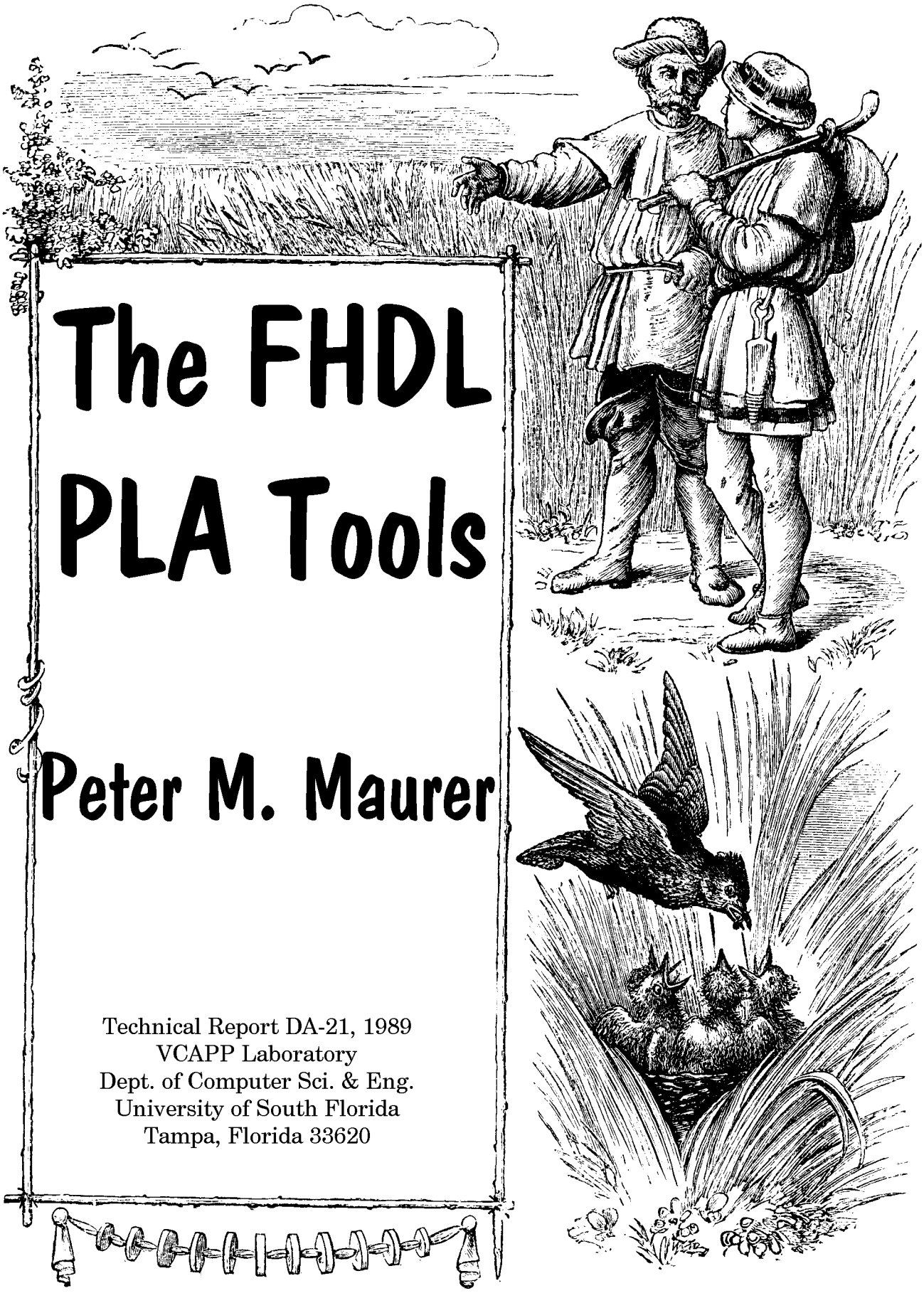

The FHDL PLA Tools

Peter M. Maurer

Technical Report DA-21, 1989
VCAPP Laboratory
Dept. of Computer Sci. & Eng.
University of South Florida
Tampa, Florida 33620

THE FHDL PLA TOOLS

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

ABSTRACT

The FHDL (Florida Hardware Design Language) PLA tools provide a means for specifying, simulating, and automatically laying out Programmed Logic Arrays (PLAs). These tools were created to facilitate VLSI design projects, to improve the quality of hardware design courses, and to serve as a basis for future research in VLSI design automation. At the specification level, the PLA tools allow the contents of a PLA to be specified as a set of logic equations. In addition, they provide features for facilitating the construction of PLA-based state machines. Once a PLA has been specified, it can be simulated at a high level in coordination with the simulation of the other portions of a VLSI design. After a PLA has been verified through simulation, it can be laid out automatically through an interface to the Berkeley PLA layout tools. The primary motivation for developing these tools was to provide a basis for future research in VLSI design automation.

