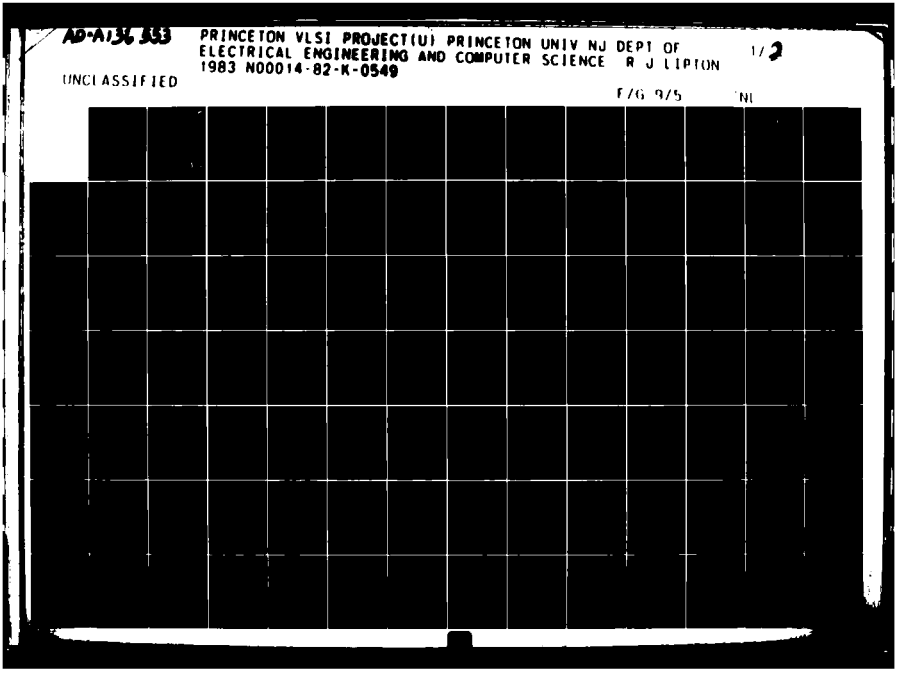
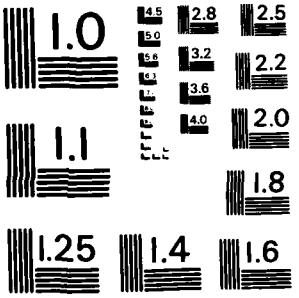
AD-A136 333 PRINCETON VLSI PROJECT(U) PRINCETON UNIV NJ DEPT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE R J LIPTON 1983 N00014-82-K-0549 1/2

UNCLASSIFIED F/G 9/5 NL





PRINCETON VLSI PROJECT: Semi-Annual Report

PERIOD ENDING: November 15, 1983

R.J. Lipton - Principal Investigator

EECS Department

Princeton University

Contract N00014-82-K-0549

FACULTY

B. W. Arden
D. Dobkin
H. Garcia-Molina
P. Honeyman
A. LaPaugh
K. Steiglitz

1. Introduction

There are three major aspects to our project. The first concerns the development of a procedural approach to the layout of VLSI circuits. The second is the continuing investigation of the census language. Finally, the third is in the area of testing of VLSI circuits.

2. Procedural Approach to VLSI

2.1. ALI2 [LaPaugh, Mata]

A complete version of ALI2 is now operational. It includes a variety of support packages. These include a library of basic cells and a switch-level simulator that is "built" into ALI2. This simulator is novel in that it can detect a number of "problems" in circuits such as race conditions.

ALI2 is now being used and evaluated by a number of VLSI designers at Princeton. It is also being used in a beginning VLSI course at Princeton. We hope to get feedback from these users shortly on ALI2 and the procedural approach to VLSI design.

Already work is under way on improvements to ALI2. One area of improvement is the elimination of any need for design rule checkers. Layouts generated by ALI2 are usually design rule correct but this is not guaranteed by the system. It appears possible to modify ALI2 slightly to make all generated layouts design rule correct.

2.2. Clay [Lipton, Lucas, North, Souvaine]

Clay, another procedural approach to VLSI design, is now operational. We are currently using it in several design projects. Indeed, a number of simple designs have completed successfully the full design-fabrication cycle. We have also just made Clay available to other institutions and have a number of users outside Princeton.

2.3. Layout Algorithms [Huang, North, Steiglitz]

The layout algorithms used by ALI2 and Clay are quite prone to "thrashing" the paging system of the VAX. For this reason a number of independent projects are underway to improve on the current implementations. Clay uses a hierarchical approach. Clay allows the user to break their layout up into several pieces that can be separately compiled into layouts. This still preserves the total flexibility of Clay layouts. Another more theoretical approach is based on a new algorithm for layout. For an important class of layout problems, this algorithm can guarantee few (relatively) page-faults. Work is now underway to implement and evaluate this new algorithm.

2.4. Referee [Lipton]

Referee is a new program for circuit comparison. It uses a new definition of when two circuits are the same. This definition is more "forgiving" then the usual definition based on graph isomorphism. Referee also has a guaranteed running time that is linear in the size of the circuit. We are planning in the future to integrate it into the ALI2/Clay systems.

2.5. Applications of Clay

2.5.1. Graphics Engine [Dobkin, Field, Souvaine]

Progress on the design of a VLSI engine for doing graphics has concentrated on the design of custom chips for scan conversion of lines. Using Clay adders of various types have been designed. These can be combined to yield complete circuits for both Bresenham's algorithm and Field's algorithm for anti-aliased scan conversion of lines, scenes, and cubic curves.

Work has begun on interfacing these circuits to other portions of our graphics system. The goal is to have the pseudo-triangle as the basic building block. This structure consists of the interconnection of three vertices via curves of arbitrary degree (<4). Circuits to compute these functions are lacking in even high-end state of the art graphics systems.

2.5.2. Recursive Layout [Lucas, Souvaine, Steiglitz]

Clay has been used to design a number of recursive circuits. These include: (1) comparers, (2) tally circuits, (3) various adders, and others.

The advantages of using Clay for such designs are several. First of all, once the basic cells have been described, the entire layout is generated by a single recursive function call. Since, in Clay, the calls remain flexible until the layout is complete, proper interconnections among the cells is assured. Moreover, by changing a single parameter, an 8-bit, a 16-bit, a 128-bit, or any size layout may be generated.

Equally important, however, is the ease with which we can resize transistors in order to improve speed. A number of layouts have used this feature and Crystal to dramatically improve their performance: one chip was speedup from 200ns to 53ns by just such a resizing which is trivial with Clay. We are now working on automating this whole resizing step.

3. Census

There are two main projects under way here.

3.1. Top/Down [Lopresti, North]

This project is investigating the use of the census approach to parallel computation as a way to speed up a large class of computations. The essential idea is that rather than speeding up the inner loop of a computation as is usual, we plan to take a top-down approach. Here, the problem is decomposed at a high

level into independent (or nearly) computations on loosely coupled processors. We are currently investigating the classes of problems that match this approach.

3.2. M^6 [Garcia-Molina, Honeyman, Lipton]

This project is investigating a new approach to the design of a super computer: we propose to interconnect large number of *memories* with a very small number of processors. Our central thesis is that a machine with a hugh amount of physical memory, in the *tens of billions* of bytes, can outperform other supercomputers on many important tasks. The project has already found a new novel way to implement such a machine which we call *ESP*. Work is now underway to develop and expand our understanding of the issues involved in building such a machine.

4. Testing

Work on VLSI testing is continuing along two basic lines.

4.1. Structured Testing [Steiglitz, Vergis]

Work here has recently found large classes of regular layouts that are easily testable. These include many important classes of systolic arrays.

4.2. Bipartite Testing [LaPaugh, Lipton]

Work continues on this approach to design for testability. The earlier methods have now been extended to CMOS circuits. Work also is continuing on building test circuits.

In addition, a new but related approach to testing is now being developed. It uses a special nand gate that is similar to that used in the Bipartite Method. However, it avoids the potential doubling of the number of gates found in the Bipartite Method. The additional cost is the number of test vectors is no longer constant but in worst case is linear in the size of the circuit. The key, however, is as before it is computationally easy to find the test vectors that guarantee 100% coverage.

5. Papers

Molding Clay: A Manual for the Clay Layout Language

Stephen C. North

VLSI Memo #3
July 1983

Molding Clay: A Manual for the Clay Layout Language

Stephen C. North

Department of Electrical Engineering and Computer Science
Princeton University
Princeton, New Jersey 08540

Bell Laboratories
Murray Hill, NJ 07974

The Clay VLSI Design Language

Clay is a procedural language for NMOS VLSI layout design.† A layout in Clay is created by writing a program which describes the devices and wires in the layout, and where they are placed. The Clay system translates the algorithmic description into CIF (Caltech Intermediate Format).

There are several advantages of a programming language over a graphical editor for VLSI design. A programming language provides a means for controlling the complexity of the design task. For instance, a structured design language can help make large layouts managable by top-down decomposition, similar to the way large programs can be written. A language, as opposed to an editor, also provides a vehicle for implementing VLSI layout algorithms, and allows the designer to write generic, parameterized cells (such as transistors, inverters, PLAs, channel routers, etc.) and then instantiate them many times.

A disadvantage to our approach is that the designer cannot see his design as he is writing the layout program, except by going through the translate-layout-plot cycle. So he must have a mental (or physical) picture of the design he is trying to create, and then express it as statements in the programming language. This is primarily a problem in writing low-level cells, which

† The fundamental design of Clay is independent of the fabrication technology; an extension for CMOS is planned.

contain many random objects and which often must be optimized for small area. Higher level structures tend to be more regular and are more naturally described algorithmically. Nevertheless, we have had satisfactory experience with designing low-level cells, and since Clay can handle arbitrary CIF objects, it is very easy to access cells created by other layout tools such as a graphical editor.

Clay was written as a package of C data types and functions. Before trying to write a Clay program, the designer should already know C. We chose C as a base language because we did not want to try to re-invent all the features of a structured programming language not related to the layout task and C is flexible enough to support the data types and function interfaces we need. Further, the Unix C compiler is efficient enough to support large layouts.

Clay adds two new data types to C: *wires* and *symbols*. Wires are horizontal or vertical runs of some layer (metal, polysilicon, or diffusion). Wires declared in a Clay program are of fixed width but variable length. The length is determined by the Clay system itself as part of the translation into a layout. A wire can be thought of as a stretchable line segment with a fixed-width field around it. A symbol is a small rigid piece of CIF, such as a transistor or contact. Symbols interconnect wires. Thus, a layout consists entirely of stretchable wires meeting at symbols. It is intentionally *not* possible to place any object at an absolute location. This flexible placement of objects, similar to stick diagrams, is an important feature of Clay.

The Clay language primitives (which we will describe in detail later) create wires and symbols and control their placement in the layout. The execution of a Clay program produces, not the CIF layout, but a list of the wires and symbols it created, and *constraints* on their placement. A program called the *solver* converts these into CIF.

To get started, consider the following simple Clay program illustrating the basic primitives (line numbers are *not* part of the program).

```
1:   #include "/va/clay/lib/header.h"
2:   main()
3:   {
4:   wiretype w;
5:   symboltype s;
6:   w = wire(POLY,MIN);
7:   s = symbol("mpcontact");
8:        ordered(LR);
9:             place(s, NULL,NULL,w,NULL);
10:            place(s, w,NULL,NULL,NULL);
11:       leaveordered();
12:  }
```

Line (1) is the include needed for the definition of Clay data types. Every Clay program must have this. Line (4) is the declaration of a wire variable. A wire variable takes on actual wires as values. A call to **wire** creates a new wire in the layout, but does not say anything about where to place it, nor how long it is. Thus, the call to **wire** in line (6) sets w to a new minimum width wire of polysilicon. In NMOS, the legal layers are POLY, METAL, and DIFF. Widths larger than MIN can be given as multiples of the predefined constant LAMBDA, for instance:

```
w = wire(METAL,10 * LAMBDA);
```

To conform with the convention that CIF dimensions are given in centimicrons for 2.0 micron NMOS, LAMBDA is currently defined as 200. For a different fabrication process or CIF scaling factor, LAMBDA can be redefined. The width of a wire is the maximum of the user-supplied width and the process minimum. That is, a wire can't be narrower than the design rules allow, but it can be wider.

Line (5) is the declaration of a symbol variable. As described before, a symbol is a rigid object that can be placed under the control of a Clay program. A symbol variable is set to such an object by a call to **symbol**, as in line (7). The argument to **symbol** is the Unix name of a CIF file. Clay uses a symbol as a template, to be copied and placed. The call to **symbol** does not put anything in the layout, but merely sets the value of a symbol variable so the symbol can be referenced later. Since **symbol** opens, reads, and closes the CIF file to get the symbol definition, it is better to set symbol variables once at the start of a program, rather than within a loop.

An important concept in Clay is that wires and symbols are placed inside ordered contexts. The **ordered** primitive creates a new context. Its argument specifies the kind of context to be created: TB for top-to-bottom, BT for bottom-to-top, LR for left-to-right, and RL for right-to-left. A context is a virtual box in the layout. A context's scope extends until a matching **leaveordered** primitive appears. In our example, the left-to-right ordered context created in line (8) continues until line (11). Usually, **ordered** and **leaveordered** will enclose a block of code, but since they are executable primitives (and not syntactic delimiters of a static scope) a Clay program can create new contexts dynamically.

Within a context, the **place** primitive places wires and copies of symbols. The general form of this primitive is:

place(sym,a,b,c,d);

The first argument is a symbol; the other four are wires or the constant NULL. The call to **place** has several effects. First, it forces the wires to meet at a point: *a* must enter from the left, *b* must enter from the top, *c* must enter from the right, and *d* must enter from below (see Fig. 1). Second, it places a copy of the symbol on top of this point. Third, the symbol is constrained to lie entirely within the current context. Fourth, ***symbols are ordered as they are placed***. It is this interplay between **ordered** and **place** that gives Clay its power. The user need never explicitly constraint the position of any wire or symbol. The positions are implied by the sequence of primitives that appear in an ordered context.

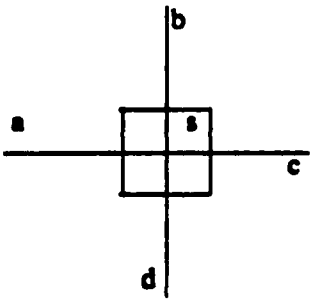


Figure 1

If a wire argument to **place** is NULL, then there is no wire entering from that direction. Note that the four wire arguments need not be distinct: if a wire goes through a symbol, not terminating inside it, then it can enter from both top and bottom, or left and right. The symbol argument can also be NULL, which forces the wires to meet and orders the point in the current context, but does not create a copy of a symbol. Note that a symbol cannot be used where the wires do *not* meet at a point (see Fig. 2). Cases like this can be created by a Clay function.

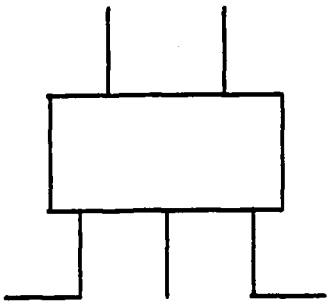


Figure 2

Once a Clay program (foo.c) has been written, it can be translated into a CIF file by the following commands:

```
% cl foo.c
% a.out
% solve
```

cl compiles the source program and loads it with the Clay runtime library. *Cl* is a slightly modified version of the *cc* compiler, with the same options. The execution of *a.out* creates the constraint files. These are put in the current directory as dot files since usually the programmer need never refer to them. Since their names are fixed (for instance: .xconstraint, .yconstraint, .definitions) each Clay program should reside in its own directory. Finally, *solver* reads these files and outputs a file named *out.cif* containing the layout. The plot of the example program is given in Fig. 3.

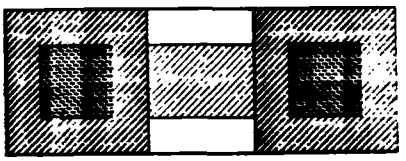


Figure 3

Correcting Errors

Syntax errors are detected by the compiler.

Run time errors are sometimes self-explanatory and sometimes aren't. If the run time system complains about a "negative constraint," a Clay primitive has written a constraint which that the right endpoint of a wire is to the left of its left endpoint. Also, CIF symbols not in the special format described later will be rejected. The run time system can *core dump* for the same reasons an ordinary C program does, such as referencing an uninitialized wire or symbol variable. *sdb* can be used to track down some of these errors.

The most common diagnostic from solver is the infamous "cycle error." This means that the Clay program wrote an inconsistent set of constraints; there is no possible layout satisfying them. For example: a cycle error occurs if the Clay program states that wire A is both above and below wire B. Look for incorrect **place** and **ordered** commands, and misuse of wires that are function arguments. Referencing an uninitialized wire variable may also cause the solver to give a warning about a "coordinate variable number out of bounds."

Many runtime or solve errors can be diagnosed with the aid of the trace package. The trace writes a log of the Clay primitives called (with indentation according to the nesting of ordered contexts) on stderr. **set_trace**(level) turns the trace on or off. The level can be TR_NOTRACE, TR_PARTIAL, or TR_TRACEALL. If enabled, trace also checks for dangling wires at the end of the program run. These are wires with one end unconstrained. Since the solver tries to move the

endpoints of wires as far down and to the left as possible, the free end will stretch to the boundary of the layout, even if it crosses over the other endpoint ("snaps back"— see Fig. 4). The solver gives warnings about wires that are degenerate or snap back and does not place them in the layout. If a wire is created by a call to **ext_wire**, rather than **wire**, its external name will be printed in the trace. The format of the call is **ext_wire**(layer,width,external_name) where external_name is a string.

Finally, the solver can generate illegal CIF if there is a bad symbol file.

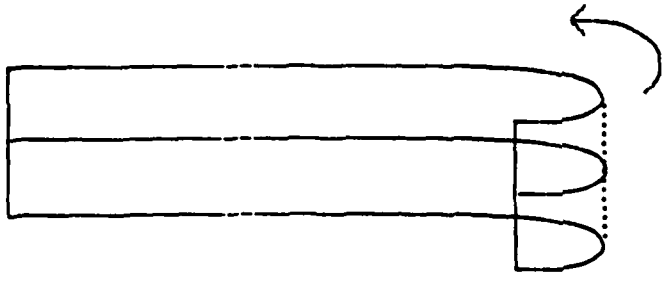


Figure 4

More Primitives

Although **ordered** and **primitive** are powerful enough to describe most Clay designs, other primitives are provided for access to internal data structures, efficiency, or flexibility.

drop(sym,a,b,c,d) takes the same arguments as **place**. **drop** glues up to four wires together in a point, and puts in a symbol over this point, but does not write any other constraints. **drop** is appropriate when symbols are being dropped in over a regular structure which has already been constrained. For instance, a PLA can be created by first laying out a grid of wires, and then dropping in contacts and transistors where needed to define its functions. Because of the risk of design rule violations, **drop** should be used carefully.

override(i) changes the default separation of wires and symbols in an ordered context. The default separation is the maximum imposed by any design rule, which is 3 * LAMBDA in the

current NMOS version of Clay. This means that Clay is not very smart about how close the design rules allow objects to be packed; it assumes the worst case. **override** is intended for hacking low-level cells, where lambdas count. Its integer argument is in centimicrons, but it can be given as a multiple of LAMBDA. Obviously it is possible to create layouts with design rule violations if **override** is used incorrectly. Note that the argument is the *change* in separation—negative to decrease it, positive to increase it.

layer(w), **width**(w), and **direction**(w) have wiretype variable arguments. **layer** returns the layer of the wire. **direction** returns its direction. (TB, LR, BT, or RL). **width** returns the width of the wire in multiples of LAMBDA. These primitives can be helpful when writing a function that needs to find out the type of its wire arguments.

position(w,type) constrains wires to run outside the layout (the outermost context). w is a wiretype variable; type is one of the following: **enter_ left, enter_right, enter_top, enter_bottom, thru_LR,** or **thru_TB**. **enter** forces one end of the wire to be outside and the other end inside; **thru** forces both ends of the wire to be outside.

freewire(w) frees the storage allocated by a call to **wire**. This is 28 bytes per wire in the current version of Clay. **freewire** can be called when the memory requirement of a Clay program becomes excessive due to the creation of many wires.

mark(w,string) is a symboltype-valued function. The one-line CIF symbol it returns puts the string argument as a label on the same layer as the wire, using a Berkeley extension to standard CIF. Placing this symbol somewhere on the wire will label it in the plot. Note that some CIF tools (such as the *crystal* timing simulator) will not recognize a label as being on a wire if it is placed on its endpoint. Instead, the wire should pass through the symbol.

connect(a,b,c,d) forces up to four wires to meet at a point and also places the appropriate symbol to electrically connect them. **connect** is usually preferable to **place** since it automatically creates symbols when needed, and therefore is easier than creating them by hand and less error-prone.

Useful Things to Know

The functions **startup** and **endup** are automatically called by the Clay runtime system at the beginning and end of its execution. These primitives should not be called by the user; we mention them only so their names can be avoided.

If the environment variable claypath is defined, the **symbol** primitive will use this to search for symbol files. claypath should contain the name of one or more directories, separated by colons. These directories are searched in order if the initial open of the file in the current directory fails.

The CIF for a symbol must be in the following canonical format. The first CIF command must be a comment containing two numbers which give the size of the symbol (x and y) in centimicrons. The size is measured as distance from (0,0). So the first line of a symbol of size 1000 x 1000 centered over the origin would be "(500 500);". The next section is a list of macro definitions. The last section is a list of macro calls and box creation commands. Note that some CIF extensions which affect scanning the CIF file, such as the Berkeley CIF include command, are not supported. Also, when a symbol is placed, the CIF origin (0,0) is centered over the point. At present, symbols *must* be symmetric, that is, the boundaries of the symbol cannot be off-center, although the contents of the symbol can be arbitrary.

A major annoyance in the current release of Clay is that there is no way to change orientation. For instance, separate symbols for horizontal and vertical pass transistors are needed. Likewise, if you have written a channel router in Clay with the channels running horizontally, you cannot easily obtain from this a router with channels running vertically except by editing a copy of the function, making the necessary changes. We intend to correct this deficiency in a future version of Clay.

In addition to TB, LR, BT, and RL, contexts may be NONE ordered. The initial context of a Clay program, before the first **ordered** call, is NONE ordered. Symbols placed in a NONE ordered context are constrained to lie inside it, but are not constrained with respect to each other.

Solving Constraints

To write low-level primitives or modify the Clay system, you must understand how Clay generates the CIF layout. The layout is contained entirely within the first quadrant of the Cartesian coordinate plane. When a symbol, wire, or context is created, it is assigned coordinate variables. Since a symbol is placed over a point, it has two coordinate variables (an x coordinate and a y coordinate). A wire has three coordinates: a horizontal wire has two x coordinates associated with it, and a y coordinate; similarly a vertical wire has one x coordinate and two y coordinates. The bounding box of a context has two x coordinates and two y coordinates. The Clay primitives can then control the positions of objects by stating constraints on the values of their coordinate variables. For instance, let vertical wire *a* have x coordinate variable 4, and wire *b* have x coordinate variable 12. (Coordinate variable names are non-negative integers. x variables are even; y variables are odd.) If the Clay program states that the center line of *b* is at least 5 LAMBDAs to the right of the center line of *a*, where LAMBDA is defined as 200, then the execution of the Clay program creates the constraint:

$$x_{12} \geq x_4 + 1000$$

In fact, all constraints generated by Clay are of the form:

$$v_i \geq v_j + d$$

Constraints on x coordinate variables are written in binary in the file .xconstraint. Constraints on y variables are written in .yconstraint. Also, since endpoints of wires can be glued together, as by **drop** or **place**, the Clay program writes a list of commands in .unionfind which force two coordinate variable numbers to be synonyms. In addition, a list of the wires and symbols created is put in .creation, and a list of symbol definitions is put in .definitions.

To obtain a CIF layout, the solver first reads .unionfind and builds a union-find tree. Next on separate passes it processes .xconstraint and .yconstraint to find a layout having smallest total area, using a linear-time algorithm based on topological sort. Finally, solver writes a CIF file by loading the CIF macros for symbols (using .definitions) and writing box creation commands for wires and macro calls for symbols (using .creation).

For dynamic storage allocation in the solver, the maximum internal coordinate variable number and symbol numbers referenced by the Clay program are written in .maxpofile, along with the coordinate variable numbers of the outermost context. These coordinate numbers are needed for hierarchical solving, described in the next section.

Hierarchical Solving

In the Clay examples given so far, an entire layout was described by a single Clay program, and all the constraints were solved in one run of the solver. If a Clay program creating a large layout generates many objects and constraints, the run time of the solver may become excessive and its memory requirements may cause page thrashing. To help avoid this, and for top-down refinement of Clay designs, we allow hierarchical partitioning of Clay layouts into cells, or non-overlapping sections of a layout. A hierarchical layout has a main cell, the parent, containing one or more child cells. Each cell is described by a separate Clay program. This containment is recursive, so a child cell may itself have children. A parent and child cell usually have wires they share that cross the boundary between them, called *parameter wires*. The parameters wires and the outermost context of a child cell are its *externally visible points*.

Since each Clay program must reside in its own directory, we need a separate directory for each cell. The logical hierarchy of cells must be reflected in their directory names. For instance, if *alu* and *control* are children of *mychip*, there is a directory *mychip* with subdirectories *alu* and *control*.

We also allow rigid CIF cells to be children. A rigid cell cannot have its own children.

In a hierarchical layout, the parent and child cells are solved separately. A child may affect the layout of its parent, since it has area and imposes a minimum distance between its parameter wires. Likewise, a parent may affect its child by stretching the distance between parameter wires. To obtain a hierarchical layout, we first compile and execute the Clay programs for all the cells to get constraint files. Then, starting with the lowest-level children (those with no children of their own) we solve to get a layout of the child cell, and append constraints on its size and position of parameter wires to the constraint files of its parent. Then we solve the parents of these cells, on

upward in the hierarchy, until we have solved all the way up to the topmost cell (the root) which has no parent of its own. Then we can solve back down the hierarchy, exporting constraints from parents to their children, and at the same time getting *out.cif* files for the individual cells. When we have solved all the leaf cells on this downward pass, the concatenation of all the *out.cif* files yields the complete layout. The *cifcat* command concatenates CIF files with macro renumbering and handles the CIF End command so the resulting file is palatable to most CIF tools. The arguments to *cifcat* are names of files to be concatenated, and it writes to its standard output (which can be redirected).

The solver works on only one cell in the hierarchy at a time. That is, in the directory of any cell, we can run *solve -u* to solve up, exporting constraints to the parent, *solve -d* to export constraints to children and get an *out.cif* file, or a simple *solve* to get *out.cif* without affecting children. Since the solver must be invoked more than once on a hierarchical layout, you may want to write a shell script to make this more convenient.

Next we will explain how to define parameter wires in a Clay program and describe the hierarchy of parent and child cells. Parameter wires and child cells are identified by name. The primitive for creating parameter wires is **ext_wire**(layer,width,name), described previously. The external name of a wire is returned by the **name**(w) primitive. If w was created by **wire**, not **ext_wire**, then **name** returns NULL. Each wire created by a call to **ext_wire** has an entry in .symtab with its coordinate numbers.

ext_ordered(direction,name) creates a context for a child cell. The context is an externally visible object with an entry in .symtab. Parameter wires can be placed between **ext_ordered** and **leaveordered**. The parameter wires between a cell and its parent should be constrained by calls to **position**.

A *floorplan* in a parent cell directory tells the solver the names of children and the names of the parameter wires. The floorplan has an entry for each child cell. The first line of each entry is of the form:

type childname directory

Type is either *flexible* or *rigid*. Flexible cells are those described by the constraint files of a Clay program execution; rigid cells are in CIF. *childname* is the name given in the **ext_ordered** call. *directory* is the name of the subdirectory containing the child. For sanity's sake, this should usually be the same as *childname*. Remember that there must be a separate directory for each child, even if they are identical copies of the same layout. For instance, if your layout has 8 input pads, there must be a separate directory for each instance of an input pad.

Following this comes a list of the child cell's parameter wires, which we call a *walk*. The walk must totally order all the externally visible wire coordinates. The x-coordinate walk comes first (implicitly beginning with the left side of the cell and ending with the right), then the y-coordinate walk (which likewise implicitly begins with the bottom and ends with the top of the cell). The walk is given simply by listing the wire coordinates, terminated by a *. A wire coordinate is one of the following:

x(wirename)	x-coordinate of vertical wire
x1(wirename)	left x-coordinate of horizontal wire
x2(wirename)	right x-coordinate of horizontal wire
y(wirename)	y-coordinate of a horizontal wire
y1(wirename)	lower y-coordinate of vertical wire
y2(wirename)	upper y-coordinate of vertical wire

where *wirename* is the name given to the wire in the call to **ext_wire**. For instance, the floorplan entry of the flexible cell in Fig. 5 is shown below:

```
flexible onebitadder onebitadder
x2(gnd) x2(cin) x(sum) x(data1) x(data2) x1(cout) x1(vdd)
*
y(gnd) y2(data1) y2(data2) y(cin) y(cout) y1(sum) y(vdd)
*
```

Since the floorplan walk imposes a *total ordering* on the parameter wires, even if they are not otherwise related to each other in the Clay program, you may need to fine-tune a floorplan entry if the ordering causes the cell to stretch unnecessarily. For instance, in the floorplan entry for onebitadder, the y-walk forces *cout* to be at the same y-value or above *cin*, even though the Clay program may allow them to float. If the cell stretches badly because of this, y(cout) should appear before y(cin) in the floorplan.

