Silicon Compilation - A Hierarchical Use of PLAs

Ron Ayres November 1978

1.0 Introduction

This paper proposes a way to *compile* a silicon layout directly from synchronous logic specification. The motivation for introducing *compilation* into the silicon world comes from its extreme success in the software world. As we see silicon area increasing and circuit complexity increasing, we might feel much in common with the early day programmers who faced increasing memory availability along with increasing program complexity.

Software and hardware revolve around the same basic concern: The *software* designer lays out a one dimensional array of *memory* whereas the integrated circuit designer lays out a two dimensional area of *silicon.* In each case, various constraints must be satisfied in order to obtain a working product. In addition, both efforts involve lots of modification.

Compilers are around to reduce the amount of specification required to obtain a working program. Custom programs can be *created* economically and confidently with the aid of compilers and programs can be *modified* quickly even though the resulting memory layout has to change drammatically. Producing custom VLSI requires these same conveniences. Where a software compiler revises addresses which reference relocated blocks of memory, a silicon compiler reroutes wires to relocated blocks of silicon. When changes are made to the logic specification for the chip, the compiler creates a new layout with all cells positioned and connected especially for the new logic specification.

This paper is divided as follows: First, synchronous logic specification and a silicon implementation for synchronous logic is described. Then, problems arising with large systems concerning both their specification and silicon implementation are cited and a solution is presented which both eases large synchronous logic specs and decreases silicon area. This particular solution introduces a flexible hierarchy into the specification of logic and accounts automatically for placement and wire routing. The ideas presented here are implemented currently as programs written in the language ICL which runs on a PDP-10.

2.0 Logic Specification for ICs

Synchronous logic is a language in which one specifies the function of an IC. In almost all IC designs, there exists at some time a synchronous logic specification. The synchronous logic spec provides means for simulating the IC before more detailed work proceeds. Synchronous logic is by no means

the most convenient high level language in which to describe ICs but it does provide a solid starting point. Other high level specification languages can translate their input into synchronous logic.

The language of synchronous logic admits any set of equations written in terms of boolean expressions and variables. In addition, synchronous logic admits the specification of a unit delay in time. For example, figure 1 shows the synchronous logic for a single bit of a resetable counter.

Any set of synchronous logic equations can be implemented in silicon via a PLA (Programmable Logic Array) coupled with some inverters and unit delay flipflops. Figure 2 shows a PLA which implements the counter-bit. A PLA by itself implements any set of logic equations (without delays) which can be written in terms of an OR of AND terms *devoid* of logical NOTs. Attach inverters to the inputs of a PLA and one can implement any set of logic equations. Finally, by adding unit delay flipflops onto the outputs, one can implement any set of synchronous logic equations. The unit delay flipflops are placed on the outputs of those equations which were specified with the $=_{next}$ as opposed to the $=$ by itself.

3.0 Hierarchical or Functional Logic Specification

Conceptual design of any IC involves partitioning the desired function into smaller functional units. The smaller functional units themselves may be partitioned *etc.* so that each of the smallest units can be implemented with confidence. For example, a frequency divider may be divided into one counter + one register. The counter can be decomposed further by defining an individual bit position for the counter.

Functional partitioning provides means to localize design concerns. Functional partitioning also alleviates the need to replicate equations for replicated components, e.g., the equations for one bit in a counter may be written once but referenced as a whole many times to form a full counter. Functional partitioning is old hat in software where programs are written in languages which support function definition and invocation.

Logic specification can be partitioned by introducing the concept of a *logic cell.* A logic cell appears to the outside world as a block box + and interface. An interface is a set of *named signals.* Logic cells are related together by writing equations which involve *signals,* each of which is specified by naming both a particular logic cell and a particular signal in that logic cell's interface. For example, figure 3 presents a definition of an N-bit counter in terms of logic cells each of which represents a single counter bit. The notation \S (looks like *apostrophe s*) is used to retrieve a signal given a logic cell and a name. Figure 4 is a copy of figure 3 with explanations superimposed.

The notation used in figure 3 is working program text which has been used to generate the illustrations for this article. The program text uses the name LOGIC_LEVEL as a synonym for *logic cell.* (LOGIC_LEVEL was intended to mean one LEVEL of LOGIC in a hierarchy).

A logic cell is a datastructure consisting of three fields:

EXTERNALS: *two sets of named pins (input and output)*
RELATIONS: *A set of synchronous logic equations which relate the inputs and outputs of both this logic cell and all of the subcells (GUTS).*
GUTS: *The set of subcells referenced by this cell.*

This hierarchical synchronous logic specification language has been used successfully to define an IC whose function is to drive an SCR to control the brightness of a lightbulb, parameterized by four kinds of instructions. The IC consists of two 6-bit registers, a 6-bit and a 2-bit shift register, a 6-bit and a 3-bit counter, and a flipflop. The logic specification was easy to write and it has been simulated successfully.

4.0 Hierarchical PLAs to Implement Hierarchical Logic Specification

To simulate the hierarchical logic specification, a program smashes the hierarchy and creates one long list of equations. A standard synchronous logic simulator takes it from there. Likewise, a single, giant PLA can be constructed automatically from the long list of equations and therefore provide a silicon implementation.

Unfortunately, a PLA can take an inappropriately large amount of area; the area equals approximately the product of the number of input terms and the number of AND terms. PLA area can be saved when the designer notices that his PLA implements actually several *independent* sets of equations. He can substitute the one large PLA with several smaller PLAs. In the best case, the designer can cut PLA area by a factor of k by implementing k independent sets of equations as k small PLAs instead of one large PLA.

