THE SEMI-AUTOMATIC GENERATION OF PROCESSING ELEMENT CONTROL PATHS FOR HIGHLY PARALLEL MACHINES.

CUCS-127-84

Theodore M. Sabety
Brian Mathies
David Elliot Shaw

Department of Computer Science
Columbia University
(212) 280-8100

The Semi-Automatic Generation of Processing Element Control Paths for Highly Parallel Machines

Abstract:

This paper describes a recently implemented program that very rapidly generates control paths for different versions of the constituent processing elements of a particular massively parallel machine, the NCN-VCN Supercomputer. The program, called PLATO, accepts as input a set of instruction opcodes, together with associated control information, and produces as output a functionally correct, highly area-efficient set of PLA's for the processing elements. Among the novel aspects of the program are the use of a channel routing algorithm to generate a Weinberger Array layout for the PLA and the generation, from a single input description, of different PLA variants corresponding to processing elements serving different functions within the machine.

By supporting the extremely rapid generation of processing elements with different instruction sets, PLATO facilitates "rapid turnaround" architectural experimentation of a sort that has previously proven impractical. Use of the program has already yielded major area and performance improvements in the NCN-VCN processing element. Many of the techniques employed in the PLATO system should prove applicable to the semi-automatic layout of processing elements for other multiprocessor machines.

All appropriate organizational approvals for the publication of this paper have been obtained. If accepted, the authors will prepare the final manuscript in time for inclusion in the conference proceedings and will present the paper at the conference.

Theodore M. Sabety

Table of Contents

1. The NON-VON Supercomputer
2. Types of Processing Elements
3. Design Goals for PLATO:
4. The PLATO Input File
5. Automatic Weinberger Array Layout:
6. Channel Routing Algorithm
7. Conclusion

1. The NON-VON Supercomputer

NON-VON [1] is a massively parallel non-von Neumann supercomputer, portions of which are now under construction at Columbia. While the machine has evolved over the past several years, its most important elements remain the same. All versions of the machine contain a primary processing subsystem (PPS), implemented using custom nMOS VLSI circuits, and a secondary processing subsystem (SPS), based on a set of "intelligent" disk drives.

The PPS comprises a large number (perhaps as many as a million) processing elements (PE's), and is constructed from custom nMOS VLSI chips, each containing a number (eight, at present) PE's. The PPS is organized to form a binary tree of PE's. In all but the latest version of the machine (NON-VON 4, which will not be discussed further in this paper), a single control processor is attached to the root of the PPS tree. The control processor broadcasts instructions which are executed simulaneously by all PE's in the PPS. (NON-VON 4 incorporates a number of processors, each capable of serving as a control processor for some subtree in the PPS; these "large processing elements" are interconnected by a high-bandwidth interconnection network.) Figure 1 provides a description of the PPS.

Our first prototype, called NON-VON 1, was designed using largely ad hoc methods. Our principal goals in constructing the NON-VON 1 prototype were to validate the essential architectural principles of the NON-VON design, to measure the area and aspect ratios of various silicon structures incorporated within the PE's, and to perform certain electrical measurements on the completed chips. For this reason, little attention was given to either area- or time-optimization in the NON-VON 1 prototype chip. The NON-VON 1 PPS chip has now been completed, fabricated, and tested through DARPA's MOSIS system, and appears at present to be fully functional.