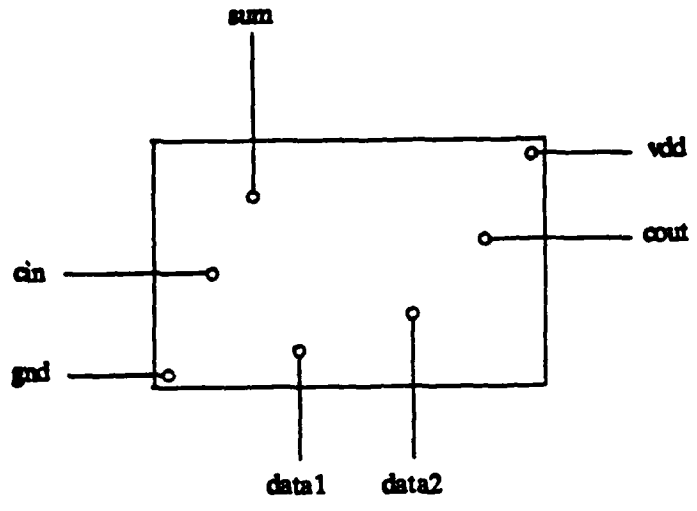
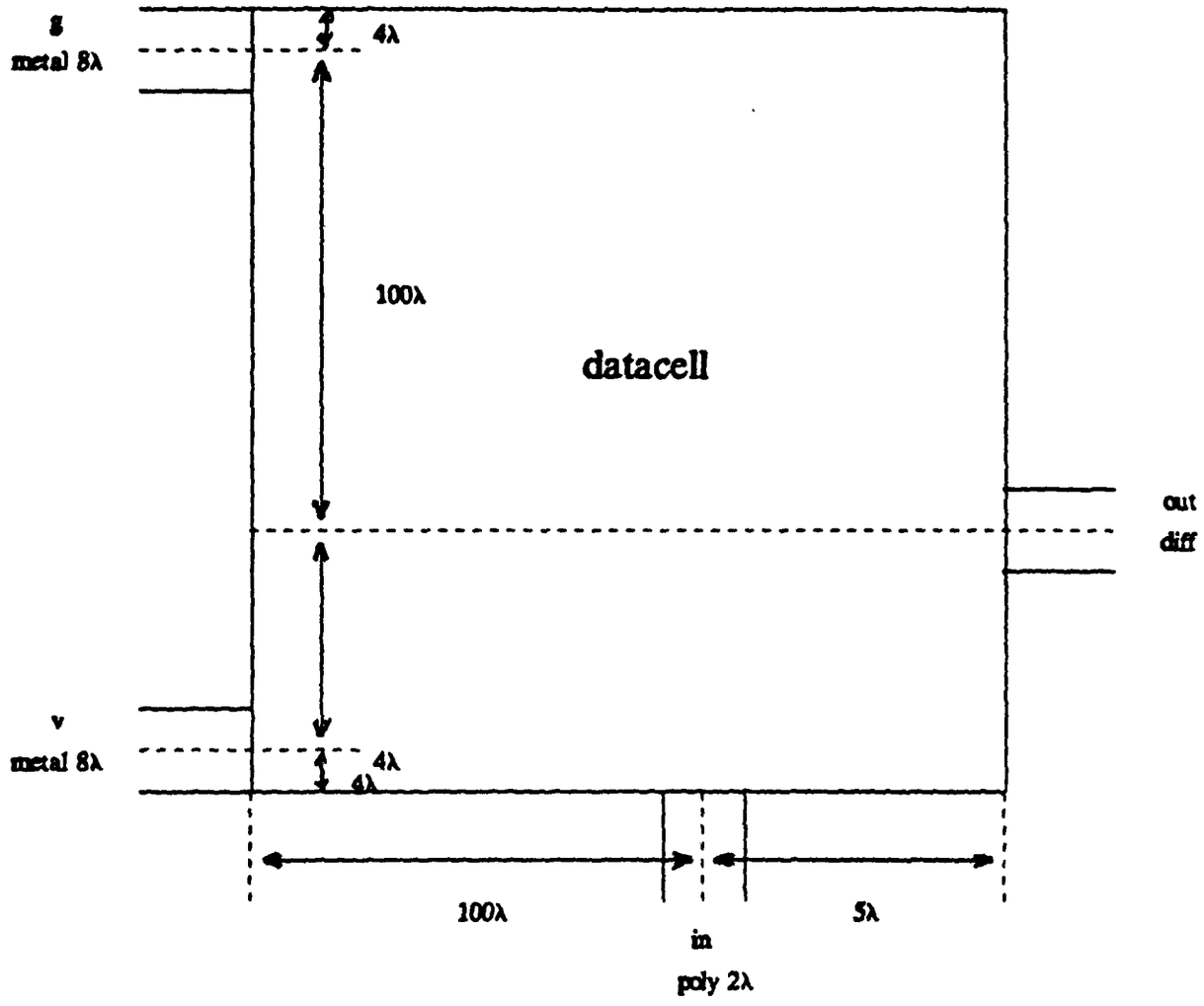


Figure 5

The walk for a rigid CIF cell is slightly different, since we also need to specify the exact separation of parameter wires measured between wire centers, and the wire types and widths. These are given in centimicrons (not LAMBDAs). The walk begins and ends with the separation from the cell boundary. The entry for each wire is the wire coordinate, its type ('m', 'p', or 'd'), and its width. The separation appears between the wire entries. The floorplan entry for the rigid cell of Fig. 6 is given below:

```
rigid datacell datacell
0 x2(v) m 1600 0 x2(g) m 1600 20000 x(in) p 400 10000 x1(out) d 400 0
*
0 y2(in) p 400 800 y(v) m 1600 6000 y(out) d 400 20000 y(g) m 1600 800
*
```

The CIF for a rigid child cell must be placed in the file *in.cif* in its directory. During solve up, the constraints implied by the rigid cell's walk are exported to its parent. Then, during solve down, rather than exporting constraints, the parent checks that it has not tried to change the separation of the rigid cell's parameter wires, and writes *out.cif* in the child directory by translating *in.cif* to its position in the layout. *in.cif* must be in the canonical format described earlier for symbols.



λ = 200 centimicrons

Figure 6

There are two other Clay primitives written for separate compilation. **arrayname(name,number)** simply concatenates the string conversion of *number* to *name*, for convenience in giving names to arrays of parameter wires. For instance, **arrayname**("data",5) is "data5". **put_in_cif(cellname,ciffile,floorplan_header)** makes it easier to incorporate rigid cells in Clay programs. Its arguments are string pointers. The first is the name of an ext_ordered context. The second is the name of a CIF file, and the third is the name of a file containing the floorplan header (X and Y walks). **put_in_cif** creates a subdirectory for the child cell (if needed), copies the named CIF file to *in.cif*, and appends an entry for the rigid cell to the floorplan in the current directory. Since the floorplan is modified every time **put_in_cif** is called, you will need to make a backup of *floorplan* and restore it whenever the Clay program is run. Otherwise, **put_in_cif** will append multiple copies of the same entry to *floorplan*.

A Simple Router in Clay

As another Clay example, consider the following function which is a one-sided channel router using a greedy allocation strategy. Metal wires enter from the top; the router connects the nets on poly. The function call is gcroute(n,a,w) where n is the number of wires, a is the connection list given as an array of n integers, and w is an array of n wires, already separated left to right. $a[i]$ gives the index in w of the next wire to the right of $w[i]$ to be connected in the net, or -1 if $w[i]$ is the rightmost wire in the net.

The router has two phases. In the first phase it assigns channels to the nets. To do this, it works from left to right in the net list, assigning the lowest-numbered channel available. The variable *pos* stores the current position in the left-to-right scan for wire nets. *channel_number*[*pos*] stores the channel number (0 is topmost) chosen for the connection of the net whose leftmost terminal is at *pos*. *chan*[*i*] stores the index of the rightmost wire in the current net connected on channel i. When *pos*>*chan*[*i*], the ith channel can be reused. No actual layout is done during the first phase.

During the second phase, the router works from top to bottom, laying out each channel. Since the wires in w are assumed to be previously separated from each other, each channel is

ordered(NONE). The router then looks through the *channel_number* array to find wires which are the leftmost members of nets running on the channel, and connects the net via a poly wire. All dynamically allocated data structures are freed before the function exits.

```
1:  /*
2:   * greedy one-sided channel router
3:   * metal wires enter from top and nets are connected by horizontal
4:   * poly runs.
5:   *          n is the number of wires
6:   *          a is the connection list.  a[i] gives the index of the next wire
7:   *               to the right of w[i] in the net or -1 if it is the rightmost.
8:   *          w is an array of wires to be connected.  they must already be
9:   *               constrainedin left to right order!
10:  */
11:
12: #include "/vb/clay/lib/header.h"
13: #include <stdio.h>
14: #define NCHANNELS 10  /*max number of channels route can have*/
15:
16: groute(n,a,w)
17: int n,a[];
18: wiretype w[];
19: {
20:      int chan[NCHANNELS];    /*rightmost terminal connected*/
21:      int nextavail = 0;      /*next available channel (lowest numbered)*/
22:      int i,j,k,pos,prev;
23:      int *phase,*channel_number;
24:      int maxused = -1;       /*highest channel number actually used*/
25:      wiretype c;
26:
27:      /*phase keeps track of which wires have been connected*/
28:      phase = (int *) malloc(n * sizeof(int));
29:      for (i = 0; i < n; i++) phase[i] = 0;  /*mark everyone as not seen yet*/
30:      /*channel_number remembers to which channel the leftmost wire
31:       in a net list has been connected*/
32:      channel_number = (int *) malloc(n * sizeof(int));
33:      for (i = 0; i < n; i++) channel_number[i] = -1;  /*mark as not used yet*/
34:      for (i = 0; i < NCHANNELS; i++) chan[i] = -1; /*not used yet*/
35:
36:      /*first phase is to compute connections to channels*/
37:      pos = 0;
38:      while (pos < n)
39:      {
40:           if (a[pos] <= pos)    /*can't go from right to left in the net list*/
41:                fprintf(stderr,"route: attempt to connect term %d to %d0',pos,
42:                a[pos]);
43:           channel_number[pos] = nextavail;
44:           i = a[pos];
45:           phase[pos] = 1;
46:           while (a[i] != -1)    /*scan middle contacts*/
47:           {
48:                phase[i] = 1;
```

```
49:                 i = a[i];
50:             }
51:             phase[i] = 1;           /*rightmost contact*/
52:             chan[nextavail] = i;    /*mark how far we used*/
53:             /*move to next position and find nextavail channel*/
54:             while ((phase[pos]) && pos < n) pos++;
55:             if (pos == n) break; /*all done*/
56:             prev = nextavail;
57:             for (nextavail = 0; nextavail < NCHANNELS; nextavail++)
58:                 if (chan[nextavail] < pos) break;
59:             if (nextavail == NCHANNELS)
60:             {
61:                 fprintf(stderr,"couldn't route in %d channels0',NCHANNELS);
62:                 exit(-1);
63:             }
64:             if (nextavail > maxused) maxused = nextavail;    /*remember max*/
65:         }
66:
67:         /*second phase is to create layout*/
68:         ordered(TB);    /*go by channels*/
69:         for (i = 0; i <= maxused; i++)
70:         {
71:             ordered(NONE);      /*use ordering of w[] within channels*/
72:             /*could speed up by having a list per channel; not worth the trouble*/
73:             for (j = 0; j < n; j++)
74:             {
75:                 if (channel_number[j] != i) continue; /*ignore if not leftmost*/
76:                 /*do leftmost terminal*/
77:                 c = wire(POLY,MIN);             /*poly wire for channel*/
78:                 connect(NULL,w[j],c,NULL);
79:                 k = a[j];
80:                 while(a[k] != -1)       /*do middle terminals*/
81:                 {
82:                     connect(c,w[k],c,NULL);
83:                     k = a[k];
84:                 }
85:                 /*do rightmost terminal*/
86:                 connect(c,w[k],NULL,NULL);
87:                 freewire(c);
88:             }
89:             leaveordered();
90:         }
91:         leaveordered();
92:         free(phase);
93:         free(channel_number);
94:     }
```

A plot of a layout created by this function is given in Fig. 7.

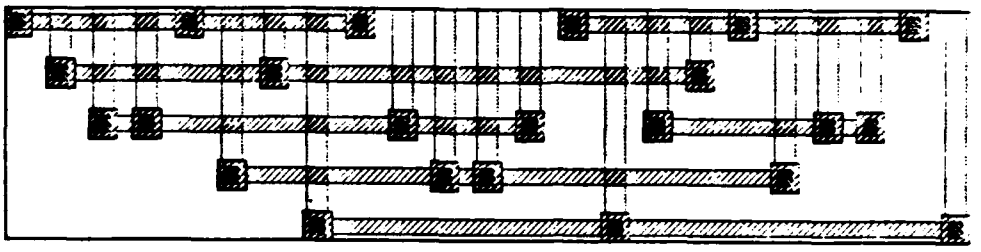


Figure 7

Acknowledgments

The Clay primitives and solver were written by R. J. Lipton and S. C. North. Our first users, J. Lucas and D. Souvaine, wrote many of the library functions and helped greatly to debug and refine the system. Support for R. J. Lipton, J. Lucas, and D. Souvaine was provided under DARPA contract N00014-82-K-0549. S. North was supported by Bell Laboratories. The trace package was written by Tom Freeman.

ALI2: A VLSI Layout System

(Draft)

J. Mata, G. Vijayan

Department of Electrical Engineering and Computer Science
Princeton University
Princeton, NJ 08540

1. Introduction

In this paper we describe the main features and usage of a language designed at Princeton to automate the layout of VLSI circuits. The language is called ALI2 and has been operational for some months at Princeton. The language ALI1, also developed at Princeton was a forerunner to ALI2.

The main thesis in the ALI project is that VLSI design can be profitably thought of as a *programming task*, as opposed to a geometric editing task. We believe that making layout design similar to software design has many advantages and that much is to be gained by consciously attempting to apply our knowledge about programming to this new activity. We have thus tried to create tools for the VLSI designer that incorporate many useful features of the software development tools that we are familiar with.

The main feature of ALI2 as a layout language is that it allows its user to design layouts at a *conceptual level*, in which only the topological relations between the layout components can be specified. Absolute positions of layout components cannot be specified.

2. An overview of ALI2

ALI2 programs are compiled by first translating the ALI2 statements into standard Pascal. Partly as a consequence of this arrangement and partly for aesthetic reasons, ALI2 programs look very much like Pascal programs.

The objects manipulated by ALI2 programs can be classified naturally into two categories: those that a normal Pascal program can manipulate (which will be called *Pascal objects*) and those that are specific to ALI2 (*ALI2 objects*). There are three ALI2 objects: *cells*, *boxes*, and *wires*. ALI2 programs can also manipulate aggregates of wires, just as Pascal programs can manipulate aggregates of variables using structured types. Although ALI2 programs will typically manipulate all three kinds of ALI2 objects, the final product of an ALI2 program is a layout consisting entirely of wires. Cells and boxes are simply used as ways to express the relations between groups of wires in a structured and systematic way.

A *cell* in ALI2 is a prototype for a rectangular section of a layout. In a cell definition, the user describes a prototype of a rectangular layout piece. In a cell creation, also called instantiation, the user requests the insertion of an instance of a previously defined cell in a given environment. Multiple instances of a prototype can be created. It is possible to define a cell prototype whose content and structure depends on the values of parameters which will be supplied to the prototype at run-time. The sizes and shapes of actual instances of a given cell will then vary according to the "actual parameters" provided when the instance is created. Thus, ALI2 cells are very much like the familiar parameterized procedures and functions.

Each cell instance is enclosed in a *cell bounding box*; cells are thus restricted to have rectangular shape. Cell boundaries may not overlap, nor may they be crossed by any wires. Wires will either be entirely contained within a given cell instances, or lie entirely outside it. Cell boundaries therefore impose a strict hierarchy on the arrangement of wires in a layout.

Wires are rectilinear objects which lie on a specific *layer*, have a given *width*, and carry a specified *signal*. Wires are used to interconnect cells and must have both of their endpoints lying on cell boundaries.

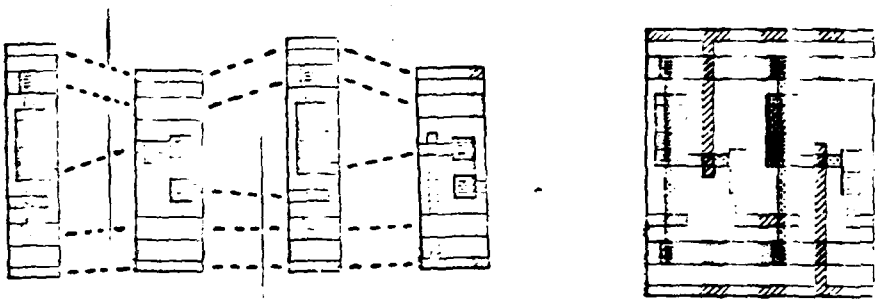


Fig. 1 - Four separate cells and the result of connecting them

The entire layout generated by an ALI2 program is itself actually an instance of a single cell defined by the program. An ALI2 program produces a set of linear inequalities involving the coordinates of the endpoints of the wires and boxes in the layout as variables. These inequalities, which embody the relations between the wires and boxes of the layout, are then solved to generate the positions and sizes of the layout elements. The program also produces connectivity information about the wires in the layout. This information can then be used by a switch level simulator that predicts the behavior of the circuit as laid out without having to perform the usual "node extraction" analysis on the resulting layout.

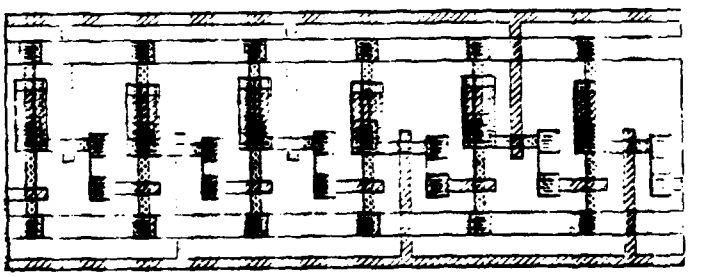


Fig.2 - Layout produced by an ALI2 program

```
chip shiftregister (output);

wiretype polywire = wire (poly, 2*lambda, nullsignal);
         diffwire = wire (diff, 2*lambda, nullsignal);
         metalwire= wire (metal,4*lambda, nullsignal);
         fivewires (lr: layer) = bus w1: polywire;
                                     w2: metalwire;
                                     w3: wire (lr, minwidth(lr), nullsignal);
                                     w4: metalwire;
                                     w5: polywire;
                                 end;
wirevar ll, rr: fivewires (poly);

cell contact (left l: wire; top t: wire; right r: wire; bottom b: wire);
begin
  create syscontact ( l.l, t.l, r.l, b.l ) (false)
end;

cell inverter ( left l: fivewires; right r: fivewires );
wirevar diff1, diff2, diff3: diffwire;
begin
  ordered ttob do begin
    create contact ( l.w1, nullwire, r.w1, nullwire );
    create contact ( l.w2, nullwire, r.w2, diff1 );
    create syspullup ( nulllist, ldiff1l, lr.w3l, ldiff2l ) (4);
    create systransistor ( l1.w3l, ldiff2l, nulllist, ldiff3l ) (false);
    create contact ( l.w4, diff3, r.w4, nullwire );
    create contact ( l.w5, nullwire, r.w5, nullwire );
  end (ordered);
end;

cell cl ( left l: fivewires; right r: fivewires );
wirevar poly1, poly2: polywire;
        diff1: diffwire;
        met1: metalwire;
begin
  ordered ttob do begin
    create contact ( l.w1, nullwire, r.w1, poly1 );
    create syscontact ( l1.w2l, lpoly1l, lr.w2l, lpoly2l ) (true);
    ordered ltor do begin
      create rotated2D systransistor ( lpoly2l, ldiff1l, nulllist, l1.w3l ) (false);
      create contact ( diff1, nullwire, nullwire, met1 );
    end (ordered);
    create contact ( nullwire, met1, r.w2, nullwire );
    create contact ( l.w4, nullwire, r.w4, nullwire );
    create contact ( l.w5, nullwire, r.w5, nullwire );
  end (ordered)
end;

cell cl2 ( left l: fivewires; right r: fivewires );
wirevar poly1, poly2: polywire;
        diff1: diffwire;
        met1: metalwire;
begin
  ordered ttob do begin
    create contact ( l.w1, nullwire, r.w1, nullwire );
    create contact ( l.w2, nullwire, r.w2, nullwire );
    ordered ltor do begin
      create rotated2D systransistor ( lpoly2l, l1.w3l, nulllist, ldiff1l ) (false);
      create contact ( diff1, nullwire, nullwire, met1 );
    end (ordered);
    create contact ( nullwire, met1, r.w2, nullwire );
    create syscontact ( l1.w4l, lpoly2l, lr.w4l, lpoly1l ) (true);
    create contact ( l.w5, poly1, r.w5, nullwire );
  end (ordered)
end;



cell shift ( left l: fivewires; right rr: fivewires );
wirevar mm1,mm2: fivewires (diff);
        mm3: fivewires (poly);
begin
  create inverter ( ll, mm1 );
  create cl ( mm1, mm2 );
  create inverter ( mm2, mm3 );
  create cl2 ( mm2, rr);
end;

cell shiftregister ( left inbus: fivewires; right outbus: fivewires ) (length: integer);
wirevar temp: fivewires (poly);
begin
  if length = 1 then
    create shift ( inbus, outbus )
  else begin
    create shift ( inbus, temp );
    create shiftregister ( temp, outbus) ( length - 1 )
  end (if)
end;

create shiftregister ( ll, rr ) ( 3 )

end.
```

Fig. 3 - An ALI2 program

3. Main Features of ALI2

3.1. Type Structure

The wires manipulated by ALI2 are declared by stating their *name* and their *type*. Wires can be of a *simple type* (a single wire) or of a *structured type* (a group of wires).

ALI2 is a strongly typed language. The ALI2 compiler will perform type checking just as compilers for conventional languages do. Type checking can be effective in catching certain errors very early during the design phase. For example, cells can be designed to accept only certain types of wires, and any violation will be reported during compilation time even before the layout is actually produced.

Wire types in ALI2 are parametric types. Parametric types are designed to make type checking more selective or weaker as the user wishes.

In ALI2 there is just one predefined wire type called *wire*. This parametric type has three parameters corresponding to the three attributes of a wire:

wire (l: wirelayer; w: integer; s: signal)

The types *wirelayer* and *signal* are predefined scalar types. The parameter *w* stands for the width of the wire.

Other parametric types can be defined by pseudo-calls to the type *wire* . For instance, the following type definition

polywire (w: integer) = wire (poly, w, nullsignal)

creates a new parametric type *polywire*. All wires of this new type will have *poly* as their layer and *nullsignal* as their signal. The following wirevar declaration

*mywire: polywire (2*lambda)*

creates a *poly* wire with width *2*lambda*.

The values used as actual parameters can be arbitrary expressions of the appropriate type. These expressions will be evaluated at run time. Thus if *k* is a variable of type integer defined in the current scope, the following would have been a legal type declaration:

*localpoly = polywire ((2*k - 1)*lambda)*

Thus the actual parameters of the parametric types of ALI2 are *bound* at run time. This allows for a great deal of flexibility and permits the construction of dynamic types within a cell.

There are three composite wire types in ALI2: *bus, bundle* and *list*. The types *bus* and *bundle* are roughly analogous to the *array* and *record* types of Pascal, and represent, respectively aggregates of wires of the same type and aggregates of wires of different types. The type *list* is peculiar to ALI2. A list is either the *nulllist* or an aggregate of one or more wires, each of any type whatsoever. This type is intended to facilitate the writing of general-purpose cells which accept a variable number of wire parameters.

The accessing of the elements of bundles and buses is done as in Pascal. Accessing of lists is similar to that of bundles. ALI2 also provides the user with a number of predefined functions that take composite or simple wires as parameters and return various interesting attributes of the wires like layer, width, number of elements, etc.

3.2. Cell Mechanism

Perhaps the most powerful feature of ALI2 is its procedure-like mechanism for the definition and creation of *cells*. The cell mechanism permits the users of ALI2 to introduce hierarchical information into their programs, and therefore into the layouts they describe.

A cell is a collection of related wires enclosed in a rectangular area. Wires that are inside a cell are of two types: *local* which are invisible to the outside, or *parameters* which can interact in a simple and well defined manner with wires outside the cell.

A cell is *defined* by specifying its local objects, its formal parameters and the relations among all of them. Once a cell has been defined, it can be *instantiated* as many times as desired by specifying the actual parameters for the instance, much the same way as one invokes a procedure or function in a procedural language. The result of instantiating a cell is to create a brand new copy of the prototype described in the cell definition with the formal parameters connected to the actual parameters.

The body of a cell will contain Pascal and ALI2 statements. Cells can be defined to be 'external' cells and separately compiled. Cells can also be 'rigid' cells to indicate that the cell definition is not given textually as part of the ALI2 program but instead the actual layout produced by a previous instantiation of the cell is to be used.

Cells are instantiated by the **create** statement, and the parameter list of the cell contains both wire parameters and other parameters.

The cell mechanism helps in the automatic generation of constraints in many ways: local wires and cells are put inside the cell bounding box, wire parameters are separated, and cells that share a parameter are automatically separated.

The cell mechanism gives the ALI2 user the ability to describe layouts in a truly hierarchical manner. A proper ALI2 design, very much like a well structured program, will consist of a hierarchy of cell instances with only a small amount of information at a given level (the parameters of the cell instances at that level) being visible from the immediately higher level. Cells can be written and debugged separately and then put together with the least effort to obtain more complicated cells.

Much of the power and generality of the cell mechanism of ALI2 comes from the absence of absolute positions and sizes in a layout specification. We believe that no cell mechanism can be said to be truly general unless the sizes of its parameter wires and local wires, as well as the relative distances between them are determined at the time the cell is instantiated.

The primitive cells in ALI2 are the predefined cells. These are the cells that appear at the leaves of the hierarchy of cells. In fact, the whole layout can be viewed as a collection of primitive cells joined together by straight line wires. The higher level cells are just rectangular regions enclosing subsets of these primitive cells.

The primitive cells in ALI2 are called *systransistor*, *syscontact* and *syspullup*. These are quite general cells that implement the transistor, contact, and pullup of nMOS. Each of these primitive cells have four parameters: four lists of wires, one for each side of the cell. The contents of an instance of a primitive cell will depend on the attributes of the actual parameter wires used in that instance. So, these cells are 'smart' cells which do a large amount of processing internally.

There are also some non-wire parameters to these cells, which also contribute to the contents of an individual instance. The systransistor cell has a boolean parameter which determines whether the transistor is implanted or not. The pullup ratio is a parameter to the syspullup cell. The syscontact cell has a boolean parameter which determines whether all the wires are to be electrically connected at the contact, or only the wires on independent layers are to be connected to each other.

The reason for making these primitive cells general and thus having fewer number of these cells, is to keep the number of technology dependent features of the language small. However, the user can define simpler versions of these cells to facilitate their repeated invocation. As

mentioned earlier, all the technology dependent features of ALI2 are hidden inside the design rules table, the primitive cells, and a few reserved identifiers. Even in the design rules table only the separation and width rules are stored, because the other design rules are enforced inside the primitive cells. ALI2 currently supports only nMOS primitive cells. Design of cells for other technologies is currently under investigation.

3.3. Placement

Placement is specified implicitly by *create* statements, or explicitly by the *ordered* and the *separate* statements. These statements are used to relatively place the various objects (wires and bounding boxes) in the layout.

The *ordered* statement is given a direction of separation, and a list of creations of objects, and its effect is to place the created objects in the order in which they are created.

```
ordered ltor do
    begin
        < bounding box 1 >
        < bounding box 2 >
        ordered ttob do
            begin
                < bounding box 3 >
                < bounding box 4 >
            end;
        < bounding box 5 >
    end
```

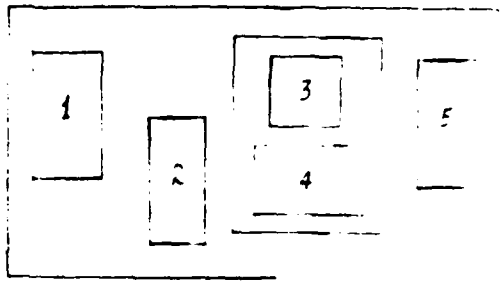


Fig. 4 - *ordered* statement

The actual objects that are ordered within an ordered statement are really bounding boxes Each ordered statement or cell create statement is associated with a rectangular bounding box The bounding box created for an ordered statement will enclose the bounding boxes created for the statements within its scope, and in addition these bounding boxes will be separated in the given direction.

Since ALI2 is an extension of Pascal, repetition statements of Pascal can be used within an ordered statement to create a succession of objects that are separated as specified

The ordered statement matches quite well with the notion of floor-plans of layouts. Once the ALI2 user has a rough sketch of the floor-plan of his layout, he can quickly translate the sketch into a series of nested ordered statements. He can then refine each of his regions in the floor-plan in a similar manner.

Both the cell structure and the ordered statement contribute to the hierarchy in the layout description. However, there is a fundamental difference in the hierarchies created by the cells and the ordered statement: wires cannot straddle the bounding box of a cell, but the same is not true for an ordered statement. Thus, wires are subject only to the hierarchy defined by the cell boundaries. The combination of strict hierarchy of the cell structure and the lenient hierarchy of the ordered statement seems to give the ALI2 user the right mixture of rigidity and flexibility that he needs.

The other placement statement – the *separate* statement – is used to separate a given list of bounding boxes and wires in a given direction of separation. Unlike the ordered statement, the separate statement is not a structured statement. Its analogy in programming languages is the *go to* statement. An ALI2 program can be written without using the separate statement, but it may be used to make small local changes in the layout to avoid rewriting major portions of the ALI2 program.