Rather than removing the hierarchy from the logic specification prior to PLA generation, we can let the hierarchy *work for us.* Figure 5 shows a PLA which implements a 16-bit counter and figure 6 shows a hierarchical use of PLAs to implement the same. The subPLAs, the PLAs generated by each of the subcells, are lined up with all their inputs and outputs facing upwards. The RELATIONS of the current logic cell are themselves translated to a PLA and placed on the righthand side of the picture. Finally, wires are placed horizontally above the subcells. These wires transmit signals between the RELATIONS PLA and each of the subcells and the connection points (EXTERNALS) of this logic cell.

This hierarchical use of PLAs as the following principle working in its favor: Local signals are represented locally in silicon. That is, signals relevant only to a subcell remain inside that subcell and do not enter PLAs of other subcells or enclosing cells. In addition, a good functional partitioning minimizes the number of input and output signals. (Notice that software functions generally take in and produce small numbers of parameters). Therefore, the RELATIONS PLA will have in general a minimal number of input and output lines, and therefore a nearly minimal area. With this setup, PLAs will *always* relate inputs and outputs of functional units.

The shape of a layout will depend not only upon the specific logic equations, but also upon the chosen hierachy. Figure 7 shows another layout for the 16-bit counter which differs only in its functional partitioning. The hierarchy for figure 7 is one level deeper; it consists of four subcells where each subcell itself is a four-bit counter generated by calling COUNTER(4). The RELATIONS for the new top level connect the four counters in series. Figure 8 shows still another 16-bit counter; this hierarchy is five levels deep and each level relates exactly two instances of the level immediately underneith. Even though the different layouts have differing areas, a layout not having minimal area might be chosen merely because it has the right shape for a slot in a larger chip. This program makes no such choices, however. The user can modify his hierarchy to obtain the shape he wants.

The layout for a logic cell always comes out with its interface on the lefthand edge. A cell is utilized as a subcell by rotating it 90 degrees so as to get its interface facing upwards.

This program provides interconnect between the PLAs it generates. The interconnect generator accepts two sets of fixed pins among which it will provide interconnect. These two sets of fixed pins lie horizontally on the bottom and vertically on the right, e.g., the interface pins of the subcells on one hand and the pins of the RELATIONS PLA on the other. In addition, the interconnect generator accepts a set of movable pins which will reside on the left and which will be the interface for this level in the hierarchy. The interconnect generator fixes the positions of the movable pins as it creates horizontal and vertical wires between the two sets of fixed pins and the one set of movable pins.

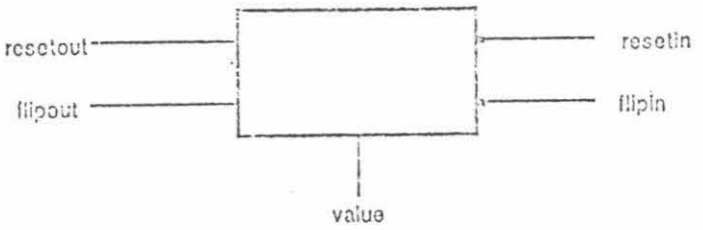
5.0 Some Optimizations

The reader may notice that with the counter shown above, the RELATIONS PLA is trivial, i.e., it represents no logic computation. This RELATIONS PLA serves merely to route signals between the subcells and the counter's interface. The equations in the RELATIONS component of the counter specification contain no logic in fact; they all have the form $A = B$.

Figures 9, 10, and 11 are copies of figures 6 thru 8 where the *trivial* equations are removed. That is, equations of the form $A=B$ have been removed and the signal A has been *indirected* to the signal B so that whenever signal A shows up in a computation, signal B appears instead.

6.0 Conclusions

The continual increase in silicon area invites compilation because there is some space for overhead and because our bigger ciruits are getting complex enough to *require* computer assistance like that required by large software systems. Modifications will need to be made more readily and with confident results. In addition, even if silicon compilation is doomed to require too much area, there are still lots of smaller custom ICs which are needed in short order.



resetout = resetin
flipout = flipin & value

$value_{next}$ = (IF flipin THEN ~value ELSE value) & ~resetin

COUNTER-BIT

```
DEFINE  COUNTER_BIT= LOGIC_LEVEL:     BEGIN  VAR RESET_IN,RESET_OUT,FLIP_IN,FLIP_OUT,VALUE=PIN;

        DO      RESET_IN:=      NEW_BIT;
                RESET_OUT:=     NEW_BIT;
                FLIP_IN:=       NEW_BIT;
                FLIP_OUT:=      NEW_BIT;
                VALUE:=         NEW_BIT;
        GIVE     EXTERNALS:     [IN_PIN:      { RESET_IN \NAMED 'RESET_IN';
                                               ' FLIP_IN  \NAMED  FLIP_IN  }

                                OUT_PINS:     { RESET_OUT \NAMED 'RESET_OUT';
                                                FLIP_OUT  \NAMED 'FLIP_OUT';
                                                VALUE     \NAMED  VALUE'  }  ]

                 RELATIONS:    {  {     RESET_OUT \EQU RESET_IN;
(Equations) ------------------>  {       FLIP_OUT  \EQU  (FLIP_IN \AND VALUE);
                                 {       VALUE     \NEXT  CIF:  FLIP_IN  THEN: NOT(VALUE) ELSE:VALUE] \AND NOT(RESET_IN)
                              ]  {       END    ENDDEFN
```

