Sadiq M. Sait, Farooqui AA•The architecture of a highly reconfigurable RISC dataflow array processor •INTERNATIONAL JOURNAL OF ELECTRONICS 83 (4): 493-518 OCT 1997

Automatic Weinberger array synthesis from UAHPL description

S. M. SAIT† and F. A. AL-KHULAIWI†

A Weinberger array (WA) (Weinberger 1967) synthesis system is described that automatically generates WAs for combinational logic circuits modelled in Universal Hardware Programming Language (UAHPL) (Masud and Sait 1986). The system also minimizes the area required by the WA by performing row compaction. An algorithm similar to that used for channel routing is employed for compaction (Hashimoto and Stevens 1971). This convenient tool for designing combinational logic circuits models at a high level of abstraction and much of the procedure is automated.

1. Introduction

Automatic design of the layouts of digital systems from descriptions in high-level language is an attractive approach, especially when the number of transistors per chip is large. Structured implementation of digital systems in nMOS is preferable for its regularity, the ease of mapping to layouts, and testing. Several design automation (DA) systems have been designed keeping these points as the primary objectives (Ayres 1983, Kang and van Cleemput 1981, Sait 1987).

UAHPL has been used to model digital systems and synthesize them in nMOS technology. Final implementations have used path programmable logic arrays (PPLA) (Olson 1984), storage logic arrays (SLAs) (Hill *et al.* 1984) and also full custom VLSI design (Sait 1987). In this paper we present the automatic synthesis of layouts for all-NOR nMOS Weinberger arrays (WAs) (Weinberger 1967) for combinational circuits modelled in UAHPL (Masud 1981). The UAHPL DA (Masud and Sait 1986) system is used to generate the interconnection list, which is the mapping of the modelled circuit. The list is converted to an all-NOR representation, optimized to delete any redundant gates and then mapped to a WA. A new row compaction procedure for row area sharing is devised to reduce the area required by the array. As a result of compaction, average savings of 50–90% have been obtained. The final output of the WA synthesis system is the stick diagram and layout data for fabrication in nMOS technology.

The next section discusses the advantages of the Weinberger array and the language UAHPL. Section 3 gives an overview of the WA generator system. The heuristics and algorithms used in the generation and compaction of the WA, and their implementation details, are discussed in §4. Results and conclusions are presented in §5.

2. Weinberger arrays and UAHPL

This section gives a brief overview of WAs and features of the UAHPL language.

Received July 1989; accepted July 1989.

† Department of Computer Engineering, KFUPM Box. No. 673, Dhahran 31261, Saudi Arabia.

0020-7217/90 $3.00

2.1. *Weinberger arrays*

A Weinberger array is a logic array similar to a programmable logic array (PLA). However, it differs from a PLA in that it consists of only one plane. Weinberger arrays can take any form, one of which is the NOR structure in which there is a single rectangular plane consisting of only NOR gates (and inverters).

Weinberger arrays are an alternative to PLAs as a method of implementing combinational logic circuits. They have certain advantages over the PLAs in that the area does not generally grow with the size of the problem as fast as that required for PLAs does. If the final implementation is in nMOS technology, then large PLAs require large widths for power and ground lines (in metal); this is to avoid the problem of metal migration (Mead and Conway 1980). The other advantage of WAs is that unlike PLAs they accommodate forms of logic other than the standard two-level sum of products (SOP) logic. Functions expressed as an arbitrarily deep SOP, that is, of multi-level circuits, can be easily realized using WAs.

2.2. *UAHPL*

Universal Hardware Programming Language is an extension of AHPL (Hill and Peterson 1978). It is a register transfer-level language that allows one to specify many low-level details for efficient implementation of digital systems in MOS technology (Sait 1987). Large iterative circuits such as artithmetic logic units (ALUs) can be conveniently expressed as a combinational logic circuit (CLU). The language has been implemented by means of a multistage compiler which supports a wide spectrum of design activities including testing (Chiang *et al.* 1982), simulation (Alsharif 1983) and VLSI mask generation (Sait 1987).

An example of a UAHPL model for a simple circuit is given in Fig. 1. The description of the UAHPL model is as follows. Registers and flip-flops are declared as MEMORY. Other declarations in this example are external inputs (EXINPUTS) and a CLU called SAMCKT. CLUs are used to describe both simple and iterative combinational circuits. Examples include circuits of adders, decoders, etc. For the purpose of simplicity we use a trivial example.