4. Layout Issues Addressed in ALI2

A sample of the main issues that we tried to address with ALI2 are the following:

- The creation of an *open ended tool*. Most layout design tools require the specification of absolute sizes and positions, thus making the creation of a general purpose library of cells a hard task, since information about the sizes and positions of the cell elements that can interact with the outside world has to be apparent to the user of the library. The absence of absolute sizes and positions makes this problem much less severe in ALI2. ALI2 has been built on top of Pascal, and is a full-fledged programming language having all the powers of Pascal, thereby making it easily extensible. The generation of tools to automate the layout process, such as simple routers or PLA generators, involves writing Pascal routines to solve some abstract version of the problem and having done so invoke ALI2 cells to generate the layouts.
- Facilitating the *division of labor*. Large layouts have to be produced by more than one designer. If the piece produced by each designer is specified in absolute positions, serious problems are likely to arise when the different pieces are put together. ALI2 allows the partitioning of tasks in such a way that the designer of a piece of the layout does not need to know anything about the positions or sizes of other pieces of the complete layout.
- Facilitating *hierarchical design*. In ALI2, the information about a given level of the hierarchy needed at the level immediately above is reduced by the absence of absolute sizes and positions, to topological relations among the layout elements of the lower level visible to the higher one.
- Facilitating easy *update of layouts*. Successful designs seem to be more or less continuously updated as improved processes become available during their lifetime. Therefore, layout tools must be easily amenable to changes in the technology or design rules. The technology dependent part of ALI2 is confined to a few design rules tables and primitive cells and only these have to be rewritten in order to update ALI2 to a new technology. Future versions of ALI2 will give its user the flexibility of writing one ALI2 program to describe a layout, and then producing different layouts for different processes by just setting certain appropriate flags when invoking the ALI2 system.
- Allowing *parametric design*. Having a layout design which produces different layouts for different values of a set of parameters is extremely useful. This is especially true for cell designs which are used repeatedly. These parameters will allow decisions about the detailed characteristics of the cell in a layout to be delayed until later in the design phase. In ALI2, the cell mechanism has been designed so that the number as well as the attributes of the wires connecting to a cell can be parameters of the cell. In addition, the cells can have other parameters that affect the insides of the cell. ALI2 offers all the wealth of a full-fledged programming language, such as do-loops, conditional statements etc., which can be used to exploit the availability of these parameters.
- To allow *easy modification of layouts*. The fact that absolute sizes and positions are absent in an ALI2 specification makes modification of a layout a very simple task. Such modifications are actually being made to a program, which is a much easier task compared to making changes in the final layout.

5. The ALI2 System

The ALI2 program takes as input an ALI2 program, with precompiled cells or rigid cells, and produces the layout in CIF (Caltech Intermediate Form) code, or alternatively a precompiled cell or a rigid cell, and connectivity information for simulation. There is a switch-level simulator, described in [11]. The CIF code is then used to interface with other CAD tools, like Berkeley VLSI Tools [9]. There is also a program that takes a CIF code and transforms it into a rigid cell, to be used by any ALI program. Also, the node information for simulation can be obtained from the CIF code.

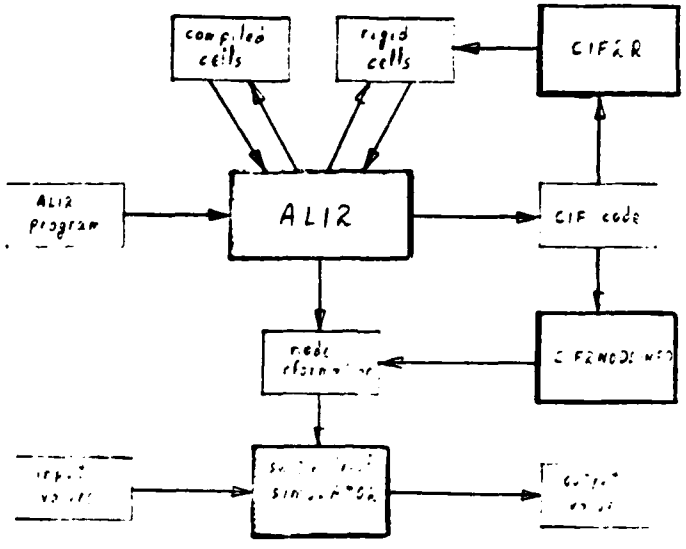


Fig. 5 - The ALI2 System

There are 6 steps in going from the text of an ALI2 program to a layout in CIF:

1- **Translation.** The ALI2 program is translated into Pascal.

2- **Compilation.** The Pascal program is compiled, producing an object file.

3- **Loading.** The object file generated by the previous step plus several other standard object modules are made into a single executable file.

4- **Execution.** The executable file is executed, producing a file of linear constraints, and optionally connectivity information.

5- **Solving.** The set of linear constraints is run through the solver program, and an internal representation of the layout is produced.

6- **Generating CIF.** The internal representation (in lambda units) is converted to CIF (centimicron units).

The whole system is implemented under Berkeley UNIX, and the system is very efficient. The translator was written using YACC. The compiler is the Berkeley Pascal Compiler. Execution doesn't take too much time, since its basic operation is to write down constraints every time a cell is instantiated. The solver takes linear time relative to the number of constraints. CIF generation is straightforward. So, what takes most of the time is read/write operations, specially for large layouts.

8. Example

One of the chips designed using ALI2 was a n-bit parallel adder, and it is being sent for fabrication. The parallel algorithm used for addition was borrowed from [13].

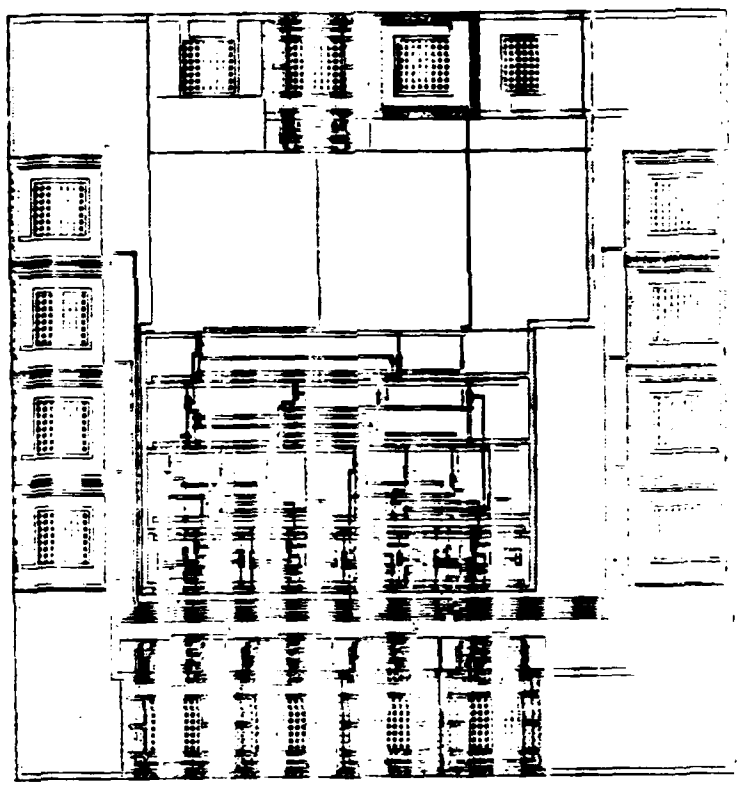


Fig. 8 - A 4-bit adder

This design illustrates the utility of several features of ALI2:

1- General purpose cells such as the aarray cell [8], that was used to generate Weinberger type cells, can be written and used very effectively.

2- It is easy to parametrize cells.

3- ALI2 has the power of a conventional programming language such as recursion, iterative statements and functions.

4- It is quite simple to divide a layout task among several designers.

5- An ALI2 program serves as a good documentation of the design of the layout.

Acknowledgements

The ALI2 system resulted of the work of many people, especially Prof. Jacobo Valdes and Prof. Richard Lipton. We would like to mention the contributions of Ron Kalin and Steve North.

This work was supported in part by DARPA under ONR N00014-82-K-0549.

7. References

[1] Hennessy, J., Elmquist, H. *The Design and Implementation of Parametric Types in Pascal.* Software -- Practice and Experience, vol. 12, 1982.

[2] Jensen, K., Wirth, N. *Pascal User Manual and Report.* 2nd ed., Springer-Verlag.

[3] Johnson, S. C. *YACC: Yet Another Compiler-Compiler.* Unix Programmer's Manual, January 1979.

[4] Kalin, R. L., Valdes, J. *Language Overview.* ALI2 Documentation and Implementation Guide.

[5] Lipton, R. J., Sedgewick, R., Valdes, J. *Programming Aspects of VLSI.* Proc. of the Ninth Annual ACM Symp. on Principles of Programming Languages, 1982.

[6] Lipton, R. J., North, S. C., Sedgewick, R., Valdes, J., Vijayan, G., *ALI: a Procedural Language to Describe VLSI Layouts.* Proc. of the 19th Design Automation Conference, June 1982.

[7] Lipton, R. J., North, S. C., Sedgewick, R., Valdes, J., Vijayan, G. *VLSI Layout as Programming.* ACM Trans. of Programming Languages and Systems, July 1983.

[8] Mata, J. M. *An Array Generator in ALI2.* Department of Electrical Engineering and Computer Science, Princeton University, 1983.

[9] Mayo, R., et al. *1983 VLSI Tools.* Report No. UCB/CSD 83/115, University of California, Berkeley, March 1983.

[10] Mead, C., Conway, L. *Introduction to VLSI Systems.* Addison-Wesley, 1980.

[11] Ramachandran, V. *An improved switch-level simulator for MOS circuits.* Proc. of the 20th Design Automation Conference, June 1983.

[12] Vijayan, G. *Design, Implementation, and Theory of a VLSI Layout Language.* Ph.D. Thesis, Princeton University, August 1983.

[13] Vuillemin, J., Guibas, L. *On Fast Binary Addition in MOS Technologies.* Proc. of the IEEE International Conference on Circuits and Systems, September 1982.

A HIERARCHICAL COMPACTION ALGORITHM WITH LOW PAGE-FAULT COMPLEXITY†

Ming-Deh A. Huang and Kenneth Steiglitz

Department of Electrical Engineering and Computer Science
Princeton University, Princeton, New Jersey 08544

ABSTRACT

The problem of VLSI layout compaction is often reduced to finding optimal solutions to systems of simple linear inequalities and equalities. The commonly used algorithms take only linear time and space by the usual worst case complexity measures, but serious problems of page thrashing often occur when the algorithms are run on systems with large sets of constraints. Page faults must be taken into account if the performance of such algorithms is to be predicted realistically.

In this paper, we first discuss page-fault complexity in the setting of *paged dags*. We then extend the discussion to the case of constraint systems that are hierarchically organized. We present algorithms that find optimal solutions to hierarchical constraint systems with strict bounds on the number of page-faults. These algorithms also run in linear time and space by the usual complexity measures.

1. Introduction

As VLSI logical design problems get more and more complex, there is a trend toward hierarchical design methodology. Several languages (e.g. ALI [LSV], CLAY [N], HILL [LM], SLIM [D]) have been developed for layout specification in which the relative geometric relations and interconnections among the geometric objects in a layout are specified instead of the absolute positions of these objects, and the layout can be specified in a hierarchical way. Usually a layout specification is stated formally as follows:

(1) **SLI Compaction**: Given a set of simple linear inequalities $\{ x_i + d_{ij} \le x_j \}$, find a solution such that $\max_i (x_i) - \min_i (x_i)$ is minimized.

In the case where the set of constraints consists of simple linear inequalities plus simple equalities, the problem can be stated as follows:

(2) **SLIE Compaction**: Given a set of simple linear inequalities $\{ x_i + d_{ij} \le x_j \}$ and a set of simple equalities $\{ x_i = x_j \}$, find a solution such that $\max_i (x_i) - \min_i (x_i)$ is minimized.

We call a set of simple linear inequalities an *SLI system*, and a set of simple linear inequalities and simple equalities an *SLIE system*.

Both problems have efficient algorithms in terms of the usual time and space complexity measures. For SLI compaction, the well known *PERT* algorithm runs in linear time and space. For SLIE compaction, we make a substitution of variables to reduce the problem to compaction for an SLI system. However, when an SLI or SLIE system generated from a VLSI layout specification is too large to fit into the working space of a computer, the system is often partitioned and stored in several pages. Page-thrashing in execution of the *PERT* algorithm then becomes a serious problem, often dominating the rest of the computation. Experiments indicate that as the size of the constraint set grows larger and larger, the problem of page-thrashing becomes more and more significant. Therefore, we must take page-faults into account in any meaningful measure of the complexity of algorithms for these problems.

One way to avoid page thrashing is to find algorithms that are efficient in the length of the layout specification, which is usually much shorter than that of the completely generated constraint set. Lengauer [L] showed that this is possible in some, but not all, cases. In this paper, we adopt a different approach. We assume that the generated constraints are stored explicitly in the secondary memory, which is divided into pages of fixed size. We will show that when the constraint systems are hierarchically organized, page-swapping can be controlled in such a way that the number of page-faults is strictly bounded. This is practical since we can often organize the generated con-

† This work was supported in part by NSF Grant ECS-8120037, U.S. Army Research Office-Durham, Grant DAAG 29-82-K-0095, and DARPA contract N00014-82-K-0584

straints in a way that reflects the hierarchical specification of a VLSI circuit.

In Section 2, we define hierarchical SLI and SLIE systems. In Section 3, we discuss page-fault complexity in the general setting of paged-dags. In Section 4, the basic ideas presented in Section 3 are applied to hierarchical SLI and SLIE systems. We present new algorithms that find optimal solutions to hierarchical SLI and SLIE systems in a hierarchical way, with strict bounds on the number of page-faults. These algorithms run in linear time and space, which is also best possible by the usual time and space complexity measures.

2. Hierarchical SLI and SLIE Systems

We assume that the simple linear inequalities and simple equalities in an SLI or SLIE system are stored in the following way: for each variable x_i, there is a list of tuples (x_j, d_{ij}), where $x_i + d_{ij} \le x_j$ is a constraint, a list of tuples (x_k, d_{ki}), where $x_k + d_{ki} \le x_i$ is a constraint, and a list of elements x_l where $x_i = x_l$ is a constraint. We assume throughout that $d_{ij} > 0$, and the constraints are acyclic.

We also assume that the storage structure consists of a fast memory, which we call the *main memory* (or the *working space*), and a slower memory, which we call the *secondary memory*. The secondary memory is partitioned into *pages* where each page has a fixed amount of space. Suppose now a set of constraints is stored in several pages and each page stores a disjoint subset of variables with their adjacency lists. We may think of each page as representing a subset of variables, and the union of these subsets is the whole set of variables. We call an SLI or SLIE system stored in this way a *paged* SLI or SLIE system.

A paged SLI or SLIE system can be organized hierarchically. More formally, let V be the set of variables of an SLIE or SLI system. Let $\hat{V} = \{ V_1, \ldots, V_n \}$ be a partition of V. We call each subset V_i of V a *block* at level 1. A block at level 1 corresponds to a page. Now we can similarly partition $\hat{V}$ into subsets of blocks at level 1 and call each subset a *block* at level 2. A block at level 2 thus contains several blocks at level 1 as members. This process can be continued to higher levels. We stop when we get to a block that contains all of V, and such a hierarchy can be represented by a tree. The root is the block at the highest level, the leaves are the blocks at level 1, which correspond to the pages. When a paged SLIE (or SLI) system is hierarchically organized, we call it a *hierarchical* SLIE (or SLI) system.

Definition Suppose blocks V_1, V_2 are both members of block V. When there is a constraint relating a variable x_1 in V_1 and a variable x_2 in V_2, we say that x_1 and x_2 are *outer-variables* of V_1 and V_2 respectively.

Note that the outer-variables of a block are the variables that interact with variables of other blocks at the same level. Usually, the number of outer-variables is small compared to the total number of variables.

Before going further, we present a slightly modified version of the *PERT* algorithm and point out how page-thrashing may occur. Let us extend slightly the definition of an SLI system to include, besides a set of simple linear inequalities, a set of constraints $\{ x_i = x_{i0} \mid x_i$ is a variable, x_{i0} is a constant $\}$. We call such an extended system a *preconditioned* SLI system. The following modified version of the algorithm *PERT* solves the compaction problem for a preconditioned SLI system in a way similar to topological sorting [K]

Algorithm SIMPLE PERT

Input : a set of simple linear inequalities together with a set $\{ x_i \ge x_{i0} \mid x_i$ is a variable, x_{i0} is a constant $\}$

Output : $\{ t(x_i) \mid x_i$ is a variable $\}$ (**comment** an optimal solution to the preconditioned SLI system where $t(x)$ is the value for variable x)

begin
$t(x_i) := 0$, $p(x_i) = x_{i0}$, for all i;
$in(x_i) := |\{ x_k \mid x_k + d_{ki} \le x_i$ is a constraint $\}|$;
$S :=$ empty queue;
(**comment** initialization)
  Find all x_i where $in(x_i) = 0$, put x_i in S;
  while S is not empty **do begin**
  Pop a variable x_i from S;
  $t(x_i) := p(x_i)$;
  for x_j such that $x_i + d_{ij} \le x_j$ is a constraint **do begin**
    $p(x_j) := \max (p(x_j), t(x_i) + d_{ij})$;
    $in(x_j) := in(x_j) - 1$;
    If $in(x_j) = 0$, then put x_j in S
    end
  end
end.

Referring to *SIMPLE PERT*, we see that whenever a variable in a page different from that of the current x_i is referenced, a page fault occurs. In practice, for large problems, this can happen quite often, as illustrated by the following simple example of an SLI system. This example also motivates the basic ideas that will be used later on.

Example 2.1 The set of variables $\{ x_{ij} \mid 1 \le i \le 3, 1 \le j \le n \}$ is partitioned into 3 disjoint blocks.

$$V_1 = \{ x_{10}, x_{11}, \ldots, x_{1n} \}$$

$$V_2 = \{ x_{20}, x_{21}, \ldots, x_{2n} \}$$

$$V_3 = \{ x_{30}, x_{31}, \ldots, x_{3n} \}$$

We assume that each block is in one page. The constraints are the following, where d is a positive integer:

$x_{ij} + d \le x_{i(j+1)}$. $i = 1, 2, 3$; $j = 1, \ldots, (n-1)$.

$x_{10} + d \le x_{20}$.

$x_{20} + d \le x_{30}$.

$x_{30} + d \le x_{12}$

Representing a constraint $x + d \le x'$ by $x \to x'$, we obtain a dag G as shown in Figure 2.1. In Figure 2.1, the variables are arranged so that the rows correspond to the blocks, and the columns represent the successive configurations of the queue S when *SIMPLE PERT* is applied. From the picture, we see that $3n$ page-faults occur when *SIMPLE PERT* is applied. Although this example exhibits bad behavior using a FIFO queue, similar examples can be contrived for LIFO and other list-management disciplines.

Now we examine the example more closely and show how page-faults can be reduced. The interaction among the blocks can be represented by a dag H on the outer-variables x_{10}, x_{12}, $i = 1, 2, 3$, as shown in Figure 2.2. The dag H actually represents the dependency relation among the outer-variables. In H, there is an arc from an outer-variable to another if the latter is reachable from the former in G without passing through any other outer-variable. If an outer-variable has no predecessor in H, then it does not depend on any variable external to the block it is in. Therefore we can compute the value for it if the block it belongs to is fetched to the working space. After the value of an outer-variable is computed, we delete it from H. It then holds inductively that at any point of the algorithm, an outer-variable with no predecessors in the remaining part of dag H is one whose predecessors in G that are external to the block it belongs to are all computed. Therefore we can compute such a variable if the block it belongs to is fetched next.

In the beginning, since x_{10} has no predecessor in H, we can fetch V_1 and compute the value for x_{10}. After we compute x_{10}, we can compute x_{11} since x_{10}, its only predecessor, is computed. However, x_{12} cannot be computed since it depends on x_{30} which is external to V_1. Since the value of x_{10} is computed, we delete it from H (Fig. 2.3). Now that x_{10}, the only predecessor to the outer-variables of V_2, is computed, we can compute all the variables in V_2 in the order $x_{20}, x_{21}, \ldots, x_{2n}$. Now we delete x_{20} and x_{22} from H (Fig. 2.4); then we see that we can fetch V_3 and compute $x_{30}, x_{31}, \ldots, x_{3n}$ since all the predecessors of x_{30} and x_{32} that are external to V_3 are already determined. Finally, x_{30} and x_{32} are deleted from H (Fig. 2.5), we fetch V_1 and the values for $x_{12}, \ldots, x_{1n}$ can be computed. In this way, only 4 page-faults occur. •

We call the dags on the outer-variables constructed in Example 2.1 the *outer-dags* associated with the SLI systems. The example illustrates the fact that the outer-dags contain information that is useful for arranging the page-fetching to reduce the number of page-faults.

In the algorithms we will present later on, algorithm *SIMPLE PERT* will only be used locally within a page. We observe that algorithm *SIMPLE PERT* is essentially topological sort on the variables with respect to the partial order induced by the inequality constraints. In the hierarchical situation this approach will be extended (1) to exploit useful partial orders on the outer-variables or interesting sets of outer-variables that are induced by the constraints; (2) to find efficient methods for computing such partial orders.

3. Page-fault Complexity for Computational Dags

In this section, we will discuss page-fault complexity in the setting of *paged computational dags*, which correspond in a natural way to paged SLI systems. We consider a general computational problem that can be characterized by a dag $G = (V,E)$. A node represents a computational step and the arc set E represents the dependency relation on the computational steps. That is if $(v,u) \in E$, then the computation u cannot be done before v. We suppose V is partitioned into subsets $V_1, \ldots, V_p$, and call each subset a *block*. We may think of a block with the adjacency lists of its nodes being stored in a page and therefore we call a dag $G = (V,E)$ with a partition $\hat{V} = \{ V_1, \ldots, V_p \}$ a *paged computational dag*, or simply *paged dag*. If $(u,u') \in E$ with $u \in V_i$, $u' \in V_j$, and $i \neq j$, then we say u is an *outer-node* of V_i, and u' is an *outer-node* of V_j.

In the sequel, we will always use "$*$" to denote transitive closure.

Definition For $u,v \in V$ we say $u \to v$ when $(u,v) \in E$. So $u \to^* v$ if and only if v is reachable from u. For outer-nodes u,u', we say $u <_o u'$ when u' is reachable from u without passing through any other outer-node. •

Definition Let U be the set of outer-nodes. We define a directed graph H on U as follows. $H = (U, E_o)$, and $E_o = \{ (u,v) \mid u <_o v$ and $u,v \in U \}$. It is clear that H is a dag. We call H the *outer-dag* associated with G. •

Definition Let $G = (V,E)$, $\hat{V} = \{ V_1, \ldots, V_p \}$ be as described before. Let $\hat{G} = (\hat{V},\hat{E})$ where $\hat{E} = \{ (V_i,V_j) |$ there are outer-nodes $u \in V_i$, $v \in V_j$, $(u,v) \in E \}$. We call $\hat{G}$ the *super-graph* associated with G and $\hat{V}$. When $\hat{G}$ is a dag, we call it a *super-dag*. •

In particular, for an SLJ system, the corresponding computational dag is $G = (V,E)$ where V is the set of variables, a node represents the determination of the value of the variable in it, and $E = \{ (x_i,x_j) |$ there is a constraint $x_i + d_{ij} \le x_j \}$. In this case, a paged dag is just a paged SLJ system, and a block corresponds to a page of the SLJ system. The outer-nodes are just the outer-variables in V. For variables x_i, x_j, $x_i \rightarrow x_j$ if and only if there is a constraint $x_i + d_{ij} \le x_j$. For outer-variables x_i, x_j, $x_i <_o x_j$ if and only if $x_i \rightarrow^* x_j$ and there is no outer-variable x_k such that $x_i \rightarrow^* x_k$ and $x_k \rightarrow^* x_j$. Note that for every paged dag $G = (V,E)$ with partition $\hat{V}$, there is a paged SLJ system whose corresponding paged dag can be represented by G and $\hat{V}$.

Definition Let $G = (V,E)$ be a dag. For $v \in V$, define $l(v) = 1$ if v has no predecessor, else $l(v) = \max \{ l(u) | u$ is a predecessor of v in $G \} + 1$. Call $l(v)$ the *level* of v in G. •

Definition Let $G = (V,E)$ be a dag. A node v in G is said to be *exposed* if and only if indegree$(v) = 0$, that is, v has no predecessor. •

Given a dag $G = (V,E)$ and a subset W of V, the following procedure recursively deletes the exposed nodes which belong to W. We will use this procedure later on.

Procedure DELETE(G, W)

(**comment** $G = (V, E)$ is a graph, W is a subset of V)
begin
  while there is a node u in W that is exposed **do**
  begin
    $G := G - u$;
    $W := W - u$;
  end.
end •

In carrying out the computational steps of a paged dag $G = (V,E)$, we assume that only one block is allowed to reside in the working space at one time. When a block is fetched to the working space, some nodes in the block can be computed. When it is not possible to proceed any further, a new block must be fetched into the working space and the original block stored back in the secondary memory, in which case a page-fault occurs.

Let $< V_{a_i} >$, $i = 1, \ldots, k$, be the sequence of page-faults that occur in the course of a computation. That is, when the i-th page-fault occurs, block V_{a_i} is fetched to the working space, and $V_{a_i} \neq V_{a_{(i+1)}}$. The number of page-faults is simply the length of the sequence. We call a sequence of blocks a *page sequence*, call a page sequence that corresponds to a full computation of the paged dag a *legal page sequence*.

Suppose that an outer-node is deleted from the outer-dag once it is computed. Then an outer-node becomes exposed exactly when all its precedent outer-nodes are computed and deleted. Now let H be the current outer-dag with all the computed outer-nodes deleted at the time the i-th page-fault occurs, and let U_{a_i} be the outer-nodes of V_{a_i}. When the i-th page-fault occurs and V_{a_i} is fetched to the working space, the exposed outer-nodes in U_{a_i} have all their precedent outer-nodes already computed. Therefore they can be computed and deleted before the $(i+1)$-st page-fault occurs. After they are computed and deleted, there may be new outer-nodes in U_{a_i} that become exposed. Similarly, they can also be computed and deleted from H before the next page-fault occurs. Inductively, we see that the outer-nodes in V_{a_i} that can be computed and deleted are exactly those that would be deleted when procedure *DELETE* is applied to H and U_{a_i}. Therefore, the following is true:

Lemma 3.1 Let $< V_{a_i} >$ be a page sequence. Let U_{a_i} be the outer-nodes of the block V_{a_i}. Let H be the outer-dag with all the computed outer-nodes deleted at the time the i-th page-fault occurs. Then (1) between the i-th and $(i+1)$-st page-faults, the outer-nodes of block V_{a_i} that can be computed are exactly those that are deleted when procedure *DELETE* is applied to H and U_{a_i}. (2) between the i-th and $(i+1)$-st page-faults, the nodes v in V_{a_i} that can be computed are exactly those v in V_{a_i} whose precedent outer-nodes u, where $u \rightarrow^* v$, are deleted before the $(i+1)$-st page-fault occurs. •

Lemma 3.1 shows that the computational effect between two page-faults can be described by a process that recursively deletes the exposed outer-nodes of a block from the outer-dag. To illustrate this idea, let us look at Example 3.1.

Example 3.1 A paged dag G and its associated outer-dag H are depicted in Figure 3.1, where x_{1j}'s are in block V_1, x_{2j}'s are in block V_2.

A legal page sequence is V_1, V_2, V_1. Let us examine the action taken with each page-fault.

V_1 is fetched to the working space (see Figure 3.2).

•node x_{10} is deleted from H, then x_{18}.

•the remaining part of G when the deletion process is applied is shown in Figure 3.2.

V_2 is fetched to the working space (see Figure 3.3):

•node x_{22} is deleted from H, then x_{24}.

•all nodes in V_2 will be deleted.

Finally, V_1 is fetched to the working space:

•node x_{14} is deleted.

•all nodes in V_1 will be deleted. •

Ideally, given a paged dag, we would like to minimize the number of page-faults for a full computation of the dag. That is, we would like to find a legal page sequence that has minimum length for the paged dag. Unfortunately, this problem is NP-complete. Namely, we have the following result:

Theorem 3.1 The following problem is NP-complete: given a paged dag and an integer k, are k page-faults sufficient for computing the paged dag?

The proof of Theorem 3.1 can be found in [H]. it uses reduction from feed-back vertex set. Theorem 3.1 implies that to achieve the minimum possible page-faulting is practically infeasible, assuming of course $P \neq NP$. However, we have the following result:

Theorem 3.2

(1) In the worst case, for a paged dag of p blocks and m outer-nodes in every block, at least mp page-faults are necessary for computing the paged dag.

(2) There is an algorithm that computes a paged dag of p blocks with no more than m outer-nodes in every block in linear time and space, with the number of page-faults no more than mp. •

Theorem 3.2 will be established in several steps, and an algorithm for (2) will actually be constructed.

We prove (1) first by constructing paged-dags for which the worst case occurs, given p, the number of blocks, and m, the maximum number of outer-nodes in a block. Choose positive integers a and b such that $a+b = p$. We form a dag $H = (U, E_0)$ so that level $2i+1$ of H, $i = 0,\ldots, m-1$, consists of a nodes u_{ik}, $k = 1,\ldots,a$, level $2i$ consists of b nodes u_{ik}, $k = a+1,\ldots, p$, $i = 1,\ldots, m$, and for node u, u' in H, (u, u') is an arc if and only if $l(u) = l(u')-1$. It is easy to construct a dag G with partition $\hat{V} = \{V_1, \ldots, V_p\}$ on V such that for $k = 1,\ldots, p$, the u_{ij}'s are in V_k, and the associated outer-dag is the same as H. By Lemma 3.1 and the way H is defined, any legal page sequence for G has length $\geq |U| = mp$. This proves (1).

Now we turn our attention to (2), first finding a legal page sequence.

Suppose $G = (V,E)$ is a paged dag with the associated partition $\hat{V}$ on V. Suppose the associated outer-dag $H = (U,E_0)$ has been determined; then the following procedure will find a legal page sequence for G.

Procedure SEQUENCE

Input: H, the outer-dag associated with a paged dag G.

Output: a legal page sequence S.

```
S := empty sequence; H := the input outer-dag. (comment: initialization)
begin
    while H is not empty do begin
        choose an exposed node u in H;
        if u is from block V_i then apply procedure DELETE on H and the subset of outer-nodes of V_i;
        append V_i to the end of the sequence S
    end
end •
```

Let m be an upper bound on the number of outer-nodes a block can have and p be the number of blocks. Then the length of the legal page sequence determined by procedure *SEQUENCE* $\leq |U| \leq mp$.

