A Computer-Aided Design System for Custom Large-Scale Integrated Circuits

M. W. Sievers
Communications Systems Research Section

This paper describes a computer-aided design system for custom large-scale integrated circuits. The system is composed of a high-level descriptive language and a SIMULA based language interpreter. The interpreter is running on the Caltech DEC SYSTEM 20 computer. It has been used to design a 16-bit self-checking comparator of medium-scale integration proportions.

I. Introduction

A general logic structure (GLS) for the design of custom integrated circuits has been discussed previously (Ref. 1). This is a matrixlike structure into which logic and wiring can be mapped. The simple and regular nature of the structure leads to a straightforward descriptive language and interpreter.

The descriptive language is based on the concept of a cell. A cell is a bounded region that may contain other cells, gates, and networks. This basis is amenable with structured, hierarchical design.

The function of the interpreter is to examine the design description and construct the cell "object" structures. Presently, it is possible to request cells to draw themselves either in a high-level notation consistent with the descriptive language or in the low-level detail required for mask making consistent with Caltech's NMOS design rules (Ref. 2).

Certain functions such as memory, pad drivers, and input protection diodes are best not constructed within the GLS structure. The facility for using such cells is not yet part of the design language. What is needed is to define cells whose input, output, and power wires are compatible with the GLS structure. Future versions of the language and interpreter will make use of these cells.

Several other modifications and additions are planned to increase the power of the design system. These will be mentioned in a later section. The design system is in its infancy at this time and is expected to grow in its capabilities.

The choice of SIMULA for the implementation of the interpreter will ease the work of making program changes. SIMULA is similar to many other block structured languages but has at least one significant dissimilarity. SIMULA is an object oriented language rather than data oriented language.

In SIMULA an object, called a CLASS, is defined that has various attributes associated with it. These attributes might be constants, variables, arrays, and procedures. Making interpreter changes requires the addition, deletion, or replacement of various CLASS attributes. Generally, these changes can be made within one CLASS without affecting other CLASSes. This is because each CLASS can be a totally self-contained object with little or no need for interaction with other CLASSes.

As already mentioned, SIMULA is an object oriented language. Section III will very briefly examine some of its more salient characteristics. Since data per se is not defined in SIMULA it is more meaningful to discuss object structures than data structures. The object structure created by the interpreter is specified in Section IV. Section V presents the descriptive language constructs and constraints. Section VI illustrates a typical design using the descriptive language. The next section introduces the high level GLS description upon which the design system is based.

II. High-Level GLS Notation

The GLS can be represented as a collection of unit gates interleaved with signal wires as indicated in Fig. 1. Each unit gate is represented by a rectangle. Signal wires are drawn as vertical lines.

A unit gate has two functions. It may be used either to create a NOR gate or as a wiring channel. The basic gate is a four port device in which any port may be programmed for either input or output. The ports are arranged two on each side of the gate. Unit gates may be coalesced into larger gates or wires. Only integral numbers of gates are defined, however. No problem occurs if gates cross power wires.

The unit gate defines the basic vertical unit of measure. For example, power wires are located on unit gate boundaries and pull-up resistors needed in the NMOS implementation are one unit gate long. Additionally, ground wires are assumed to be present at each unit gate boundary, unless certain conditions exist which are mentioned in Section V.

Signal wires adjacent to each unit gate column may be used to carry either power or signals vertically through the structure. These wires may be cut into arbitrary sizes as suit the design implementation. Signal wires exist for carrying signals horizontally but are omitted in Fig. 1 to reduce the clutter.

Figure 2 shows a static gated D flip-flop constructed in the high level notation. Flip-flop inputs and outputs are carried on horizontal wires. Arrows pointing into gates are input terms. Gates outputs are indicated by dots inside the gate. Dots at the intersection of vertical and horizontal wires indicate connection. Wide horizontal lines indicate a power bus and thinner vertical lines running from gates to a power bus represent a combination of pull-up and power connection.

Ground wires are not explicitly shown in Fig. 2 for reasons already mentioned. However, it may become necessary to periodically tie ground wires to a ground bus in larger designs. A facility exists for doing this (see Table 6 in section V).