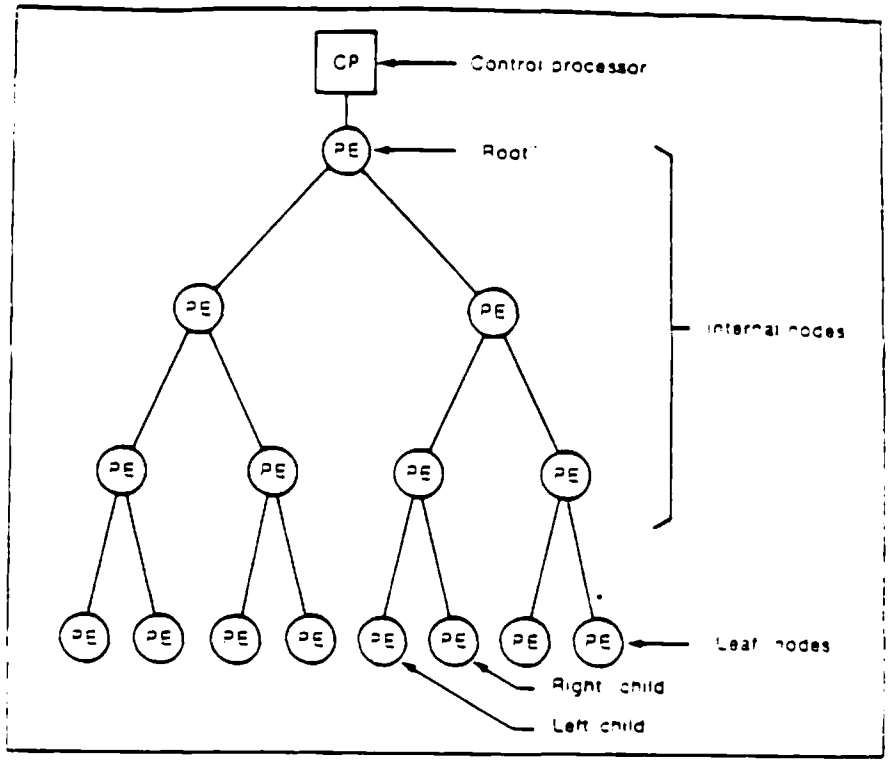


FIGURE 1. NON-VON primary processing system.

A second prototype, NON-VON 3, is now under development. (The name NON-VON 2 was assigned to an interesting architectural exercise that we do not currently plan to carry beyond the "paper-and-pencil" stage, although its central ideas may well influence future NON-VON designs.) NON-VON 3 will be similar in most respects to the original NON-VON 1 design, but is expected to incorporate a number of improvements suggested by the results of our initial experiments in chip design and software development. In particular, the NON-VON 3 SPE will feature:

1. An area-efficient eight-bit ALU to replace the one-bit ALU incorporated in the prototype NON-VON 1 SPE chip.

2. Fewer local registers, based on NON-VON 1 area measurements and software simulation results.

3. A far better floor plan, formulated using precise measurements taken from the prototype chip.

4. A generalization of certain NON-VON 1 instructions to support the more efficient execution of many common instruction sequences.

5. Less silicon area devoted to control path logic.

Our plans called for the NON-VON 3 instruction set to be closely based on, and with few exceptions, more general than the one employed in NON-VON 1. (Some of the additions we plan to incorporate in fact correspond to commonly used macros in our existing NON-VON 1 software.) It was also deemed important that all existing NON-VON 1 software be simply and mechanically translatable into NON-VON 3 instructions, so that none of our work to date would be lost. (Translated programs would take advantage of some, but not all of NON-VON 3's enhancements.) In the future, of course, NON-VON 3 software will be written using NON-VON 3 instructions, allowing the exploitation of all of these features.

Early in the development cycle of NON-VON 3, it was recognized that the successful accomplishment of these ambitious area and performance goals would be greatly accelerated by the availability of a highly automated system for the specification, design, layout and testing of the constituent processing elements. To be useful, such a system would have to rapidly and reliably generate "correct" layouts, allowing the user to experiment with alternative processing element architectures with the confidence that the resulting layout would in fact faithfully realize the more abstractly specified design. Within such a semi-automatic development environment, changes in the instruction set might be realized in hardware in a fraction of the time that would otherwise be required, facilitating extensive experimentation with and "fine tuning" of the PE architecture.

2. Types of Processing Elements

With minor exceptions, all PE's in the PPS tree are physically identical. Those differences that do exist are based on the manner in which communication among adjacent PE's is accomodated. These differences in communication regime

define four classes of processing elements:

1. Leaf nodes that are left child of some node
2. Leaf nodes that are right child of some node
3. Internal nodes that are left child of some node
4. Internal nodes that are the right child of some node

By convention, the root is considered an internal node and a left child. Each type of PE must be aware of its type so that it can properly participate in the communication instructions that operate between the PE's. For example, when executing a SEND LEFT CHILD instruction, each left child in the tree must latch data into an I/O register as well as send data to its left child; the right children only send data and do not latch any incoming data.