ICL TEXT

FIGURE 1

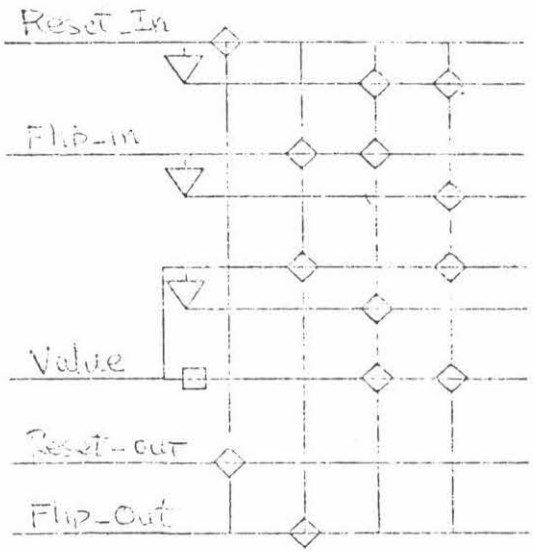
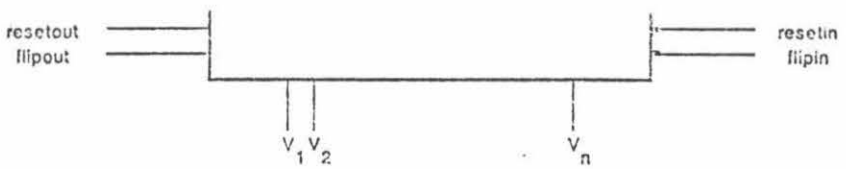


FIGURE 2



LET X = COUNTER-BIT [1 .. N]

resetin of X[i] = resetout of X[i+1]
flipin of X[i] = flipout of X[i+1]
i =1 to n - 1

resetin of X[n] = resetin
flipin of X[n] = flipin

resetout = resetout of X[1]
flipout = flipout of X[1]