Two constraints on designing with the GLS occur that may not be obvious from Fig. 2. The first is that all gates must start on odd numbered columns although wires may be placed in any column. This restriction is due to the GLS low level implementation and is made to enforce consistent column usage throughout a design. The second constraint requires that no gate input be made in the same row on the right side of a gate in column i - 1 or on the left side of a gate in column i + 1 as a contact made in column i. This again is due to the GLS implementation. Violating this rule would result in the creation of gates that probably would not be capable of successfully driving their fanout.

III. A Brief Look at SIMULA

SIMULA is a block structured language most similar to ALGOL. It is unique in that it is oriented toward objects, called CLASSes, rather than toward data. A CLASS is an instance prototype that consists of three major parts:

(1) Head.

(2) CLASS definition body.

(3) Initialization body.

The head part names the CLASS, defines parameters needed to create instances of it, and establishes it within a hierarchical CLASS structure. The CLASS definition body is the set of constants, variables, and procedures that make up the CLASS. Procedures are not executed unless called. The initialization body is a special procedure that is executed upon the creation of each CLASS instance.

CLASS instances are created by the construct

NEW classname (parms);

where classname is the name of an existing CLASS and parms are the parameters required by that CLASS. Reference variables may be declared which are used as pointers to CLASS instances. Thus if a and b are declared as reference pointers to CLASS c then after execution of the segment

a: – New c (parms);

b: – New c (parms);

two instances of CLASS c exist with a pointing to one and b the other. Instances remain accessible until all of their reference pointers have been destroyed. If

a: – b: – New c (parms);

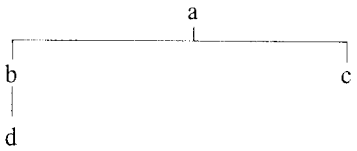
the single instance of object c created remains until both a and b no longer point to it.

Each instance created effectively has a full copy of all attributes defined by the associated CLASS. Thus procedures within several instances of the same CLASS may be executed in parallel without the usual concerns of program sharing. In reality of course, only a single copy of each CLASS exists and SIMULA maintains the required data structures for segment sharing.

CLASSes may be made subsets of other CLASSes by prefixing the CLASS head with the name of another CLASS. So

```
a CLASS b;
a CLASS c;
b CLASS d;
```

establishes the following CLASS hierarchy.



Each sub CLASS has all of the attributes of each of its super CLASSes. A super CLASS, however, has none of the attributes defined within its sub CLASSes. Further, each creation of a sub CLASS object results also in the creation of instances of all of its super CLASSes.

Attributes of instances are always visible from within the instance and can be visible externally unless otherwise protected. It is also possible to define virtual attributes. For example a super CLASS s can declare a VIRTUAL PROCEDURE p. Sub CLASSes of s each define a PROCEDURE p. A reference made to the virtual PROCEDURE p in s will cause the actual procedure to be executed in the sub CLASS. For example, if

```
CLASS s;
VIRTUAL: PROCEDURE p;
s CLASS b;
BEGIN
PROCEDURE p;
END;

s CLASS c;
Begin
PROCEDURE p;
END;
```

then if d, and e are made CLASS s references and are assigned instances by

```
d: – NEW b;
e: – NEW c;
```

a reference to p in the s object d will pass control to p in object c. Similarly, a reference to p in s object e will pass control to p in object c. The advantage of this is that it is possible to declare a CLASS as a super CLASS for many objects and to refer to any instance of these many objects by the single super CLASS. A reference to any virtual attribute of the super CLASS will result in a reference to the actual attribute in the sub CLASS that qualifies the super CLASS. The alternative to this capability is to test each super CLASS reference to determine which of its sub CLASSes qualify it and then access the attribute of the sub CLASS directly. Testing is slower and considerably less elegant than the VIRTUAL declaration approach.

IV. Interpreter Object Structure

The high-level language interpreter is based on the object hierarchy shown in Fig. 3. CLASS thing is an object that basically has no attributes. It is defined within CLASS things on top of which the interpreter is written; i.e., things is external to it. Object thing is defined to be a super CLASS for all objects. CLASSes celldef, network, nodes, gaterep, and pwrwire are all thing sub CLASSes.

Object celldef is the cell definition prototype. Its attributes and their meaning are listed in Table 1. The network object is defined by two nodes objects. Table 2 lists its attributes. CLASS nodes is composed of virtual attributes only. It is defined to be the super CLASS for all node objects. Therefore, for example, requests generated in CLASS network procedures to produce high- or low-level descriptions of its defining nodes can be directed toward nodes CLASS objects. CLASS network need not test to determine what type of node is actually present. Virtual attributes of nodes are indicated in Table 3.