THE FHDL PLA TOOLS

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

1. Introduction

The Programmed Logic Array, or PLA, is a fundamental component of many of today's Very Large Scale Integrated (VLSI) circuits[1]. PLAs can be used for many purposes, the most common of which are implementation of combinational logic and the construction of large complex state machines. Because of this, PLAs have been the subject of much research for the last several years. Most of this research has been focused on creating tools to simplify the specification of PLAs, and on tools for automatic layout and optimization of PLAs[2]. At the University of South Florida, we have created a set of hardware design tools called the Florida Hardware Design Language (FHDL)[3]. This set of tools enables hardware designers (particularly students) to specify designs in a textual format and simulate them to verify their correctness. These tools also provide an interface to several automatic layout and optimization programs available from the University of California, Berkeley.

To clarify the following discussion, consider Figure 1 which shows the basic structure of a PLA.

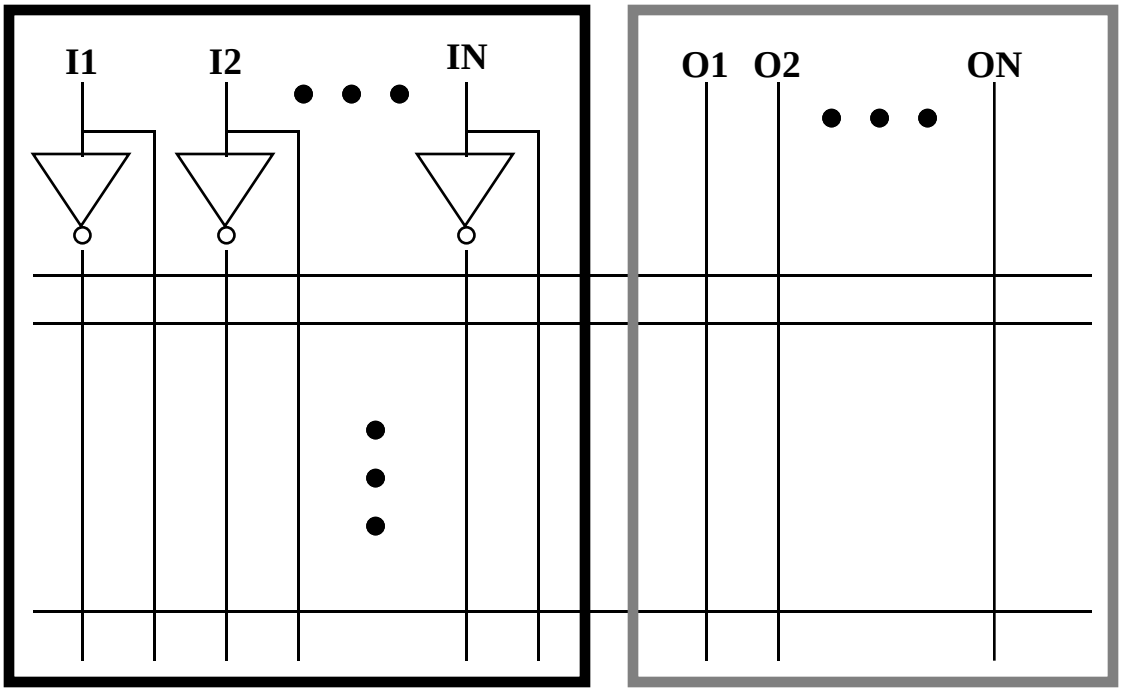

Figure 1. The Basic Structure of a PLA

In Figure 1, the area in the black box is called the *AND plane* of the PLA, while the area in the grey box is called the *OR plane*. The horizontal lines are called *wordlines*. Each wordline is connected to one or more of the inputs I1 through IN. Each input is available in both complemented and uncomplemented form, which gives three possibilities for each wordline. The wordline may be connected to the complemented input, the uncomplemented input, or not connected to either. The three types of connections are known as 0 or *false*, 1 or *true*, and *don't care* respectively. A wordline is *selected* when all the vertical lines to which it is connected all have the value 1. If a wordline is connected to the complemented form of input *a* and the uncomplemented forms of *b* and *c* then the wordline will be selected when the term *a'bc* is true. Thus the AND-plane computes the AND function for terms of a logical expression. Similarly the output lines may be either connected or not connected to the word line. An output has the value 1 whenever it is connected to a selected wordline, and the value 0 otherwise. When more than one wordline is selected, the outputs contain the logical OR of the values selected by the individual wordlines. If an output is connected to two wordlines, one of which computes the term *abc* and the other of which computes the term *de*, the output will be 1 (true) whenever the expression *abc+de* is true. Hence the OR-plane computes the OR function for a logical expression.

