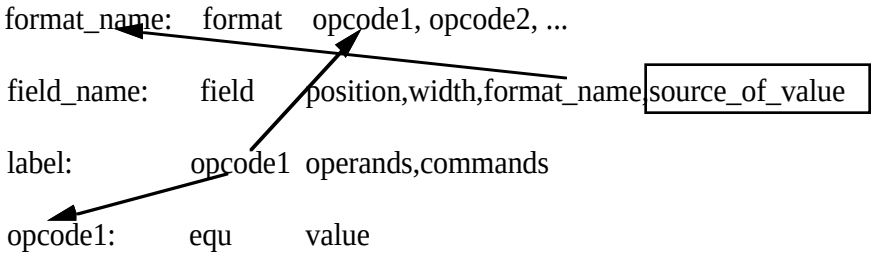


Figure 3. The structure of the ROM Programming Language.

Each block of type "rom" defines a programming environment and the contents of one ROM. A programming environment consists of one or more formats, each of which is defined by a "format" statement. The format statement lists the opcodes of the word-creation statements (or word-statements) for the format. Each format contains one or more field descriptors that define the components of a word and the syntax of the statements that create these words. Each field descriptor is created by a "field" statement that also has parameters to determine the source of the field's value. The value may be a constant or a default, or it may be obtained from some portion of the word-statement. A word-statement can assign a value to a field in three different ways. First, the value of the opcode can be automatically assigned to a field. The usual value for an opcode is zero, but this can be changed with an "equate" statement. Next, the value of an operand can be automatically assigned to a field. If operand number *n* is automatically assigned to a field, then operands 1 through *n*-1 must also be automatically assigned to fields. Finally, any field in the format, including operand and opcode fields, may be explicitly assigned a value using a command of the form "value->field_name".

As in other ROM programming languages, values may be specified as expressions containing arithmetic operators and parentheses, equates may be used to clarify the meanings of various values, and complex commands may be defined using statements of the following form.

command_name: command <command 1>,<command 2>, ...

The ROM preprocessor supports several different ROM addressing schemes. Although most addressing details concern hardware external to the ROM, it is necessary to know the format of an address when constructing jump-type micro-instructions. The simplest scheme is one in which the jump-type instruction contains a complete ROM address.

Placing a label on a word-statement causes the label to be equated to the ROM address of the generated word. This label may be used to create the value of any field. As is usual, the symbol "*" represents the address of the current word. When the "*" symbol appears in the default value for a field, the symbol (and the containing expression if any) is re-evaluated for each word in which the field appears. Thus one could support sequential micro-instruction execution in a two-address-per-word branching scheme by assigning default values of "*+1" to both address fields.

For other branching schemes, the ROM-compiler allows a portion of an address to be assigned to a field under the assumption that the remainder of the address will be supplied by external logic.

5. The PLA Language

The FHDL PLA language is quite similar to the FHDL ROM language. Each block of type "pla" defines a programming environment and the contents of one PLA. Each programming environment contains one or more formats, each of which contains one or more field descriptors. Field descriptors define the format of the OR plane of each wordline as well as the syntax of the statements that create the wordlines of the format. With only one exception, the full power of the ROM language is available for programming the OR plane of the PLA. Because addresses are not meaningful in PLAs, the addressing features of the ROM language are not included in the PLA programming language.

The AND plane of the PLA is programmed using conditions, which replace the statement labels used in the ROM programming language. A simple condition is an expression of the form "input_field_name=value." (A special format called the input format contains the descriptors of all input fields.) The connectives & (and), | (or) and ~ (not) may be used to form complex conditions. To simplify the programming of state machines and sequential microcode, the PLA language provides "begin" and "end" statements that allow conditions and commands to be applied to an entire group of statements. This concept is illustrated in Figure 4. Figure 4 gives an example of a three-state state machine with several sequential substates per state. It is assumed that this PLA will be coupled with a clocked external register that stores the values of "ostate" and "osub" and returns them in the input variables "istate" and "isub."

```
go_to_first:          command     FIRST->ostate,ZERO->osub
go_to_second:         command     SECOND->ostate,ZERO->osub
go_to_third:          command     THIRD->ostate,ZERO->osub

istate=FIRST:         begin       FIRST->ostate
isub=ZERO             wordline    ONE->osub
isub=ONE              wordline    TWO->osub
isub=TWO:             begin
carry=1:              wordline    go_to_second
carry=0:              wordline    go_to_third
                      end
                      end

istate=SECOND:        begin       SECOND->ostate
isub=ZERO             wordline    ONE->osub
isub=ONE              wordline    TWO->osub
isub=TWO              wordline    THREE->osub
isub=THREE:           begin
x=1&y=0:              wordline    go_to_first
x=0&y=1:              wordline    go_to_third
(x=0&y=0)|(x=1&y=1):  wordline    go_to_second
                      end
                      end

istate=THIRD:         begin       THIRD->ostate
isub=ZERO             wordline    ONE->osub
isub=ONE:             begin
z=1:                  wordline    go_to_first
z=0:                  wordline    go_to_second
                      end
                      end
```

Figure 4. An Example of a PLA-Based State Machine.