Nodes subclasses transistor, contact, gndcontact, pin, and powercontact each contain the actual procedure definitions declared VIRTUAL in CLASS nodes. Transistor has an additional local attribute that flags which side of the gate it is on. This attribute is not visible at the nodes level.

The gaterep object is the gate prototype. Its attributes are found in Table 4. Object pwrwire is the power wire definition. Table 5 lists its attributes.

V. Design Language

The design language has the syntax of SIMULA and contains constructs for creating cells, making instances of cells, and preparing output files of cell descriptions. In addition to these constructs, any legal SIMULA code may be included in a chip description.

A chip is described as a collection of cell definition blocks. The start of each such block is indicated by

```
create (cellname);
```

where cellname is a text string. Each time a create is executed, the interpreter makes a new instance of celldef. Every cell definition block is terminated by an endcreate statement. This statement takes the cell just defined and enters it into a cell dictionary for future reference.

The body of a cell definition block contains the constructs that define elements within the cell. At present, a cell definition block cannot contain a create statement. This restriction is arbitrary and could be eliminated. Table 6 lists the body constructs.

After a cell has been defined, it may be translated into an output file in either its high- or low-level form. High-level output is requested by hllplot (cellname) and low-level output by goryplot (cellname). An output file may be displayed or translated into a mask set. It is necessary, however, to request a goryplot for actual mask making.

Two language precautions arise due to the interpreter implementation. These are:

(1) When cells overlap and any of these has a power wire contained within the overlapped region, then all cells must declare a power wire within that region. This is necessary because each cell is responsible for drawing its own ground wires. Ground wires are drawn across a cell on unit gate vertical boundaries unless a power wire or subcell is found in its path. If one declares a power wire, that wire is not visible outside of that cell definition. A cell sharing a common region with that cell will insert a ground wire under the power wire declared in the first cell since its existence is unknown. The result could be a short from power to ground at a pull-up resistor.

(2) All cells in which a gate is declared must have a suitably located power wire declared. Again, power wires declared outside of a cell definition are unknown within the cell. Gates must have clear access to power wires to which they can be connected.

In addition to these precautions, there are a few limitations due to interpreter implementation:

(1) Nested cell definitions are not permitted.

(2) Cells may be stretched only once.

(3) No provision has been made to include cells defined outside of the design language.

(4) Only NMOS devices can be created.

These precautions and limitations are fairly minor and do not hinder the specification of a complete chip. Modifications to the interpreter are planned to remove most of them.

VI. A Typical Design

The static register cell in Fig. 2 is implemented in the design language. Figure 4 shows the language description of the register. Figure 5 shows a black and white copy of a color plot of the high level description produced by the code in Fig. 4. Figure 6 shows a black and white copy of a color plot representing the mask level details of Fig. 5.

As a final example, Fig. 7 illustrates a black and white version of a color plot showing a 16-bit self-checking comparator (Ref. 3) designed by the system. The pads and drivers were added by merging a file containing their description with the file produced by the interpreter. This chip is part of one of the Caltech class chip projects and will be fabricated by November 1979.

VII. Conclusions

A descriptive design language and interpreter are now available for defining the logic "core" of an integrated circuit. Additions are planned to increase the design system capabilities to include cells not created in the GLS. Further additions might also include logical and electrical simulation of the devices created.

References

1. Sievers, M. W., "A General Logic Structure for Custom LSI," in *The Deep Space Network Progress Report 42-50*, Jet Propulsion Laboratory, Pasadena, Calif., April 15, 1979, pp. 97–105.

2. Mead, C., and Conway, L., Introduction to *VLSI Systems*, 1978, ch. 2., text in preparation.

3. Carter, W. C., Wadia, A. B., and Jessup, D. C., "Implementation of Checkable Acyclic Automata by Morphic Boolean Functions," Symposium of Computers and Automata, Polytechnic Inst. of Brooklyn, April 1971, pp. 465-482.