The expressions computed by a PLA must be in "sum-of-products" form. That is, there may be no parentheses, and the NOT function may not be used. (The AND function has precedence over the OR function.) Variables may be primed (complemented) or unprimed (uncomplemented). Expressions of this form are sometimes called two-level expressions, since they can be computed using two levels of logic. To translate a sum-of-products logical expression into PLA format one first allocates a wordline for each term, and then connects each input to the wordline depending on whether the input is complemented, uncomplemented, or not present in the term corresponding to the wordline. If a single logical expression is being implemented, the PLA will have a single output which is connected to all wordlines. However in most cases, the PLA will be used to implement several logical expressions simultaneously. In this case, each output corresponds to a single logical expression, and is connected to the wordlines that correspond to the terms of the expression.

Although the primary function of a PLA is to compute logic functions, it can be viewed in another way. When one is using the PLA to construct the control function for a complex state machine it is convenient to think of a PLA as a ROM in which it is possible to program not only the outputs but the address-decoder as well. When viewed in this manner, each wordline has an address of the form "0010xx0x1." These addresses are similar to ROM addresses, with the addition of "x" the "don't care" value. A PLA can be microcoded in much the same way as a ROM. (In fact, in the remainder of this paper the terms "microcoded PLA" and "PLA-based state machine" are used interchangeably.) The microinstructions are stored in the OR-plane, while the AND-plane contains the address of each word. The primary differences between ROM microcoding and PLA microcoding is that not all addresses need to be defined, and that two or more words can be selected simultaneously. Sequencing of PLA microcode is accomplished by using hardware similar to that pictured in Figure 2.

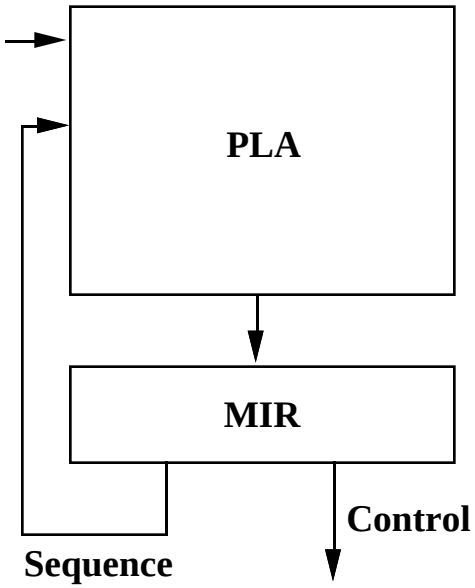

Figure 2. A PLA Microcoding Scheme

Since PLA wordlines do not normally have sequential addresses, each microinstruction must supply the address of the next microinstruction to be executed. As with ROM microcoding, a portion of the address is supplied by status lines from the hardware. Because the addressing structure of the PLA is programmable, the method for handling status lines is more straightforward than that typically found in ROM microcoding.

2. Introduction to the PLA Tools

The FHDL PLA tools provide a means for constructing, simulating, and automatically laying out PLAs. The motivation for constructing the these tools was threefold. The first aim was to facilitate VLSI research at the University of South Florida. Before the introduction of these tools, PLAs had to be constructed by hand, and had to be simulated at the transistor level to verify their correctness. The Berkeley tools could be used to automatically generate the layout, but this required the designer to specify a separate file of PLA specifications, and still forced the designer to simulate the PLA at the transistor level. The FHDL PLA tools allow the description of the PLA to be included with the description of the remainder of the circuit, and provide a means to simulate the PLA at a higher (and more efficient) level than the transistor level. The automatic layout and optimization tools provided in the Berkeley tool set were modified to take FHDL PLA descriptions as input, thereby creating a continuous path from high-level specification to layout and optimization.

The second motivation for creating the FHDL PLA tools was to enhance the quality of hardware courses taught at the University of South Florida. Since the PLA-based state machine is an important technique for implementing computer architectures and other highly complex sequential circuits, we naturally wish to teach the concepts of PLA-based state machines in our computer architecture and sequential circuits courses. The availability of the FHDL specification and simulation tools makes it possible to provide laboratory exercises in PLA-based state machine design without the expense of providing actual hardware. The tools also make it possible to for students to design and test hardware that can be implemented only in custom VLSI. Without high-level specification and simulation tools, there is no practical way to give students extensive experience with such designs.

The third (and as far as the authors are concerned, the most important) motivation was to provide a springboard for future research in various aspects of PLA design automation. Design automation research typically involves a great deal of programming and requires programming skills in many different areas. In addition, it requires knowledge of hardware structures, and how those structures are used to create circuits and systems. The average master's student can easily become overwhelmed by the huge amount of code that must be written to test even the simplest of ideas. One intent of the FHDL PLA tools was to provide a framework into which new algorithms could be placed. This allows students to focus more closely on the problem to be solved rather than on the code in which it is to be embedded. Furthermore, the existence of code for parsing and manipulating logical expressions gives the student working examples upon which further work can be based.