The declaration part is followed by the procedural part, which consists of numbered steps defining the state sequential machine. The number of steps is equal to

```
MODULE: EXAMPLE1.
  MEMORY :  AC1[2];AC2[1].
  EXINPUTS : CLOCK ; RESET.
  CLUNITS  : CKT[2] <: SAMCKT<. 1 .>.

 BODY
  SEQUENCE    : CLOCK.
  1  AC1 <= CKT(AC2,AC2,AC2);
     => (1).
  ENDSEQUENCE
  CONTROLRESET (RESET) / (1).
END.

CLU:SAMCKT(X) <. N .>.
 INPUTS: X[2*N+1].
 OUTPUTS: Y[N+1].
  BODY
  Y[0] = (X[0]@X[1]);
  Y[1] = (X[1]@X[2]).
  END.
END.
```

Figure 1. UAHPL model of example 1.

the number of states in the finite state machine. This part begins with the statement BODY SEQUENCE: CLOCK and is terminated by the keyword ENDSEQUENCE. Each step in the procedural part consists of a step number followed by a register transfer statement (←), and/or connection statements (=), and, if needed, by a branch (→) statement. Each pulse on signal line CLOCK advances the circuit from one step to the next. UAHPL permits the usual operators such as AND (&), OR (+), EXOR (@), NOT (−) and concatenation (,).

The identifier following the delimiter '<:' in the CLU gives the generic name of the CLU. If the CLU has already been compiled it is not necessary to describe it: otherwise it must be described. The number enclosed in curly brackets is used as a parameter whose use is explained later. The generic CLU description is used as a template by the compiler to generate copies of the combinational circuit. UAHPL uses an Algol-like syntax for CLU description and is very suitable for describing iterative networks. Another involved example of a CLU that uses the iterative construct is given in Fig. 2.

The parameter in curly brackets in the CLU description allows the compiler to generate CLUs of various sizes from the same template. In Fig. 2 the parameter 'I' is used to specify the bit size of the INCR (incrementer) unit. Referring to Fig. 2, to do the incrementing the least significant bit is complemented and the other output bits are obtained by EXclusive ORing the corresponding input bit with the logical AND of all the lower significant input bits. FOR and IF statements are compiler directives whereby the compiler generates the iterative network at compile time. In this paper the UAHPL description of CLUs only is mapped to NOR WAs.

3. Overview of the system

This section provides an overview of the system. We briefly discuss the UAHPL compiler and net-list generator, the conversion of the net-list representation to an all-NOR circuit, optimization of the converted circuit to delete redundant inverters and gates and the generation of the WA personality.

3.1. *UAHPL compiler and net-list*

The task of generating WAs from UAHPL descriptions is carried out in stages. Tools available in the UAHPL DA system are also used (Masud and Sait 1986). The first stage of the DA system performs syntax and semantic analysis of the model and decomposes it into tables which are used by subsequent stages. The task of the second stage is to produce a logic-level design of the system in terms of flip-flops,

```
CLU:INCR{I) <. N .>.
  "I-BIT INCREMENTER CONSTRUCTED WITH EXCLUSIVE OR GATES"
  "AND GATES & AN INVERTER"
  INPUTS : X{I}.
  OUTPUTS: Y{I}.
   BODY
        FOR J=(I-1) TO 0 STEP -1
         CONSTRUCT
          IF J=I-1 THEN Y{J}=¬X{J}
            ELSE Y{J}=X{J}@(&/X{J+1:I-1})
          FI
        ROF.
   END. "INCR"
```

Figure 2. UAHPL model of CLU for I-bit incrementer.

various logic gates and their interconnection. The design is stored in a net-list (Masud 1981).

Figure 3 gives a partial net-list for the CLU in the UAHPL model given in Fig. 1. It comprises the gatelist and the IOlist. Figure 4 gives the corresponding logic circuit.

The gatelist is stored in a matrix data structure under the name COLS. The three fields of COLS are GATE#, GATE-TYPE AND I-LINK. I-LINK is a pointer to the IOlist. Whence inputs to the gates are obtained. The IOlist is stored in another matrix called CONNECTION, and this also has three fields. The first field in the IOlist is a sequence number and is not stored. The first two of the next three fields have the GATE#, which is the number of the gate connected to the input, and the third field is a pointer to the matrix itself, which is null if the fan-in is less than two. The complete circuit can be constructed and manipulated using COLS and CONNECTION. The next sections will describe the process of mapping the interconnection list provided by the second stage of the UAHPL compiler to efficient NOR Weinberger arrays for fabrication in nMOS technology.

GATE #	GATE TYPE	ILINK
1	INPUT	1
2	INPUT	3
3	INPUT	4
4	OUTPUT	9
5	OUTPUT	13
6	EXOR	6
7	EXOR	11

Figure 3(*a*). Partial gatelist for SAMCKT in Fig. 1.