Table 1. celldef attributes

Attribute	Comment
x, y	Unit gate location of lower left corner
networklist	List of networks
subcellist	List of subcells
pwrlist	List of power wires
gatelist	List of gates
mybox	A rectangle defining the bounding box
boundingcell	The next highest celldef in a hierarchical design.
stretched	Boolean variable indicating whether or not the cell has been stretched
PROCEDURE vertstretch	Produces a vertically stretched copy of the cell; stretching is nonlinear in that only objects above a given row number are moved up
PROCEDURE horizstretch	Produces a horizontally copy of the cell; the nonlinear comment above applies here to column number
PROCEDURE squoosh	Produces a cell both horizontally and vertically stretched.
PROCEDURE makegore	Does a recursive descent through cell hierarchy to make detailed mask description
PROCEDURE makehll	Makes a high level description of the cell; subcells are represented by their bounding box
PROCEDURE bbox	Computes the bounding box.

Table 2. network attributes

Attribute	Comment
node1, node2	Nodes objects defining the network endpoints
PROCEDURE morenodes	Adds additional nodes to network defined by node1, node2
PROCEDURE makehll	Requests nodes objects to make their high level description, draws in a connecting wire
PROCEDURE makegore	Requests nodes objects to make detailed description, connects nodes objects with a wire and inserts crossunders as required.

Table 3. nodes attributes

Attribute	Comment
VIRTUAL PROCEDURE hllx	Returns high-level x location from nodes sub CLASS
VIRTUAL PROCEDURE hlly	Returns high-level y location from nodes sub CLASS
VIRTUAL PROCEDURE gorx	Returns low-level x location from nodes sub CLASS
VIRTUAL PROCEDURE gory	Returns low-level y location from nodes sub CLASS
VIRTUAL PROCEDURE myy	Returns unit gate row number from nodes sub CLASS
VIRTUAL PROCEDURE myx	Returns column number from nodes sub CLASS
VIRTUAL PROCEDURE makehll	Requests nodes sub CLASS to make its high-level description
VIRTUAL PROCEDURE makegore	Requests nodes sub CLASS to make its low-level description
VIRTUAL PROCEDURE xgets	Replaces nodes sub CLASS column number with passed parameter
VIRTUAL PROCEDURE ygets	Replaces nodes sub CLASS unit gate row number with passed value
VIRTUAL PROCEDURE copynode	Creates a copy of the nodes sub CLASS

Table 4. gaterep attributes

Attribute	Comment
x, y	Location of bottom-most unit gate
size	Length in unit gates
side	Side, top or bottom, to which power will be connected
PROCEDURE hllx PROCEDURE hlly PROCEDURE goryx PROCEDURE gory	Same function as listed in Table 3 as applied to gates
PROCEDURE makegore	Makes low level details if gate, cross-unders are inserted as necessary, connects gate to pull-up and power
PROCEDURE makehll	Makes high level gate description, connects gate to power.

Table 5. pwrwire attributes

Attribute	Comment
x, y	Row and column location of left-most point
length	Length in columns
PROCEDURE hllx PROCEDURE hlly PROCEDURE gorx PROCEDURE gory PROCEDURE makehll PROCEDURE makegore	Same as Table 3 as applied to power wires

Table 6. Interpreter constructs

Construct	Comment
plop (cellname, location);	Creates an instance of cell named cellname and places it at position location; location is the row and column of the lower left corner of the cell
vertput (cellname, location, row, howmuch);	Creates an instance of cell cellname at location; all elements above row are moved up by howmuch
horizput (cellname, location, col, howmuch);	Creates instance of cell cellname at location; all elements to the right of col are moved 2 * howmuch columns.
vhput (cellname, location, row, howmuchr, col, howmuchc);	Does both vertical and horizontal stretches
gate (loc, size, side);	Creates an instance of gaterep at location loc, size unit gates long, with power connected to side ("top" or "bottom")
power (loc, length);	Creates an instance of a power wire starting at loc, length columns long
connect (nodel, node2) < • morenodes (node) > ;	Creates a network between nodes objects node1 and node2; additional nodes are added via morenodes
input (loc, row, side);	A gate input nodes object for connect, creates a transistor instance at gate defined at loc, at row ports from the bottom of the gate, on side ("left" or "right")
output (loc, row);	Gate output from gate defined at loc at row ports from the gate bottom; a contact instance is created
con (loc, row);	Vertical to horizontal wire contact, loc, row same as above
pwrcon (loc);	Connection to a power bus
gndcon (loc);	Connection to ground bus
io (name, loc, row);	Cell input-output node, creates a pin instance labeled name