2. Logical Expressions.

The first component of the FHDL PLA tools is a compiler that translates a high-level description of a PLA into bit-level wordline specifications. In its simplest form, a PLA specification is a list of logical expressions, along with the OR-plane values that will be produced when the expressions are true. As Figure 3 illustrates, the expressions do not have to be in sum of products form.

```
a&b:              word    exp1
a&(c | d):        word    exp2
!(e&f)&(!g | !h): word    exp3
```

Figure 3: A Simple PLA Specification.

The PLA of Figure 3 contains eight inputs, three outputs and three expressions. (In a running example there would be additional statements to specify the position of the inputs

and outputs. The keyword "word" is used to distinguish wordline specifications from formatting statements.) The operators are &-AND, |-OR, and !-NOT. When the expression a&b is true, the output exp1 will also be true. It is possible for all expressions to be true simultaneously, thereby causing all outputs to be true simultaneously.

The translation from expressions to wordlines is not one-to-one. The expression "a&b" produces a single wordline, while the expression "a&(c | d)" produces two. The PLA compiler automatically generates as many wordlines as needed for a particular expression, but before wordlines can be allocated, the expression must be converted into sum-of-products form. This is done by application of DeMorgan's laws to remove the NOT operator (except as applied to variables) and successive application of the distributive laws to remove parentheses. Figure 4 gives an example of how the expression "!(e&f)&(!g|!h) is converted to sum-of-products form.

```
!(e&f)&(!g | !h)
(!e | !f)&(!g | !h)
((!e | !f)&!g)|((!e | !f)&!h)
!e&!g | !f&!g | !e&!h | !f&!h
```

Figure 4. The Conversion of an Expression to Sum-Of-Products Form.

Once the sum-of-products expression has been generated, one wordline is generated for each term in the expression. No attempt is made to minimize expressions at this point. This problem has been deliberately left open to provide an opportunity for a master's project at some future date.

The focus on expressions rather than wordlines has two purposes. First, it allows users of the PLA tools to code their expressions in the most convenient form and lets the compiler do the laborious work of translating them into the proper form. Second, it provides examples of expression-manipulating code for students who are doing projects in design automation.

Many of the features of the FHDL PLA tools have been provided primarily to emphasize to design automation students that they must think in terms of how their tools will be used, rather than in terms of reducing their own work. The focus on expressions is one example. Another example is the ability to use multi-bit fields both in expressions and in OR-plane specifications. For example, the user could code the following statement.

```
a=5&b=2:        word   15->r
```

In this example, it is assumed that a, and b are 3-bit fields, and that r is a 4-bit field. The compiler interprets this statement as follows. (Subscripts denote the position of the individual bits within the field.)

$$a_0\&!a_1\&a_2\&!b_0\&b_1\&!b_2: \quad \text{word} \quad r_0,r_1,r_2,r_3$$

When multi-bit fields are used in expressions, the first step in converting the expression into sum-of-products form is breaking the fields into individual bits. After this step, the operation proceeds as described above.

3. Support for Microcoded PLAs

Any PLA can be defined using nothing more that a list of logical expressions. However, the complexity of microcoded PLAs can make the list of expressions quite unmanageable unless a structured approach is used. The FHDL PLA compiler provides "begin" and "end" statements to facilitate microcode structuring, and also provides the ability to reference complex conditions by name. OR-plane specifications can also be collected into meaningful groups and referenced by name. To illustrate the grouping of OR-plane commands, consider the hardware pictured in Figure 5.

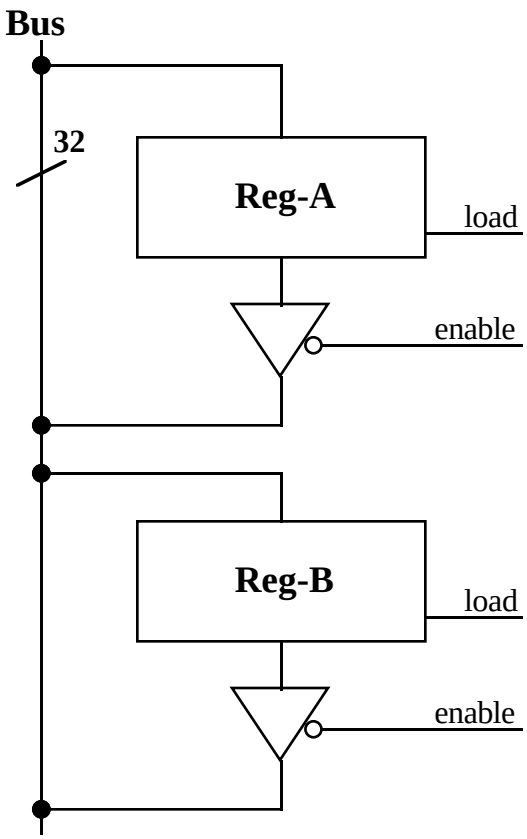

Figure 5. An Example of a Shared Bus.