One possible technique used to differentiate between the types of PE's would be to encode the PE type on two control lines that enter the PLA at the inputs to the AND-plane. In this scheme, one wire would distinguish between left and right children, while the other would distinguish between leaves and internal nodes. When the individual chips were wired together to form a complete PPS, these inputs would be permanently wired to the appropriate constant logic values to "bind" the type of each PE. The disadvantage of this approach, however, is that each PE would have to contain a considerable amount of logic that would never be used, resulting in a waste of silicon area. To save silicon "real estate", PLATO generates a different PLA for each type of PE, each containing only that logic which is relevant to a PE of that type.

While the particular set of processor classes enumerated above, along with their associated communication semantics, are specific to NON-VON-like tree-structured machines, the presence of "boundary conditions" distinguishing various classes of processing elements is common to most parallel architectures. In parallel machines configured as a linear array, for example, three types of PE's (leftmost, rightmost, and intermediate) may be

defined. Machines based on the orthogonal mesh, on the other hand, may require as many as nine PE types (central PE's, north, south, east and west PE's, and the four corner PE's).

3. Design Goals for PLATO:

Several goals were formulated when work began on the PLATO program:

1. The engineer should be able to define the PLA's for all PE types with a single high-level definition.
2. The program should produce the smallest possible PLA consistent with a given set of VLSI design rules.
3. The program should be integrated with all other layout and simulation tools employed for PE design.
4. The program should execute with absolutely no intervention by the design engineer.

The last goal is intended to minimize the possibility of errors introduced by the human user, insuring that all layouts correctly realize the intended high-level function, and need not be extracted and simulated before insertion in the PE layout.

4. The PLATO Input File

Among the advantages of the PLATO program is the fact that only one input file need be created to generate all four types of PLA's. The system makes use of mnemonic labels wherever possible to aid in the isolation of errors and to make it easier to identify PLA inputs and outputs in the finished layout. The same labels are used by a register transfer level simulator for NON-VON processing elements that is now being designed at Columbia, and which will interface directly with PLATO, as will be discussed shortly.

To use the PLATO system, the engineer compiles a list of instruction opcodes with appropriate state variable inputs, and for each opcode, a list of the control lines that must be excited to execute the instruction. The input file contains three types of commands:

1. Commands that define the input file format.
2. Commands that describe the placement of inputs and outputs in the layout.
3. Commands that describe the logical functionality of the array.

Figure 2 provides a simple example of a typical PLATO input file. The first command line assigns the names INPUT_1, INPUT_2, INPUT_3, AND INPUT_4 to the first four opcode (and, in general, state) bits that will be encountered in left-to-right order. The second command line specifies the order in which these opcode and state bits should enter the AND plane of the PLA (listed from the bottom to the top of the PLA, assuming that the bits enter the AND plane from the right). The third command line in the example file specifies the order in which the output lines of the array are to appear (listed from left to right with the wires leaving the PLA at the bottom).

Figure 2

```
/* This is an example of a PLATO input file. All comments are ignored. */

/* list of input labels for input file ordering */
INPUT_1, INPUT_2, INPUT_3, INPUT_4;

/* list of same labels, but for PLA ordering */
INPUT_4, INPUT_2, INPUT_1, INPUT_3;

/* Then, a list of output labels are placed in the desired order. */
OUTPUT_1, OUTPUT_2, OUTPUT_3;

/* Mnemonics --- Opcode Bits ---  List of Control Lines */
MOV_A_B                 0000    OUTPUT_1, OUTPUT_2;
MOV_A_C                 0101    OUTPUT_2, OUTPUT_3;
ADD_A_B                 1101    OUTPUT_1, OUTPUT_3;
SUB_A_B                 1111    OUTPUT_3;
```

The rest of the file specifies the decoding function of the PLA. To specify how an instruction is to be decoded, the engineer makes a list consisting of every instruction opcode, together with all appropriate state variable inputs and a list of the control lines that must be excited to execute the

instruction properly. The processor's state machine is represented similar The present state is considered one of the input bits and the next state is defined as if each bit in the state machine were a control line.