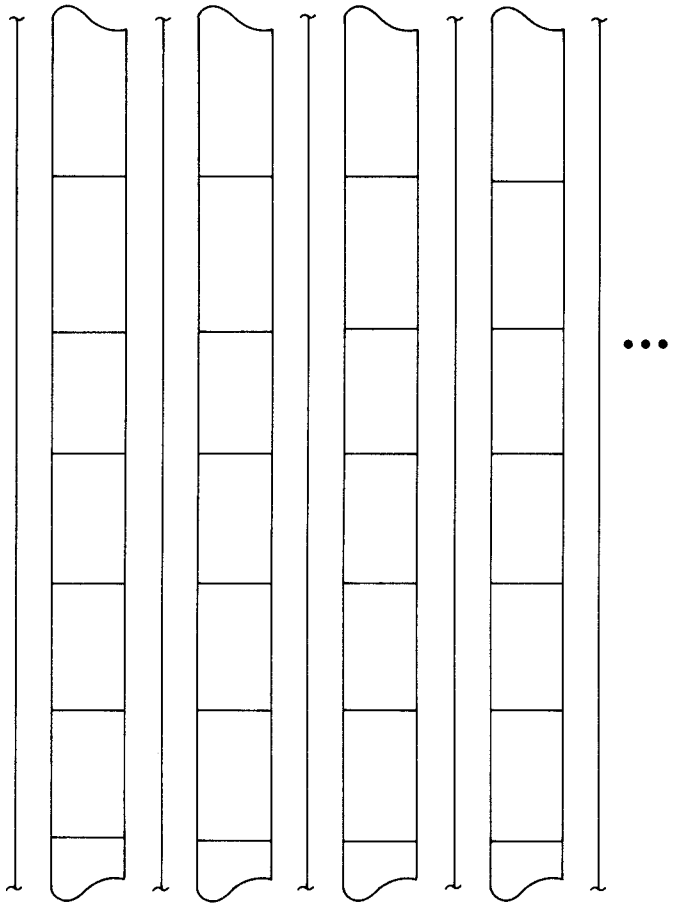


Fig. 1. High-level representation of GLS matrix

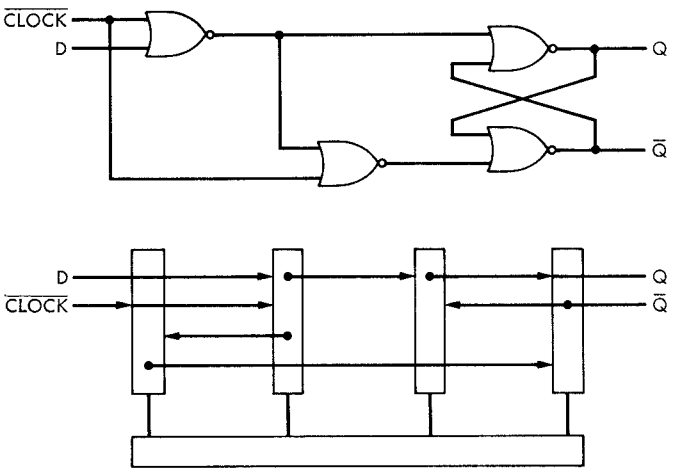


Fig. 2. Static gated D flip-flop

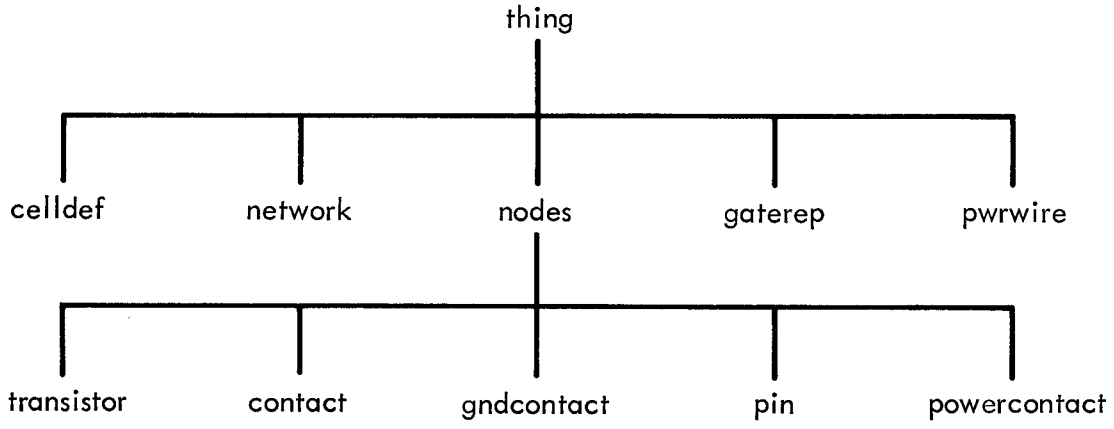


Fig. 3. Interpreter object hierarchy

```
create ("flipflop");
power (0, 0, 7);
gate (1, 1, 2, "bottom");
gate (3, 1, 2, "bottom");
gate (5, 1, 2, "bottom");
gate (7, 1, 2, "bottom");
connect (io ("D", 0, 1, 4), input (3, 1, 4, "left"));
connect (output (3, 1, 4), input (5, 1, 4, "left"));
connect (output (5, 1, 4), input (7, 1, 4, "left"))
    . morenodes (io ("Q", 8, 1, 4));
connect (io ("CK", 0, 1, 3), input (1, 1, 3 "left"))
    . morenodes (input (3, 1, 3, "left"));
connect (io ("NQ", 8, 1, 3), output (7, 1, 3))
    . morenodes (input (5, 1, 3, "right"));
connect (output (3, 1, 2), input (1, 1, 2, "right"));
connect (output (1, 1, 1), input (7, 1, 1, "left");
endcreate;
```