Instead of actually constructing the outer-dag and applying procedure *SEQUENCE*, we will describe an algorithm for computing a computational dag that will result in the same legal page sequence as is determined by procedure *SEQUENCE*. The algorithm is based on the argument that was used to prove Lemma 3.1.

Given a set of newly computed nodes in a block V, the following procedure *LOCAL* will compute as many nodes in V as possible. In this procedure as well as in the algorithm we will describe, $S(V)$ will denote a set of newly computed nodes in V, $in(v)$ will denote the in-degree of v for a node v.

```
Procedure LOCAL( S(V), V )
begin
    do until S(V) is empty begin
        pop a node v from S(V);
        for u such that ( v, u ) is an arc do begin
            in(u) := in(u) - 1;
            if in(u) = 0 then do begin
                if u ∈ V, then put u in S(V);
                if u ∈ U_j for some j then put u in S_0(V_j);
                end
            end
        end
end
```

Notice that $S_0(V_i)$ is small since it is a subset of U_i. We may assume that these $S_0(V_i)$ are kept in the main memory throughout the computation. The following algorithm computes a computational dag G with the

same page sequence determined by procedure *SEQUENCE*.

Algorithm COMPUTE DAG
begin
  for each block V **do begin**
    $S(V) := \{ v \in V \mid \overline{in}(v) = 0 \}$;
    LOCAL($S(V)$, V);
    end
  do until all $S_0(V)$ are empty **begin**
    for non-empty $S_0(V)$ **do** *LOCAL*($S_0(V)$, V);
    end
end

We can see by induction that between two page-faults, procedure *LOCAL* computes the nodes which are characterized by Lemma 3.1. And the execution of algorithm *COMPUTE DAG* will result in the same number of page-faults as is determined by procedure *SEQUENCE*, which is bounded by mp. Finally, the total running time and space is linear in the number of arcs, as is clear from the algorithm. •

From the above discussion, we see that the outer-dags associated with the paged-dags play an essential role in studying the page-fault complexity. The ideas presented in this section will be extended in the next section to deal with the general hierarchical situation. Our goal is to obtain an analog of (2) of Theorem 3.2 for the general case of hierarchical SLIE systems, where equality constraints are also involved.

4. Hierarchical Compaction for SLI and SLIE systems

In this section, we will extend the ideas presented in the last section to deal with hierarchical SLI and SLIE systems. All the algorithms presented in this section run in total linear time and space with a strict bound on the number of page faults. In Section 4.1, we discuss an important subclass of SLI systems that are characterized by an acyclic property. In Section 4.2, we discuss the general case of hierarchical SLIE systems.

4.1 Hierarchical Local Pert for Acyclic SLI Systems

First let us extend the definition of super-graph to hierarchical SLI systems.

Definition Let V be a block in a hierarchical SLI system, and let $\hat{V} = \{ v_1, \ldots, v_n \}$ be the set of member blocks of V. Define the directed graph $\hat{G}_V = (\hat{V}, \hat{E}_V)$ where $\hat{E}_V = \{ (v_i, v_j) \mid$ there are outer-variables x_i of v_i, x_j of v_j such that $x_i <_0 x_j \}$. Call $\hat{G}_V$ the *super-graph* associated with V. When $\hat{G}_V$ is a dag, call it a *super-dag*. •

The SLI systems we are interested in in this section are those for which every block has an acyclic super-graph. For such an SLI system, the associated super-graph of every block V at any level, $\hat{G}_V = (\hat{V}, \hat{E}_V)$, is a dag, and therefore $\hat{E}_V$ defines a partial order $<_V$ on $\hat{V}$. Namely, we put $v_i <_V v_j$ if and only if $(v_i, v_j) \in \hat{E}$.

Let V, W be two blocks at the same level. If there are $x \in V$, $x' \in W$, such that $x <_0 x'$, then we say V is a *preceding* block of W. We say a block is *computed* if all the variables in it are computed.

For a block V at the bottom level, if all the preceding blocks of V have been computed, then all outer-variables x external to V such that $x <_0 x'$ for some $x' \in V$ have been computed. Therefore, V can be computed next. We see that this holds inductively for blocks at all levels. Based on this idea, the following algorithm will determine the values for all the variables of a block V by proceeding from one member block of V to another in an order that is consistent with the partial order $<_V$, assuming that the preceding blocks of block V are all computed.

Algorithm (HIERARCHICAL LOCAL PERT)
Input: a set of simple linear inequalities on the variables of a block V, $\hat{G}_V = (\hat{V}, \hat{E}_V)$ the associated super-dag, a set $Pre(V)$ containing inequalities of the forms $x \geq x_0$ where x is an outer-variable of V and x_0 is a constant, and similarly a set $Pre(v_i)$ containing inequalities $x \geq x_0$ where x is an outer-variable of v_i, for each $v_i \in \hat{V}$.
Output: $\{ t(x) \mid x \in V \}$ (**comment**: $t(x)$ is the value of the variable x in V)
begin
for $x \in V$, $t(x) := 0$; $p(x) := 0$; (**comment** initialization)
If V is a bottom level block **then**
begin
$p(x) := \max \{ a \mid x \geq a$ is in $Pre(V) \}$, for every outer-variable x of V;
SIMPLE PERT on V;
end
If V is not a bottom level block **then**
begin
Select the member blocks of V in an order that is consistent with the partial order $<_V$ on $\hat{V}$, **for** each selected block v_i in $\hat{V}$ **do**
  begin
  $p(x) := \max \{ a \mid x \geq a$ is in $Pre(v_i) \}$, for every outer-variable x of v_i;
  HIERARCHICAL LOCAL PERT on v_i
  for v_j such that $(v_i, v_j) \in \hat{E}$ **do**
    $Pre(v_j) := Pre(v_j) \cup \{ x_1 \geq t(x) + d \mid x + d \leq x_1$ is a constraint with $x \in v_i$, $x_1 \in v_j \}$.
  end
end
end

To find an optimal solution to an acyclic SLI system, we first construct the super-dags $\hat{G}_V$ for all the blocks. This takes linear time and space and p page-

faults where p is the total number of pages. We then apply Hierarchical Local Pert on the outer-most block V with $Pre(V) = \{ x \geq 0 \mid x$ is an outer-variable in $V \}$. It is easy to see that every page is fetched to the main memory once. Therefore, we have the following:

Proposition 4.1 For a hierarchical SLJ system with acyclic super-graphs for all the blocks at all levels, an optimal solution can be found in linear time and space, causing the number of page-faults $2p$ where p is the number of pages.•

4.2 Compaction for General Hierarchical SLJ and SLJE Systems

In this section, we discuss a hierarchical algorithm for finding an optimal solution to a general hierarchical SLJE system. This algorithm also applies to SLJ systems since SLJ systems are also SLJE systems.

Definition Denote by the relation $R_1(x_1,x_2)$ (or $x_1R_1x_2$) the condition where there is a constraint $x_1+d_{12} \leq x_2$, and by $R_2(x_1,x_2)$ (or $x_1R_2x_2$) the condition where there is a constraint $x_1 = x_2$. We say $x \rightarrow x'$ if there are $x_1, x_2, \ldots, x_k$, $a_0, a_1, \ldots, a_k$ with $a_i \in \{1, 2\}$, such that $R_{a_0}(x,x_1)$, $R_{a_1}(x_1,x_2), \ldots, R_{a_k}(x_k,x')$, and there is exactly one i such that $a_i = 1$.

The relation $\rightarrow$ just defined can actually be viewed as the extension of the relation $\rightarrow$ defined for paged dags in Section 3. The relation $\rightarrow$ on the variables has the following interpretation. The value of a variable x cannot be computed until all the variables x' such that $x \rightarrow^* x'$ are computed. It is clear that $\rightarrow$ must be acyclic, otherwise the SLJE system is unsolvable. (Recall that we assume $d_{ij} > 0$ for inequality constraint $x_i + d_{ij} \leq x_j$.)

Suppose V is a block of an SLJE system. Let $Z(V)$ be the set of the outer-variables of the member blocks of V. We now define the relation $<_o$ on the outer-variables in $Z(V)$.

Definition Let V be a block of an SLJE system, and let x, x' be two outer-variables in $Z(V)$. We say $x <_o x'$ (in V) if and only if $x \rightarrow x'$ or there exist variables $x_1, \ldots, x_m$ in V such that $x \rightarrow x_1$, $x_1 \rightarrow x_2, \ldots, x_m \rightarrow x'$ and none of x_i, $i = 1, \ldots, m$, is an outer-variable in $Z(V)$.•

Note that the definitions of $\rightarrow$ and $<_o$ when restricted to the SLJ systems are consistent with the ones defined in section 3. For an SLJE system, if we disregard the equality constraints, we get an SLJ system which we call the SLJ subsystem of the original SLJE system. When we restrict $\rightarrow$ to the SLJ subsystem, we call it $\rightarrow$ *with respect to the SLJ subsystem*, and likewise for $<_o$.

From the definition of $<_o$, we see that $<_o$ must be acyclic. For every block V, we can construct an outer-dag on the outer-variables $Z(V)$ with respect to $<_o$ as before. Indeed for SLJ systems, the method that worked for paged dags in Section 3 can be extended in a simple way to work for hierarchical SLJ systems. However, for SLJE systems, some complications arise due to the presence of equality constraints. For instance in Example 3 of Section 3, if we add the following constraints: $x_{10} = x_{20}$, $x_{12} = x_{22}$, and $x_{14} = x_{24}$, then $x_{2C} <_o x_{12}$, and the outer-dag H becomes as shown in Figure 4.1. We now cannot apply procedure *DELETE* to determine a legal page sequence. For example, although x_{10} is exposed in H, we cannot compute it immediately, for otherwise, since $x_{10} = x_{20}$, we would have been able to compute x_{2C} without looking at block V_2. The existence of an equality $x = x'$ implies that when one of the two variables is computed, so is the other. To take this fact into account, we need to consider the equivalence classes on the outer-variables formed with respect to the equalities.

Definition We partition the set of variables in an SLJE system into equivalence classes with respect to R_2, the relation defined by equalities. We write $[x]$ for the equivalence class $[x]$ is in, and call $[x]$ an *equality class*.•

Clearly, $[x_1] = [x_2]$ if and only if $x_1R_2^*x_2$

Definition We say a block V *touches* $[x]$ if $[x] \cap Z(V) \neq \phi$. In this case, we let $[x]_V$ be $[x] \cap Z(V)$, and call $[x]_V$ the *restriction* of $[x]$ to V. When a block V is specified, we sometimes write $[x]$ or $[x]$ in V for $[x]_V$.•

We note that if for every variable x_j in an equality class $[x]$, the values of all variables x_i such that $x_i \rightarrow^* x_j$ are computed, then the value $t[x]$ of all the variables in $[x]$ can be computed as follows:

(1) $p(x_j) \leftarrow \max \{ t(x_i)+d_{ij} \mid x_i+d_{ij} \leq x_j$ is a constraint $\}$.

(2) $t[x] \leftarrow \max \{ p(x_j) \mid x_j \in [x] \}$.

Now we extend the definition of $<_o$ to the equality classes.

Definition Let $[x_1]$, $[x_2]$ be two equality classes that touch a block V.

(1)If V is a bottom-level block, then write $[x_1] <_o [x_2]$ (in V) if and only if there are $x \in [x_1]$, $x' \in [x_2]$ such that $x <_o x'$ in V.

(2)If V is not a bottom-level block, then write $[x_1] <_o [x_2]$ (in V) if and only if there are $x \in [x_1]$, $x' \in [x_2]$ such that either $R_1(x,x')$ or $[x_1]_u <_o [x_2]_u$ where u is a member block of V that touches both $[x_1]$ and $[x_2]$.•

Lemma 4.1 Let V be a block in a hierarchical SLIE system. Then

(1) the relation $<_e$ defines a partial order on the equality classes that touch V.

(2) if x, x' are two outer-variables in V, then $x <_e {}^* x'$ implies $[x] <_e {}^* [x']$

Proof

Without loss of generality, we assume that there is at least one outer-variable in each block so that every block touches at least one equality class.

Part (1) is clear from the definition. Part (2) we prove by induction on the level of V in the hierarchy.

For V at the bottom level, the assertion follows immediately from the definition. Suppose V is not at the bottom level. If x, x' are in the same member block u of V, then by induction, $x <_e {}^* x' \Rightarrow [x]_u <_e {}^* [x']_u$, and therefore $[x] <_e {}^* [x']$.

If x, x' are in different member blocks, say $x \in v_0$, $x' \in v_m$, putting $x = x_{(1)}$, $x' = x_{m2}$, then $x <_e {}^* x'$ implies that there are member blocks of V, $v_1, \ldots, v_{m-1}$ and outer-variables $x_{i1}, x_{i2} \in v_i$, $i = 0, \ldots, m$ such that $x_{i1} <_e {}^* x_{i2}$, $x_{i2} R_1 x_{(i+1)1}$, for $i = 0, \ldots, m-1$. Now by induction $x_{i1} <_e {}^* x_{i2} \Rightarrow [x_{i1}]_{v_i} <_e {}^* [x_{i2}]_{v_i}$, $i = 0, \ldots, m \Rightarrow [x_{i1}] <_e {}^* [x_{i2}]$. Also $x_{i2} R_1 x_{(i+1)1} \Rightarrow [x_{i2}] <_e [x_{(i+1)1}]$, $i = 0, \ldots, m-1$. Therefore, $[x] <_e {}^* [x']$. •

By Lemma 4.1, for every block, we can form a dag on the equality classes with respect to $<_e$. From Lemma 4.1, we also see that for a block V and an equality class $[x]$ that touches V, if all equality classes $[x']$ such that $[x] <_e [x']$ have been computed, then the number $\max \{ t(x_1) + d_1 \mid x_1 + d_1 \le x, x_1 \in V \}$ can be computed. When V is the highest-level block, then this value is exactly the value for all the variables in $[x]$. Therefore, we can proceed from equality class to equality class in an order that is consistent with the partial order $<_e$.

As in Section 3, we need not construct the dag that is defined with respect to the partial order $<_e$. As a preparation stage, we need only compute the equality classes. This can be done by applying breadth-first-search hierarchically, treating an equality $x = x'$ as an edge (x, x'). We also compute $in[x]$ which is the sum of $in(x)$ for $x \in [x]$, where $in(x)$ denotes the number of x' so that there is a constraint $x' + d \le x$. All these computations can be done by examining each bottom-level block once. To simplify the presentation, we also assume that within each bottom-level block, there are only inequality constraints binding variables in that block. This can be achieved by forming equivalence classes with respect to equality constraints within each bottom-level block, choosing one representative for each class, and substituting all the variables in a class by the representing one.

The main procedure *LOCAL EQ-PERT* is described below. It takes two arguments V and $Y(V)$, where V is a block, $Y(V)$ is a set of newly computed variables if V is at the bottom level, and a set of newly computed equality classes otherwise. It will compute as many variables and equality classes in V as possible, starting from the given $Y(V)$. In the procedure as well as in the algorithm, $t(x)$ will denote the current lower bound on a variable x, $t[x]$ will denote the current lower bound on an equality class $[x]$. The final values of $t(x)$ and $t[x]$ will be the values of x and $[x]$ respectively. Also, V_0 will denote the highest-level block.

```
Procedure LOCAL EQ-PERT ( Y(V), V )
Begin
If V is at bottom level then while Y(V) is not empty do
begin
    for x ∈ Y(V) do begin
        for a constraint x + d ≤ x1 do begin
            t(x1) := max( t(x1), t(x) + d );
            in(x1) := in(x1) − 1;
            if in(x1) = 0 then if x1 is not an outer-
            variable then put x1 in Y(V) else do begin
                t[x1] := max( t[x1]; t(x1) );
                in[x1] := in[x1] − 1;
                if in[x1] = 0 then put [x1] in Y(V0);
                end.
            end.
        end.
    end.
if V is not at bottom level then for each member block
u that touches at least one equality class in Y(V) do
begin
    Y(u) := { [x]u  [x] ∈ Y(V) },
    LOCAL EQ-PERT ( Y(u), u ).
    end.
end.
```

Now we describe the algorithm.

```
Algorithm COMPUTE SLIE
begin
t(x) := 0 for all variables x;
t[x] := 0 for all equality class [x];
for each bottom-level block V do begin
    for variable x in V such that in(x) = 0 do begin
        if x is not an outer-variable then put x in
        Y(V);
        if x is an outer-variable and in[x] = 0 then
        put [x] in Y(V0);
        end.
    LOCAL EQ-PERT ( Y(V), V );
    end.
do until Y(V0) is empty begin
LOCAL EQ-PERT ( Y(V0), V0 ).
end.
end.
```

Let p be the number of pages in a hierarchical SLIE system. The preparation stage for computing equality

classes takes p page-faults. Other than that, algorithm *COMPUTE SLIE* examines each bottom-level block no more than m times, where m is a bound on the number of outer-variables in a bottom-level block. We therefore have finally,

Theorem 4.1 Algorithm *COMPUTE SLIE* runs in time $O(N)$ where N is the total number of constraints, causes a number of page faults no greater than $(m+1)p$ where p is the number of pages, and m is the maximum number of outer-variables a page (that is, a bottom level block) can have. •

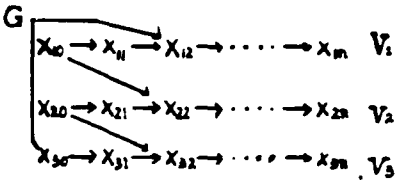


Fig 2.1

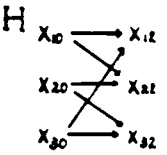


Fig 2.2

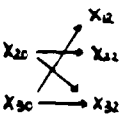


Fig 2.3

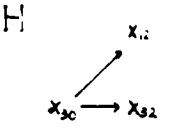


Fig 2.4

Fig 2.5

Reference

[AHU]A.V. Aho, J.E. Hopcroft, J.D. Ullman, *The Design and Analysis of of Computer Algorithms*, Addison-Wesley, Reading, Mass, (1974), 378-387.

[D]A.E.Dunlop, "SLIM - The Translation of Symbolic Layouts into Mask Data," *Proc 17th IEEE Design Automation Conference* (1980), 595-602.

[GJ]M.R. Garey, D.S. Johnson, *Computers and Intractability A Guide to The Theory of NP-Completeness*, Freeman (1979).

[H]M. A. Huang, thesis in preparation

[K]D.E. Knuth, *The Art of Computer Programming*, Vol. 1, Addison-Wesley, Reading, Mass (1968).

[LSV]R.J. Lipton, R. Sedgewick, J. Valdes, "Programming Aspects of VLSI", *Proc 8th ACM-POPL Conference* (1982), 57-65

[LM]T. Lengauer, K. Mehlhorn, "HILL - Hierarchical Layout Language, A CAD System for VLSI Design," TR A82/10, FB 10, Universität des Saarbrücken, West-Germany (1982)

[L]T. Lengauer, "The Complexity of Compacting Hierarchically Specified Layouts of Integrated Circuits", *Proc IEEE Conference on Foundations of Computer Science* (1982), 358-365

[N]S.C. North, "Molding Clay: A Manual for the CLAY Layout Language", *VLSI memo#3*, Princeton University.

[MC]C. Mead, L. Conway, *Introduction to VLSI Systems*, Addison-Wesley, Reading, Mass. (1980)

[V]G. Vijayan, *Design Implementation and Theory of a VLSI Layout Language*, Ph.D Thesis, Princeton University, (1983)

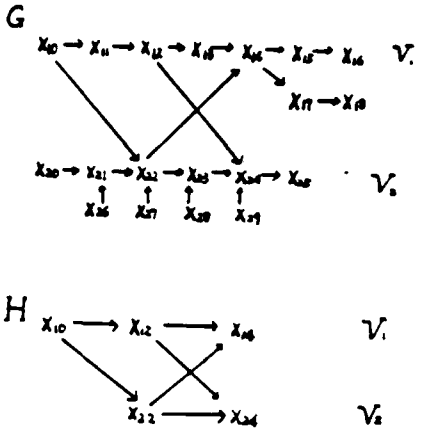


Fig 3.1

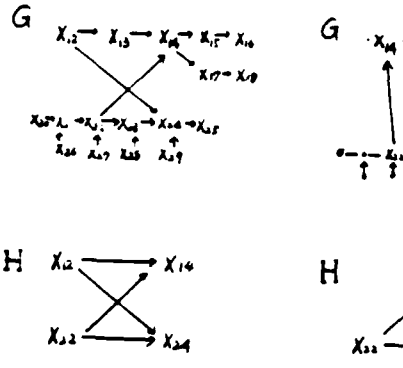


Fig. 3.2

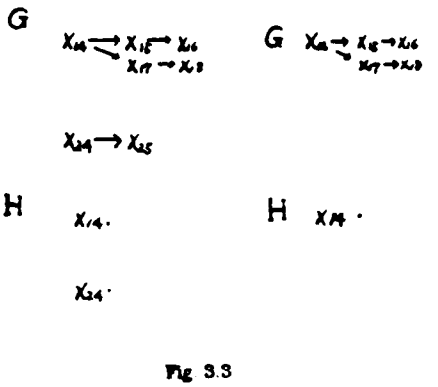


Fig 3.3

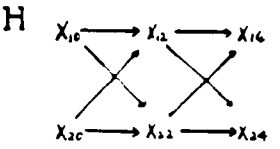


Fig 4.1

Testability Conditions for Bilateral Arrays of Combinational Cells

Anastasios Vergis and Kenneth Steiglitz

Department of Electrical Engineering and Computer Science
Princeton University, Princeton, New Jersey 08544

ABSTRACT

Two sets of conditions are derived that make one-dimensional bilateral arrays of combinational cells testable for single faulty cells. The test sequences are preset and, in the worst case, grow quadratically with the size of the array. Conditions for testability in linear time are also derived. The basic cell can operate at the bit or at the word level. An implementation of FIR filters using (systolic) one-dimensional bilateral arrays of cells, which can be considered combinational at the word level, is presented as an example.

1. Introduction

The use of iterative arrays of identical cells in current VLSI technology is becoming more frequent due to their many advantages, like ease of design, fabrication and testing. Moreover, many problems are efficiently solved with the use of "systolic arrays", which are highly iterative structures operating synchronously. An important problem associated with these structures is fault detection, that is, derivation of test input sequences to the array, such that the output sequences of the normal and any faulty array (under some fault assumptions) are different. In this paper we give some testability conditions for a special class of arrays (defined below) that improve upon the condition reported in [3]. Derivation of the test input sequences is also described.

2. Assumptions, Definitions and Notation

Figure 1a shows a bilateral array of combinational cells. The basic cell is shown in Figure 1b. At each time unit it produces left and right outputs, depending on its left, right and vertical inputs.

Let R be the set of right-moving signals, L the set of left-moving signals and Z the set of vertical cell inputs. Let $g_R: R \times Z \times L \to R$ be the right-moving signal mapping, and $g_L: R \times Z \times L \to L$ be the left-moving signal mapping.

A *fault* in a particular cell alters g_R, g_L, or both for one or more arguments (r, s, l). However, we assume that the cell remains combinational.

We assume initially that to test a cell completely, we must apply all input combinations $R \times Z \times L$ to that cell. This assumption makes testing of the cells independent of how they are realized and independent of the fault model for permanent faults [1]. We shall examine later the case when only a subset of $R \times Z \times L$ suffices to test the basic cell. We further assume that to test the array completely (for single faulty cells), we must test completely every cell in the array.

The left, vertical and right inputs of cell j at time t are denoted as $r^j(t)$, $s^j(t)$, $l^j(t)$ respectively. Hence, cell j at time $t+1$ will produce left and right outputs $l^{j-1}(t+1)$, $r^{j+1}(t+1)$.

Figure 2 shows the *space-time transformation* [2] of the array in Fig. 1a. Each row represents the array at each time unit. This makes the operation of the array easier to visualize. Note that this transformation maps the one-dimensional synchronous bilateral array into a two-dimensional asynchronous unilateral array.

If L_0 is a subset of L, define $g_R(r, s, L_0) = R_0$, where R_0 is the set $\{ g_R(r, s, l) \mid l \in L_0 \}$. If R_0 contains just one element r', we write $g_R(r, s, L_0) = r'$ (instead of $\{r'\}$). Similarly for $g_R(R_0, s, l)$, $g_R(r, Z_0, l)$, and similarly for g_L.

If $r \in R$, define $R_r = R - \{r\}$. Similarly $L_l = L - \{l\}$. An input or output labeled r/R_0, where $R_0 \subset R$, and r does *not* belong to R_0, means normal input or output r, and faulty input or output some member of R_0. If R_0 contains just one element r', we write r/r'. Sometimes, when no confusion arises, r/R_0 will also be called a *fault*. For simplicity r/R_r will sometimes be written as $r/\bullet$.

We also define $g_R(r_1/r_2, s, l) = r'_1/r'_2$ if $g_R(r_1, s, l) = r'_1$ and $g_R(r_2, s, l) = r'_2$ and $r_1 \neq r_2$, $r'_1 \neq r'_2$. We define similarly $g_R(r, s, l_1/l_2) = l'_1/l'_2$, and similarly for g_L. Another definition is: $g_R(r/R_0, s, l) = r'/R'_0$ if $g_R(r, s, l) = r'$, and $g_R(R_0, s, l) \subset R'_0$. Similarly for $g_R(r, s, l/L_0)$ and for g_L. (Note that $g_R(r/R_0, s, l)$ is not uniquely defined, since R'_0 may be any superset of $g_R(R_0, s, l)$, not containing r'.) If, according to the above, the input of a cell is a fault r/R_0 and its output is r'/R'_0 we say that this cell *propagates* the fault r/R_0. Notice that according to our convention r' does not belong to R'_0, so we can distinguish between the normal and the faulty output.

In the sequel, p denotes the total number of cells in the array.

3. Some General Testability Conditions

Let V be the following set of conditions:

- C1: for every $r \in R$ there exist $r' \in R$, $s \in Z$ such that $g_R(r', s, L) = r$.
- C2: for every $l \in L$ there exist $l' \in L$, $s \in Z$ such that $g_L(R, s, l') = l$.
- O1: for every $r_1, r_2 \in R$ with $r_1 \neq r_2$, there exist $l \in L$, $s \in Z$ such that $g_R(r_1, s, l) \neq g_R(r_2, s, l)$.
- O2: for every $l_1, l_2 \in L$ with $l_1 \neq l_2$, there exist $r \in R$, $s \in Z$ such that $g_L(r, s, l_1) \neq g_L(r, s, l_2)$.

Theorem 1: Any bilateral array of combinational cells that satisfies conditions V is testable for single faulty cells.

1983 IEEE International Conference on Computer Design/VLSI in Computers, Rye, New York, October 31-Nov. 3, 1983.

The proof is similar in spirit to the proof of the next theorem, and it is actually simpler. So it is omited. We only mention that, under conditions V, to test the entire array completely we need $p\cdot(p+1)\cdot max(|R|,|L|)\cdot|R|\cdot|Z|\cdot|L|$ tests.

Let W be the following set of conditions (see Figs. 3 and 4):

C1: for every $r\in R$ there exist $r'\in R$, $s\in Z$ such that $g_R(r'/\bullet, s, L) = r/\bullet$.

C2: for every $l\in L$ there exist $l'\in L$, $s\in Z$, $r'\in R$ such that $g_L(r', s, l') = l$ and $g_R(r'/\bullet, s, l') = r/\bullet$.

O1: for every $r\in R$ there exist $s\in Z$ such that $g_R(r/\bullet, s, L) = r'/\bullet$.

O2: for every $l_1, l_2\in L$ with $l_1\neq l_2$, there exist $r\in R$, $s\in Z$ such that $g_L(r, s, l_1)\neq g_L(r, s, l_2)$.

Conditions W hold if 1) the basic cell just transmits unaltered the right-moving signal, 2) $g_R(R,Z,L) = L$, 3) for any two different right inputs there exist r, s that produce two different left outputs. This is a quite reasonable set of assumptions.

Theorem 2: Any bilateral array of combinational cells that satisfies conditions W is testable for single faulty cells.

Proof: Assume we want to test cell j for inputs (r_0, s_0, l_0). These test inputs will be applied at time $t=2p-j$ if the test begins at time $t=1$. Hence $r_0 = r^j(2p-j)$, $l_0 = l^j(2p-j)$, $s_0 = s^j(2p-j)$ (see Fig. 5, shaded cell).

First we must make the left input of cell j at time $2p-j$ be $r_0 = r^j(2p-j)$. Condition C1 guarantees the existence of $r^{j-1}(2p-j-1)$, $s^{j-1}(2p-j-1)$ such that $r^j(2p-j) = g_R(r^{j-1}(2p-j-1), s^{j-1}(2p-j-1), L)$; hence it suffices to apply $r^{j-1}(2p-j-1)$, $s^{j-1}(2p-j-1)$ as left and vertical inputs to cell $j-1$ (at time $2p-j-1$). Inductively, apply $r^{j-2}(2p-j-2)$, $s^{j-2}(2p-j-2)$ to cell $j-2$, etc., until we reach the leftmost cell. The tricky part is to apply right input $l_0 = l^j(2p-j)$ to cell j. Condition C2 guarantees the existence of $r^{j+1}(2p-j-1)$, $l^{j+1}(2p-j-1)$, $s^{j+1}(2p-j-1)$ such that $l^j(2p-j) = g_L(r^{j+1}(2p-j-1), s^{j+1}(2p-j-1), l^{j+1}(2p-j-1))$, and $g_R(r^{j+1}(2p-j-1)/\bullet, s^{j+1}(2p-j-1), l^{j+1}(2p-j-1)) = r^{j+2}(2p-j)/\bullet$. Hence it suffices to apply input $(r^{j+1}(2p-j-1), l^{j+1}(2p-j-1), s^{j+1}(2p-j-1))$ to cell j+1 (at time $2p-j-1$). Left input $r^{j+1}(2p-j-1)$ and right input $l^{j+1}(2p-j-1)$ can be applied (recursively) in the same way we applied inputs r_0 and l_0 to cell j (using C1). This solves the controllability problem.