Figure 5 contains an example of a shared bus which can be used to transfer data in either direction between two registers. To transfer data from Reg-A to Reg-B one must activate the enable control of Reg-A and the load control of Reg-B. It is assumed that the bus can also be used for other types of transfers, so the load and enable controls must be distinct. One OR-plane output can be assigned to each of the four control lines, and each wordline that wishes to perform a transfer can include both OR-plane bits in its specification. However, the microcode can be made more readable by including a command definition as illustrated in Figure 6.

```
Move_A_to_B:   command   load-B,enable-A
Move_B_to_A:   command   load-A,enable-B

a&b:           word      Move_A_to_B
a&!b:          word      Move_B_to_A
```

Figure 6. An Example of a Command Definition.

The FHDL PLA compiler provides a mechanism similar to that pictured in Figure 6 to allow the definition of complex conditions. Figure 7 illustrates this mechanism.

```
Read_Done:      condition    Bus_Error | Data_Ready

Read_Done:      word         Handle_Data
!Read_Done:     word         Wait
```

Figure 7. An Example of a Complex Condition Definition.

Figure 7 illustrates a segment of code that might be used as part of the control of a microprocessor. It is assumed that a memory-read operation is in progress. The first wordline performs the steps necessary when the read operation completes, while the second waits for the operation to complete. The completion of the read operation can be signaled by one of two status lines. Bus_Error signals that an error has occurred, while Data_Ready signifies that the operation completed normally. In addition to complex conditions and commands, the FHDL PLA compiler also provides mechanisms for defining named constants.

The purpose of the "begin" and "end" statements is to specify conditions and commands that apply to a group of wordlines. (At the present time the PLA compiler does not support scoping of names, but this feature will probably be added in the near future.) Figure 8 contains an example of a begin-end block.

```
(a | b)&(d | c):    begin       8->R_Type,Activate_Read
e:                  word        Move_A_to_B
f:                  word        Signal_Done
g:                  word        7->R_Type
                    end
```

Figure 8. An Example of a Begin-End Block.

When a condition is specified on a begin statement, the condition is ANDed with the conditions specified for the "word" statements of the begin-end block. OR-plane specifications specified on the begin statement are included with the OR-plane specifications on the "word" statements. If an OR-plane field is specified on both the begin statement a "word" statement, the value specified by the "word" statement takes precedence. Figure 9 illustrates how the "word" statements of Figure 8 would be coded without begin and end statements.

```
((a | b)&(d | c))&e:    word    Move_A_to_B,8->R_Type,Activate_Read
((a | b)&(d | c))&f:    word    Signal_Done,8->R_Type,Activate_Read
((a | b)&(d | c))&g:    word    7->R_Type,Activate_Read
```

Figure 9. The Effect of Removing the Begin and End Statements.

One of the most complex problems that can be solved using microcoded PLAs is building the control structure of a microprocessor. There are several portions of the control that must perform a series of sequential steps. There are other portions, most notably the instruction decoder, that must perform n-way branches. The sequence of control steps is normally broken into a number of states, such as INSTRUCTION-FETCH, and DECODE. There is normally one state for each instruction that can be executed by the microprocessor. Within each state, a number of sequential sub-states are defined. One method of implementing such a control structure is to break the sequencing portion of the Micro-Instruction Register into two sections, one that specifies the state, and one that specifies the sub-state. Recall that all microinstructions must supply the address of the next micro-instruction to be executed. Large-scale branching is performed by specifying a new state, while small-scale sequencing is performed by leaving the state unchanged while supplying a new sub-state. The task of specifying such a control structure is greatly simplified by the use of begin-end blocks, as well as the command, condition, and constant definition features. To illustrate, consider the example of Figure 10, which contains part of the control for a microprocessor.

```
State=IFETCH:          begin    IFETCH->Next_State
Sub=0:                 word     PC_TO_MAR,1->next_sub
Sub=1:                 word     ACTIVATE_READ,2->next_sub
Sub=2&!Read_Done:      word     ACTIVATE_READ,2->next_sub
Sub=2&Read_Done:       word     PC_TO_ALU,ALU_ADD_ONE,3->next_sub
Sub=3:                 word     LOAD_PC,DECODE->Next_State
                       end

State=DECODE:          begin    0->next_sub
IType=ADD_INST:        word     ADD->Next_State
IType=SUB_INST:        word     SUB->Next_State
                       …
                       end

State=ADD:             begin    ADD->Next_State
Sub=0:                 word     {begin fetching operands}
…
                       end
```

Figure 10. Part of a Microprocessor's Control.