The condition for a particular wordline is obtain by ANDing the condition on the wordline statement with the conditions on the "begin" statements of any begin/end blocks containing the wordline statement. Thus the condition for the first wordline statement of Figure 4 is "istate=FIRST&isub=ZERO." By the same token, the commands on the wordline statement are combined with any commands appearing on the begin statements of the enclosing begin/end blocks.

Although the PLA preprocessor allows the OR connective (|) to be used in conditions, this connective cannot be implemented in the AND plane alone. When the OR connective is used in a conditional expression, the expression is reduced to sum-of-products form, and one wordline is generated for each product term. Thus the OR connective is actually implemented in the OR plane of the PLA. The NOT connective is implemented by breaking multi-bit input fields into single-bit fields and applying DeMorgan's laws.

The PLA language also allows complex conditions to be defined using the condition statement illustrated below.

condition_name: condition <conditional expression>

This statement performs a function analogous to that of the "command" statement illustrated in the previous section.

6. The Macro Language

The FHDL Macro Language provides a method for extending the ROM, PLA, and ASM languages. Although it can be used for conventional purposes such as creating parameterized functional blocks, it was also designed to provide a means for constructing quick prototypes of high-level synthesis tools. Because of this, the FHDL macro processor has a far more extensive list features than would be found in a conventional macro processor. The input to the macro processor is a set of statements in the standard FHDL format. The output may be in any format, but is typically code that is acceptable to either the ROM preprocessor, the PLA preprocessor, or the functional simulator generator. A macro definition has the following form.

```
macro_name:     'macro
                ...
                'endmacro
```

Macros may be included with the rest of the FHDL text or may be placed in a macro library. A macro is invoked using a macro-call statement which has the following form.

```
<label>:  macro_name  <operands>
```

When a statement of this form is encountered, the processing of the current input is suspended, and the body of the macro is processed as if it were included in the input at that point.

Macros contain two types of statements, output statements and control statements. An output statement is an ordinary FHDL statement which is not also a macro call. When an output statement is encountered, it is copied into the output. If the statement contains any macro-processor expressions, they are removed, evaluated, and replaced by their values. Macro-processor expressions are expressions that contain macro variables, or macro-operators or both. A macro-variable has the form <apostrophe><name>, while a macro-operator may be any one of a wide variety of arithmetic and logical operators.

Control statements are used for a variety of functions, including defining variables, assigning values to variables, constructing if-then-else conditionals, and constructing while loops. The statements comprising the body of a macro are normally processed sequentially, but "if" statements and "while" statements can be used to alter the normal flow of statement processing. Because the macro language contains (compile time) assignment statements "if" statements and "while" statements, it is a universal programming language. In addition, one macro may call another either as an ordinary macro or as a function-call that returns a value to an expression. Macros may even be recursive, as the example given in Figure 5 shows.

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

Figure 5. A Recursive Macro.

Although recursive macros such as that pictured in Figure 5 are relatively uncommon, this macro illustrates some of the more common features of all macros. The variable '0 specifies the first operand of the macro call (or in this case, the first argument). The double

equal sign "==" is used to test for equality. The "<-" operator on line 3 causes the value 1 to be assigned to the variable 'factorial, which in this case is the "returned-value" variable of the macro. A more conventional macro is illustrated in Figure 6. This macro creates a chain of not gates. The input is taken from the first operand of the macro call, while the output is taken from the second operand. The third operand determines the number of gates in the chain.

```
not_chain:  'macro
            'str            'in,'out
            'int            'i,'j
            'in<-'0
            'i<-0
            'j<-0
            'while          'i<('2-1)
            'out<-x#'j
            'j<-'j+1
            not             'in,'out
            i<-'i+1
            'in<-'out
            'endwhile
            not             'in,'1
            'endmacro
```

Figure 6. A Conventional Macro.

This macro illustrates the use of compile-time loops. Most of the code is straightforward. The 'str and 'int statements define variables of type string and integer, and the operator # is used to denote concatenation. Type coercion is automatic. Figure 7 illustrates a typical macro call for this macro, and the output produced in response to the macro call.

```
not_chain   a,b,4


not         a,x0
not         x0,x1
not         x1,x2
not         x2,b
```

Figure 7. A Macro Call and Its Output.