	Gate#	Gate#	Next Pointer
1	102	0	0
2	1	2	5
3	102	0	0
4	102	0	0
5	3	0	0
6	2	1	0
7	6	7	0
8	6	0	0
9	6	0	0
10	4	0	0
11	3	2	0
12	7	0	0
13	7	0	0
14	5	0	0
15	4	129	0
16	140	0	0

Figure 3(*b*). Partial IOlist for SAMCKT in Fig. 1.

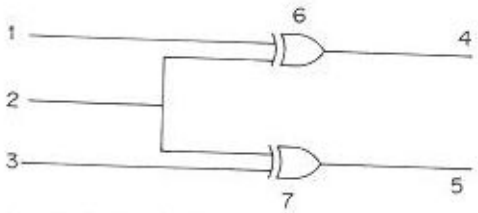

Figure 4. Logic circuit of SAMCKT in Fig. 1.

3.2. *Conversion of net-list to NOR gates*

The linked list shown in Fig. 3 can be manipulated to meet the demands or recommendations of the target technology. When modelling and designing digital hardware one often uses more than the minimal number of logical element types than are absolutely necessary. This is convenient because it is tedious to work in terms of a minimal set. At the same time, only a minimal set may be available or recommended for actually building the circuit. There is, then, a need for automatic translation from the form preferred by the designer to that required for implementation. The designer may prefer to think in terms of logical operators like AND, NAND, OR, etc., while the basic hardware module preferred may be NOR.

Each gate used in the circuit contains a record in the gatelist. Since we want to synthesize the NOR Weinberger array, it is required to convert all the gates in the gatelist to NOR gates. In order to convert the logical circuit to all-NOR, the gates in the gatelist must be replaced by NOR gates, and additional inverters (also here called one-input NORs) must be added at the input or output of the NOR gate to maintain the functionality of the circuit. As an example, to convert AND gates to NOR gates the record in the gatelist containing the AND gate is replaced by a NOR gate, and additional inverters required at the input of this gate are added as additional records in the gatelist. The I/O pointers in the IOlist corresponding to the input/output connections of the gates are updated. In order to convert the EXOR gate to NOR gates the equivalent circuit shown in Fig. 5, containing only NOR gates replaces the EXOR gate, and thus four additional records are added to the gatelist for each EXOR gate. The modified gatelist and IOlist are given in Figs 6(*a*) and 6(*b*), respectively. The all-NOR SAMCKT is shown in Fig. 7.

3.3. *Deleting redundant inverters*

The net-list provided by the Stage-2 UAHPL compiler is already optimized. Details of the optimization algorithms are found in work by Masud (1981). After conversion of the circuit to all-NOR, additional inverters may appear in series/parallel with the inverters already present in the circuit. These are removed by a simple optimizing routine and the corresponding records are dropped from the gatelist. The function of the optimizing routine to remove the redundant gates in the converted NOR list may be stated as follows.

In the process of converting the circuit to NOR gates and inverters the circuit may have two points providing the same logic signal at all times. For example points '*p*' and '*q*' and points '*r*' and '*s*' in Fig. 8(*b*). These points are merged and the redundancy in the circuit is removed.

The optimized NOR circuit is now mapped into a Weinberger personality as explained in the next section.

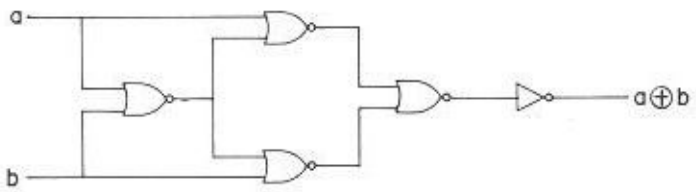

Figure 5. NOR equivalent of EXOR gate.

GATE #	GATE TYPE	ILINK
1	4025	x
2	4025	x
3	4025	x
4	4026	11
5	4026	12
6	INV1	1
7	INV2	2
8	NOR3	3
9	NOR4	4
10	NOR5	5
11	NOR6	6
12	NOR7	7
13	NOR8	8
14	NOR9	9
15	NOR10	10

Figure 6(*a*). All-NOR gatelist from Fig. 3a.

	Gate#	Gate#	Next Pointer
1	11	0	0
2	15	0	0
3	1	2	0
4	8	2	0
5	1	8	0
6	9	10	0
7	2	3	0
8	3	12	0
9	2	12	0
10	13	14	0
11	6	0	0
12	7	0	0

Figure 6(*b*). IOlist from Fig. 3(*a*) for all-NOR circuit.