In Figure 10, begin-end blocks are used to group the wordlines for each state, and to supply common conditions and OR-plane specifications. Note that each wordline must supply a next state and a next sub-state which are then passed back into the PLA to control the selection of the next wordline. The condition to test for the current state is placed on the begin statement to propagate it to all wordlines in the state. For states that require sequential control, the default assignment to the "Next_State" output is to remain in the same state. This, of course, can be overridden on the "word" statement. The example of Figure 10 contains several predefined commands and constants whose definitions are not shown.

Note how the substate is used to sequence the operations for instruction fetch. Substate 0 moves the PC to the Memory Address Register. Next, the READ operation is activated and the control waits (in substate 2) until the "Read_Done" signal becomes true. Then the instruction is moved from the Memory Data Register into the Instruction Register, the PC is updated, and the control moves to the decoding phase.

In the decoding phase an n-way branch is performed to move to the proper state for the instruction being executed. Since all states that contain sequential sub-states begin with sub-state zero, the begin statement is used to propagate the value zero to all wordlines. Remember that all wordlines in this state are evaluated simultaneously, and no sequencing of operations is implied. Also note that the value of the sub-state is "don't care" in the DECODE state.

In the state "ADD," the operands of the instruction are fetched, and the proper micro-operations to complete the execution of the instruction are performed. In an actual microprocessor, operands would probably be fetched by a separate operand-fetch state.

4. Simulation

The PLA compiler translates a high-level PLA specification into a collection of wordline specifications. Each wordline specification contains an AND-plane specification in the form "01xx100," where the "x" represents "don't care," and a binary OR-plane specification. These wordline specifications can then be used either for simulation or for automatic layout. We have investigated two different algorithms for simulating PLA's at the high level. In both algorithms the OR-plane specifications are stored in a linear array. The difference in the algorithms is in the way AND-plane specifications are handled.

The first algorithm is a "vertical" algorithm that tests every wordline to see if it is selected, and ORs the OR-plane specifications of the selected wordlines to create the output. The second algorithm is a "horizontal" algorithm that tests the first bit of every wordline, then the second bit, and so forth. The running time of the first algorithm is a linear function of the number of wordlines, while the running time of the second is a linear function of the number of AND-plane bits in a single wordline. A choice can be made between the algorithms based on the shape of the AND-plane. (At the present time, the second algorithm is still under development, so the automatic choice of algorithm is not yet operational.)

The vertical algorithm stores AND-plane specifications as two binary arrays, the mask array and the value array. In the value array, all bits specified as 1 or 0 retain their values, but those bits specified as "don't care" are set to zero. In the mask array, all bits that are specified as 1 or 0 are set to 1, while all bits that are specified as "don't care" are set to zero. Thus if the AND-plane specification is "01xx100," the value array is contains "0100100," and the mask array contains "1100111." When the PLA is simulated, all inputs are collected into a single binary field which is the same width as the mask and value arrays. When a wordline is tested, the mask-array element is ANDed with the input field to zero out all "don't care" positions, and the result is compared with the value array. If the modified input field is equal to the value-array element, the wordline is selected, and the corresponding OR-plane specification is ORed into the result field. (The result field is initialized to zero.) Since any number of wordlines can be selected simultaneously, each wordline must be tested for each set of inputs.

The "horizontal" algorithm stores the AND-plane data as a tree. Before the tree is created, the list of wordlines is sorted into ascending order based on the value of the AND-

plane specifications. The sort operation uses the the ordering 0<x<1 for bit values, so that for the first bit, all wordlines with a specification of 0 come first, and all wordlines with a specification of 1 come last. After the sort operation, any wordlines with identical AND-plane specifications are merged by ORing their OR-plane specifications together. The collection of wordlines is traversed from left to right, and a tree such as that pictured in Figure 11 is created.

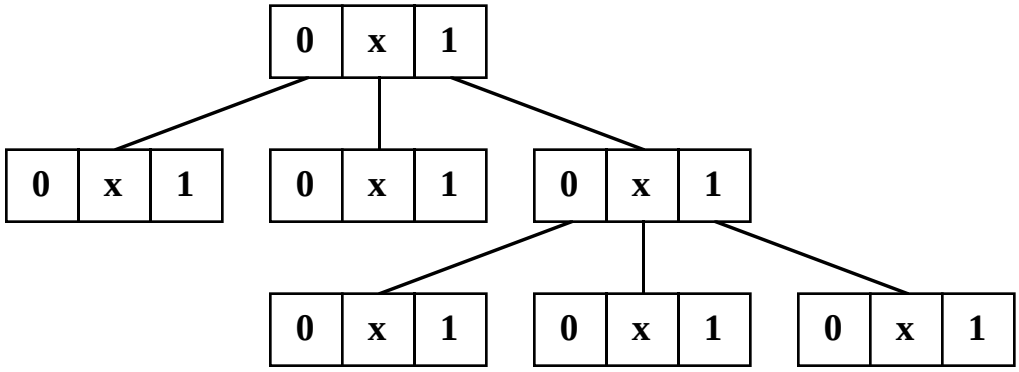

Figure 11. The Search Tree for the Horizontal Algorithm.