We have not yet used the strong part of condition C2, namely the fact that $g_R(r'/\bullet, s, l') = r/\bullet$. The usefulness of this will become apparent in the sequel.

Assume that the normal right and left outputs of cell j (on input (r_0, s_0, l_0)) are r and l respectively; assume that we test for the error $l/\hat{l}$ in the left output. We can simultaneously test for all errors $r/\bullet$ in the right output. Propagation of the error $l/\hat{l}$ to the leftmost output is done using O2 and C1.

We have not yet discussed the "southeast" portion of Fig. 5, that is the portion below the right-to-left diagonal that passes through the shaded cell. First we have to propagate the fault $r/\bullet$ $(= r^{j+1}(2p-j+1)/\bullet)$ to the rightmost output. But cell j may fail to function correctly at any previous time, so for instance (see Fig. 5) cell j on input $r^j(2p-2m+j)$ (for some m in $\{j+1, j+2, \cdots, p\}$), may not output $r^{j+1}(2p-2m+j+1)$, so cell m may not output $l^{m-1}(2p-m+1)$, hence cell j may not receive l_0 as right input, and due to a fault, it may output the expected outputs l, r. Under this worst case scenario the two faults will be masked and we will get the expected observable outputs $l^0(2p)$ and $r^{p+1}(3p-2j+1)$ (assuming $j\geq(p+1)/2$). This is avoided as follows:

First, to propagate the fault $r/\bullet$ to the rightmost output, using condition O1 we find $s^{j+1}(2p-j+1)$ such that cell $j+1$ on input $r^{j+1}(2p-j+1)/\bullet$ outputs $r^{j+2}(2p-j+2)/\bullet$; inductively we propagate this fault to the rightmost output $(r^p(3p-2j-1)/\bullet)$. Similarly, we propagate the fault $r^m(2p-m)/\bullet$ for $m=j+1, j+2, \cdots, p$. Notice that the potential previous fault $r^{j+1}(2p-2m+j+1)/\bullet$ of cell j has been "automatically" propagated to cell m as $r^m(2p-m)/\bullet$ by the strong part of condition C2 "when" we were solving the controllability problem. So, if cell j outputs something different from $r^{j+1}(2p-2m+j+1)$, we shall detect it at the rightmost output by getting something different from the expected $r^{p+1}(3p-2m+1)$. This solves the observability problem.

The above procedure is repeated for every $\hat{l}$ in L_l; then we have tested cell j for input (r_0, s_0, l_0). This is repeated for every (r, s, l) in $R\times Z\times L$; then we have tested cell j completely. ∎

The testing time shown in Fig. 5 is $2p$. Hence to test cell j for input (r_0, s_0, l_0) we need $2p\ |L_{l_0}|$ tests, hence to test completely cell j we need $2p\cdot|L|^2\cdot|R|\cdot|Z|$ tests, and to test completely the array we need $2p^2\cdot|L|^2\cdot|R|\cdot|Z|$ tests. Note that if some cell in the space-time transformation is "used" at time t, it is never used at time $t+1$, hence the obvious pipelining reduces the testing time to one-half of the above number of tests.

4. One-step Testability

Parthasarathy and Reddy [4] introduced the notion of one-step testability for unilateral arrays. We extend this notion to bilateral arrays as follows.

Definition: A cell in a bilateral array of combinational cells is one-step testable for input (r, s, l) if the number of time units needed to test this cell for input (r, s, l) is independent of $|R|$, $|L|$.

Definition: A cell in a bilateral array of combinational cells is one-step testable if it is one-step testable for all inputs (r, s, l) in $R\times Z\times L$.

Definition: A bilateral array of combinational cells is one-step testable if all its cells are one-step testable.

The notion of one-step testability is important for the following reason: if an array is one-step testable, the time needed to test it is greatly reduced since, if the expected output of a cell under test is, say, l, it is not necessary to apply different test inputs for each fault $l/\hat{l}, \hat{l}\in L-\{l\}$.

The following conditions are useful for one-step testability (see Fig. 6):

OST1: for every $r\in R$ there exist $l\in L$, $s\in Z$ such that $g_R(r/R_r, s, l) = r'/R_{r'}$.

OST2: for every $l\in L$ there exist $r\in R$, $s\in Z$ such that $g_L(r, s, l/L_l) = l'/L_{l'}$.

Let OST be conditions OST1 and OST2 together. Let V-OST (W-OST) be conditions V (W) and OST together.

If W-OST hold, instead of testing for the fault $l/\hat{l}$ for each $\hat{l}$ in L_l as in the proof of theorem 2, we can test for the fault l/L_l. Thus, to test cell j for input (r_0, s_0, l_0), we only need $2p$ time units. Therefore, if we want to test all cells for inputs a subset I of $R\times Z\times L$, we need

$\Sigma p^2 \cdot |I|$ time units. If V-OST hold, similarly as above, by testing for all output faults simultaneously (on a given input), we need $p\,(p+1)\,|I|$ tests if we want to test all cells for a subset I of $R \times Z \times L$.

5. Conditions for Testability in Linear Time

Let L be the following set of conditions (see Fig. 7):

OC1: For every $r_1, r_2 \in R$ with $r_1 \neq r_2$ and for every $l \in L$ there exists $l' \in L$ and $z \in Z$ such that $g_R(r_1, z, l') \neq g_R(r_2, z, l')$ and $g_L(r_1, z, l') = l$.

OC2: For every $l_1, l_2 \in L$ with $l_1 \neq l_2$ and for every $r \in R$ there exists $r' \in R$ and $z \in Z$ such that $g_L(r', z, l_1) \neq g_L(r', z, l_2)$ and $g_R(r', z, l_1) = r$.

Theorem 3: Condition C1 of V and L is a sufficient set of conditions for testability of the entire array in a number of time units proportional to the length of the array.

Proof: We shall give a constructive proof. To keep things simple, we shall work on an example; the generalization is straightforward. Consider the space-time transformation of an array of 6 cells as in Fig. 8. Assume that each cell must undergo 2 tests, that is it must be tested for inputs (r_i, l_i) $(i=1,2)$ for faults $r'_i/\hat{r}_i$, $l'_i/\hat{l}_i$ (i.e. r'_i, l'_i are the normal outputs and $\hat{r}_i, \hat{l}_i$ are the faulty outputs). In Fig. 8 cells under test are depicted as full squares; so the inputs and outputs of these cells are "given".

Let's group the cells of the space-time transformation diagonally, forming different groups on the left and right of the cells under test, as in Fig. 8. Call these groups left and right diagonal groups. Note that each left diagonal group of c cells has one left input, one right output, c right inputs and c left outputs. Its right inputs will typically be faults that have to be propagated to its left outputs. Fig. 9 shows a left diagonal group containing cells k through m, at time units t through $t+m-k$. Assume that the faults $l^{k+1}(t+i)/\hat{l}^{k+1}(t+i)$ for $i=0,1,\cdots,m-k$ are given and have to be propagated to its left outputs; also output $r^{m+1}(t+m-k+1)$ must be generated. We shall show how to find its left input $r^k(t)$ that does that. Using condition OC1 we can find input $r^m(t+m-k)$ that propagates the fault $l^m(t+m-k)/\hat{l}^m(t+m-k)$ and generates the output $r^{m+1}(t+m-k+1)$. Now $r^m(t+k-m)$ has to be generated, so inductively we can find $r^k(t)$. Hence each left diagonal group can propagate to its left outputs any faults on its right inputs and, at the same time, it can generate any right output. Entirely symmetrical things hold for each right diagonal group.

We apply the construction stated above, starting from the left diagonal group at the top of Fig. 8 (which contains just one cell); that is, we find its left input so that the fault $r'_1/\hat{r}_1$ is propagated, and output r_2 is generated. Then, for the next left diagonal group (second from the top), we know the faults that it has to propagate. If we were testing the leftmost cell for one more input, we would also know the output that has to be generated by this diagonal group; since this is not the case here, we can choose it arbitrarily. Therefore, we can apply again that construction to find its left input that does this. We proceed this way, from top to bottom. (Inductively, each diagonal group will have to propagate some faults and generate an output; as we proved above, there exists an input that does this.) We repeat the same procedure for the right diagonal groups, proceeding again from top to bottom. Obviously it does not hurt if some of the left inputs of a right diagonal group are not faults that have to be propagated, as for the first, third etc. groups in Fig. 8.

For the first $p-2$ left diagonal groups their left inputs are "inside" the array, so we have the additional problem of how to generate them. Condition C1 of V takes care of this, (as in theorem 2).

If each cell requires T tests, $(T+1)\cdot p$ is the number of inputs that have to be applied to the left boundary of the array (see Fig. 8); additionally, $p-1$ time units are required for the propagation to the right boundary of the faults in the leftmost cell. Hence the entire array can be tested in in $(T+1)\cdot p+p-1$ time units. Again, the obvious pipelining saves about half of these time units. ∎

6. Conditions for One-step Testability in Linear Time

Let us consider now one-step testable arrays. It turns out that it suffices to replace the fault r_1/r_2 (resp. l_1/l_2) in conditions L by the fault $r/*$ (resp. $l/*$). This way we obtain the following set of conditions L-OST.

OC1: For every $r \in R$ and for every $l \in L$ there exists $l' \in L$ and $z \in Z$ such that $g_R(r/*, z, l') = g_R(r, z, l')/*$ and $g_L(r, z, l') = l$.

OC2: For every $l \in L$ and for every $r \in R$ there exists $r' \in R$ and $z \in Z$ such that $g_L(r', z, l/*) = g_L(r', z, l)/*$ and $g_R(r', z, l) = r$.

Theorem 4: Condition C1 of V and L-OST is a sufficient set of conditions for one-step testability of the entire array in a number of time units proportional to the length of the array.

Proof: Analogous to the proof of theorem 3. If each cell is tested for a subset I of its inputs, the number of time units required is $(|I|+1)\cdot p+p-1$. ∎

Remark: All the above results are easily generalized for the case when g_R, g_L are not identical for every cell, that is we have g^i_R, g^i_L for the i-th cell; it suffices to replace the conditions for g_R, g_L by conditions for g^i_R, g^i_L for every i.

7. Application

Figure 10 shows the basic cell of a two-way pipeline systolic array used for FIR filtering [5], [6]. For this cell we have $|Z| = 1$ (no z-inputs), $g_R(r, l) = r$, $g_L(r, l) = l + a \cdot r$. This array can be considered as a bilateral array of combinational cells at the word level (the basic time unit is the time required to produce the outputs). It easy to see that conditions W-OST are satisfied. (Here we have the case when g_R, g_L depend on the cell.) Therefore, if a subset I of $R \times L$ suffices to test the basic cell, $\Sigma p^2 \cdot |I|$ tests suffice to test the array. Also, if we ignore overflow problems assuming for all r, $l_1 + a \cdot r \neq l_2 + a \cdot r$ for $l_1 \neq l_2$, conditions L-OST hold; hence for this case $O(p\,|I|)$ tests suffice to test the array. Note that the inequality above does not have to hold for all r, l_1, l_2; obviously, it suffices to hold for the signals that appear as inputs or outputs in the test described in the proof of theorem 3.

ACKNOWLEDGEMENT

This work was supported in part by NSF Grant ECS-8120037, U. S. Army Research-Durham Grant DAAG29-82-K-0095, and DARPA Contract N00014-82-K-0549.

REFERENCES

[1] M. A. Breuer and A. P. Friedman, *Diagnosis and Reliable Design of Digital Systems*, Computer Science Press, 1976.

[2] F. C. Hennie, *Iterative Arrays of Logical Circuits*, Cambridge, MA: MIT Press, 1961.

[3] F. G. Gray, and R. A. Thomson, "Fault Detection in Bilateral Arrays of Combinational Cells," *IEEE Trans Comput*, vol. C-27, pp. 1206-1213, 1978.

[4] R. Parthasarathy and S. M. Reddy, "A Testable Design of Iterative Logic Arrays," *IEEE Trans Comput*., vol. C-30, pp. 833-841, 1981.

[5] H. T. Kung, "Let's Design Algorithms for VLSI Systems," *Proc. Conf. on Very Large Scale Integration Architecture, Design, Fabrication*, California Institute of Technology, Jan. 1979, pp 65-90.

[6] H. T. Kung and C. E. Leiserson, "Algorithms for VLSI Processor Arrays," in *Introduction to VLSI Systems*, C. Mead and L. Conway, Addison-Wesley, 1980.

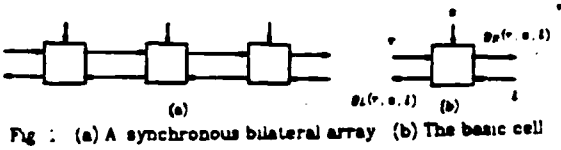


Fig : (a) A synchronous bilateral array (b) The basic cell

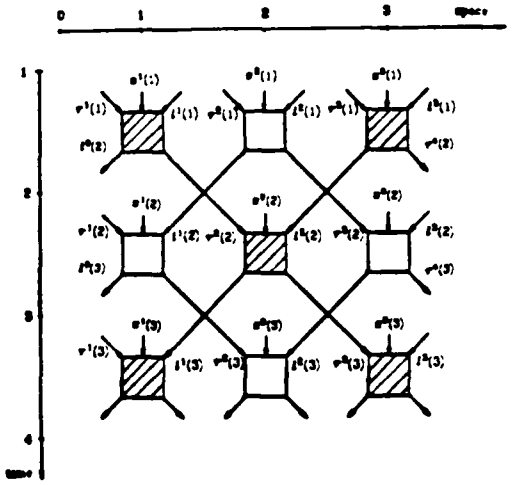


Fig 2 Space-time transformation into asynchronous unilateral array

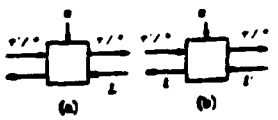


Fig 3 (a) Condition C1 of W
(b) Condition C2 of W

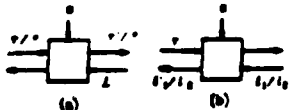


Fig 4 (a) Condition O1 of W
(b) Condition O2 of W

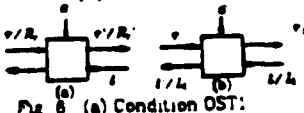


Fig 6 (a) Condition OST1
(b) Condition OST2

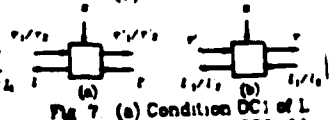


Fig 7 (a) Condition OC1 of L
(b) Condition OC2 of L

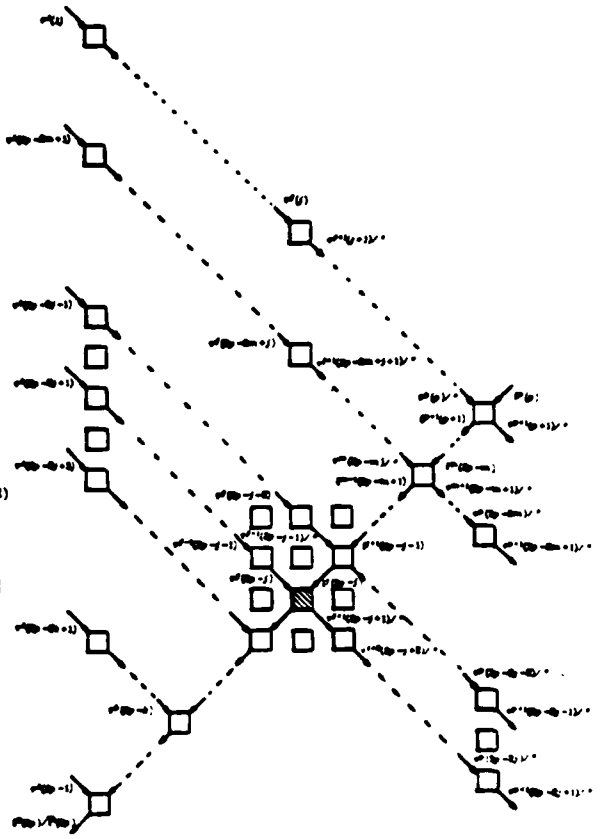


Fig 5 The test described in thm 2 (vertical inputs are not shown for simplicity)

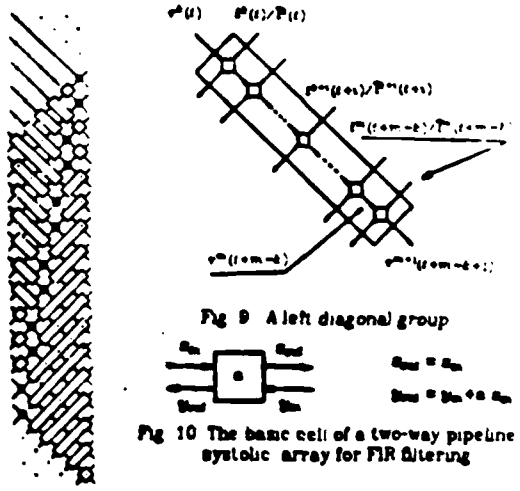


Fig 8 Testing an array that satisfies conditions L. Cells under test are depicted as full squares the other cells are depicted as points

Fig 9 A left diagonal group

Fig 10 The basic cell of a two-way pipeline systolic array for FIR filtering

Total Fault Testing using the Bipartite Transformation*

Andrea S. LaPaugh
Richard J. Lipton

Department of Electrical Engineering and Computer Science
Princeton University
Princeton, N.J. 08544

Abstract:

We present a method of creating circuits which are easily tested for all single stuck-at faults using a constant number of test vectors. The method is the combination of a number of techniques. It uses special "controllable" logic gates and latches. It requires that combinational logic subcircuits be bipartite, which is achieved by transformation if necessary. The method was previously presented for nMOS combinational logic. In this paper, we extend this method to both CMOS and to sequential circuits. We also discuss alternate methods of achieving bipartiteness during testing.

1. Introduction

In [La83], we presented a new approach to the production testing of VLSI circuits. This approach gives 100% single stuck-at fault coverage of circuits using a constant number of test vectors. It also covers many multiple faults. Generating test vectors is very fast; in fact, it does not require any searching, only a one pass analysis of the circuit. Our method has tremendous advantages over traditional methods in getting guaranteed high fault coverage without the high costs of searching for good test vectors and applying large sets of test vectors. One of its great advantages is that the set of test vectors is small enough to be stored on-chip, giving deterministic self-testing. The approach does have penalties -- primarily in circuit area but also in speed. However, we believe the advantages will outway the costs in many situations.

Our approach is the combination of three techniques which, in fact, could be used separately. The first is the use of circuits with the property of being bipartite; the second is the use of special controllable logic elements; the third is the use of small amounts of logic to observe the values of internal nodes. Originally, the approach was presented for nMOS technology and primarily combinational logic. The purpose of this second paper is to give extensions of the method to sequential logic and to CMOS. Also, we will explore variations of the method which address tradeoffs in fault coverage, area, speed, and fault tolerance. We are particularly concerned with providing alternatives when the area penalty of our basic technique is too costly.

* Supported by DARPA #N00014-82-K-0549

2. The Approach

Our approach is based on the ease with which any wire in a bipartite circuit can be controlled to the value 0 or 1. A bipartite circuit is a combinational logic circuit whose gates can be colored black and white such that no two gates of the same color are connected. Given such a coloring of the gates, the wires can also be colored so that each wire inherits the color of the gate to which it is input. Circuit output wires inherit the opposite color of the gate from which they are output.

Call a logic gate *inverting* if it has output 1 when presented with all 0's as input and 0 when presented with all 1's as input. Inverter, NAND, and NOR gates are inverting. If a bipartite circuit consists of inverting logic gates, then every wire can be forced to the values 0 and 1 using just two input vectors. (This was observed for NOR-equivalent circuits by Akers [Ak74].) The property is central to our approach; we call it the *parity principle*.

Parity Principle: *If we set all the black input wires of an inverting logic bipartite circuit to the value V and all the white input wires of the circuit to the value $\bar{V}$, then all black (respectively white) wires take on the value V (respectively $\bar{V}$).*

We use the stuck-at fault model [Br76] in which each input to a logic gate and each output from a logic gate may be independently stuck-at 0 or stuck-at 1. This model includes stuck on or stuck off faults for a MOS transistor since these are equivalent to stuck-at 0 or stuck-at 1 faults on the gate (control input) of the transistor.

To use the parity principle in a test strategy, two test vectors are applied, one with all white input wires set to 0 and all black input wires set to 1 and the other with these values reversed. These will excite any stuck-at fault at the output of a logic gate. However, note that in a fault-free circuit, all inputs to an individual logic gate will be equal. Thus not all stuck-at faults at the inputs of logic gates will be tested. For example, given a (two input) NOR gate, to detect an individual input stuck-at 0 requires that value 1 be applied to the stuck-at input and 0 be applied to the other input. Therefore, using the parity principle to test bipartite inverting logic circuits allows one to check that each logic gate output can take on the values 0 and 1 but does not catch each stuck-at fault in a transistor or at an input.

To catch all stuck-at faults in a bipartite inverting logic circuit, we replace each logic gate with a special gate. This gate uses extra global control inputs to catch all stuck-at faults for the gate. In [La83], an nMOS two input NOR gate was presented for use in circuits containing only NOR gates and inverters. An nMOS NAND for NAND/inverter circuits is similar and NOR and NAND

To be presented at International Test Conference, Oct. 1983.

gates (requiring more control variables) for use in mixed NAND/NOR circuits are extensions of these. In the next section, we present a controllable NOR for CMOS.

Given a circuit of controllable gates, the number of test vectors increases as a function of the number of global control inputs used. For a circuit consisting only of controllable two input NOR gates (inverters are created by using an NOR gate with both inputs the same), the number of test vectors increases from two to five. The purpose of the five test vectors will become clear in the next section when the CMOS NOR is presented in detail. Because combinational circuits are feedback-free, any faulty circuit will have at least one gate whose inputs come from fault-free gates but whose output is incorrect. This is detected by observing the output of every gate.

Normally, observing the output of every gate would be an impossible task for a LSI or VLSI circuit. There are far too few pins available, and mechanical probing is difficult. A scanning electron microscope can be used for such observation [Ki82], but its use is not practical for production testing of a large number of chips and prohibits field testing. However, in our approach there are very few events, i.e. combinations of values of gate outputs, that we wish to observe. The basic technique using a bipartite circuit with controllable NOR gates requires the observation of three events: all white gates output 0 and all black gates output 1; all white gates output 1 and all black gates output 0; all gates output 1. Handling special input pads and sequential logic will increase the number of events by two.

Because there are so few events using our approach, we can add extra circuitry to observe these events. Each event is observed by using a large fan-in AND to observe all outputs which should be 1 and a large fan-in OR to observe all outputs which should be 0. By physically distributing these large fan-in gates, we can keep the increase in circuit area small. Note that the number of such large fan-in gates needed is proportionally to the number of distinct events to be observed. Most test methods require the observation of a large number of events. Thus, although the observation circuitry could be added to any circuit, the amount of circuitry needed would be prohibitive for most test methods. This method of observing internal values can be used with other test methods requiring the observation of only a few events (e.g. [Ha74, Sa74]).

3. The Approach for CMOS

The CMOS controllable NOR gate is a modification of the standard ratioless pullup-pulldown CMOS NOR [Ho83] shown in Figure 1a. Figure 1b shows the new gate with additional inputs C_1 and C_2. This gate is used in circuits consisting only of inverters and NORs; the inverter is creating by using a NOR with both inputs the same. Inputs C_1 and C_2 are *global* to the circuit -- every gate in the circuit contains them. For normal operation, C_1 and C_2 have the value 1. Table 1 shows the values to be used to excite each possible stuck-at fault in the gate. Note that an n-type (respectively p-type) transistor stuck on is equivalent to its input stuck-at 1 (respectively stuck-at 0); and n-type (respectively p-type) transistor stuck off is equivalent to its input stuck-at 0 (respectively stuck-at 1).

In Table 1, note that whenever one of C_1 and C_2 takes on the value 0 and the other takes on the value 1, a fault-free gate is inverting with respect to normal inputs x and y. Thus, for these values of C_1 and C_2, the parity

Fault			Inputs				Output	
			x	y	C_1	C_2	Correct	Faulty
x	net	SA 1	0	0	1	0	1	0
		SA 0	1	1	1	0	0	1
C_1	net	SA 1	1	1	0	0	1	0
		SA 0	1	1	1	0	0	1
x	n-type	SA 1	0	0	1	0	1	short
		SA 0	1	1	1	0	0	float
x	p-type	SA 1	0	0	1	0	1	float
		SA 0	1	1	1	0	0	short
C_1	n-type	SA 1	1	1	0	0	1	short
		SA 0	1	1	1	0	0	float
C_1	p-type	SA 1	1	1	0	0	1	float
		SA 0	1	1	1	0	0	short

TABLE 1
for y faults and C_2 faults, reverse values on C_1 and C_2 with same results

principle holds and gates can be tested simultaneously -- white and black gates being tested for opposite faults. To test stuck-at 1 faults at C_1 or C_2, both are set to 0. In this case, the normal inputs and outputs of all gates should be 1; again, all gates can be tested simultaneously.

Table 1 shows that each single transistor fault in the CMOS gate results in the gate output being either electrically isolated, denoted *floating*, or on a path from VDD to VSS, denoted *short*. In nMOS, because the logic is ratioed, a short provides a valid logic 0 [Me80]. However, in CMOS, this may not be the case. Also, in nMOS a float can occur only if the depletion mode pullup transistor which is normally on is stuck off. In CMOS however, all the individual stuck off faults for transistors cause floating outputs. This is necessarily the case in ratioless CMOS since for each input combination, there should be either a path from VDD or a path from VSS to the output of a gate. To detect a stuck off transistor, such a path must be broken, causing a floating output. Similarly, any single transistor stuck on will cause a VDD-VSS short.

To detect a short, we may choose in the design of the controllable gate to size transistors so that such shorts appear as a valid logic value. This may cause some transistor faults to be undetectable. The most desirable such ratioing is to make the p-type C_1 and C_2 transistors with high resistance and the n-type C_1 and C_2 transistors with low resistance. This is also desirable for good performance of the logic gate in normal operation. The remaining transistors are adjusted so that any short leaves the output at a logic 0 (as for nMOS). Then, an n-type transistor stuck on will be detected as yielding an incorrect logic value; a p-type transistor stuck on will give the correct logic value but at a higher voltage within the allowable range.

Sizing transistors to handle shorts negates one of the great advantages of ratioless CMOS -- the ability to design logic gates so that transitions to both 1 and 0 are at approximately the same speed. If a VDD-VSS short

can be detected by detecting high leakage current on the power bus [Ma82, Ac83]. this sizing can be eliminated. The controllable gate can be designed for optimum fault-free performance. This is the method of choice for detecting shorts.

To detect erroneous but logically valid outputs and floating outputs, we must observe the values of each output gate. Possible distributed AND and OR configurations for CMOS are shown in Figure 2. These are similar to circuits which may be used in nMOS. In CMOS we use only n-type transistors and propagate only logic 0 through them. To use either type of observation circuit, we hold all circuit inputs at a test value while doing the following. The observation logic output is set to logic 1 by a separate source while holding the observation logic input at 1. Then, the separate source is disconnected and the observation logic input is set to 0. For the AND configuration, this 0 will propagate to the output if all wires being observed are correctly at value 1. For the OR configuration, the 0 will propagate to the output if some wire being observed is incorrectly at value 1. Figure 2 shows the basic construction. In practice, drivers may be needed so that the observation logic does not have too severe a delay. Note that this delay only affects the time to test the circuit – the observation logic is not used during normal operation. Since there are very few test vectors to be applied, we can afford a longer delay for each test vector than if we were using a method in which thousands of test vectors were applied. This delay is a disadvantage in that it may prohibit "at speed" testing of the circuit.

To detect floating outputs, we use the fact that such an output will hold it's old value for several circuit delays. We sequence the test vectors so that the correct values of gate outputs alternate between 0 and 1. Thus, when a gate output is floating, it will hold an old value different from the expected value. Note that there are in fact only five test vectors.

Vector	Black Inputs	White Inputs	C_1	C_2
v_1	0	1	1	0
v_2	1	0	1	0
v_3	0	1	0	1
v_4	1	0	0	1
v_5	1	1	0	0

If we use the sequence $v_1, v_2, v_3, v_4, v_1, v_2, v_3, v_5$, then the outputs of black gates should take on the sequence of values 1,0,1,0,1,1,0,1 and the outputs of white gates should take on the sequence of values 0,1,0,1,0,1,1,1. In this case, for each gate, each test vector is applied at least once when the previous output of the gate is opposite from the expected output for the test vector.

4. Sequential Logic

We have described our approach as applied to combinational circuits. We now consider sequential circuits composed of bipartite combinational logic subcircuits separated by simple latches. We will consider only latches with one data input, one output, and one latch signal. When the latch signal is high, the value of the data input is stored. The output is equal to the last input value stored and is always available. Given such a sequential circuit with bipartite combinational subcircuits, a scan path technique [Wi81] could be used to gain access to the latches. Our technique could be used for the combinational portions by loading and unloading the latches through the scan paths. The sequential loading and unloading of the latches is slow. Even worse, for CMOS where the sequence of test vectors is important, the sequential loading may cause serious problems in testing for floating output nodes. The exact design of the scan path latch will determine what is feasible. However, there is an alternative. If a special controllable latch is use in place of the simple latch, sequential circuits with bipartite combinational subcircuits can be tested using our approach without the use of scan path circuitry.