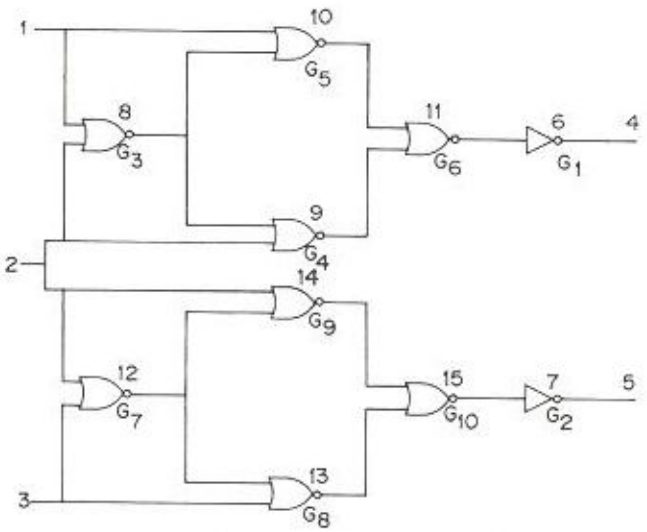

Figure 7. NOR equivalent of SAMCKT.

3.4. *NOR Weinberger array generator*

To specify the exact function of the layout in a succinct way, we define a personality matrix for the Weinberger array. The array is represented as a matrix which contains $-$, $+$, and *. A '+' means that there is a transistor at the intersection of a column and a row. A '$-$' means there is no transistor at the intersection of a

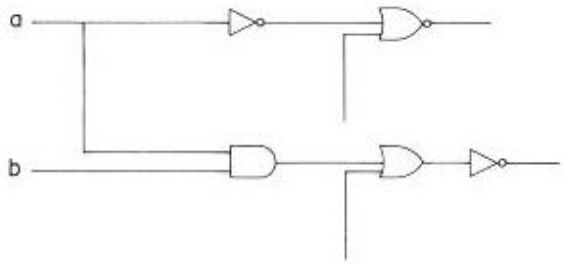

Figure 8(*a*). Sample circuit.

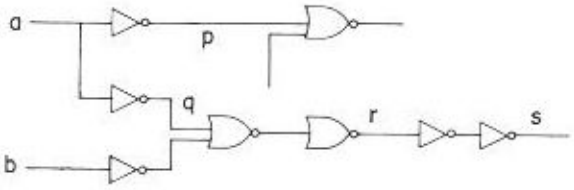

Figure 8(*b*). Redundant all-NOR sample circuit.

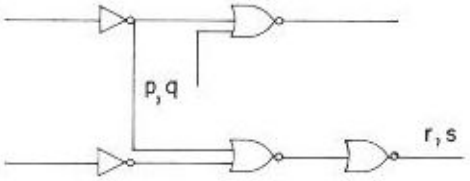

Figure 8(*c*). Optimized all-NOR sample circuit.

column and a row, and a '*' means that there is a contact-cut (sometimes called an *internal input*).

Mapping the net-list to the Weinberger personality is fairly straightforward. In a Weinberger array the number of columns equals the number of NOR gates and inverters in the circuit. The number of rows is equal to the number of columns plus the number of external inputs.

A row is assigned for each column (NOR gate) and for each external input. The net-list is then scanned and '+'s (for transistors) are inserted in the Weinberger personality depending on where the gates receive the inputs. Similarly a '*' is placed in the column that generates the internal input (indicating the contact cut in the array). A '—' will indicate the absence of a contact-cut or transistor in the personality. The personality of the circuit in Fig. 7 is shown in Fig. 9. The stick diagram for the corresponding nMOS transistor circuit is given in Fig. 10. In Fig. 10 rows a1, a2, a3 are for external inputs, 01 and 02 are the outputs and the rest of the rows are for internal inputs.

The personality matrix of the generated WA is sparse, since only a few transistors and contact cuts are used in large arrays. The next step is to optimize the array such that the area used is reduced and at the same time the structure and advantages of the WAs are retained. One method is to perform column folding, where a column can share the circuitry of two columns in the unfolded array. In this case the maximum reduction in area is only 50% of the original area. In this paper

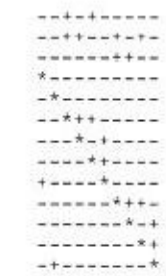

Figure 9. Weinberger personality for SAMCKT.