The macro processor provides a full range of arithmetic operators, including exponentiation and remainder; a complete set of comparison operators; the AND, OR and NOT logical connectives; a complete set of bitwise logical operators, including exclusive-or ones-complement, and left/right shifts; and a complete set of string-handling functions. Numbers may be specified in binary, decimal, octal, or hexadecimal.

In addition to string and integer variables, the macro processor also provides list data structures which are modeled after those supported by the LISP language. A complete set of list handling functions is provided. Type coercions are usually automatic, but an complete set of type-coercion functions is provided to override the automatic coercions.

Variables may be either local or global. The scope of a local variable is the macro definition, with new copies of local variables created for each recursive call. Global variables, which must be declared by all macros that use them, have universal scope and retain their values from one call to another. When a macro library is used, it may contain

two macros named "$START" and "$END." The "$START" macro is invoked before any input is processed, and the "$END" macro is invoked after all input is processed. These macros may be used to initialize global variables, and to perform initialization and cleanup for specialized macro packages.

For some highly specialized macro packages it may be necessary to generate several circuits simultaneously. To support this activity, the macro package allows output to be directed into several named sections. Although output is added to these sections simultaneously, it will not be intermixed in the final output. In the final output, the content of the named sections will appear consecutively in the order that the sections were declared. There is no limit to the number of sections that may be used.

The macro processor is currently being used with a small library that generates varying-width carry-lookahead adders and other varying-width arithmetic circuits. The macro processor provides other features that cannot be described here due to lack of space.

8. Extensibility.

The FHDL system was designed to be a basis for future research in CAD tools. Recently the language has been extended to incorporate switch-level simulations along with the other types of simulations. To incorporate a switch-level simulation into an FHDL simulation, one includes a block such as the following in the FHDL code.

```
name: mos
    ... data extracted from the layout ..
    endmos
```

The data between the "mos" and "endmos" statements is obtained by extracting transistor information from a MAGIC [7] layout using the standard MAGIC extraction program. Alternatively, the data can reside in a separate file that is referenced by the "mos" statement. This data is then passed to the COSMOS compiler[8], and the output is incorporated into the FHDL output using a special interface routine. The switch-level simulation generated by COSMOS runs simultaneously with other simulation code generated by the FHDL compiler.

Another example of the extensibility of the FHDL system is work done for a comparative simulation study done at the University of South Florida[9]. Eight simulators were required for this study in addition to the compiled unit delay simulator provided by the FHDL system. These were three different interpreted three-valued zero-delay simulators, an interpreted two-valued zero-delay simulator, an interpreted three-valued unit-delay simulator, an interpreted two-valued unit-delay simulator, and two unit-delay compiled simulators. All eight simulators were built upon the code generation routines provided by the FHDL system. (The parser was identical for all simulators.) Modification of the existing code generation routines required about one month of programming time for one person. This level of productivity was possible only because the basic structure of each simulator was already present in the existing FHDL code-generator. Even though the difference between compiled and interpreted simulation appears to be great, the code used to schedule gate simulations was easily adaptable to interpreted simulation. For the compiled unit-delay simulators, some modifications were necessary to two existing routines. Several additional routines were written, but these were either very small or straightforward adaptations of existing routines. If these simulators had to be designed from scratch, the total development time could easily have been several months.

These examples demonstrate that the FHDL system is a valuable resource for future research in CAD tools. Future extensions to the system will provide for new types of simulation, and will enhance the layout facilities described in the next section.

9. Automatic Layout.

In addition to its ability to simulate circuits containing several different types of blocks, FHDL can also be used to automatically lay out ROMs, PLAs, and logic-blocks containing standard cells. Figure 8 gives a picture of the flow of the FHDL code, and illustrates how the layout process takes place.