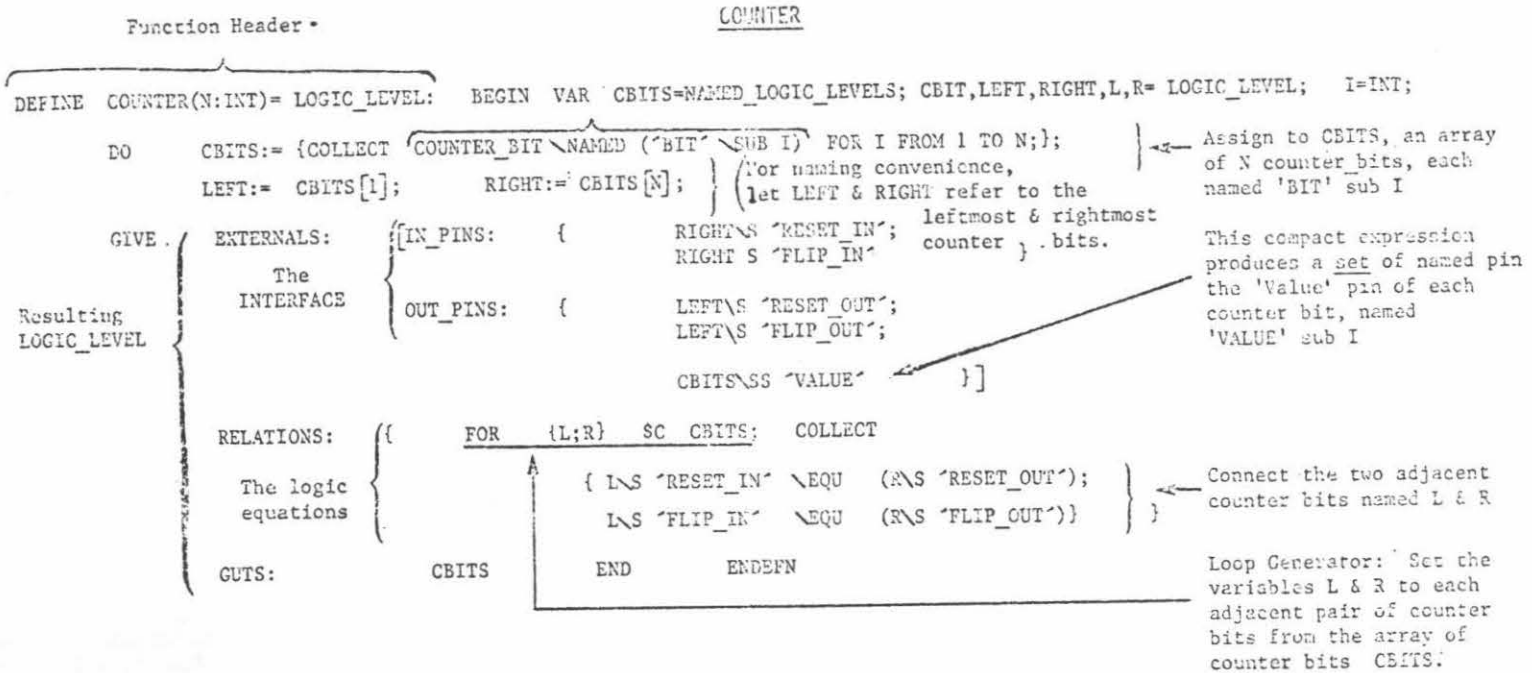
V[i] = value of X[i] for i =1 to n

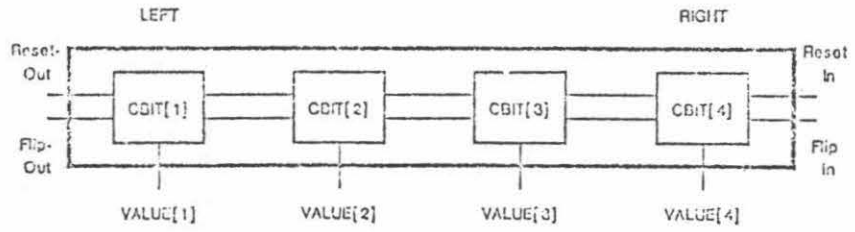
COUNTER

```
DEFINE  COUNTER(N:INT)= .LOGIC_LEVEL:   BEGIN  VAR  CBITS=NAMED_LOGIC_LEVELS; CBIT,LEFT,RIGHT,L,R=LOGIC_LEVEL;  I=INT;

        DO      CBITS:= (COLLECT   COUNTER_BIT \NAMED ('BIT' \SUB I)  FOR I FROM 1 TO N;);

                LEFT:=  CBITS[1];        RIGHT:= CBITS[N];

        GIVE    [EXTERNALS:      [IN_PINS:       {          RIGHT\S 'RESET_IN';
                                                            RIGHT\S 'FLIP_IN'     }

                                 OUT_PINS:       {          LEFT\S 'RESET_OUT';
                                                            LEFT\S 'FLIP_OUT';

                                                            CBITS\SS 'VALUE'       }]

                RELATIONS:              FOR {L;R} $C CBITS;      COLLECT

                                              { L\S 'RESET_IN'     \EQU  (R\S 'RESET_OUT');

                                                L\S 'FLIP_IN'      \EQU  (R\S 'FLIP_OUT')}      }

                GUTS:             CBITS ]      END     ENDDEFN
```

ICL TEXT

FIGURE 3


COUNTER
Function Header
DEFINE COUNTER(N:INT)= LOGIC_LEVEL: BEGIN VAR CBITS=NAMED_LOGIC_LEVELS; CBIT,LEFT,RIGHT,L,R= LOGIC_LEVEL; I=INT;
DO CBITS:= {COLLECT COUNTER_BIT \NAMED ('BIT' \SUB I) FOR I FROM 1 TO N;};
LEFT:= CBITS[1]; RIGHT:= CBITS[N];
For naming convenience, let LEFT & RIGHT refer to the leftmost & rightmost counter bits.
Assign to CBITS, an array of N counter_bits, each named 'BIT' sub I
GIVE
Resulting LOGIC_LEVEL
EXTERNALS:
The INTERFACE
IN_PINS: { LEFT\S 'RESET_IN'; RIGHT S 'FLIP_IN'
OUT_PINS: { LEFT\S 'RESET_OUT'; LEFT\S 'FLIP_OUT';
CBITS\SS 'VALUE' }]
This compact expression produces a set of named pin the 'Value' pin of each counter bit, named 'VALUE' sub I
RELATIONS:
The logic equations
{ FOR {L;R} SC CBITS: COLLECT
{ L\S 'RESET_IN' \EQU (R\S 'RESET_OUT');
L\S 'FLIP_IN' \EQU (R\S 'FLIP_OUT')}
}
Connect the two adjacent counter bits named L & R
GUTS: CBITS END ENDEFN
Loop Generator: Set the variables L & R to each adjacent pair of counter bits from the array of counter bits CBITS.


LEFT
RIGHT
Reset-Out
Flip-Out
Reset In
Flip In
CBIT[1]
CBIT[2]
CBIT[3]
CBIT[4]
VALUE[1]
VALUE[2]
VALUE[3]
VALUE[4]


LEFT
LEFT\s 'Reset-In'
LEFT\s 'Flip-In
RIGHT\s 'Reset-Out'
RIGHT\s 'Flip-Out'
RIGHT