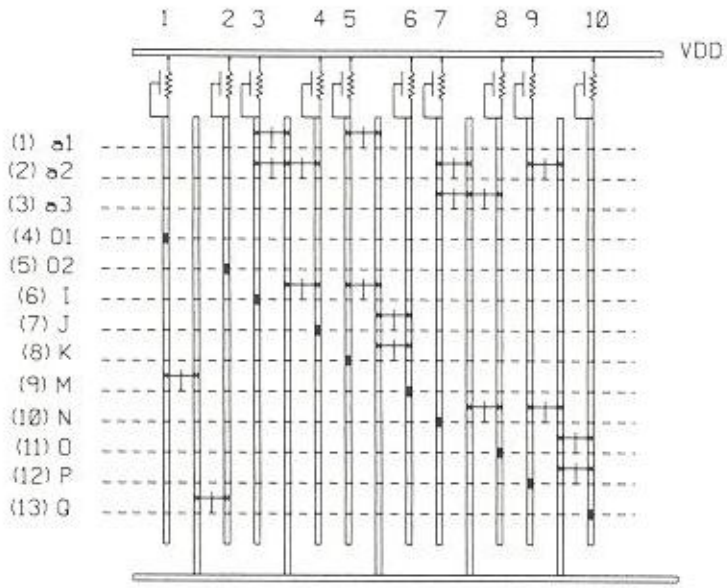

Figure 10. Stick diagram for NOR WA of SAMCKT.

we present an efficient row compaction algorithm that gives much better results than column folding. The next sections discuss the heuristics and algorithms used in area reduction by row compaction of the WAs.

4. Heuristics, algorithms and implementation details

The optimizing algorithms for row compaction consist of two steps: (*a*) A column-ordering heuristic, and (*b*) a row compaction algorithm. This is similar to the channel routing algorithm due to Hashimoto and Stevens (1971) without the vertical constraint.

4.1. *Column-ordering algorithm*

As stated earlier, in the WA, for each column of the array there is a corresponding row. This row carries the signal from the output of that column to the other columns in the array. In nMOS this horizontal row runs in polysilicon and feeds the inputs of the NOR gates of the succeeding columns. The polysilicon wire and the

transistors to which it inputs in a particular row form the circuit of that row. However, not the entire area of the row may contain transistors, contacts or connection lines. In row compaction, the area of a single row may be used to implement the circuits used in several rows of the uncompacted array, thus reducing the overall area. The efficiency of row compaction is largely dependent on the relative position of the columns. This is because the *effective size* of a row will be the distance between the extreme transistors/contacts on that row, for a given ordering of columns, and reordering may affect/change this size. The *effective row* is that part of the row that has a transistor or contact at its extreme ends and no transistor or contact cut beyond the effective part. For example in Fig. 10 the effective size of row No. 1 (a1) is between columns 3 and 5. Similarly, in row No. 13 (Q) the effective size is from columns 2 to 10.

The ordering of a column is similar to the assignment problem. A heuristic ORDER, explained below, is used to reduce the complexity. This is only a sketch of the algorithm: more details can be obtained from Fig. 11.

Logic function can be divided into levels. The logical net-list is a mapping of the circuit, and the columns in the initial ordered WA are the ones corresponding to the gates in the circuit.

Step 1. Sequentially scan the initial ordered list of columns and remove any gates (columns) that receive external inputs.

Step 2. In the process of scanning,
(2.1) if any gate appears whose input gates have already been removed, or
(2.2) if any output gate appears, then,
remove the gate and update the initial list.

Step 3. Continue the above steps until the end of the initial ordered list is reached.

```
Procedure ORDER(order1,order2,COLS, CONNECTION)
/*order1: Initial order of columns */
/*order2: Final order, the result */
Declare SEL, mark : Boolean
BEGIN
   NC←# of columns
   j←1, mark[i]←False, 1 ≤ i ≤ NC
      Repeat
         For i←1 To NC DO
         Begin
            if mark[order1(i)]=False then
            Begin
               SEL←True
                  For each input K of COLS(order1(i),2)DO
                  Begin
                     SEL←SEL.AND.mark[order1(k)]
                  End
                  If NOT SEL then
                  Begin
                     order2(j)←order1(i)
                     j←j+1, mark[order1(i)]←True
                  End
            End
         End
      Until (j > NC)
ENDORDER
```

Figure 11. Pseudo code for ORDER algorithm.

Step 4. Go to the beginning of the updated initial list and repeat Steps 2 and 3 until the initial list is empty.

The order in which the gates are removed from the initial ordered list is the new order of columns in the WA. The procedure called ORDER that gives the new ordering of columns is given in Fig. 11. The new order for the columns in Fig. 10 is as follows.

Initial order: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10

Final order: 3, 4, 5, 6, 7, 8, 9, 10, 1, 2

From Fig. 7 note that because gate No. 3 receives external inputs its column is chosen first. Then gate No. 4 receives inputs from gate No. 3 (already placed) and external inputs, and so it comes next. The process continues until gates Nos. 5 and 6 are placed. The next gate in sequence is No. 7, which is chosen, and this continues until the end of the initial ordered list. Gates Nos. 1 and 2 were not chosen in the first scan because the initial ordered list is scanned sequentially. In the second pass gate No. 1 is chosen because its input (from gate No. 6) was placed in the previous pass, and so is gate No. 2.