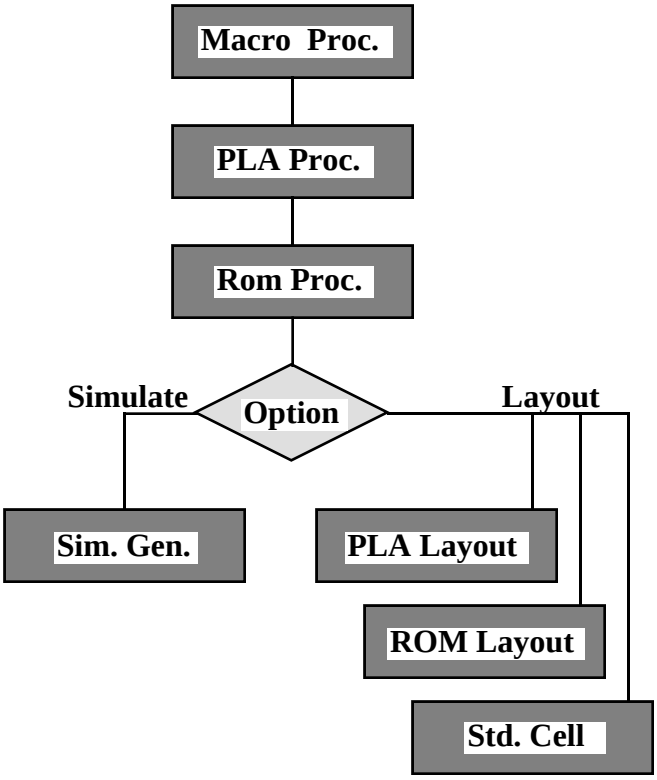


Figure 8. The Flow of the FHDL Code.

A simple option to the compiler is used to select layout instead of simulator output. Three different layout routines may be invoked depending on the type of input. The PLA layout routine makes use of a standard tiling program provided with the Berkeley MAGIC package [7]. The ROM layout routine is similar, except that it must synthesize the ROM decoder. The standard cell layout program makes use of the Timberwolf placement program[10] as well as the MAGIC automatic routing features. As yet, the production of layout is not completely automated. For a complete discussion of the FHDL layout features, see [11].

7. Conclusion.

The strength of FHDL lies in its extensibility and in its ability to describe all portions of a circuit, including logic and state machines as well as ROM and PLA contents. At the present time FHDL is still in its infancy. It has been used to simulate a number of arithmetic circuits, both combinational and sequential, and we anticipate that it will be used to support laboratory work for courses in Printed Circuit Board design. By January of 1989, FHDL should be fully integrated into the WSI project currently under way at USF. We also anticipate that FHDL will be used by other VLSI projects.

There is currently a great deal of work being done to enhance and extend FHDL. The translators described in this paper are continually being tested, and improved. The main

priority at this time is enhancing the scheduling algorithm of the functional simulator to be able to simulate hierarchical circuits even in the presence of cycles[4]. This will greatly enhance the ability of the functional simulator generator to incorporate separately compiled subroutines and to selectively invoke more detailed simulators on certain portions of a circuit. Once this work has been completed, we anticipate integrating the functional simulator generator with various logic simulators, switch-level simulators, timing simulators, circuit simulators, and so forth. Depending on our needs, we may integrate the generator with commercial simulators or with locally-developed simulators . In either case we plan to transform the functional simulator generator into a mixed-level simulator that is capable using several different simulation techniques simultaneously on different parts of the circuit.

We believe that the FHDL system will be of great benefit to our continuing research efforts for many years to come.

REFERENCES

1. *VHDL Language Reference Manual*, IEEE Standard 1076, IEEE Computer Society Press, Los Amitos, CA, 1987.
2. *Electronic Design Interchange Format Version 2 0 0*, Recommended ANSI Standard EIA-548, Electronic Industries Association, Washington D.C., 1988.
3. L. Wang, N. Hoover, E. Porter, J. Zasio, "SSIM: A Software Levelized Compiled-Code Simulator," Proceedings of the 24th Design Automation Conference, 1984, pp. 473-478.
4. Z. Wang, P. Maurer, "Scheduling High-Level Blocks for Functional Simulation," Department of Computer Science and Engineering, University of South Florida, Tampa, FL 33620, Technical Report Number CSE-88-24, 1988.
5. P. Maurer, "Design Verification of the WE 32106 Math Accelerator Unit," IEEE Design and Test of Computers, June, 1988, pp. 11-21.
6. N. Bhate, A. Tokuta, "The FHDL Schematic Capture Program," Proceedings of Southeascon-90.
6. J. K. Ousterhout, G. T. Hamachi, R. N. Mayo, W. S. Scott, G. S. Taylor, "The Magic VLSI Layout System," *IEEE Design and Test of Computers*, Vol. 2, No. 1, Feb. 1985.
7. Bryant, R. E., D. Beatty, K. Brace, K. Cho and T. Sheffler, "COSMOS: A Compiled Simulator for MOS Circuits," Proceedings of the 24th Design Automation Conference, 1987, pp. 9-16.
8. P. Maurer, Z. Wang, "Techniques for Unit-Delay Compiled Simulation," Submitted for publication.
9. Sechen, C. and A. Sangiovanni-Vincentelli, "The Timberwolf Placement and Routing Package," *Custom Integrated Circuits Conference*, May 1984.
10. C. Morency, P. Maurer, "HDL Driven Chip Layout within the FHDL Design Framework," Proceedings Southeastcon-90.