FIGURE 4

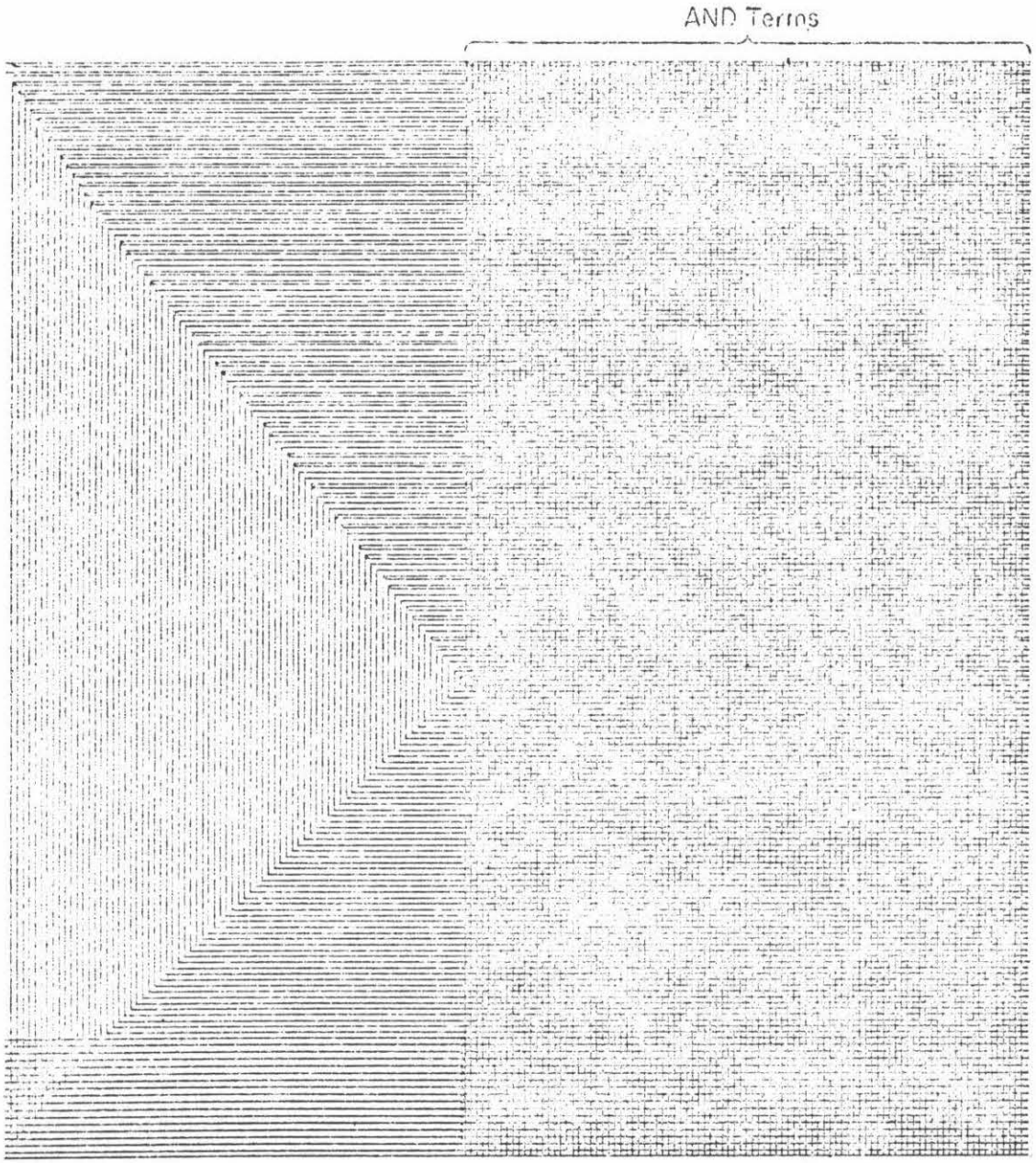


Figure 5

16-bit counter as
one PLA
area ≈32000

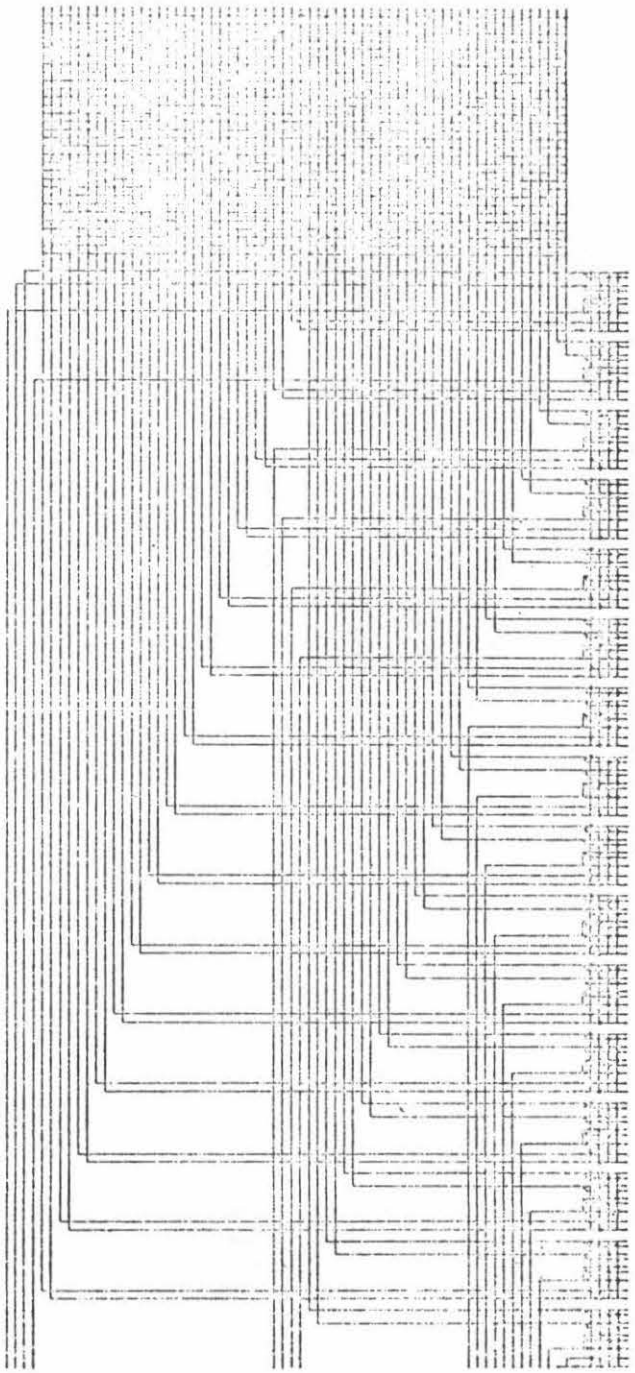