4.2. *Row compaction algorithm*

In order to compact the array the problem is now considered as a channel routing problem (Hashimoto and Stevens 1971), where the columns that mark the extreme ends of the effective row are the nets, and the effective rows themselves are the tracks of the uncompacted array. Unlike the situation for the channel routing problem, since the pull-ups are only on the top there is no vertical constraint on the tracks, only a horizontal constraint. The algorithm for compaction is given in Fig. 12.

```
Procedure COMPAC(order2, ROWS)
Declare placed : ARRAY[1....n] of Boolean
BEGIN
   NR←# of rows
   A_s←set of all columns
   placed[i]←False, 1 ≤ i ≤ NR
      For all rows in ROWS DO
      Begin
         find S_i, I_i
      End
      A←A_s
      For i←1 To NR DO
      Begin
         A←A_s
         For j←i+1 To NR DO
         Begin
            if A∩S_j=S_j  then
            Begin
               put row j in row i
               placed[j]←True
               A←A-(S_j∪I_j)
            End
         End
         A←A_s
      End
ENDCOMPAC
```

Figure 12. Pseudo code for COMPAC algorithm.

In Fig. 12 the data structure ROWS consists of one field. This field is a pointer to the CONNECTION data elements. It will give the columns for each row that have either a transistor or a contact cut. For example, ROWS(1) will point to CONNECTION, which will contain the elements 3, 5. Similarly ROWS(13) will point to CONNECTION, which will contain 2, 10.

The algorithm in Fig. 12 is best explained with the help of an example. Consider the Weinberger array shown in Fig. 10. As a first step the sets S_i for each row 'i' are built. These sets will contain the numbers of the columns in which row 'i' has a transistor or a contact cut. For example: $S_1 = \{3, 5\}$, $S_2 = \{3, 5, 7, 9\}$, $S_3 = \{7, 8\}$, $S_4 = \{1\}$, $S_5 = \{2\}$, $S_6 = \{3, 4, 5\}$, $S_1 = \{4, 6\}$, $S_8 = \{5, 6\}$, $S_9 = \{1, 6\}$, $S_{10} = \{7, 8, 9\}$, $S_{11} = \{8, 10\}$, $S_{12} = \{9, 10\}$, $S_{13} = \{2, 10\}$.

As the next step the rows are sorted; first the rows corresponding to the inputs are considered. The rest of the rows are taken in the order of their ordered columns. Hence the new order of rows for this example will be:

1, 2, 3, 6, 7, 8, 9, 10, 11, 12, 13, 4, 5. (1, 2, 3 correspond to the inputs, row No. 6 corresponds to column No. 3, row No. 7 to column No. 4, and so on.)

Let A_0 be the universal set of all columns in the ordered WA. In the example under discussion $A_0 = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$. We also define a set I_j which, out of the array of ordered columns, has the number of those columns in the effective row 'j' that have no transistors or contact; for example $I_1 = \{4\}$, $I_2 = \{5, 6, 8\}$, $I_7 = \{5\}$, $I_9 = \{7, 8, 9, 10\}$, $I_{11} = \{9\}$, $I_{13} = \{1\}$, and the rest of the sets are $\varnothing$.

Then the sets $(S \cup I)_k = S_k \cup I_k$ are computed for all k rows. The $\{S \cup I\}$ sets for the inputs are $\{S \cup I\}_1 = \{3, 4, 5\}$, $\{S \cup I\}_2 = \{3, 4, 5, 6, 7, 8, 9\}$, $\{S \cup I\}_3 = \{7, 8\}$. The same sets for the other rows, taken in the order of the ordered columns, are: $\{S \cup I\}_6 = \{3, 4, 5\}$, $\{S \cup I\}_7 = \{4, 5, 6\}$, $\{S \cup I\}_8 = \{5, 6\}$, $\{S \cup I\}_9 = \{1, 6\}$, $\{S \cup I\}_{10} = \{7, 8, 9\}$, $\{S \cup I\}_{11} = \{8, 9\ 10\}$, $S \cup I\}_{12} = \{9, 10\}$, $\{S \cup I\}_{13} = \{1, 2, 10\}$, $\{S \cup I\}_4 = \{1\}$, and $\{S \cup I\}_5 = \{2\}$.

The procedure for compaction can now be summarized as follows.

Step 1. Initialize set AA_0.