Fig. 4. Program for describing static D flip flop

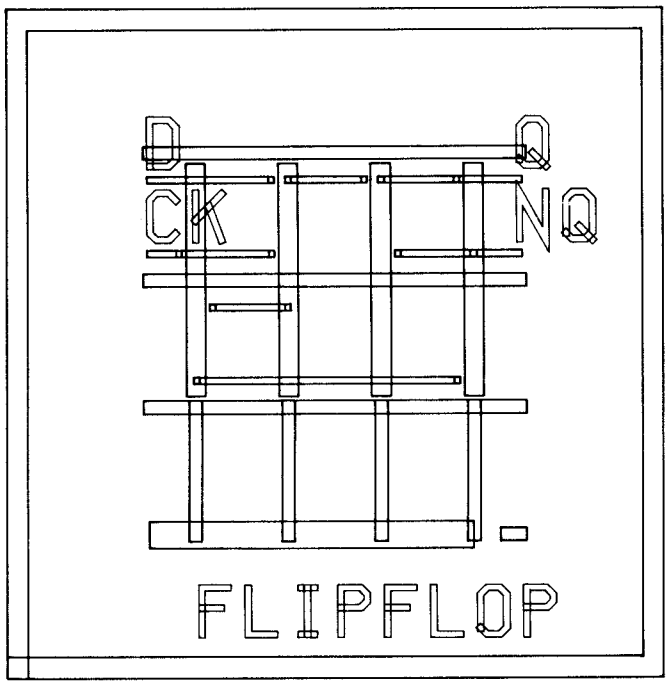


Fig. 5. High-level flip-flop description