Figure 6
16-bit counter
area = 11500

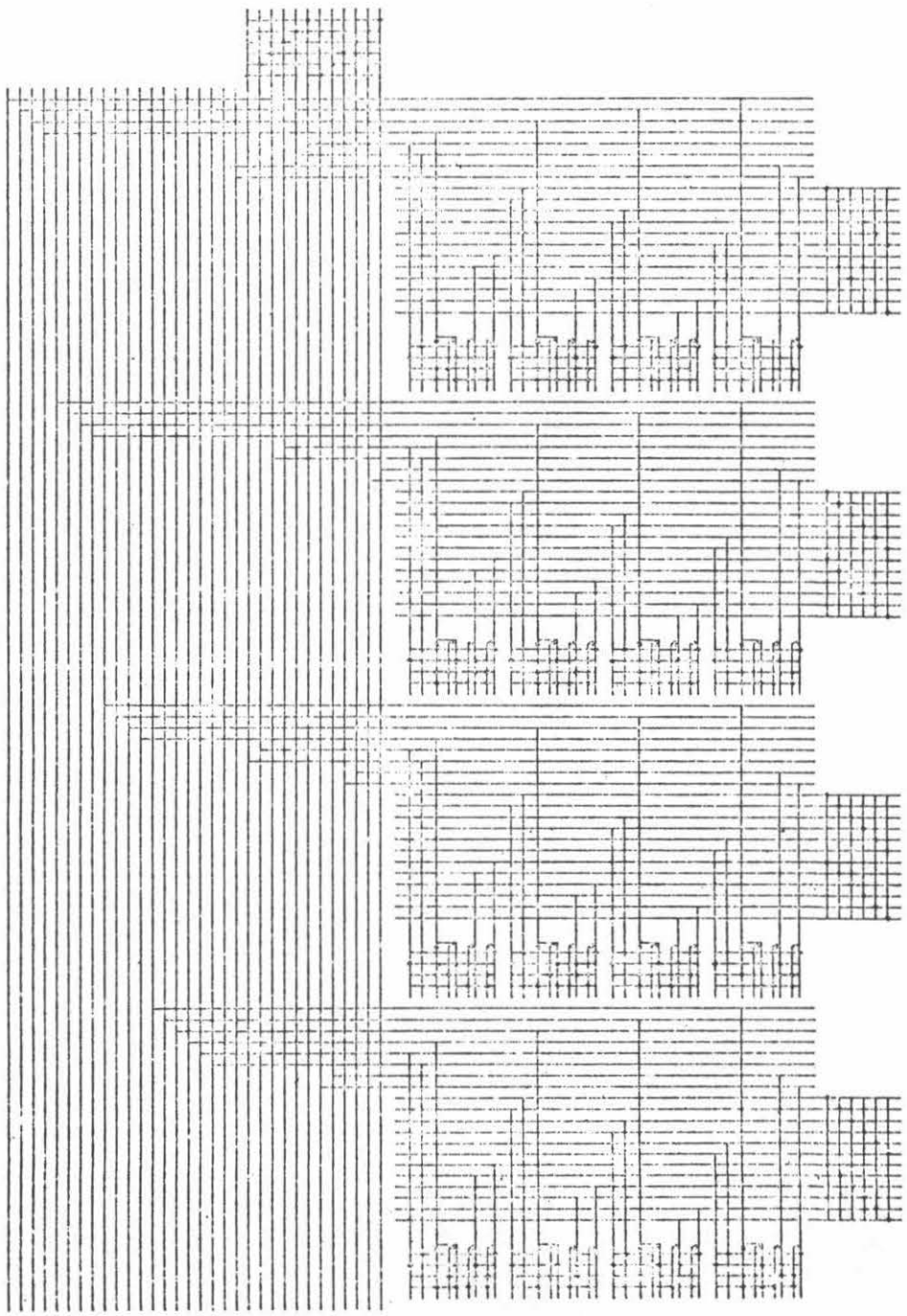


Figure 7

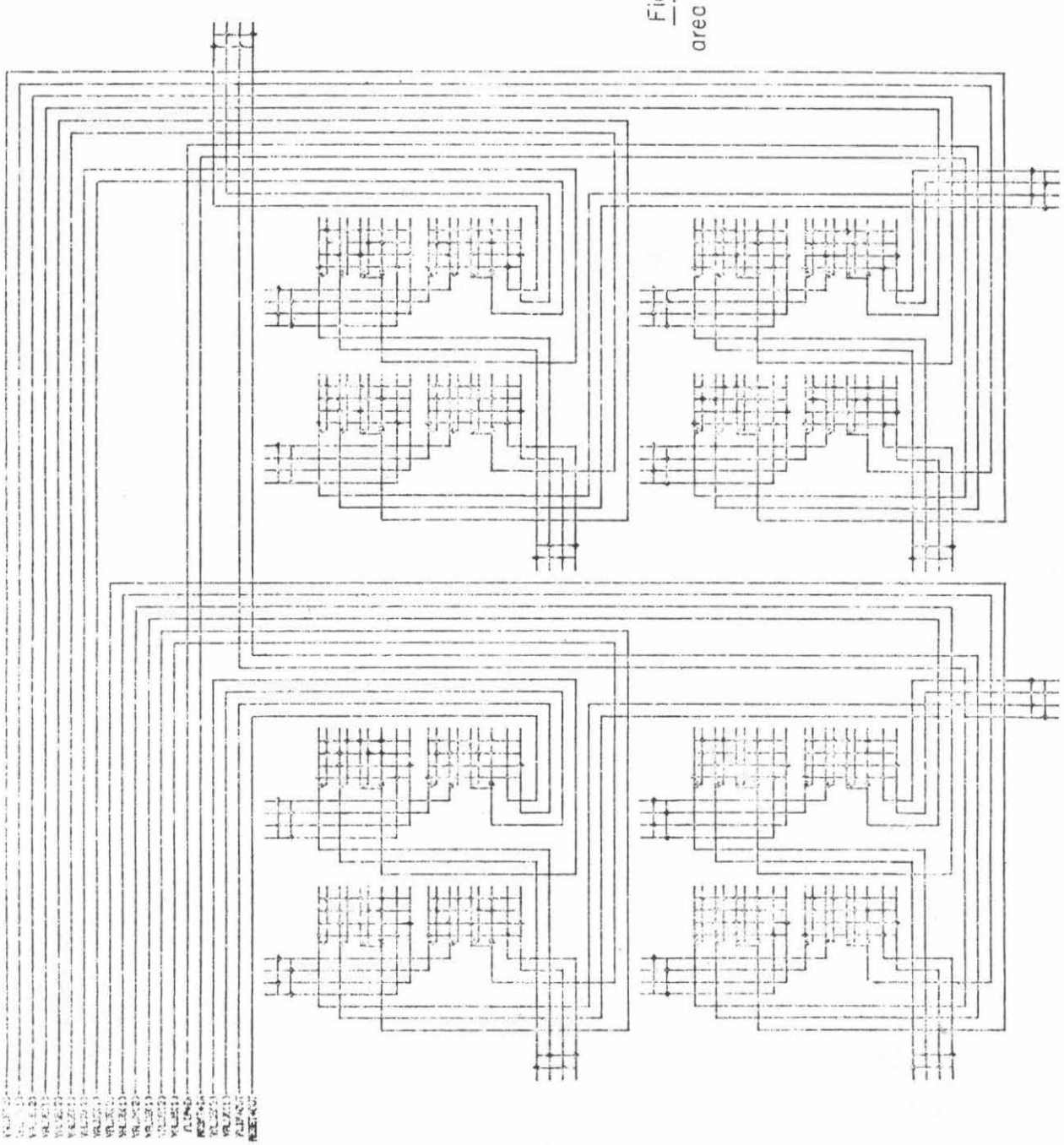


Figure 8