The testing of sequential circuits using the controllable latch breaks up the circuit into controllable combinational pieces just as scan path techniques do. However, instead of loading the latches with values during testing, each controllable latch has special test modes. In each test mode, the outputs of a latch are the specific values needed for the current test, regardless of the stored value. Thus, in addition to the data input and latch signal, the controllable latch will have mode inputs. In the most general case, the latch output may be connected to both black and white gates. In this case, the latch will have five modes: (i)all outputs equal to last stored input (normal), (ii) all outputs equal to 1, (iii) all outputs equal to 0, (iv) all outputs to white gates equal to 0 and outputs to black gates equal to 1, (v) all outputs to white gates equal to 1 and outputs to black gates equal to 0. (Note that for a NOR/inverter circuit mode (iii) is not needed, but in a NOR/NAND/inverter circuit it would be.) Modes (ii) through (v) are test modes and ignore the value actually stored in the latch. During testing the outputs (black and white) of the controllable latch are observed in the same fashion as the output of a logic gate. Figure 3 gives one possible design for such a gate in nMOS.

The test sequence using controllable latches is as follows:

I. Test combinational logic. Set the modes using (ii) through (v) above as appropriate to test the combinational logic pieces. Observe that correct test values are on outputs of each latch.

II. Test latching of input. The data input to each latch is either white or black; we call the latch white or black accordingly. Since the combinational logic has been tested, we can use it to set a known value in each latch.

   A. Using mode (iv), set the inputs to all white latches to 0 and the inputs to all black latches to 1. Raise all latch signals so these values should be stored.

   B. Lower all latch signals. Use mode (i) to propagate latched values to observable outputs

   Repeat A and B using mode (v) in A and again using mode (iv) in A. For each latch, this will test the transition from storing a 0 to storing a 1 and the transition from storing a 1 to storing a 0. In B, only the values at latch outputs are observed since the distinction between white and black wires is not maintained at this step. This introduces two new events to observe.

Note that to execute II, the latch signals of all latches must be controllable *independent* of the values in the circuit under test. In many circuits this will happen naturally – the control signals and data are received

and manipulated separately. If this independence does not hold, there are several alternatives. The most direct is to modify the latch signal logic so that under test the latch signals can be controlled directly. Note that only one latch signal can be used within a latch, otherwise the latch will not be completely testable. Of course, the logic creating the modified latch signal must be tested. Another alternative is to test only some latches at a time. For example, there may be two sets of latches, with the latch signals and data inputs for each dependent on the output of the other but the latch signals and data inputs for each latch in a given set independently controllable. Then we can test one set at a time by appropriately setting the outputs of the other set. Such a method requires careful analysis of the dependencies of the circuit and brings us once again to the problems of test vector generation. Thus, it is not in keeping with the spirit of our approach. However, it may prove to be a desirable method, especially if there are very few groups of latches which interact.

5. Finding Bipartite Circuits

Our approach requires that the combinational logic sections of a circuit be bipartite. In [LaS3], we have shown that any circuit can be transformed into a bipartite version by at worst doubling the number of gates. This transformation does not increase the fan-out or the length of input/output paths. We also require the use of controllable input pads similar to the controllable latch described above. In test mode, a controllable input pad allows black wires and white wires from the pad to take on opposite values.

Since the cost of transforming an arbitrary combinational circuit into a bipartite circuit may be very high, we wish to identify technologies and design methodologies which produce bipartite circuits naturally. Two common design structures which are bipartite are the NOR-NOR and NAND-NAND PLAs. (Note that each gate of a PLA may have high fan-in, requiring a proportional number of control signals.) PLA's are in fact examples of level logic. A level logic which is constructed in levels of gates so that gates in the i^{th} level receive inputs from gates in the $i-1^{st}$ level and send outputs to gates in the $i+1^{st}$ level. Any level circuit is bipartite. It is interesting to note that to string together PLAs, one must take care at the inputs. Often, a PLA uses an input value and its complement as shown in Figure 4. When the input is treated as a circuit input with a controllable pad, this does not present a problem. However, if the input is to come as the output of some other PLA, the combined circuit may not be bipartite. Instead of employing the circuit transformation, one may use a pseudo input pad between the two PLAs so that the input value and its complement can be decoupled (take on the same value) during test. This illustrates one important theme which emerges from our test approach:

> *Allow variables and their complements to assume values independently during test.*

The usefulness of this theme has been noted by others as well (e.g. [Sa82]). The pseudo input pad essentially breaks a "bad" edge. The concept of breaking "bad" edges will be expanded upon below. Note that the result is not always fully testable.

A design methodology which produces bipartite circuits is domino CMOS [Ho83]. In domino CMOS, each composite gate is actually a precharded inverting gate followed by a standard CMOS inverter. These composite gates can be connected in any feedback-free fashion to produce a bipartite circuit (color all precharged gates white and all standard inverters black).

The transformation presented in [LaS3] is advantageous because it does not increase the fan-out or lengths of paths of the circuit. Thus it does not increase the two major sources of delay. However, the area penalty for a particular circuit can be very large. A circuit may be "almost" bipartite in one of two ways. Its transformation may require only a small number of extra gates, or there may be a black/white coloring for it in which few edges go between gates of the same color (offending edges). Note that the latter does not imply the former, as shown in Figure 5. We would like a circuit with few offending edges to have a low cost modification to a bipartite version, but it may not under the original transformation. In the remainder of this section we discuss ways of modifying such a circuit at the offending edges so that the parity principle can be used. These techniques will not incur the area cost but have other costs. It should be noted that both finding a minimum size transformed circuit and finding a coloring with a minimum number of offending edges are computationally difficult (i.e. NP-complete [Ga79, GaS2]). Therefore, we do not expect to solve either problem optimally, but to find good solutions.

In each of the techniques, we will conceptually insert an extra gate on each offending edge during test so that the edge is split into two and the circuit becomes bipartite. The most straightforward solution is to actually do this as shown in Figure 6a. This introduces an extra delay during normal operation of one pass transistor (transmission gate) on each offending edge. Even worse, the normal functioning of the "edge" cannot be tested using our approach. Thus this solution costs both speed and fault coverage, but has a small area penalty when there are few offending edges.

By using an inverter whose output can be isolated as shown in Figure 6b, part of the testing problem of our initial "splitting" method can be alleviated. This inverter must have a pullup which will not dominate the pulldown paths within the preceding logic gate. In this respect, it is designed to be fault tolerant. A third control signal is required.

Two alternatives present themselves to solve the remaining lack of fault coverage. One is to introduce a second path for normal operation (making a double edge) and through this redundancy make the construction fault tolerant. The layout of such a construction with redundant paths is very important, since the likelihood of both paths being broken must be low. This alternative increases the area penalty at each offending edge and does not improve the fault coverage.

The second alternative is to modify the preceding logic gate so that we have further control over it. Note that the controllable NOR can be set to output a 1 regardless of the values of its normal inputs (the controllable NAND can be set to output a 0). Suppose we further modify the NOR, using another control input, to output a 0 regardless of the value of its normal inputs. If an offending edge is from such a controllable NOR, we can check that the normal connection from the gate to the next gate is working, i.e. that the "edge" works correctly in normal operation. To do this, we generate a 0 and a 1 as outputs of the gate independent of the gate's normal inputs. For each value, we select the normal connection through the edge and observe the value of the edge beyond the test mode circuitry. This test will introduce two new events at such edges. This alternative

requires a more complicated gate preceding an offending edge, thus further increasing the area penalty. It also requires more test vectors and observation logic than the simple strategy for "splitting" an edge. Again, if few edges are involved, the area penalties may still be less than using the original transformation.

The techniques described above all "split" an offending edge during testing so that the parity principle will hold. We may instead, reroute inputs so that each gate receives inputs from gates of the correct color during test. To do this, for each offending edge, we introduce an alternate edge to the same input as the offending edge but from an arbitrary gate of the correct color. The offending edge is selected for normal operation and the alternate edge is selected for test. This involves the addition of wires and pass transistors as shown in Figure 7. As for our simple splitting technique, a pass transistor delay is added to each offending edge and faults in the pass transistors for normal operation are undetected. This technique does not require the addition of an inverter. If routing the alternate connections is easy, very little additional area is required. Note that this technique tests a different circuit than the one desired, but one with the same gates. Thus all the gates are tested but not all of the connections are tested.

6. Concluding Remarks

We have presented a method of creating easily testable circuits, focusing on the method as applied to CMOS and sequential circuits. Our method requires bipartite combinational logic, specially designed logic gates and latches, and the addition of observation logic. The special components and observation logic could be used separately, but are designed to work with the bipartite properties to produce a circuit requiring very few test vectors. We have also presented some alternatives to the transformation presented in [La83] which try to minimize the area penalty of making a circuit bipartite for testing purposes.

We have concentrated on the stuck-at fault model. However, in addition to all stuck-at faults, our method will catch any logic gate fault which causes the gate to become non-inverting. Also, any bridging fault between a white wire and a black wire, causing both to take on the same values, will be detected. However, such bridging faults between wires of the same color are not detected. It is interesting to note that faults causing CMOS logic gates to have floating outputs are difficult to deal with in conventional testing methods since a large set of test vectors must be sequenced [Ch83]. In our method, the small number of test vectors allows us to easily sequence them by inspection to produce tests for these faults. Another advantage of the method is the ability to design self-testing chips using it, allowing the possibility of in-field testing.

Given a circuit which has bipartite combinational logic, there remain some costs in area and speed associated with the method we have described. The controllable gates and latches, routing of control signals, and observation logic require extra area. The controllable logic gates may be somewhat slower than their standard counterparts, but this penalty can be minimized during gate design (normally at the expense of area). The observation logic will present a small extra load on each gate whose output is observed; this will also introduce delay. Consequently, we do expect a circuit designed for this testing method to run somewhat slower than if designed without the testability structures. We believe this added delay will be small. However, future work comparing versions of circuits designed with and without the testability structures is necessary.

The practicality of the method we have presented ultimately depends on the relative values of chip fabrication cost, chip performance, and cost of faulty chips reaching the field. The additional circuitry will decrease the yield of chips designed using our method and thus increase the cost of fabrication. The higher fault coverage must more than compensate for the increase in faulty chips to decrease the number of faulty chips ending up in the field. We believe our method will prove advantageous when high confidence in chip correctness is required. However further study is necessary to determine the actual penalties and gains of the method in various design domains.

7. References

[Ac83] Acken, J.M., "Testing for Bridging Faults (Shorts) in CMOS Circuits," *20th Design Automation Conf*., IEEE, June 1983, pp. 717-718.

[Ak74] Akers, S.B. Jr., "Fault Diagnosis as a Graph Coloring Problem," *IEEE Trans on Computers*, C-23, No. 7, July 1974, pp. 706-712.

[Br76] Breuer, M.A., Friedman, A.D., *Diagnosis and Reliable Design of Digital Systems*, Computer Science Press (Potomac, Md.), 1976.

[Ch83] Chiang, K-W., Vranesic, Z.G., "On Fault Detection in CMOS Logic Networks," *20th Design Automation Conf*., IEEE, June 1983, pp. 50-56.

[Ga79] Garey, M., Johnson, D. *Computers and Intractability A Guide to the Theory of NP-Completeness*, W.H. Freeman and Co. (San Francisco), 1979.

[Ga82] Garey, M., Johnson, D. private communication.

[Ha74] Hayes, J., "On Modifying Logic Networks to Improve Their Diagnosability", *IEEE Trans on Computers*, C-23, No. 1, Jan. 1974, pp. 56-62.

[Ho83] Hodges, D., Jackson, H., *Analysis and Design of Digital Integrated Circuits*, McGraw-Hill (New York), 1983.

[Ki82] Kinch, R., Pottle, C., "Automatic Test Generation for Electron-Beam Testing of VLSI Circuits," *International Conference on Circuits and Computers (ICCC)*, 1982, pp. 548-551.

[La83] LaPaugh, A., Lipton, R., "Total Stuck-at Fault Testing by Circuit Transformation," *20th Design Automation Conf*., IEEE, June 1983, pp. 713-718.

[Ma82] Malaiya, Y.K., Su, S., "A New Fault Model and Testing Technique for CMOS Devices," *International Test Conf*., IEEE, Nov. 1982, pp. 25-35.

[Me80] Mead, C., Conway, L., *Introduction to VLSI Systems*, Addison-Wesley (Reading, Ma.), 1980.

[Sa74] Saluja, K., Reddy, S., "On Minimally Testable Logic Networks", *IEEE Trans on Computers*, C-23, No. 5, May 1974, pp.552-554.

[Sa82] Saluja, K., "An Enhancement of LSSD to Reduce Test Pattern Generation Effort and Increase Fault Coverage," *19th Design Automation Conf*., IEEE, June 1982, pp. 489-494.

[Wi81] Williams, T.W, Parker, K.P., "Design for Testability - A Survey," *IEEE Trans on Computers*, Vol. C-31, No. 1, Jan. 1982, pp. 2-15.

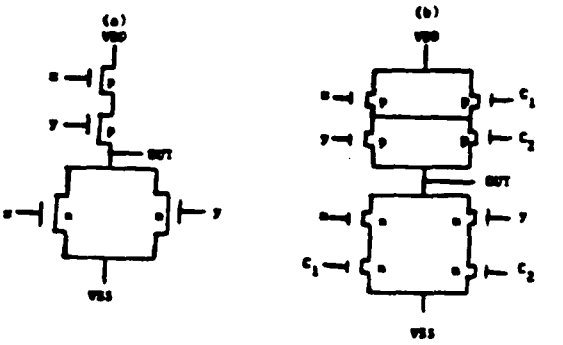
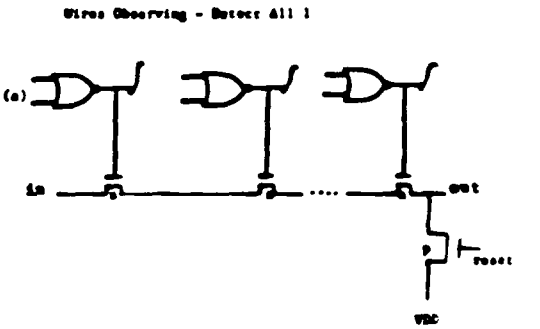


FIGURE 1

(a) Standard CMOS NOR

(b) Simple Controllable CMOS NOR



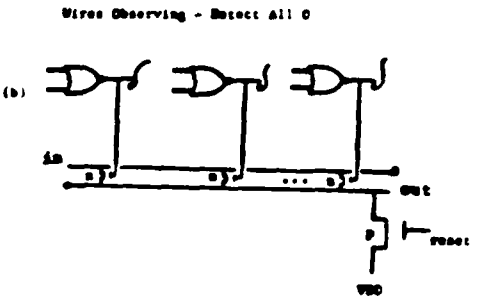


FIGURE 2

Basic Observation Logic for CMOS

(a) Distributed AND

(b) Distributed OR

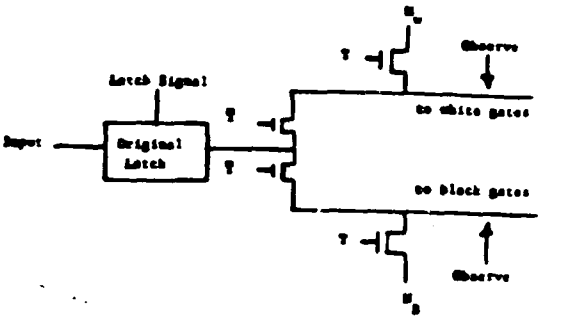


FIGURE 3

An nMOS Controllable Latch

T, M_w, and M_B are mode inputs

mode (i) T = 0

modes (ii)-(v) T = 1

M_w = value of white output

M_B = value of black output

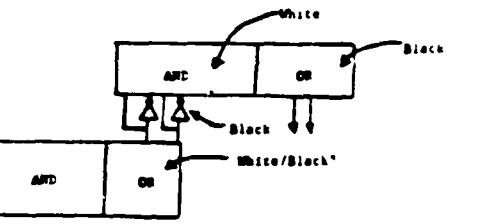


FIGURE 4

PLA Structure

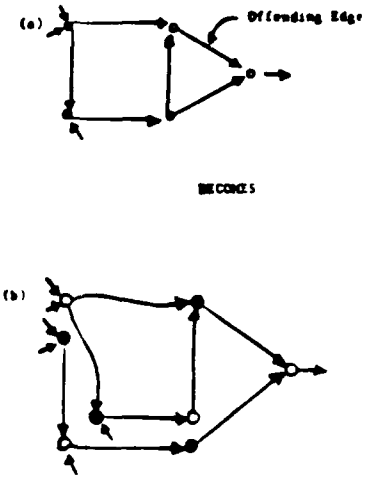
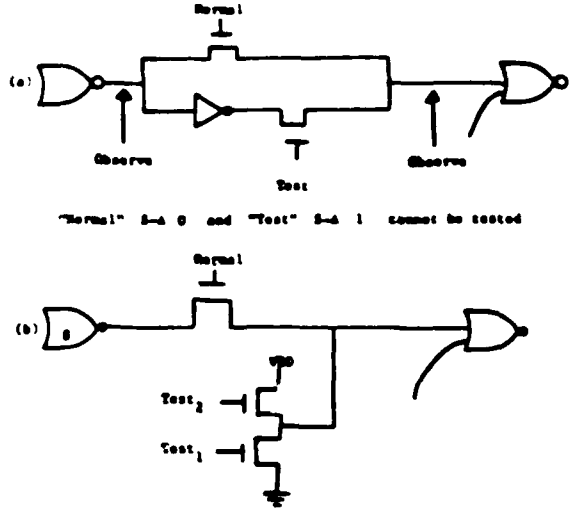


FIGURE 5

(a) Graph for circuit with one offending edge.

(b) Transformation of circuit - 5 gates becomes 8.



$Test_2$ controlled pullup is designed so that gate 8 output will dominate if $Test_2$ is S-A 1

FIGURE 6

nMOS Configurations for "Splitting" Edge

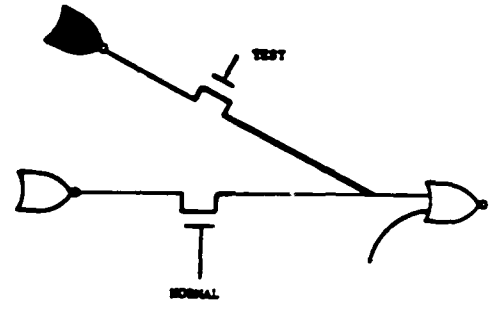


FIGURE 7

Rerouting for Test

The Impact of Processing Techniques on Communications,
NATO Advanced Study Institute, Château de Bonas,
(Gers), France, 11-22 July 1983

HIERARCHICAL, PARALLEL AND SYSTOLIC ARRAY PROCESSING

Kenneth Steiglitz

Electrical Engineering and
Computer Science Department
Princeton University
Princeton, New Jersey 08544

1. SIGNAL PROCESSING AND VLSI

Many signal processing algorithms are highly regular, data-independent, and access the data in fixed patterns. For these reasons the current technological advances in very large scale integrated circuits hold especially great promise for signal processing, and in fact we now see the development of many highly integrated processors of a more or less specialized nature. At one end of the spectrum, we see programmable signal processing chips that are really microprocessors, with program, memory and logic separated as in a general-purpose machine. At the other extreme, we see highly-specialized, custom chips that perform fixed tasks; typically the data moves through the chip along fixed, regular paths, the arithmetic logic is distributed in space, and the "program" is really "hard-wired" into the topology. This talk is devoted to a study of this latter, custom variety of architecture.

The range of algorithms that are commonly used for digital signal processing is not very great; a few very important algorithms are used intensively. They fall roughly into four categories: convolution and filtering; Fourier transforms; matrix calculations (see, for example, [36]); and iterative

algorithms for adaptive filtering. All these applications are characterized by two important characteristics that make special-purpose, highly dense hardware very attractive:

- ***high-volume, real- or nearly real-time data-flow requirements***, and
- ***effective algorithms with regular patterns of data access and fixed operation sequences***.

There are direct architectural consequences of these characteristics. The regularity of the patterns of data access and operation sequences makes possible a high degree of pipelining and multiplexing. This, in turn, makes possible a high-volume data flow. Furthermore, the regularity of the algorithms is reflected in a regularity of VLSI circuit structure, so that a hierarchical approach to layout design and specification becomes possible, and that greatly simplifies the design and development of large-scale, custom VLSI circuits for digital signal processing. The rest of this talk is devoted to these architectural consequences: In the next section we will discuss some general aspects of parallelism, and why the need for parallelism justifies the development of custom, single-purpose chips. Section 3 is devoted to highly-pipelined and systolic structures, using filtering as an example, with a review of some useful topologies. Section 4 deals with how some of the important structures can be combined in a hierarchical way . We will review mathematical models of VLSI computation and available lower and upper performance bounds in Section 5. Section 6 will deal with the practical matter of maximizing throughput by appropriate choice of latching density.

2. PARALLELISM AND THE CUSTOM/PROGRAMMABLE DECISION

There is a general tradeoff between specialization and programmability in digital signal processing chips. The obvious advantages of flexibility afforded by programmability must be weighed against the higher potential throughput of a custom chip. The choice between the two is dictated by the cost in time and money of designing and testing a chip, and by the need for very high-throughput real-time processing. This tradeoff changes with time and technology: as chip design becomes a highly automated process, and as more real-time, high-volume applications arise (such as in the fields of communications and robotics), we are likely to see the proliferation of very specialized signal processing chips.

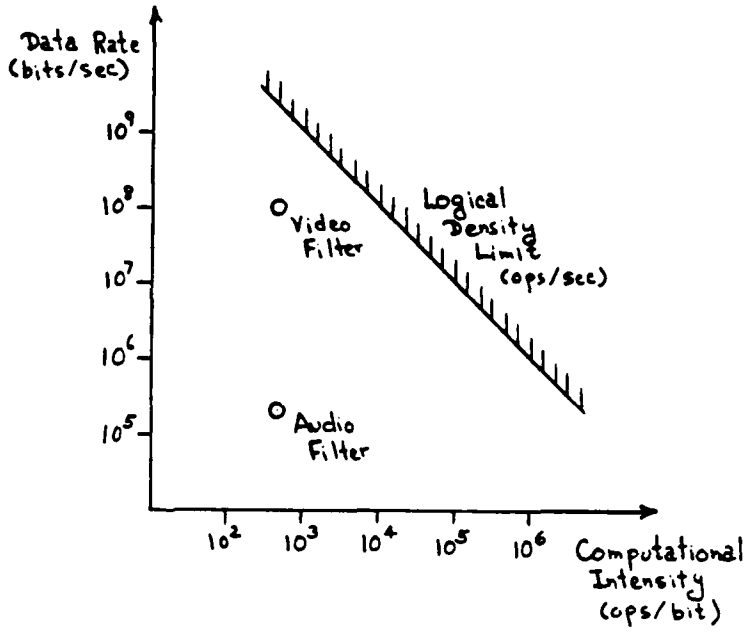


Figure 1. Signal Processing Tasks in the intensity-rate plane (after [7,9]).

Figure 1 shows one way of looking at the range of signal processing applications. We plot the data rate in bits/sec as ordinate, and the computational intensity requirement of a given task in operations/bit as abscissa. For example, a low-order FIR filter has a low computational intensity, whereas a high-order filter has a correspondingly high intensity. A 50th-order FIR filter at a sampling rate of 20,000 words/sec (a typical audio rate) and 10 bits/word, with say 100 logical operations (at the gate level) per fixed-point multiply-by-constant, corresponds to a data rate of 2×10^6 bits/sec, and a computational intensity of 5×10^2 operations/bit. For a chip of fixed size and for a given technology, the product of the two coordinates in operations/sec is bounded from above by some constant: the total number of operations possible if every piece of the

chip were doing useful work all the time. For a chip with 10^5 gates and a clock period of 100 nsec, this is 10^{12} operations/sec. This boundary is shown by a hyperbola in Fig. 1 (a straight line in log-log coordinates). At the same time, there is an upper limit on the data rate, determined by the number of I/O pins and the speed of the I/O drivers.

We are thus constrained to work within the region shown. Whenever an operation is carried out that does not contribute directly to the processing of the signal, as counted by the measure of computational intensity, we move away from the boundary. Consider, for example, the operation of a programmable signal processor with a stored program. Every instruction fetch or store, every instruction decoding, and every test for branching, is wasted in the sense that a part of the chip is doing work that is not essential. Also wasted, of course, is any part of the chip that remains idle during any particular clock cycle.

We are lead to the conclusion that the most efficient use of chip area, the dearest resource at present, should avoid programmability, and should make concurrent use of as much of the chip as possible. When demands on performance are very high, at the limits of applications technology, we are lead to the design of custom, single-purpose chips with fixed data-flow paths. Thus some filtering tasks at audio bandwidths may be best implemented now with programmable chips, but applications at video rates, like robot vision, demand custom designs.

In this talk we will use the operation of convolution for our examples. It is no doubt the most widely-used of all the digital signal operations, and is also representative in terms complexity and throughput requirements. We write it as

$$y = w \otimes x = \sum_k w_k x_{n-k} = \sum_k x_k w_{n-k} \qquad (1)$$

The function w_k will be called the ***weight sequence***, and will usually be of finite duration, so that the limits of the summations in (1) will be finite. We will distinguish two situations in which convolution is usually implemented: ***general convolution***, where the weights w_k are variable on a short-term basis, as fast as the signal x_k; and ***filtering***, where the weights w_k are fixed (or at least infrequently changed). We will make no distinction between convolution and ***correlation***, which is simply convolution with one of the signals time-reversed.

Convolution can be applied in many ways. At the bit level, with Boolean product, and Boolean sum with carry, it means

binary multiplication. At the signal level it means filtering or correlation. At the logical level it means pattern matching. This observation allows us to develop highly regular VLSI topologies by first developing a structure at the *word* level for convolution. A similar structure is then used recursively to build a multiplier at the *bit* level. The result is a hierarchical structure that is highly regular, being uniform in topology all the way down to the bit level. In the next two sections we carry out just this plan.

3. SYSTOLIC AND COMPLETELY-PIPELINED STRUCTURES

Highly concurrent VLSI circuits can be characterized by the following desirable properties [7,9]:

- *Local-Connectedness* : This means that computational elements are connected only to nearby neighbors.
- *Flow-Simplicity* : This means that each element is used only once per elementary computation.
- *Cell-Simplicity* : This means that each element takes only constant time for its computation; that is, its computation time does not depend on the such parameters as the number of bits in a word, or the number of coefficients in a filter.

Systolic arrays [21,22,27] can be characterized as those that are both *locally-connected* and *flow-simple*. Wires are short and the data flows through the structure in a smooth way. However, each "cell" may be very complex (a multiplier, possibly), and may take time dependent on the problem parameters.

Completely-pipelined circuits [7,9] are another class of highly concurrent, pipelined circuits, characterized by the properties of being *flow-simple* and *cell-simple*. These circuits are more general than systolic ones in that long wires are allowed, but more restricted in the sense that the computational cells operate in time independent of the problem parameters (such as word-size).

In what follows we will concentrate on the simplicity and regularity of some computational structures and ignore some problems that are important at a practical level, but which would obscure the presentation. For example, the question of efficient use of area will be ignored for now, but will be discussed in Section 5. The problem of distributing power, ground, and clock lines will likewise be ignored; some

discussion of these points in the present context can be found in [9]. We will also not worry about the signs of numbers in describing multiplication, but assume that extension bits are added to two's-complement numbers so that the answer is always in range (see [13,31,33] for some discussion of this issue).

3.1 *Word-Serial Filtering*

Figure 2a shows the conventional signal flow graph F for FIR filtering (see [37], for example): the input signal x_k is delayed along a chain of registers, and during each clock period the appropriate samples are multiplied by the corresponding weights w_{n-k} and summed. At the right we see the computation that must be performed every clock cycle. Notice that for this example of a 4-coefficient filter, not only do we need to perform 4 multiplications and 3 additions between clock pulses, but the input of the second addition depends on the result of the first, and the input of the third depends on the result of the second. This means that we cannot perform these additions in parallel, and therefore that the throughput rate is limited by the time for three additions.

Figure 2b shows another signal flow graph F^t, the ***transpose*** (see [34], for example) of signal flow graph F (called B1 in [21]). The transformation of transposition entails reversing the direction of every arc, replacing summing nodes by branching nodes, replacing branching nodes by summing nodes, and interchanging input and output. Here the sequence of sums is replaced by a ***broadcast*** of the input signal x_k at any time; the computation during each cycle is again shown at the right. This broadcasting, or ***fanout***, of a signal carries with it a certain penalty in terms of delay, but is generally much faster than sequential add operations, so that this signal flow graph can be implemented with a much higher throughput if the three additions are implemented with three adders operating in parallel. The commercial chip described in [47] uses this transposed structure.