In Figure 11, each wordline corresponds to a path from the root to a leaf, and each level in the tree corresponds to a bit position in the input. (Note that there is only one path from the root to a particular leaf.) Leaves may be at many different levels, because AND-plane specifications that end with a string of "don't cares" cause the tree to be truncated at that point. Since the number of leaves cannot exceed the number of wordlines, the total number of vertices in the tree cannot exceed twice the number of wordlines. The leaves of the tree contain the index values of the corresponding OR-plane specifications. A First-In-First-Out queue is used to process a set of input values. Each queue element contains a bit position as well as a pointer to a vertex of the tree. Initially only the root of the tree is placed on the queue. When an element is removed from the queue, the appropriate bit is tested, and one or more successors of the current vertex are added to the queue. If the input bit is 1, the 1 and x successors are added, otherwise the 0 and x successors are added. It is possible for any of the three successors to be null, indicating that there is no AND-plane specification corresponding to that path. Null successors are never added to the queue. When a leaf is removed from the queue, the corresponding wordline is selected and the OR-plane specification is ORed into the output field. (As before, the output field is initialized to zero.) There is a considerable amount of room for optimizing this algorithm. For example, the algorithm may run more efficiently if the AND-plane bits are processed in

other than left-to-right order. Furthermore, it may be more efficient in some cases to test several bits simultaneously rather than one bit at a time. We are currently investigating these problems.

In most cases a PLA is simulated along with other hardware. The details of the simulator scheduling algorithms are beyond the scope of this paper, but in general the simulation of a PLA can be scheduled as if it were a simple gate such as an AND or an OR.

5. Automatic Layout

Once a PLA has been simulated and completely verified, it can be laid out automatically. This operation makes use of the Mpla program, which is part of the Berkeley MAGIC package[4]. The details of Mpla's operation are available elsewhere, but in an effort to make this paper self-contained, an overview of its operation is presented here. The layout of a PLA can be divided into several standard cells which can be abutted to create the PLA. Figure 12 illustrates the arrangement of some of these cells.

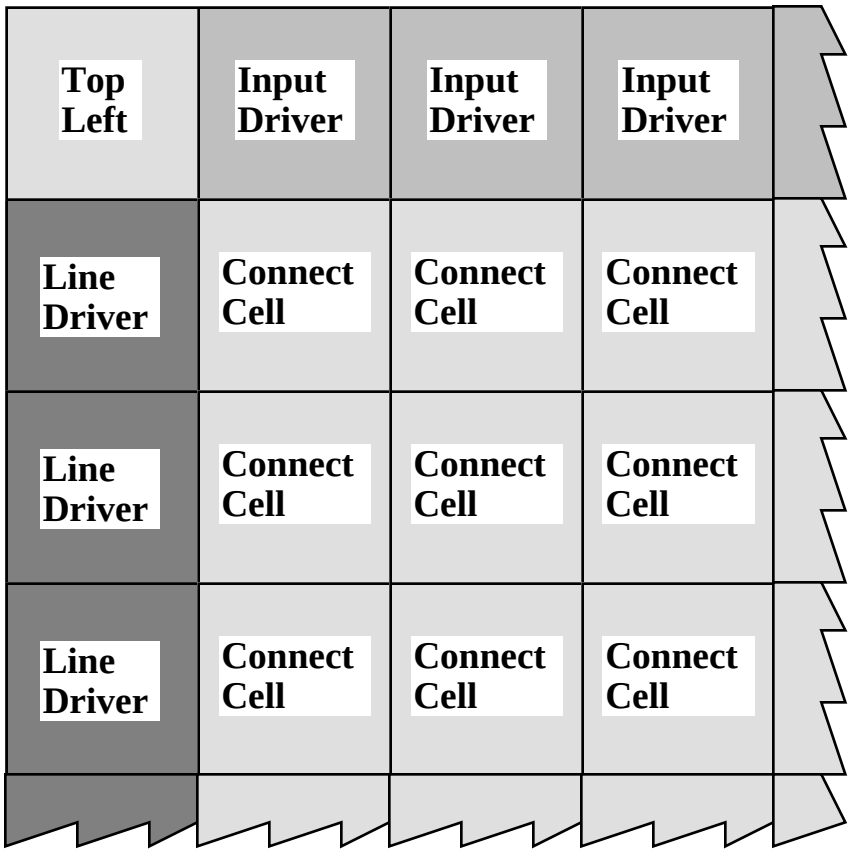

Figure 12. PLA Layout Format.