area ≈ 7000

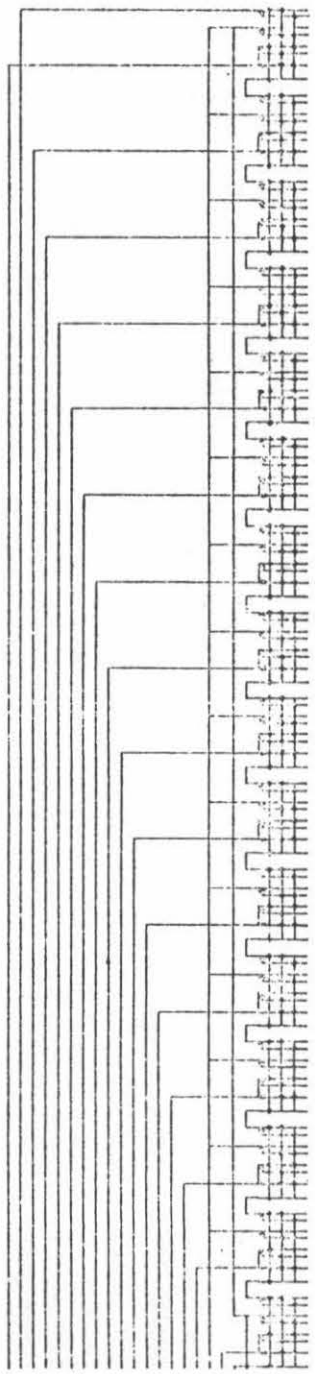


Figure 9
16-bit counter
with "trivial" equations removed
area ≈ 3000

Figure 10
area ≈ 2200