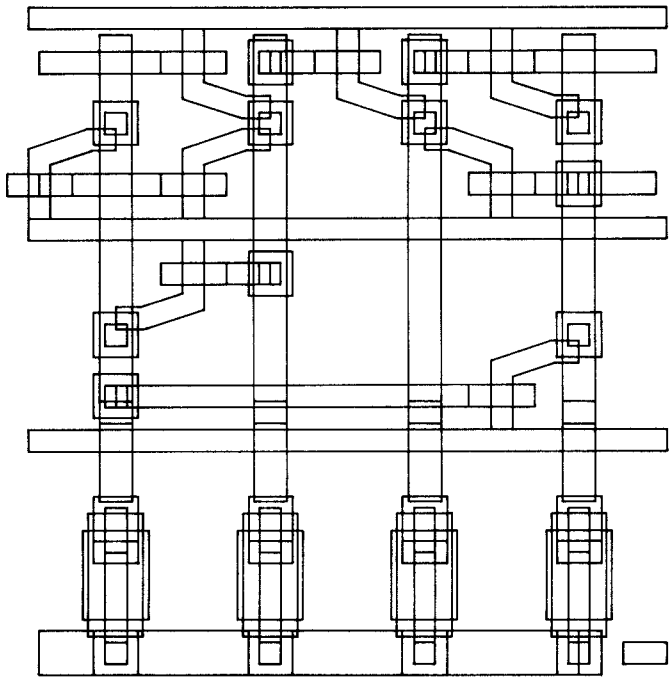


Fig. 6. Mask level flip-flop description

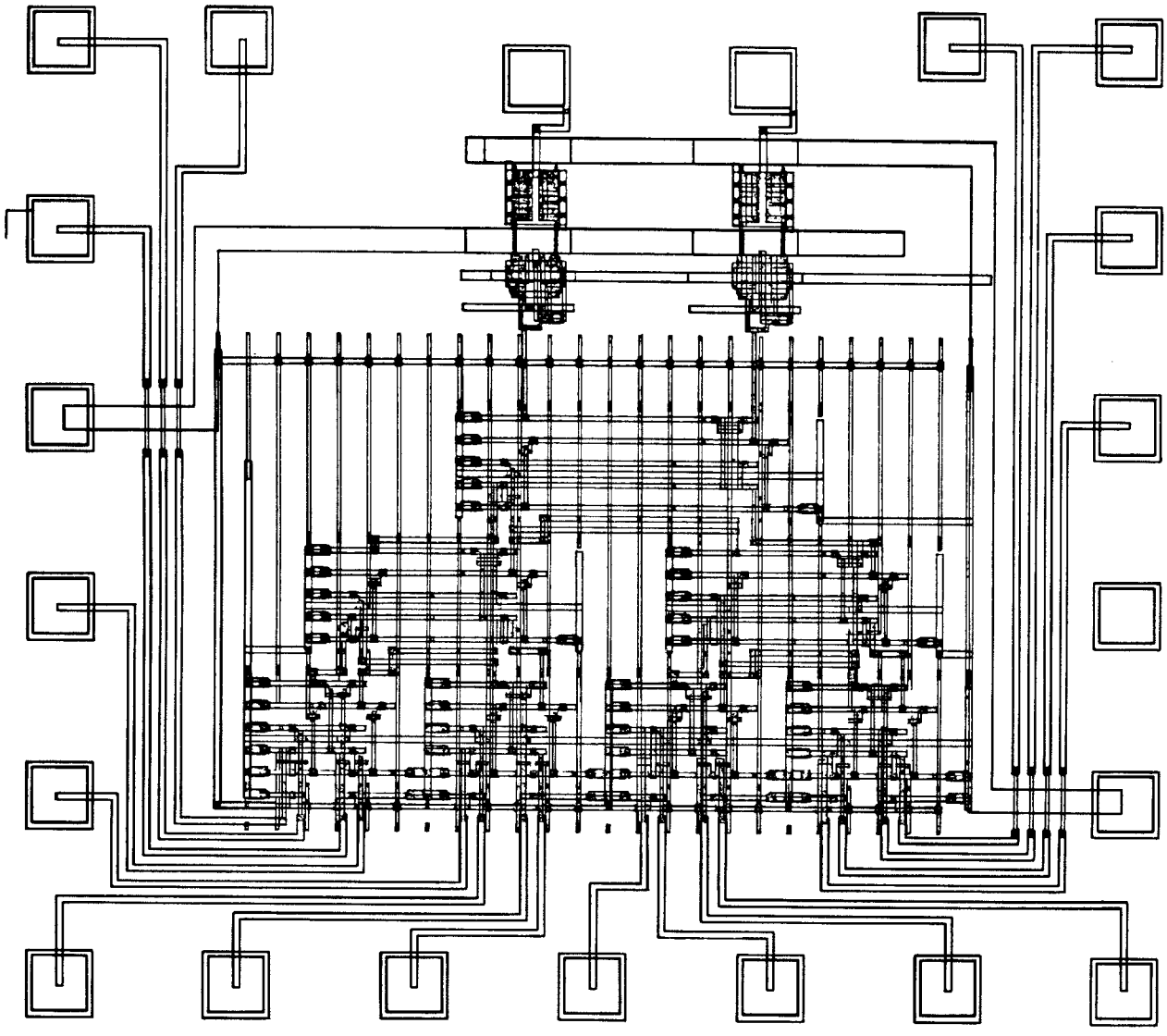


Fig. 7. 16-bit self-checking comparator