The fanout problem of graph F^t and the sequential delay of graph F can be avoided by using the graph F^2, shown in Fig. 2c. Here on every clock pulse the input signal moves to the right, and the output signal moves to the left. The transfer function of F^2 is $H(z^2)$ if the original graph F has transfer function $H(z)$, so that meaningful output is obtained only every other clock cycle (this is the structure W1 of [21]). Thus the input and

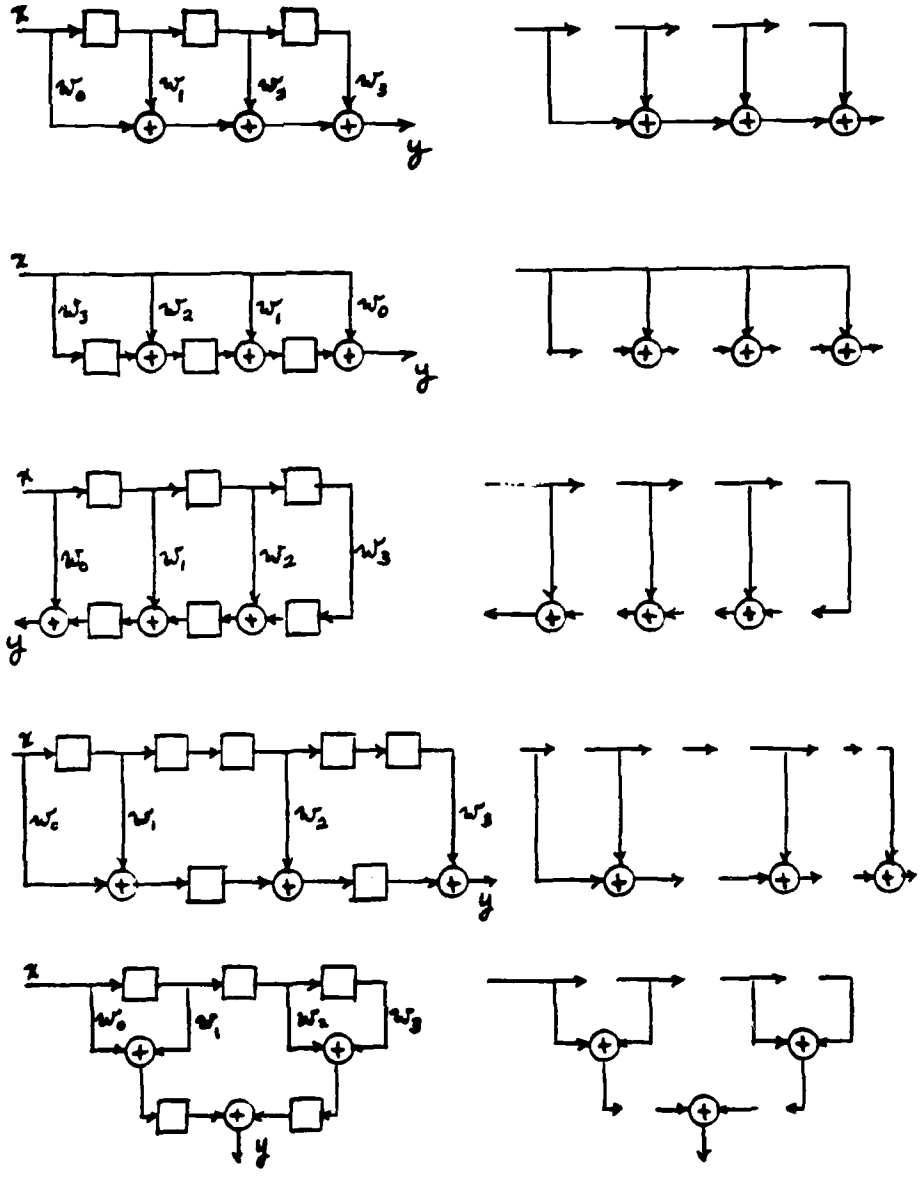


Figure 2. Structures for word-serial filtering. The boxes are delay elements. The computation during each clock period is shown at the right. From the top: a) F, b) F^4, c) F^2, d) F^3, e) F^5.

output signals must be interleaved with zeros (or two independent filtering operations can be interleaved).

It seems that there is no way to avoid all the difficulties with a single structure. For example, the summing nodes of F can be separated by registers (delays), and corresponding extra delays inserted between inputs, producing the circuit F^d shown in Fig. 2d (called W2 in [21]). Here the input and output signals move in the same direction, but at different speeds. But this graph has more registers than F or F^t, and has a delay before the output appears.

Finally, Fig. 2e shows the structure F^Δ that results when the additions in F are performed using a binary-add tree [7,9,37]. This is a convenient structure for visualizing the convolution operation, and may be useful for general convolution (as opposed to filtering). Care must be taken, however, that the tree is laid out in a way that does not take up too much area on the chip. The recursive configuration of an H-tree is useful for that purpose [32].

Are the preceding structures, by our terminology, systolic or completely-pipelined? Graph F can be laid out to be locally-connected, but is not flow-simple if a single adder is used sequentially (and that is only reasonable since multiple adders would not be usable in parallel). Neither is it cell-simple, since the time for the elementary computation (that between clock pulses) depends on the number of coefficients in the filter. The structure F is therefore neither systolic nor completely-pipelined.

Signal flow graph F^t is not locally-connected, because the length of the longest wire, used to broadcast x, depends on the filter order. It is, however, flow- and cell-simple, so it is by our definition completely-pipelined, but not systolic.

Graph F^e is both systolic and completely-pipelined; it can be laid out so as to be locally-connected, and is flow-and cell-simple.

Finally, signal flow graph F^Δ is completely-pipelined, but not systolic, since any layout (including H-trees) will have wires whose lengths depend on the filter order.

3.2 *Bit-Serial Multiplication*

The same structures used for word-serial filtering can be used for bit-serial multiplication by a constant, with the difference that each summing node is a full adder with three

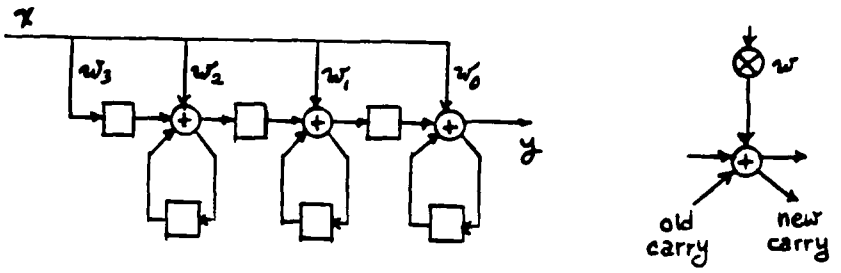


Figure 3. M^1: The structure F^1 adapted for bit-serial multiplication.

inputs and two outputs. The inputs to each full adder are the two addend bits and the preceding carry bit, the outputs are the sum bit and the carry bit. Figure 3 shows the multiplier corresponding to graph F^1; it is really no more than a simple implementation of the ordinary shift-and-add elementary-school multiplication algorithm. We will denote by M, M^1, M^2 and M^Δ the multipliers corresponding to F, F^1, F^2, and F^Δ, respectively.

3.3 *Word-Parallel Filtering*

We now consider word-parallel filtering, and, in the next section, the corresponding operation of bit-parallel multiplication. Figure 4 shows a diamond array with the signal x entering from the top left and the filter weights from the top right (when the weights w_k are fixed, they need not be transmitted through the array as shown but can be stored in place). Notice that the signal values x_k corresponding to a given signal are arranged on a horizontal line, and hence skewed in time so that successive values enter the diamond array at successive clock pulses. The next horizontal line will have another signal in it, and blocks of filtered signals emerge from the bottom of the array at successive clock pulses. The sides of the diamond array have length proportional to the filter order.

Figure 5 shows the detail of a node of the array: Each node in Fig. 4 contains a multiplication by a weight and an adder, and each arc has a delay element (a latched register). We will call the overall structure F^4 (for *Array Filter*).

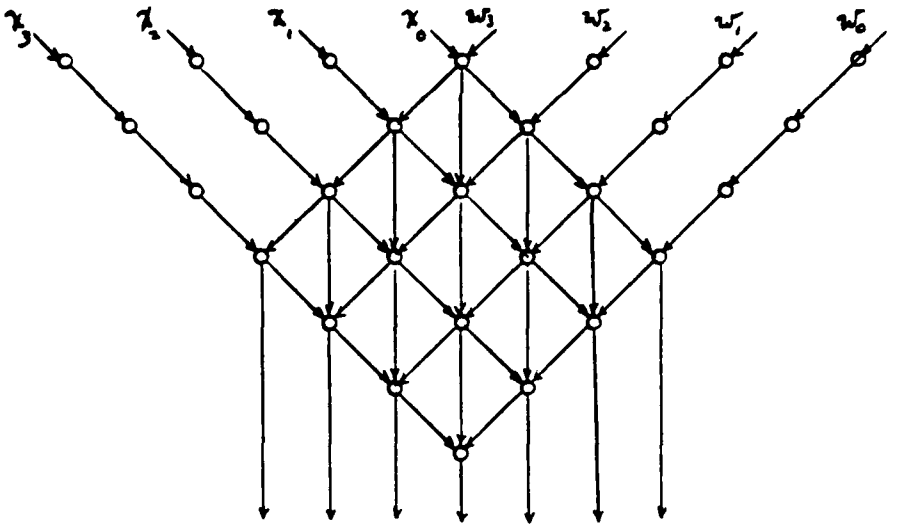


Figure 4. A structure for word-parallel filtering, F^4.

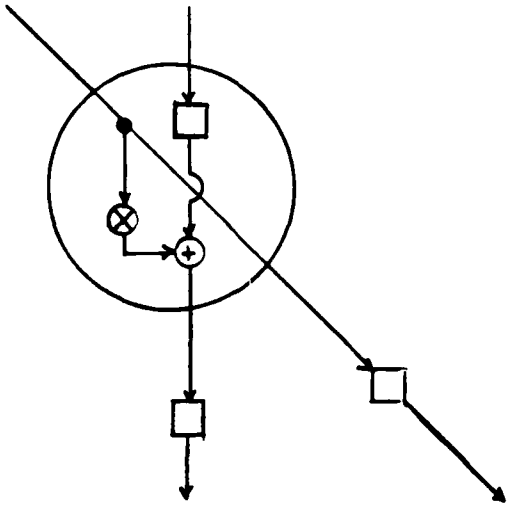


Figure 5. *Detail of a node in* F^4.

3.4 *Bit-Parallel Multiplication*

As before, the filtering structure becomes a multiplication structure when the multipliers are replaced by Boolean product and the summers by full adders with carries. In this case the carries propagate down and to the left, which direction corresponds to the next-higher bit of the product. An extra triangle is needed at the lower left so that the carry bits can propagate all the way to the left (see [31], for example). The reader will recognize this signal flow graph as nothing more than an array multiplier (we will call it M^4), with every value latched between clock pulses (see [31,37]). This parallel multiplication structure has the property that the full adders at the top of the diamond can accept inputs without extra logic, so that the multiplier can function as an accumulator as well, and this fact is useful in FIR filters and other applications [10,15].

This hexagonally-connected array multiplier is locally-connected, cell-simple, and flow-simple, and is therefore both systolic and completely pipelined by our definition.

3.5 *Other Useful Structures*

We have already seen the ***linearly-connected array*** (in all the serial examples), the hexagonally-connected array (in F^4 and M^4), and the ***leaf-connected tree*** [7,9] (in F^Δ and M^Δ). We now mention some of the other regular topologies that have proven useful in constructing computational networks for VLSI. A structure called ***cube-connected cycles*** is used in [35] for bit-parallel multiplication. A tree-like structure can be used to shorten the delay (latency) of a parallel multiplier, and the resulting structure, called a ***mesh-of-trees*** can be found in [8,25]. Leighton also discusses an analogous topology called a ***tree-of-meshes*** [25].

We have seen above a variety of different topologies, all of which perform similar computational tasks. Some work has been done towards developing a unified treatment of computational structures of this type, and showing how they can be expressed conveniently and derived from each other. For more about such mathematical representations see [12,14,20,45,46].

4. HIERARCHICAL METHODOLOGY

An important feature of the regular topologies exemplified in the preceding section is that they can be combined in a recursive, or hierarchical, way. The most obvious application

of this idea is to use bit-serial multiplication within a word-serial filter, yielding a bit-serial, word-serial filter that is fully-pipelined. On each clock pulse, every bit moves, every piece of hardware (silicon) is used, and one output bit appears. Bit-serial adders are needed at the summing nodes. Such structures have been discussed widely in the literature recently [6,13,16,23,30], and are attractive at this time because a reasonably high-order filter can fit on one chip, and the interconnection problems caused by high pin counts are greatly alleviated by the bit-serial nature of the computation.

Suppose for illustration that the multiplier M^t is used within the filter F^t, resulting in what we will call $F^t(M^t)$. Figure 6 shows a schematic representation of this filter, which is similar to those described in [6,13]. In theory, then, we have the ingredients for 5 × 5 = 25 different bit-serial, word-serial filters, all of which have slightly different timing and layout details.

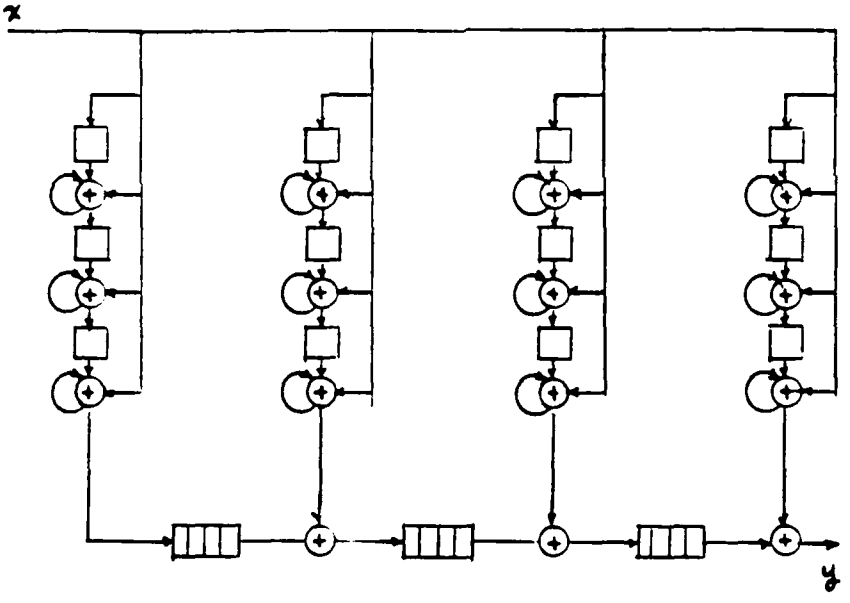


Figure 6. Recursive use of the structure F^t: $F^t(M^t)$.

To go one step further, we can combine serial and parallel structures. For example, at the other extreme from the completely bit-serial filter just described, we can assemble $F^A(M^A)$, producing a bit-parallel, word-parallel filter — one that produces a completely filtered block of signal samples once every clock pulse. (Now we need bit-parallel adders at the summing nodes.) Of course, the amount of area is greatly increased over the bit-serial filter, but so is the throughput.

In the same way, we could use a bit-serial multiplier within a word-parallel filter (yielding $F^A(M^t)$, for example), which produces a bit-serial, word-parallel filter, which for B-bit words, produces a complete block of filtered samples every B clock pulses. With only the 6 different structures discussed here, there are 36 possibilities, each having its own characteristics in terms of layout area and throughput.

An important advantage of this approach to VLSI layout is that some of the problems associated with design and layout are greatly simplified, since the overall problem is broken down into natural pieces, each of which can be handled in relative isolation. Such a design methodology is well-suited to the use of high-level layout languages and silicon compilers (see, for example, [17,18,29,38]).

Another advantage of the hierarchical approach is in the crucial but often neglected area of testing. Because the complexity of testing arbitrary circuits can grow exponentially with the size of the circuit, it is a great advantage to be able to break a circuit into blocks whose function can be tested independently of the other blocks. Much the same approach has proven very valuable in the design of large software systems. Some recent results in the testing of regular bilateral arrays can be found in [19,40,42].

5 MODELS AND BOUNDS FOR VLSI

The signal processor has heretofore been concerned mainly with speed of operation. High throughput on a general-purpose computer is achieved by managing *time*. But now the designer of systems has a new resource to manage: *area*. It is no longer sufficient to specify a sequence of instructions for data processing. We must now specify a geometric layout. Of course the requirements of high speed and small area are mutually conflicting. Consider, for example, the multipliers discussed above. A B-bit-serial multiplier like M^t will generally have a throughput rate of one product every B clock pulses

before the answer is ready, and generally takes area proportional to B. On the other hand, a B-bit-parallel multiplier such as M^4 has a throughput rate of one product every clock pulse (once the pipeline is full), but area proportional to B^2. Thus there appears to be a conservation law at work, and we expect that bounds can be obtained on such quantities as ***throughput per unit area***. We will describe some such bounds below.

5.1 *Some Terminology*

We need to define some important terms precisely. First, the *delay* or *latency* T of a signal processing device is the time between the arrival of the first bit of the input signal at the input port, and the time that the last bit of the answer appears at the output port. This is the usual usage of the term "computation time." But in many signal processing applications we are concerned more with the throughput rate than with the delay. We define the time between successive outputs with pipelines full as the *period* P of a chip, and the reciprocal of the period as the *throughput*.

If a quantity is bounded from above by a constant multiple of $f(B)$ for sufficiently large B, where B is any parameter of interest (often the number of bits), we say the quantity is $O(f(B))$. So, for example, the array multiplier M^4 requires area $O(B^2)$. A corresponding *lower* bound is written $\Omega(B)$.

5.2 *A VLSI Model*

We will next describe a mathematical model for a VLSI chip, one that is abstract and simple enough so that results can be proved about it, but one that is also realistic enough so that the results provide some guidelines, or at least hints, about reality. The model we describe is attributed by Vuillemin to the three sources [4,32,41]; this and similar models have been used by many others. There is a fairly large literature on models and bounds that we will not attempt to survey completely here (see, for example, [1,2,4,5,24,26,28,35,39,41,43]).

The basic premise is that there is a minimum feature size λ_w, and minimum delays τ_w and τ_g, dictated by the technology. The important assumptions are then that a) no two wires can have their midpoints closer together than λ_w, b) every logical unit (such as a gate) must have area at least λ_w^2, c) passing a signal through a wire entails a delay of at least τ_w, ***independent of the wire's length***, and d) passing a signal through a gate entails a delay of at least τ_g.

As discussed in [2], the assumption that the delay is independent of wire length is true only in certain regimes. Depending on the technology, the time for propagation of signals may be independent of wire length, as we assume here (the *synchronous* model [4]), or proportional to the logarithm of the wire length (using repeaters), or proportional to the square of the wire length (diffusion case).

5.3 *Lower Bounds*

The essential result is expressed in a nicely general form by Vuillemin [43]. He defines a wide class of functions called *transitive* functions, which includes integer product, convolution, linear transform, and matrix product.

Theorem [43]: Any circuit that computes a transitive function has wire area

$$A = \Omega(D^2) \tag{2}$$

where D is the data rate in bits/sec.

The period P is related to the data rate D in a simple way: $P = N/D$, where N is the number of input bits. Therefore the bound above can also be written

$$AP^2 = \Omega(N^2) \tag{3}$$

We can also observe that if any one of N input bits can affect the output, and if there is a constant bound on the allowed fan-in, then the circuit must have at least $\log N$ stages. This implies the following lower bound on the latency T:

$$T = \Omega(\log N) \tag{4}$$

A good example to illustrate the use of these bounds is B-bit multiplication. The bounds above tell us that $AP^2 = \Omega(B^2)$, and $T = \Omega(\log B)$. The array multiplier described above has $A = O(B^2)$, $P = O(1)$, and $T = O(B)$. It is therefore asymptotically optimal under the measure AP^2, but possibly has more delay than necessary. In fact, the delay can be reduced to the asymptotically optimal $O(\log B)$ with the area increasing only from $O(B^2)$ to $O(B^2 \log B)$ [8] (see also [44]).

An interesting measure of goodness, that takes into account both period and latency, is AP^2T^2. By the arguments above the lower bound on multiplication is then $AP^2T^2 = \Omega(B^2 \log^2 B)$. The array multiplier mentioned above [8]

has the upper bound $AP^2T^2 = O(B^2\log^3 B)$, which is therefore no more than one log-factor away from asymptotic optimality, by this measure at least.

Compare this result with the bit-serial multiplier M^1, for example. That structure has area $A = O(B)$, period $P = O(B)$, and latency $T = O(B)$, so that $AP^2T^2 = O(B^5)$. This is an indication that the overall efficiency of silicon utilization is not as good asymptotically as that of the parallel array multipliers, but one should not conclude too much from this argument. For one thing, we may need a multiplier with small area simply because an array multiplier will not fit on a chip, in which case we must settle for the possible instead of the asymptotically good. It is also quite likely that the constants of proportionality favor the simple linearly-arranged designs such as M^1, so that for reasonably-sized B the measures above may be misleading. The asymptotic measures give us useful guidelines for comparing designs that are similar, but there are so many factors in choosing a multiplier for a particular application at a particular point in technological development, that mathematical analysis should be interpreted with caution.

6. PIPELINING AND LATCHING FOR THROUGHPUT

In the designs discussed above it was assumed that every signal value was latched (that is, held in a register) at every stage in the signal flow graph. So, for example, the array multiplier M^4 has a register after every full adder (this was stressed in [31]). This means that every part of the circuit can be used for holding intermediate results —that every part can function as a pipeline. This approach leads to high throughput at the expense of delay. In contrast, array multipliers that are commercially produced on packaged chips do not generally have a high degree of pipelining in this sense of the term; the answer is usually produced in one or two clock cycles, and the carry signals ripple through the structure, settling in time for each new clock pulse. In [3], for example, an array multiplier with combinational logic that is 113 gates deep is mentioned. Thus, commercial single-package multipliers are optimized for latency and not throughput, and are therefore not necessarily "fast" for custom chip designs for signal processing applications.

However, latching at every possible stage of a circuit does not necessarily lead to the highest throughput. First, the latches themselves take time to operate; their input stages

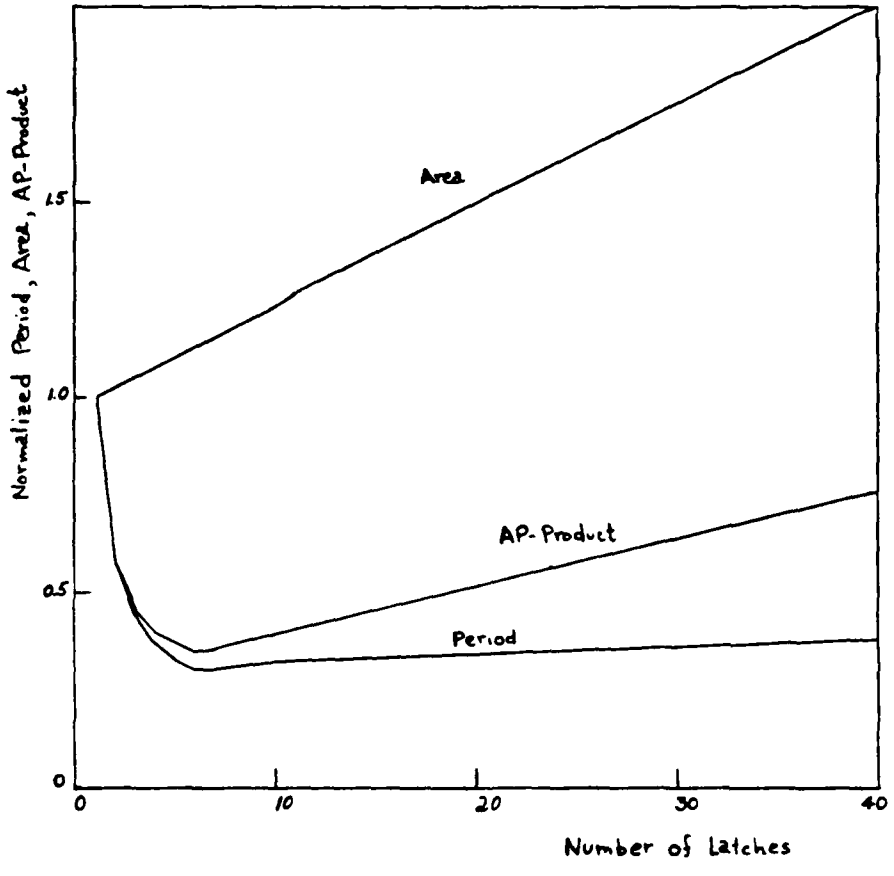


Figure 7. Area, period, and area-period product, as functions of the number of intermediate latching stages m, in a typical pipelined array multiplier (after [11]).

must be charged, and they must charge the input stages of the next layer of logic. Second, the clock driver must drive the additional capacitance of the latches, and for a given driver this lengthens the clock rise- and fall-times and decreases the possible clock rate.

If we start with one stage of a circuit that has combinational logic that is many gates deep, and we introduce m intermediate stages of latching, we decrease the period by a factor

of roughly $1/m$, up to the point that the latching and clock-driving time becomes comparable to the propagation time of signals through one stage of logic. After that point there are diminishing returns to the addition of more latching. In [11] the generic situation of a block of combinational logic is modeled mathematically, and the optimal choice of the number of additional latches, m, was studied. Figure 7 shows typical curves of area, period, and area-period product as a function of m for a circuit with a depth of 100 gates. As can be seen, the period as a function of m decreases sharply to a minimum and stays almost constant, the area increases steadily with m, and the product has a well-defined minimum. We may wish to minimize this product AP instead of the period P; $1/AP$ can be written as $(1/P)/A$ — the *throughput-per-unit-area*. In any case, the optimal values of m for minimizing P or AP will be close to each other.

A typical example of such a situation occurs in the implementation of the bit-parallel array multiplier M^4 discussed in Section 3.4. Here the analysis predicts that the period of a 16-bit array multiplier can be decreased from 210 nsec to 66 nsec by the addition of 5 stages of intermediate latches, with an attendant increase in area of only 13%.

7. CONCLUSIONS

The design and development of custom chips for signal processing tasks is very challenging, calling as it does on the signal processing expert to make decisions at many design levels: he must manage overall system architecture, circuit topology, timing, area utilization, and layout. At the same time, making good use of such resources can lead to reliable low-cost devices that have very high throughput in many signal processing applications.

The key to effective design is a high degree of pipelining using regular, repetitive structures and fixed data-flow paths. Such structures can be hierarchically organized, making the design and layout problems manageable.

8. ACKNOWLEDGEMENTS

I want to thank Prof. Peter R. Cappello of the Computer Science Department, University of California, Santa Barbara,

California. Some of the work discussed here is due originally to him, and forms part of his Ph. D. dissertation (Princeton University, 1982).

This work was supported in part by NSF Grant ECS-8120037, U. S. Army Office-Durham Grant DAAG29-82-K-0095, and DARPA Contact N00014-82-K-0549.

REFERENCES

1. Abelson, H. and P. Andreae, "Information Transfer and Area-Time Tradeoffs for VLSI Multiplication," *CACM*, Vol. 23, Jan. 1980, pp.20-23.

2. Bilardi, G., M. Pracchi, and F. P. Preparata, "A Critique and an Appraisal of VLSI Models of Computation," in *VLSI Systems and Computation*, H. T. Kung, Bob Sproull, and Guy Steele (eds.), Computer Science Press, Rockville, Md., 1981.

3. Böticher, K., A. Lacroix, M. Telmi, D. Wesseling, "Integrated Floating Point Signal Processor," *Proc. 1982 IEEE Int. Conf. on Acoustics, Speech, and Signal Processing*. Paris, May 1982, pp. 1086-91.

4. Brent, R. P. and H. T. Kung, "The Chip Complexity of Binary Arithmetic," *Proc. 12th Annual ACM Symposium on the Theory of Computing*, Los Angeles, Ca., April 1980, pp. 190-200.

5. Brent, R. P. and H. T. Kung, "The Area-Time Complexity of Binary Multiplication," *JACM*, Vol. 28, No. 3, July 1981, pp. 521-534.

6. Cappello, P. R., and K. Steiglitz, "Digital Signal Processing Applications of Systolic Algorithms," in *VLSI Systems and Computations*, H.T. Kung, Bob Sproull, and Guy Steele (eds.), Computer Science Press, Rockville, Md., 1981.

7. Cappello, P. R. and K. Steiglitz, "Bit-Level Fixed-Flow Architectures for Signal Processing," *Proc. 1982 IEEE Int. Conf. on Circuits and Computers*, New York, N. Y., Sept. 29 - Oct. 1, 1982.

8. Cappello, P. R. and K. Steiglitz, "A VLSI Layout for a Pipelined Dadda Multiplier," *ACM Trans. on Computer Systems*, Vol. 1, No. 2, May 1983, pp. 157-174.

9. Cappello, P. R. and K. Steiglitz, "Completely Pipelined Architectures for Digital Signal Processing," *IEEE Trans. on Acoustics, Speech, and Signal Processing*, Vol. ASSP-31, No. 4, August 1983, in press.

10. Cappello, P. R. and K. Steiglitz, "A Note on 'Free' Accumulation in VLSI Filter Architectures,' submitted for publication.

11. Cappello, P. R., A. S. LaPaugh, and K. Steiglitz, "Optimal Choice of Intermediate Latching to Maximize Throughput in VLSI Circuits," *Proc. 1983 IEEE Int. Conf. Acoustics, Speech, and Signal Processing*, Boston, Mass., April 14-16, 1983, pp. 935-938. (Also *IEEE Trans. on Acoustics, Speech, and Signal Processing*, in press.)

12. Cappello, P. R. and K. Steiglitz, "Unifying VLSI Array Designs with Geometric Transformations," 1983 IEEE Int. Conf. on Parallel Processing, Aug. 1983.

13. Caraiscos, C. and B. Liu, "Bit Serial VLSI Implementations of FIR and IIR Digital Filters," *Proc. 1983 Int. Symp. on Circuits and Systems*, May 1983.

14. Culik II, K. and J. Pachl, "Folding and Unrolling Systolic Arrays," Research Report CS-82-11, Faculty of Mathematics, University of Waterloo, Waterloo, Ontario, Canada, April 1982.

15. Denyer, P. B. and D. J. Myers, "Carry-Save Arrays for VLSI Signal Processing," in *VLSI 81: Very Large Scale Integration*, John P. Gray (ed.), Academic Press, London, 1981.

16. Denyer, P. B., "An Introduction to Bit-Serial Architectures for VLSI Signal Processing," Draft of a paper presented at Advanced Course on VLSI Architecture, University of Bristol, U.K., July 1982.

17. Denyer, P. B. and D. Renshaw, "Case Studies in VLSI Signal Processing using a Silicon Complier," *Proc. 1983 IEEE Int. Conf. on Acoustics, Speech, and Signal Processing*, Boston, Mass., 1983, pp. 939-942.

18. DeMan, H., J. Van Ginderdeuren, and N. Goncalves, "Custom Design of Hardware Digital Filters on I.C.'s," *Proc. Custom Integrated Circuits Conf.*, Rochester, N. Y., 1982.

19. Gray, F. G. and R. A. Thompson, "Fault Detection in Bilateral Arrays of Combinational Cells," *IEEE Trans. on Computers*", Vol. C-27, 1978, pp. 1206-1213.