Step 2. For each row 'k', compute $A \leftarrow A = \{S \cup I\}_k$.

Step 3. Any other row 'j' that can share the empty area with row 'k' must satisfy two conditions:
(*a*) it must not be a row corresponding to another input of the circuit;
(*b*) it must satisfy the condition $A \cap \{S \cup I\}_j = \{S \cup I\}_j$.

If any row 'j' satisfies these criteria, then its position is moved to row 'k' and set A is updated. The above steps are repeated until no row can be merged with row k. Then the next row is taken and the procedure is repeated on the rows that have not already been merged.

As an example

$$A - \{S \cup I\}_1 = \{1, 2, 6, 7, 8, 9, 10\}$$

Row No. 9 can be merged with this row since $A \cap \{S \cup I\}_9 = \{1, 2, 6, 7, 8, 9, 10\} \cap \{1, 6\} = \{1, 6\}$.

The new set A becomes

$$A = A - \{S \cup I\}_9 = \{2, 7, 8, 9, 10\}$$

The next row that satisfies the criteria of merging is row No. 5, so row No. 5 is also moved to row No. 1. Note that row No. 3 also satisfies the second condition, but since it is an external input (and all inputs are to appear on the sides) it is not considered. Thus row No. 1 in the compacted array has rows 1, 9 and 5 of the uncompacted array. The compacted WA and the corresponding personality of Fig. 10 are given in Fig. 13. The uncompacted personality generated for the UAHPL model of Fig. 2 for $I = 3$ is given in Fig. 14 and the corresponding compacted personality is given in Fig. 15.

5. Results and conclusions

In this paper we have presented a methodology whereby optimal layouts of WAs can be generated from high-level descriptions in UAHPL. A new method to reduce the area of WA by row compaction has been decided. On an average a 50–90% reduction in the area of WA after compaction has been observed. The developed system is now part of the silicon compiler of the UAHPL DA system (Sait 1987). Several simple and iterative practical circuits have been modelled in UAHPL and the compacted WAs were generated automatically. The results of compaction for some circuits are shown in the Table. It is apparent from the pseudocode and algorithms that the complexity of the WA generator is $O(n^2)$, where n is the number of gates in the circuit. Since the final implementation is structured, mapping of the generated compacted personalities of WAs to nMOS layouts is straightforward.

```
NEW ORDER OF COLUMNS IS :-
        3:        4:        5:        6:        7:        8:
        9:       10:        1:        2:

+-+*----+*
++--+-+*-+
----++*+*-
*++ *++
 *-+ *-+
  *+
```

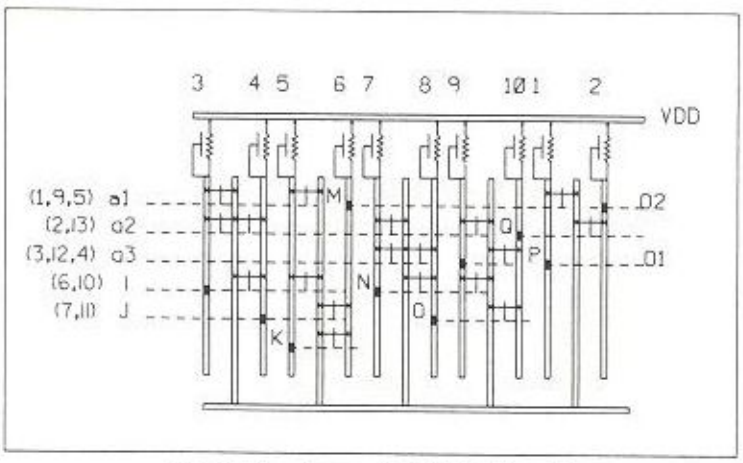

Figure 13. Compacted WA of Fig. 10.