The PLATO input file is easier to use than a truth table because "don't care conditions are considered acceptable logic values for input bits. This facilitates the separation of state machine specification from instruction execution specification because the next-state information need not be included in each instruction decoding command line. This separation contributes greatly to PLATO's ease of use and tends to minimize the number user errors. PLATO converts this input file format into a truth table format which is then used by such other design aids as logic minimization programs.

The sample input file presented in Figure 2 shows the specification of four instructions. The MOV_A_B instruction, which causes the contents of the A register to be transferred to register B, is executed by asserting two control lines: OUTPUT_1 and OUTPUT_2. In this example, the control line OUTPUT_1 would be the "read A" register control line and the OUTPUT_2 control line would be the "write B" register control line. Any number of control lines may be specified: in the case of the subtract instruction, only one control line is asserted.

Two extra input bits are represented in the input file for the NCN-VCN processing element: the "leaf/not-leaf" and "left-child/not-left-child" lines that were discussed earlier. These two bits are analyzed by the PLATO program upon scanning of the input file and are used to separate the single input file into four truth tables, each representing the function of one of the corresponding types of PLA. Figure 3 shows a small piece of the NCN-VCN 3 PLATO input file. Typically, this file would have more lines than the number of opcodes in the instruction set. The present NCN-VCN 3 PLATO input file, for example, has 147 lines.

Figure 3:

```
/* ******************   NON-VON 3  PLA PLATO inputs  ******************

/* format:  (lf/nl), mneumonic, IR0-IR7, S0, S1, EN1, Is, LC, Ctrl-lines
/* input lines into PLA (source) */
/* IR reg */     IR0, IR1, IR2, IR3, IR4, IR5, IR6, IR7,
/* PLA */        S0, S1,
/* PE */         EN1, Is, LC;

/* same input lines, but ordered for PLA */
                S1, S0, IR0, IR1, IR2, IR3, IR4, IR5, IR6, IR7,
                EN1, LC, Is;

/* Ctrl lines driven by PLA (dest) */

/* dpth */  RD1RAM8, WR1RAM8, WR1RAM1, RD1RAM1, RD1MAR, RD1EN1, SETEN1,
            WR1EN1, WR1MAR, ROTR, WR1B8, WR2B1, WR1B1, ROTL, RD1B1,
            RD1B8, WR1A8, WR2A1, WR1A1, RD1A1, RD1A8, ADD, SUB, LOGICAL,
            WR2C1, WR2C8, WR1C8, WR1C1, RD1C1, RD1C8, WR2IO8, WR1IO8,
            WR2IO1, WR1IO1, RD1IO1, RD2IO1, RD1IO8, RD2IO8,
/* RESOLVE */
            KCA, KCB,
/* IO */     DR0, DR1, DR2, IO0, IO1, IO2, IO3, IO4, IO5, IO6, IO7,
 /* PLA */  S0next, S1next;


/* (0): nl  non-leaf  */
/* (1): lf  leaf      */
/* (X): don't care    */


    general         XXXXXXXX XXX1X IO6;
       "            0XXXXXXX 10X1X S0next;
       "            11X1XXXX 10X0X S0next,S1next;
       "            10X00XXX 00XXX IO0,IO5,IO6;

    MOV8_A8_B8      00000001 101XX WR1B8,RD1A8;


       "                "      100XX RD1A8;
    ADD_IO8         01010001 100XX RD1IO8 ADD;
       "                "      101XX WR2C1,WR2C8,RD1IO8 ADD;
    COMPARE_IO8     010101X1 100XX RD1IO8 SUB;
       "                "      101XX WR2A1,WR2B1,RD1IO8 SUB;
    ENABLE          011101XX 10XXX SETEN1;
    BROADCAST8_A8   10000000 101XX WR1A8,IO0,IO5,IO7,DR0;
       "                "      100XX IO0,IO5,IO7,DR0;
    REPORT8_MAR     10001011 101XX RD1MAR,IO0,IO5,IO7,DR1,DR2;
    SEND8_RC_B8     10010001 100XX RD1B8,IO0,IO2,IO4,DR0;
       "                "      101X0 RD1B8,WR2IO8,IO0,IO2,IO4,DR0;
       "                "      101X1 RD1B8,IO0,IO2,IO4,DR0;
    RESOLVE         11100XXX 10XXX KCA,IO0,IO5,IO7,DR1,DR2;
       "                "      11XXX KCB,IO0,IO5,IO7,DR1,DR2;
       "                "      00XXX DR0;
```