Although the arrangement pictured in Figure 12 is overly simplistic, it gives the flavor of the technique used by Mpla to create a circuit. A number of pre-defined cells are used to "tile the plane" with a layout of the PLA. In many cases, the content of the PLA determines the type of cell that is placed in a certain location. For example, there are three different "connect cells" that can be used, depending on whether a wordline is connected to the complemented input, the uncomplemented input, or unconnected.

The organization of the Mpla code is pictured in Figure 13. Because there was a nice separation between the code for parsing the input, and the code for creating the layout, it was relatively simple to replace the parser with the FHDL front end, as pictured in Figure 14.

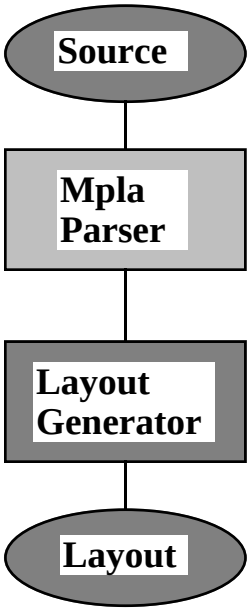

Figure 14. The Structure of Mpla.

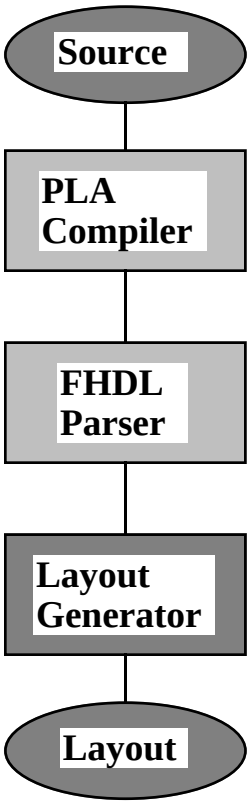

Figure 14. The Structure of the FHDL PLA Layout Generator.

The program structure illustrated in Figure 14 is essentially identical to that used to generate simulators. The only difference is that the layout generator is replaced with a simulator generator. The FHDL front end is currently being integrated with other Berkeley MAGIC tools which perform PLA minimization and folding.

6. Conclusion

The FHDL PLA tools provide a powerful method for describing, simulating, and automatically laying out PLAs. However, the PLA tools are more than just a design aid for VLSI projects. The PLA tools are a basis upon which future research in PLA synthesis and optimization will be built. In the short term, the tools will serve as a common front end to existing PLA software, but in the long term they will provide the core of a set of tools completely developed at the University of South Florida. There are two aspects to this research. The first is discovery of new techniques for optimizing and laying out PLAs. The second is using PLAs as the target for higher-level synthesis algorithms. Although much work has been done on PLA optimization, the fundamental problem is NP-complete

and must be approached heuristically. Because it is almost always possible to discover new and better heuristics for a problem, there is still room for much research in this area.

Although the creation of high-level synthesis algorithms is limited only by the creativity of the researcher, there are well defined areas that have arisen from existing research. The most important area is synthesis of state machines, either from a specific description of the machine[5] or from algorithmic specifications[6]. One segment of this research attempts to minimize PLAs by choosing an appropriate state assignment[7]. The FHDL PLA tools will serve as a basis for future research of this type.

Finally, one cannot ignore the pedagogical uses of the PLA tools. Fabrication of PLAs in the laboratory can be quite time consuming and very expensive, which precludes extensive experimentation with PLA-based designs in lower-level hardware courses. However, specification and simulation of PLAs is quite simple using the FHDL PLA tools, which permits students to become familiar with PLA-based designs without the expense of actually fabricating them. If one were to ignore all other aspects of these tools, their usage in improving the quality of hardware design courses is sufficient justification for their existence.

REFERENCES

1. C. Mead, L. Conway, *Introduction to VLSI Systems*, Addison-Wesley Publishing Company, Reading, Mass, 1980.

2. R. K. Brayton, G. D. Hachtel, C. T. McMullen, A. L. Sangiovanni-Vincentelli, *Logic Minimization Algorithms for VLSI Synthesis*, Kluwer Academic Publishers, Boston, 1984.

3. P. M. Maurer, Z. Wang, C. Morency, A. Tokuta, N. Bahte, "The Florida Hardware Design Language," *Proceedings of Southeastcon-90*, to Appear, Apr, 1990.

4. 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.

5. C-J. Tseng, A. M. Prabhu, C. Li, Z. Mehmood, M. M. Tong, "A Versatile Finite State Machine Synthesizer," *1986 IEEE International Conference on Computer Aided Design*, pp. 206-209, 1986.

6. S. Hayati, A. Parker, "Automatic Production of Controller Specifications from Control and Timing Behavioral Descriptions," *Proceedings of the 26th Design Automation Conference*, pp. 75-80, 1989.

7. T. Villa, A. Sangiovanni-Vincentelli, "NOVA: State Assignment of Finite State Machines for Optimal Two-Level Logic Implementations," *Proceedings of the 26th Design Automation Conference*, pp. 327-332, 1989.