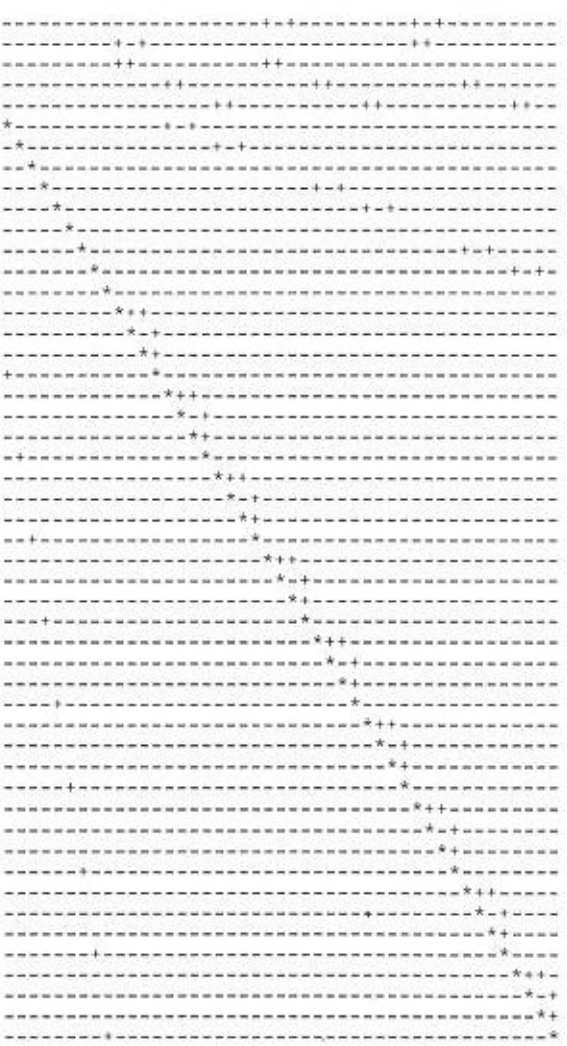

Figure 14. Uncompacted personality of Fig. 2, I = 3.

```
NEW ORDER OF COLUMNS IS :-
          10:          11:          12:          13:          22:
          24:          25:          34:          35:          36:
           1:           4:           7:          14:          15:
          17:          26:          27:          28:          29:
          39:          40:          41:           2:           5:
          18:          19:          20:          21:          30:
          32:          33:          42:          43:          44:
           3:           6:           9:
```

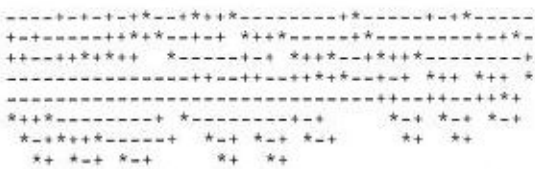

Figure 15. Compacted personality of Fig. 14.

Case	No. of cols	No. of rows before compaction	No. of rows after compaction	Area gained (%)
1	16	18	9	50
2	26	30	11	63
3	10	13	6	54
4	45	50	8	84
5	40	46	11	76

ACKNOWLEDGMENTS

The authors thank the King Fahd University of Petroleum and Minerals for support under COE465. They also thank Mr Ahmed Al-Sheikh and Mr Abdul Rashid Bhatti for help in drawing the figures.

REFERENCES

ALSHARIF, M. M., 1983, Functional level simulator for universal AHPL. M.S. thesis, University of Arizona.

AYRES, R. F., 1983, *VLSI: Silicon Compiler and the Art of Automatic Chip Design* (Englewood Cliffs, N.J.: Prentice Hall).

CHIANG, C. H., HILL, F., MOHSENI, A., and CHEN, D., 1982, Fault detection test generation at the register transfer level. *Proceedings of the I.E.E.E. First Annual Phoenix Conference on Computers and Communications*, pp. 58–63.

HASHIMOTO, A., and STEVENS, S., 1979, Wire routing by optimizing channel assignment within large apertures. *Proceedings of the Eighth Design Automation Conference*, 1978, pp. 155–169.

HILL, F. J., NAVABI, Z., CHIANG, C., CHEN, D., and MASUD, M., 1984, Hardware compilation from an RTL to a storage logic array target. *I.E.E.E. Transactions on CAD/ICAS*, **3**, pp. 208–217.

HILL, F. J., and PETERSON, G. R., 1973, *Digital Systems: Hardware Organization and Design* (New York: Wiley) (second edition published 1978).

KANG, S., and VAN CLEEMPUT, W. M., 1981, Automatic PLA synthesis from a DDL-P description. *18th Design Automation Conference*, pp. 391–397.

MASUD, M., 1981, Modular implementation of a digital hardware design automation system. Ph.D. dissertation, Department of Electrical Engineering, University of Arizona.

MASUD, M., and SAIT, S. M. 1986, Universal AHPL—a language for VLSI design automation. *I.E.E.E. Circuits and Devices magazine.*

MEAD, C., and CONWAY, L., 1980, *Introduction to VLSI Systems* (Addison Wesley).

OLSON, T. A., 1984, Automatic AHPL description to path programmable logic array. M.S. thesis, University of Arizona.

SAIT, S. M., 1987, VLSI mask generation from register transfer level descriptions: an automated approach. Ph.D. dissertation, Department of Electrical Engineering, KFUPM, Dhahran.

WEINBERGER, A., 1967, Large-scale integration of MOS complex logic: a layout method. *I.E.E.E. Journal of Solid State Circuits* **2**, 182–190.