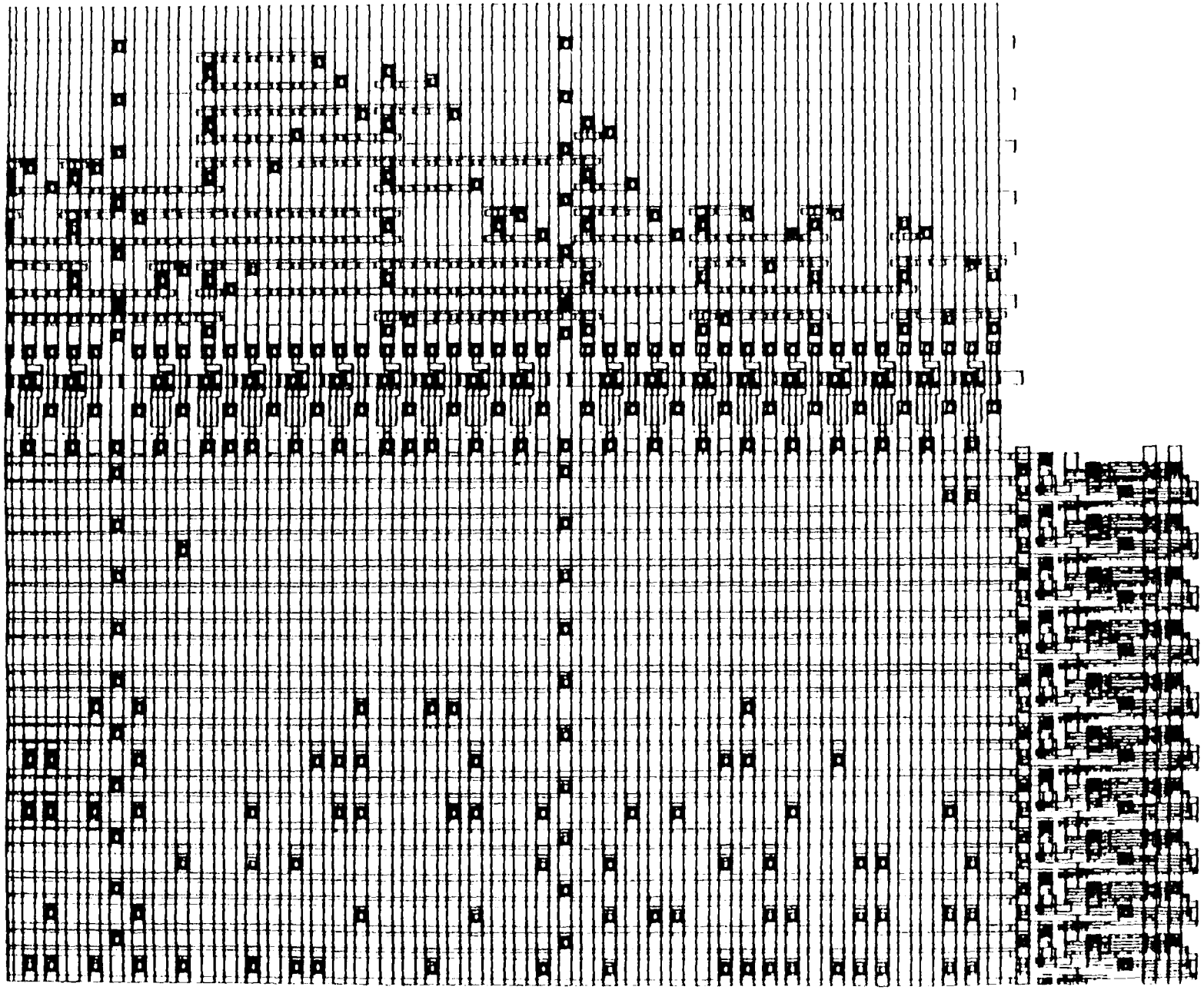
5. Automatic Weinberger Array Layout:

In order to achieve an efficient use of silicon area, PLATO generates a PLA array using a variation on the Weinberger Array [3] layout technique. In a Weinberger Array, the highly regular structure of a conventional PLA is compressed into a functionally equivalent, but smaller form. The resulting layout is less regular, and conceptually more complex, than a traditional PLA.

By way of background, an ordinary PLA layout consists of an AND-plane and an OR-plane. The AND-plane comprises a set of regularly spaced columns incorporating logic gates capable of generating the logical conjunctions of its inputs. In the context of the processing element application, those inputs are the instruction opcodes. The OR-plane is constructed similarly from a set of regularly spaced gating elements that are used to generate the logical disjunction of the outputs of the AND-plane gates. The OR-plane is rotated 90 degrees from the orientation of the AND plane, allowing the outputs of the AND-plane to connect to the inputs of the OR-plane.

In constructing most processing elements of the kind used in highly parallel machines, the population of transistors in the AND-plane far exceeds the that in the OR-plane. For this reason, a considerable amount of silicon area is typically wasted when a conventional PLA is used to realize the control path logic in such a processing element. The Weinberger Array is capable of providing significant area savings in such applications.

This technique uses an conventional array structure for the AND-plane, but obviates the need for a full OR-plane. An example of a Weinberger PLA generated by PLATO is provided in Figure 4. The AND-plane is shown on the bottom and the OR-plane on top. The instruction opcode bits enter the AND-plane from the right. Control lines exit the entire array from the bottom. The columns in the AND-plane feed into the top and make contact with wires that run horizontally in polysilicon. These wires extend only as far as is required for them to form the gates of all transistors in the OR-plane that



F=ps: <mathies>nonleafpla.bst.1 (10 November 1983 16: 10: 14) S=main WO= (1250,

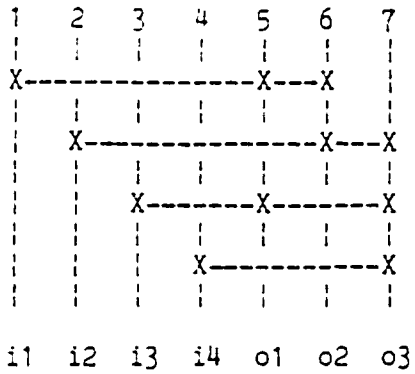
require the particular result being carried by the wire. If the layout is designed appropriately, several different wires can often share a single track in the OR-plane. Compaction is achieved through the shared use of tracks; careful placement of AND-plane columns yields a layout with a minimal or near minimal number of tracks.

The authors are not aware of any earlier PLA generation tools that generate Weinberger Array layouts automatically. Typically, the layout engineer must manufacture the Weinberger array by hand. PLATO, on the other hand, applies a channel-routing algorithm (described in the next section) to automatically specify the wiring of the Weinberger array. Unlike the usual channel routing problem, one end of the wire in the channel is connected to an AND-plane output while the other is the gate of a transistor in the OR-plane. The automatic Weinberger array layout algorithm incorporated in PLATO has successfully produced a PLA for NCN-VCN 3 that is approximately 25% smaller than the corresponding one produced using conventional PLA generation techniques.

In the rare instances in which two wires in the array share the same track and have transistor gates at the ends that meet, the highly compact AND-plane layout primitives used by PLATO can result in design rule violations in the Weiberger array. PLATO detects these cases and automatically provides extra room in the array to resolve each conflict as illustrated in Figure 5. Empirically, however, such cases have been found to occur quite infrequently. Out of a total of 208 AND-plane columns in a NCN-VCN 3 PLA, for example, only 4 incorporate extra space to avoid design rule violations. PLATO's "management by exception" approach thus permits effective compaction of the control path without the introduction of design rule errors.