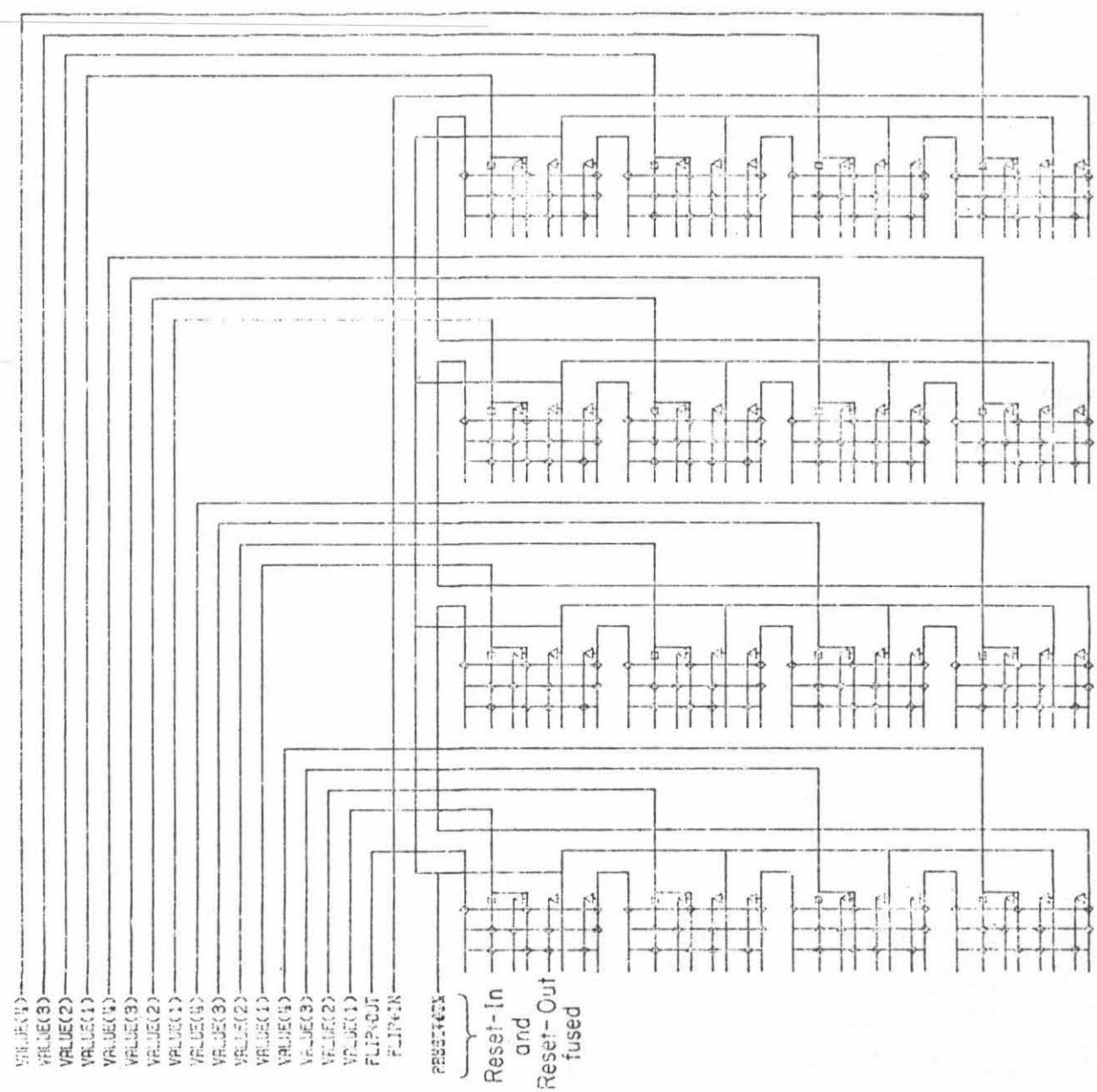
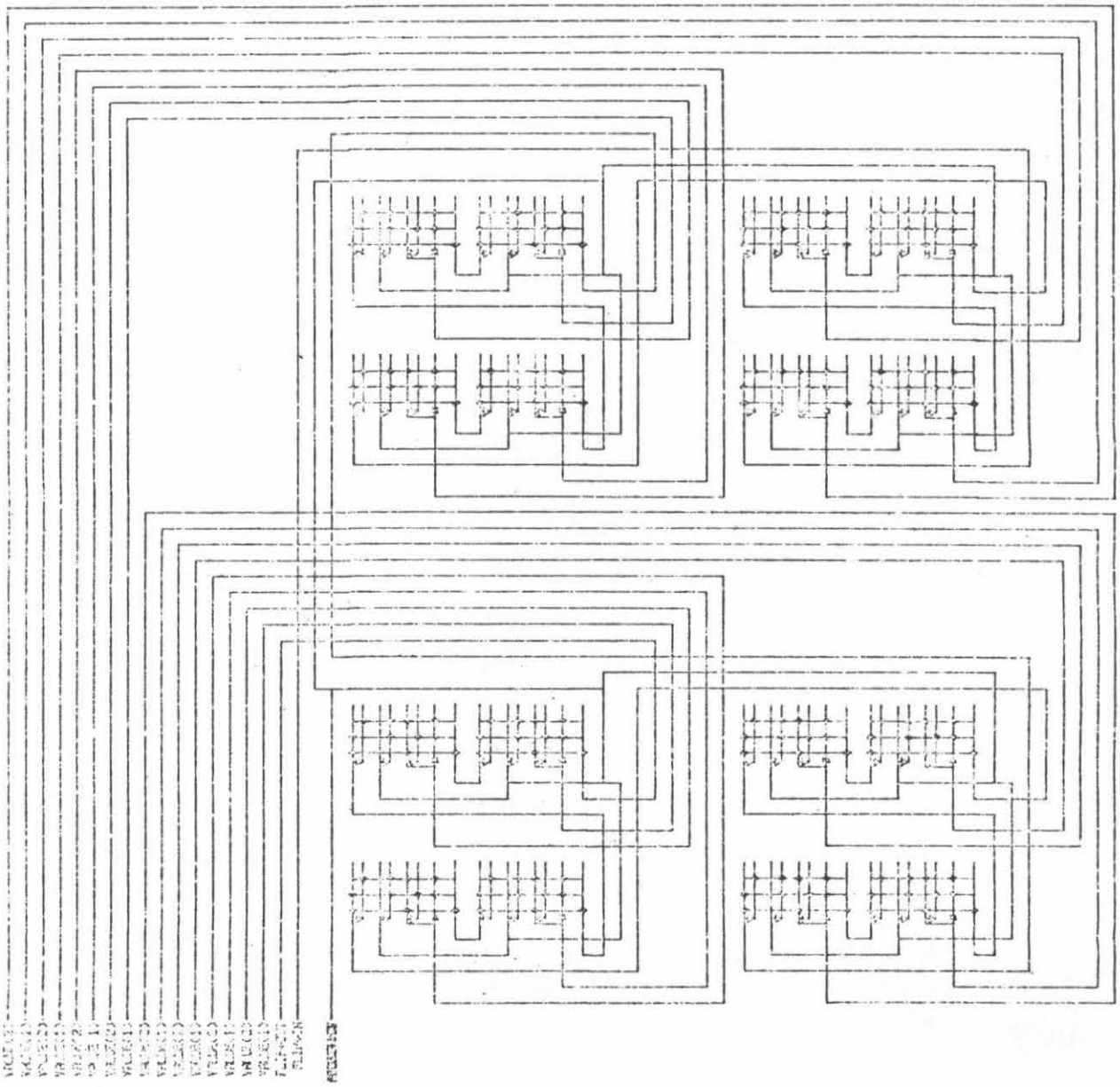


Figure II
area ≈ 4000



COMPUTER-AIDED DESIGN SESSION