Figure 5

Its initial net-representation: 4 Tracks



6. Channel Routing Algorithm

The channel routing algorithm consists of two basic phases: The first phase sets up a data structure that represents the placement of columns in the AND-plane with control lines that leave the OR-plane and are routed through the AND-plane. The second phase permutes this data structure, changing the relative positions of all of the columns in the AND-plane in such a way as to produce an OR-plane with a minimum number of tracks, and hence the least waste of silicon area.

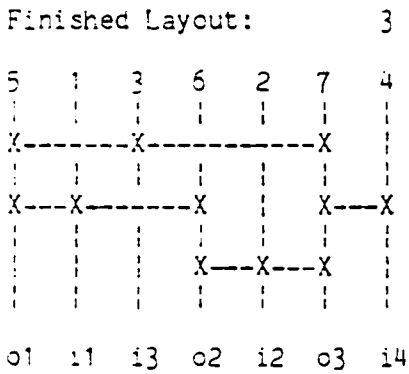
The initial form of the data structure, called an ordering, is generated from the minimized truth table that represents the function to be realized by the PLA. The ordering is a list of the desired AND-plane results appended to a list of the control line outputs, enumerated in the order in which they are to appear in the layout. Each AND result is generated by a column in the AND-plane. The AND columns may be permuted at will, but the control line columns must retain the order specified by the user.

The ordering is augmented by a net-list representation of the OR-plane. Like the AND-plane, this net-list is initially generated from the truth table. Figure 5 shows an example of this data structure. Along the bottom, the

labels i1 through i4 represent AND columns and the labels o1 through o3 represent control line columns. The rest of the figure shows a typical net-list that represents a connection scheme between the inputs and the outputs of the OR-plane. At this stage, whether a certain connection between a horizontal wire and a vertical column is a contact or a transistor gate is immaterial to the problem of minimizing the number of tracks in the array.

Once the initial setup is completed, a depth-first search for a connection scheme with the least number of tracks is performed. Figure 6 shows the result of applying this algorithm to the net-list depicted in Figure 5. Note that the order of the control line columns is preserved, although their positions have changed. The ordering of the AND plane columns has been successfully permuted to allow a connection scheme of only three tracks, the best possible result for this example. At this point, PLATO determines whether a connection is a contact between a wire and a column or a transistor gate.

Figure 6



PLATO generates the actual layout description, expressed in Caltech Intermediate Form (CIF), in two stages. First, the AND-plane is generated from by placing the layout primitives in those positions described by the portion of the truth table. In the second stage, the OR-plane is constr

by generating primitives in positions specified by the net-list. The final layout is produced after labels are attached to appropriate places in the layout.

7. Conclusion

The PLATO tool employs three techniques to minimize the area required for the control paths of processing elements for highly parallel machines:

1. The generation of control paths through the automatic generation, using a channel-routing algorithm, of Weinberger Arrays.
2. The automatic generation of multiple types of PLA adapted to the distinct types of PLA's incorporated in different PE's.
3. The use of highly compact layout primitives, together with an automatic procedure for resolving any resulting design rule violations.

Based on area comparisons between the NON-VON 1 and NON-VON 3 PE layouts, it appears that each of these three techniques has proven responsible for an area reduction on the order of 25%. The novel techniques embodied in the PLATO system have thus been responsible in large part for our ability to embed a number of processing elements in one PPS chip, which is the one of the essential cornerstones of the NON-VON approach to massively parallel computation.

References

1. D.E. Shaw ,"The NON-VON Supercomputer" , Columbia Computer Science [illegible] , August, 1982

2. Carver Mead and Lynn Conway, Introduction to VLSI c. 1980 Addison-Wesley Publishing Co. Reading, Mass.

3. A. Weinberger, "Large-scaled integration of MOS complex logic: A layout Method", IEEE J. Solid-State Circuits, vol. SC-2, pp. 182-190,Dec., [illegible]