20. Johnsson, L. and D. Cohen, "A Mathematical Approach to Modeling the Flow of Data and Control in Computational Networks," in *VLSI Systems and Computation*, H. T. Kung, Bob Sproull, and Guy Steele (eds.), Computer Science Press, Rockville, Md., 1981.

21. Kung, H. T., "Why Systolic Architectures?" Carnegie-Mellon Univ., Dept. of Computer Science, CMU-CS-81-148, Nov. 1981.

22. Kung, S. Y., and D. V. Bhaskar Rao, "Highly Parallel Architectures for Solving Linear Equations," *Proc. 1981 Int. Conf. on Acoustic, Speech, and Signal Processing*, Atlanta Ga., 1981, pp. 39-42.

23. Kung, H. T., L. M. Ruane, and D. W. L. Yen, "A Two-Level Pipelined Systolic Array for Convolutions," in *VLSI Systems and Computations*, H. T. Kung, Bob Sproull, and Guy Steele (eds.), Computer Science Press, Rockville, Md., 1981.

24. Leighton, F. T., "New Lower Bound Techniques for VLSI," *Proc. 22nd Annual Symposium on Foundations of Computer Science*, Nashville, Tenn., Oct. 1981.

25. Leighton, F. T., "A Layout Strategy for VLSI Which is Provably Good," *Proc. 14th Annual ACM Symposium on the Theory of Computing*, San Francisco, Ca., May 1982.

26. Leiserson C. E., "Area-Efficient Graph Layouts (for VLSI)," *Proc. 21st Annual Symposium on Foundations of Computer Science*, Syracuse, N.Y., 1980.

27. Leiserson, C. E. and H. T. Kung, "Algorithms for VLSI Processor Arrays," Section 8.3 of *Introduction to VLSI Systems*, C. Mead and L. Conway, Addison-Wesley Publishing Co., Menlo Park, Ca., 1980.

28. Lipton, R. J. and R. Sedgewick, "Lower Bounds for VLSI," *Proc. 13th Annual ACM Symposium on the Theory of Computing*, May 1981, pp 300-307.

29. Lipton, R.J., J. Valdes, R. Sedgewick, "Programming Aspects of VLSI," *Proc. 9th Annual ACM Symposium on Principles of Programming Languages*, Albuquerque, N.M., Jan. 1982.

30. Lyon, R. F., "A Bit-Serial VLSI Architecture Methodology for Signal Processing," in *VLSI 81: Very Large Scale Integration*, John P. Gray (ed.), Academic Press, London, 1981. (*Proceedings of the First International Conference on Very Large Scale Integration*, University of Edinburgh, August 18-21, 1981.)

31. McCanny, J. V., J.G. McWhirter, J. B. G. Roberts, D. J. Day, T. L. Thorp, "Bit Level Systolic Arrays," *Proc. 15th Asilomar Conf. on Circuits, Systems, and Computers*, Nov. 1981.

32. Mead, C. and I. Conway, *Introduction to VLSI Systems*, Addison-Wesley, Menlo Park, Ca., 1980.

33. Myers, D. J., "Multipliers for LSI and VLSI Signal Processing Applications," M. Sc. Project report MSP5, University of Edinburgh, U.K., Sept., 1981.

34. Oppenheim, A. V., and R. W. Schafer, *Digital Signal Processing*, Prentice-Hall, Englewood Cliffs, N. J., 1975.

35. Preparata, F. P. and J. E. Vuillemin, "Area-Time optimal VLSI Networks Based on the Cube Connected Cycles," Rapport INRIA #13, Rocquencourt, France, 1980.

36. Priester, R. W., H. J. Whithouse, K. Bromley, J. B. Clary, "Signal Processing with Systolic Arrays," *Proc. 1981 IEEE Int. Conf. on Parallel Processing*, 1981, pp. 207-215.

37. Rabiner, L. R and B. Gold, *Theory and application of digital signal processing*, Prentice-Hall, Inc., Englewood Cliffs, N.J., 1975.

38. Sastry, S. and S. Klein, "PLATES: A Metric-Free VLSI Layout Language," *Proc. MIT Conf. on Advanced Research in VLSI*, Cambridge, Mass., 1982.

39. Savage, J. E., "Area-Time Tradeoffs for Matrix Multiplication and Related Problems in VLSI Models," *J. Computer and Systems Science*, April 1981.

40. Sung, C. H., "Testable Sequential Cellular Arrays," *IEEE Trans. on Computers*, Vol. C-25, Jan. 1976, pp. 11-18.

41. Thompson, C. D., "Area-Time Complexity for VLSI," *Proc. 11th Annual ACM Symposium on the Theory of Computing*, April 1979, pp. 81-88.

42. Vergis, A. and K. Steiglitz, "Testability Conditions for Bilateral Arrays of Combinational Cells," 1983 IEEE International Conference on Computer Design: VLSI in Computers, New York, Oct. 31 - Nov. 3, 1983.

43. Vuillemin, J., "A Combinatorial Limit to the Computing Power of VLSI Circuits," *Proc. 21st Annual Symposium on the Foundations of Computer Science*, 1980, pp. 294-300.

44. Vuillemin, J., "A Very Fast Multiplication Algorithm for VLSI Implementation," *Integration*, Vol. 1, 1983, pp. 39-52.

45. Weiser, U. and A. L. Davis, "Mathematical Representation for VLSI Arrays," Technical Report UUCS-80-111, Dept. of Computer Science, University of Utah, Salt Lake City, Utah, Sept. 1980.

46. Weiser, U. and A. L. Davis, "A Wavefront Notation for VLSI Array Design," in *VLSI Systems and Computations*, H. T. Kung, Bob Sproull, and Guy Steele (eds.),Computer Science Press, Rockville, Md., 1981.

47. Williams, F. A., "An Expandable Single-IC Digital Filter/Correlator," *Proc. 1982 IEEE Int. Conf. on Acoustics, Speech, and Signal Processing*, Paris, May 1982, pp. 1077-80.

ESP: AN ARCHITECTURE FOR A MASSIVE MEMORY MACHINE

Hector Garcia-Molina
Richard J. Lipton
Jacobo Valdes

Department of Electrical Engineering and Computer Science
Princeton University
Princeton, N.J. 08544

ABSTRACT

This paper argues the case for a computer with massive amounts of primary storage, on the order of billions of bytes. We argue that such a machine, even with a relatively slow processor, can outperform all other supercomputers on memory bound computations. This machine would be simple to program. In addition, it could lead to new and highly efficient programs which traded the available space for running time. We present a novel architecture for such a machine, and show how it can lead to reduced memory access times.

Note: An extended version of this paper has been submitted to the IEEE Transactions on Computers.

October 28, 1983

ESP: AN ARCHITECTURE FOR A MASSIVE MEMORY MACHINE

Hector Garcia-Molina
Richard J. Lipton
Jacobo Valdes

Department of Electrical Engineering and Computer Science
Princeton University
Princeton, N.J. 08544

1. INTRODUCTION.

This paper argues the case for a computer with a primary memory substantially larger than what is currently (or will be in the near future) available on a single machine. We do not have a specific target size for such a *massive memory machine* (MMM), but for arguments sake let us say we want a few *billion* bytes of main physical memory. This size is certainly larger than what any manufacturer offers today, or will probably offer in the near future. Our thesis is that such a MMM is justified, even today, by the importance of certain applications in which *memory bound computations* occur naturally. For these computations, a classic Von Neumann machine with a relatively slow processor and massive amounts of physical memory, would vastly outperform even the "supercomputers" currently being researched and would be, in addition, far easier to program.

In Section 2 we present the case for a MMM, including its economic feasibility. In Section 3 we then discuss how an efficient MMM could be built.

2. THE CASE FOR A MMM.

Research efforts in the supercomputer field have tended to concentrate at the computational intensive end of the spectrum, disregarding the memory intensive applications altogether. The typical supercomputer being investigated today is a multiprocessor having up to one million processors, capable of executing up to billions of operations per second and yet have as "little" as sixty four megabytes of physical memory [Comp80, Comp81, Comp82, Evan82].

There are many applications for which such a machine (as well as any conventional machine) would be limited by its disk to memory transfer rates. For example, consider a program which accesses a four gigabyte (4×10^9 bytes) data structure with an essentially random pattern. A machine with one hundred or less megabytes of memory can be expected to generate a page fault in just about *every* memory access, rendering its potential processing power meaningless as a measure of its performance.

More precisely let us compare such a supercomputer with one hundred megabytes of memory and a MMM with four gigabytes of memory. Further, let us assume that the supercomputer is "infinitely fast" while the MMM runs only at one MIPS (Million Instructions per Second). Of course the supercomputer will vastly out perform the MMM on compute bound tasks. However, for the memory bound program we are discussing, assume that the supercomputer creates a page fault every f instructions , and that its disks are capable of servicing 100 instructions a second. Then on this task the MMM still computes at its one MIPS rate while the supercomputer is reduced to computing at about $100f$ instructions a second. Clearly if f is small enough the MMM will be faster than the supercomputer: if f is about 100 then the speedup is 100:1! While not all tasks will cause the supercomputer to "thrash" in this way, we believe that there are a large collection of important tasks that will cause such behavior.

2.1 Applications.

An MMM will produce significant improvements for any task which references, in a relatively random fashion, a large address space. Here we will review three areas in which such tasks abound, but this list is by no means exhaustive.

(a) **Databases** [Date81, Wied77]. It is well known that many database applications are IO bound, that is, limited by the speed at which data can be transferred from disks. Clearly, if the entire database (or a substantial fraction) could reside in main memory, then the IO bottleneck would be eliminated.

Not only will existing queries be answered faster, but it will now be possible to pose new interesting queries that previously required unreasonable times to answer. Thus, users can get more useful information out of the system.

(Reliability may be a problem in a massive memory database. We will return to this issue later.)

(b) **VLSI Design** [Mead80]. The size of VLSI circuits being designed is growing at a fast rate. Today there are circuits with a half million transistors, and predictions of integrated circuits with as many as one hundred million transistors by the mid 90's. VLSI design tools will perforce deal with massive amounts of data, notwithstanding much cleverness in the use of hierarchical design and the encoding of information.

Many of the VLSI design algorithms have good asymptotic running times, but have very poor locality of reference. Thus, they are naturally candidates for an MMM. For example, a layout system we have designed [Lipt82] uses topological sorting for placing objects. The algorithm for sorting requires linear time, but unfortunately also requires linear space and has almost no locality. Thus, beyond a certain layout size, its actual running time is determined by the memory available: at a given point, increasing the layout size by 30% sends our computer into uncontrolled thrashing and increases the running time ten fold!

(c) **Artificial Intelligence** [Nils80, Wins77]. The concept of vast data structures built mainly by the use of pointers, and hence lacking any locality of reference when accessed, brings the words *"Lisp"* and artificial intelligence (AI) to mind immediately. Garbage collection [Cohe81] and paging times contribute substantial fractions to the total running times of many AI programs. It seems fair to say that a good fraction of AI research involves memory bound computations.

Certain AI programs, such as DENDRAL [Buch78] or MACSYMA [Mart71], have succinct inputs and generally produce succinct outputs, and yet may build enormous intermediate data structures. These programs are even better suited to a MMM than others. They would not even need to incur in the overhead of loading the massive memory as a data base or VLSI program

would.

2.2 The economical feasibility of a MMM

Clearly VLSI has made computing in general cheaper. It is also clear, although not as well understood by everybody, that VLSI has made certain kinds of computing cheaper than others. One example of this differential impact involves memory and processing power: over the past few years, the price of logic circuits has decreased about 20% per year; during that same span, memory prices have decreased at twice that rate: almost 40% per year. Clearly that trend, if continued, should be very good news indeed for applications that require memory bound computations.

In fact, there are good reasons to believe that the figures given in the previous paragraph represent more than a local kink in the prices of these commodities, brought about by a vicious fight for market share in a particularly important market. Memories are the most regular integrated circuits (ICs), and thus among those which would profit immediately from higher fabrication densities. We believe that memories will be always the first circuits to profit from progress in integrated circuit manufacturing technology.

At today's prices, the cost of the ICs necessary to build a one gigabyte memory is below one million dollars. This is not out of proportion with the investment necessary to equip a state of the art installations for research or production work in some of the areas identified earlier. Furthermore, if the price trends hold, the ICs necessary to build a four gigabyte memory would cost approximately 200,000 dollars by the end of the present decade.

2.3 New Programming Techniques.

A MMM is straightforward to program. Existing programs can be run on it, and if they are memory intensive, they will run very fast. However, the impact of a MMM may be even more far reaching. A MMM may alter the way we program and this in turn may yield even grater improvements [Gray83, Wein83].

For example, consider the concurrency control mechanism of a database system. Since user programs (called transactions) encounter long delays as they wait for disk pages to be brought into main memory, the database system executes several transactions concurrently. Since the transactions are not independent (they are reading and writing the same database), their actions cannot be interleaved in arbitrary ways. The concurrency control mechanism (typically using locking) ensures that only interleavings that preserve data consistency are run. Very roughly, about 10% of the CPU instructions are spent doing concurrency control.

When the database system is transferred to a MMM, the disk delays disappear, and concurrency control may no longer be needed. The data required by each transaction is already in memory, so if transactions are short (as they are in many commercial systems) they can simply be scheduled sequentially. So in addition to making data available faster, a MMM may eliminate the overhead of concurrency control.

In general, having massive amounts of memory will change our programming techniques. Data structures for secondary storage (e.g., B-trees, extendible hashing) will become obsolete. Table lookup will be practical in many more cases. For instance, instead of computing trigonometric functions with a series, we may want to have a large table of values and use simple interpolation. Digital searching [Knut73], which improves search times at the expense of memory space, will be commonplace.

3. THE ESP ARCHITECTURE.

We have argued that main memory is a useful resource in many applications, and that a computer with massive amounts of memory (e.g., gigabytes) is economically feasible.

But are there any technological challenges in building a MMM? Is it not just a matter of connecting all the desired memory to the chosen processor in a conventional way, i.e., with a very long bus? (See Figure 1.)

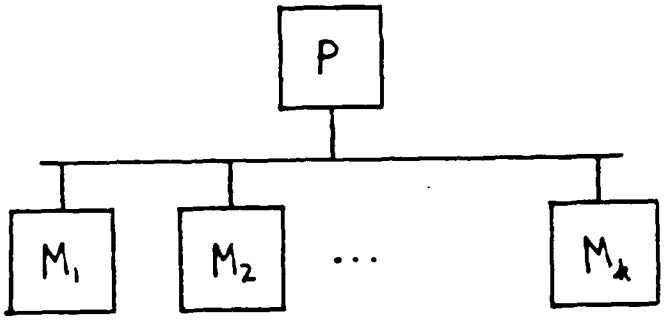


Fig. 1: *A Conventional Architecture MMM*

A conventional architecture is a reasonable one, but as we will discuss shortly there are other architectures that may be superior. The conventional architecture has two main weaknesses: memory access times and reliability.

- **Memory access times.** Given current IC densities, a four gigabyte memory requires about one thousand devices (memory cards) on a single bus. Even with clever arrangements and higher densities, hundreds of devices per bus seem unavoidable. Building a special purpose bus to support that many devices is feasible, although not trivial. However, regardless of how the bus is implemented, as the size of the memory grows, the access times grow because of the physical distances and/or capacitance effects. At the same time, memories are becoming faster, so that the larger access times make us lose part of the advantage of having a massive memory.
- **Reliability.** As the size of the memory grows, the probability that one of its components fails also grows. A conventional architecture has no provision for graceful degradation, and hence the entire machine would be unavailable with unacceptably high probability. For database applications, some type of memory redundancy is also necessary in order to avoid loss of data.

In the rest of this section we present a new architecture which directly addresses the first of these weaknesses. The reliability issues are briefly discussed

in the conclusions section, and in a separate, more detailed report [Garc83b].

3.1. A Novel Architecture.

Our basic premise is that the time to access memory over a long bus (i.e., one that drives hundreds of devices) is substantially larger than the access time over a short bus (i.e., one driving a single memory board). The meaning of "substantially" depends on how the buses are implemented, but for the time being let us assume that access times over a long bus are at least an order of magnitude larger than over a short bus.

A classical solution for improving access times over a long bus is to add a memory *cache* [Kapl73, Smit82] to the processor. (See Figure 2.) The idea is that commonly accessed data reside in the cache, and are hence available with smaller delays (both because the cache bus is shorter and because the cache memory is generally faster). Unfortunately, caching does not improve access times significantly for the programs we have in mind. A cache may be useful for holding some commonly accessed values, but as discussed in Section 2, we are concerned with programs that reference their data structures in essentially random ways. Thus, for most of the recently referenced data, the probability of being accessed next is low.

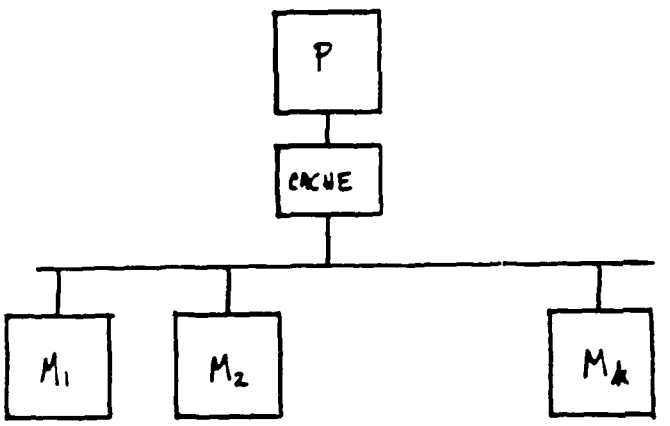


Fig. 2: *A MMM with a Cache*

If we cannot bring the data to the processor as fast as we would like, we could instead "take the processor to the data". This is precisely what the *ESP MMM* does. A schematic description of it is shown in Figure 3. (The name ESP will be explained shortly.)

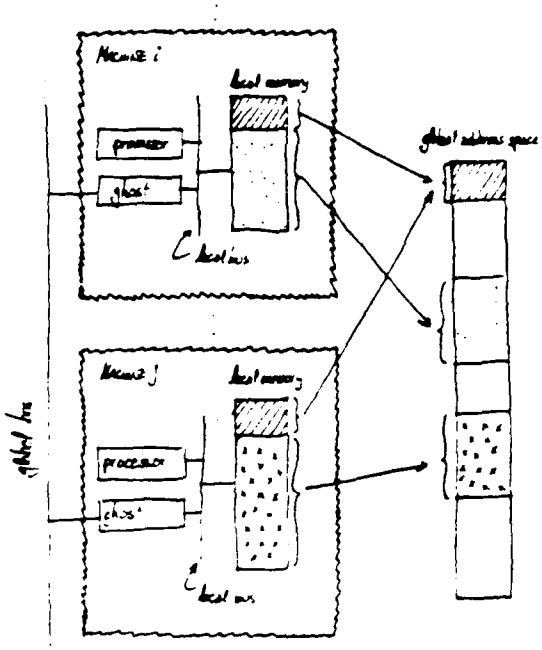


Fig. 3: *The ESP MMM*

The ESP MMM consists of a collection of standard Von-Neumann machines, interconnected by a system-wide (or global) bus that permits the broadcast of values from one machine to all the others. Each individual machine has its own processor and local memory connected via a local (short) bus. The gateway of each machine to the global bus is an *ESP* device connected both to the system bus and the local bus. (The number of machines is not critical to the architecture, but we expect a system with a few gigabytes to have a relatively small number of machines, possibly up to one hundred. This means that each individual machine has a substantial amount of memory.)

The individual processors share the same address space. This address space is distributed among the local address spaces as follows (see Figure 3). A small fraction of the global address space is replicated in each local address space; the remainder of the system address space is covered in a non-overlapping manner by the local address spaces. An *ESP* device connected to each local bus is responsible for servicing requests that involve non-local addresses.

Even though the ESP MMM has multiple processors, it is a single instruction stream, single data stream machine (SISD) [Flyn72]. All processors execute the *same* program, which is loaded into the replicated portion of the system address space. As long as that program references locations in the shared subspace all processors will execute in lockstep and no communication through the system bus will take place. References outside the shared address space are broadcast and received on the global bus, as is illustrated by the following example.

Consider a program which references memory words w_1 through w_9. Assume that w_5, w_6, w_7 are in machine 2, and the rest of the words in machine 3. Figure 4 shows the time at which each processor receives a referenced word. In this figure we assume that fetching a word from local memory takes one time unit, and that broadcasting a word over the system bus takes two units. (We choose two units only to simplify the example. As discussed earlier, we expect the system delays to be orders of magnitude larger than the local ones.)

At time 0, all processors start; since they all run the same program, they all request word w_1. Processor 3 has w_1 locally, so one time unit later it receives it. From then on, processor 3 works at full speed, accessing words w_2, w_3, and w_4. At time 4, processor 3 requests word w_5, but since it is not local, a delay ensues.

In the meantime, the ESP at machine 3 has been broadcasting words w_1 through w_4. Word w_1 arrives at processors 1 and 2 at time 3, and the following words arrive at one unit intervals. Note that the words are "pipelined" on the bus, so that there is only *one* system bus end-to-end delay involved. Hence, after the initial delay, processors 1 and 2 start receiving and processing the words at full speed.

During this time we say that processor 3 "has the lead", i.e., is ahead of the others. But when processor 2 references w_5, it finds this word in its local memory

and takes the lead. The other processors must now wait until the ESP at machine 2 broadcasts w_5 and the following words. In a similar fashion, the lead changes back to processor 3 when w_8 is referenced.

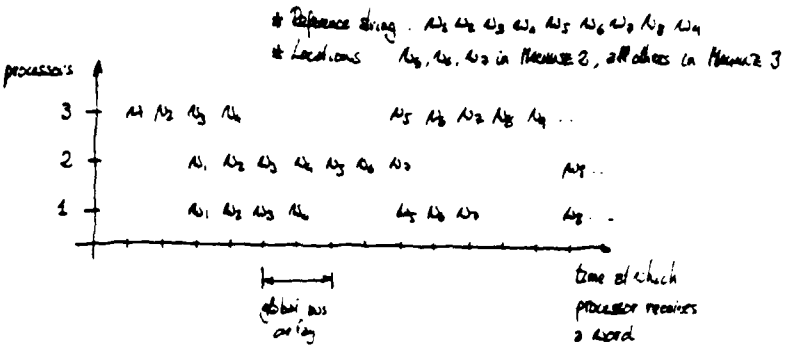


Fig. 4: *Execution in an ESP MMM*

In summary, an ESP examines each word request made by its local processor. If the address refers to the shared subspace, the ESP does nothing. If it refers to the local non-replicated memory, then the ESP reads the fetched word off the local bus and broadcasts it over the system bus. In case of a reference to remote memory, the ESP waits for the next word broadcast over the system bus, and then places it on the local bus. (This is why we picked the name "ESP" for these controllers: the remote words required appear on the system bus without having been requested, as if the controllers has ExtraSensory Perception.) In any case, the processor is not aware of the ESP controller (except for time delays); it operates as if it had a long bus linking it to all the memory units. Each local memory module must know the addresses of the data it holds, honor requests for its data, and ignore all other requests. (This is how memory modules in a conventional architecture operate.)

While the common program generates requests for data local to machine m, the processor at m takes the lead. All other processes continue execution at the same rate as m, with their ESPs supplying the data they need. These "trailing" processors, will be behind the leader by an amount of time equal to the one-way

delay time between ESPs through the system bus. When a reference to an address local to another machine occurs, that machine takes the lead.

Writes to memory can be ignored by the ESPs. When the program calls for storing into the replicated address space, all processors will execute the instruction and will update their copies. When the program modifies non-replicated storage, the processor with the data will modify it, and the rest need do nothing. (When we discuss reliability in Section 4, we will see that special precautions must be taken when writing into the non-replicated address space.)

The replicated address space is used to store the program and commonly accessed values. In addition, each processor may have registers and a cache to hold recently accessed data.

Two important things to note about the system bus are that it acts as the system "clock" and that there is no contention. The data transmitted over the bus are the timing signals that keep all processors in synchrony. (In the example of figure 4, processor 2 picks up the lead when it receives word w_4 from processor 3.) Since non-replicated data is found only at a single machine, only one ESP will ever broadcast at a time. This means that the bus protocols will be very simple, and hence transmissions can be fast.

The ESP architecture has the following advantages over a conventional one:

(1) The local machines have conventional architectures. They may be used independently when the MMM is not needed.

(2) For fully random references, memory access times are cut by roughly a *factor of two*. In a conventional machine, the address must be transmitted on the system bus and the referenced datum must be transmitted back. In an ESP machine, no addresses have to be transmitted on the global bus: each datum appears on the system bus without having been requested. That is, since references are random, each memory access will cause a lead change. But these lead changes only involve a one-way broadcast, and thus, half the delay encountered in a conventional architecture.

(3) The ESP MMM will reward "locality of reference" by minimizing "lead changes" in programs that exhibit it. That is, if two or more references fall within the same memory module, then the access times are reduced to local bus times. The fewer the lead changes, the faster the ESP MMM will execute.

Locality in this context, however, has a wider meaning than in a conventional memory cache or virtual storage system. Here, locality of reference means that two references are local to the lead machine, and this machine may have a substantial chunk of memory (probably tens of megabytes). In the next sub-section we will explore these issue in more detail.

What is the price we pay for these advantages? Obviously, we have replicated processors and some data. Given current pricing trends, the cost of this extra hardware should be reasonable, at least compared to the cost of the massive memory. What we have *not* sacrificed is simplicity and ease of programming. The processors and memory modules are conventional. The ESP architecture is transparent to the user program. The task of distributing the global address space to the spaces of the individual machines can be relegated to a sophisticated loader.

3.2. Program Locality.

The potential performance improvements of an ESP MMM over one with a conventional architecture hinge on two main factors:

(i) The "locality" exhibited by the program, and

(ii) The memory access times over the system and local busses.

In this sub-section we study the first factor in more detail. The bus times are discussed in the following sub-section.

The ESP MMM utilizes several mechanisms to improve memory access times: (1) registers and caches at each processor to hold recently accessed values; (2) a replicated address space to hold the program and commonly accessed values; and (3) the ESP mechanism, which lets the leading or controlling processor move to the memory module where the data resides. The first two mechanisms can be

easily incorporated to a conventional MMM, so the decisive factor is clearly the ESP mechanism.

What does the ESP mechanism give us that the others do not? In order to answer this question, let us postulate a simple *data* reference pattern. (We are not interested in the *instruction* reference pattern, since the entire program is replicated in all machines.)

Suppose that the M memory words of the MMM are divided into *blocks* of B words each. A block is the unit of data transfer between the memory and a cache. We assume that the location of the next referenced block depends only on the location of the most recently accessed one. Specifically, Figure 5 gives the probability distribution of the next reference. There is a set of a blocks, centered on the last referenced block, that have a high probability p of being accessed next. All other blocks have a much lower probability q. (For simplicity, we assume that when the last reference is within $a/2$ blocks of the ends of the memory, the distribution wraps around.) We assume that a is odd.

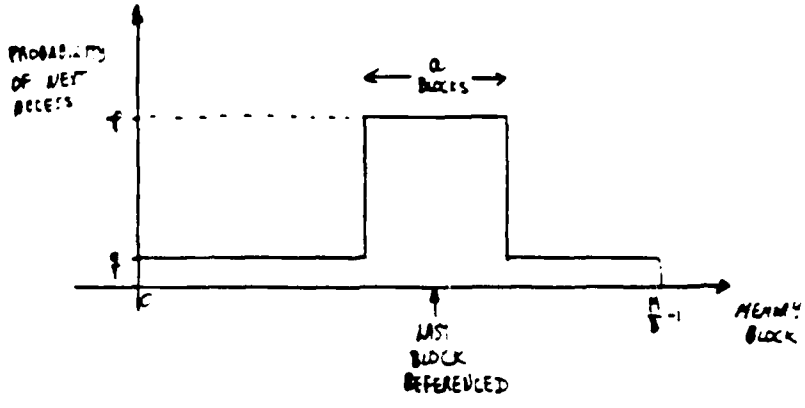


Fig. 5: *The Probability Distribution.*

Our experience tells us that this is, in an idealized way, the way programs reference their data (e.g., see [Siss68, Smit82]). For example, consider a program that simulates a VLSI chip. When a transistor is referenced, several contiguous words may be referenced. The next transistor reference is likely to be to a connected one, and if the circuit is represented in a reasonable way, it will be close

to the previous one. Here "close" may mean within a few thousand bytes, so our high probability window, a, may be relatively large.

The parameters a and p define the *locality* of the program. As a shrinks and/or p grows, the program exhibits more locality, and as a grows and/or p approaches q, the references become more random (i.e., the distribution becomes flatter).

Note that this distribution ignores other types of data locality that may also be exhibited by programs. For instance, programs may have time locality (i.e., tend to reference recently accessed data) or may access certain fixed locations with high probability. Since these types of localities are exploited by data caches, the distribution we have selected to study will highlight the strengths of the ESP mechanism, not of caches. This is precisely what we want to do.

Using this probability distribution, we have analyzed the performance of an ESP mechanism (where processors have no registers or caches) and of a simple cache. The analysis is described in [Garc83]. Figure 6 presents some typical results. The figure shows the hit ratio for the cache (h_c) and the ESP mechanism (h_e), as a function of a, the high probability window. For the cache, the hit ratio is the probability that the next referenced word is in the cache. For the ESP, it is the probability that the next word falls in the same machine as the previous word. (In the figure, locality decreases from left to right.)

If on each memory reference the cache can fetch a significant portion of the "high probability of next access" window, then the cache performs very well. (That is, if a is close to 1 block.) In this case, either the program has very high locality or the system bus feeding the cache is very wide. In this case the ESP does not have any advantages over the cache.

At the other extreme (very large a), references are fully random and both mechanisms have a hit ratio of 0. In this range, the ESP is superior by roughly a factor of two because, as we discussed earlier, addresses need not be broadcast.

In between is a large range of localities where the ESP performs substantially better than the cache (from a equal to 4 or 5 until a is roughly the number of blocks in a memory module of the ESP.) In this area, most references using the ESP mechanism are local. On the other hand, with a cache, most references

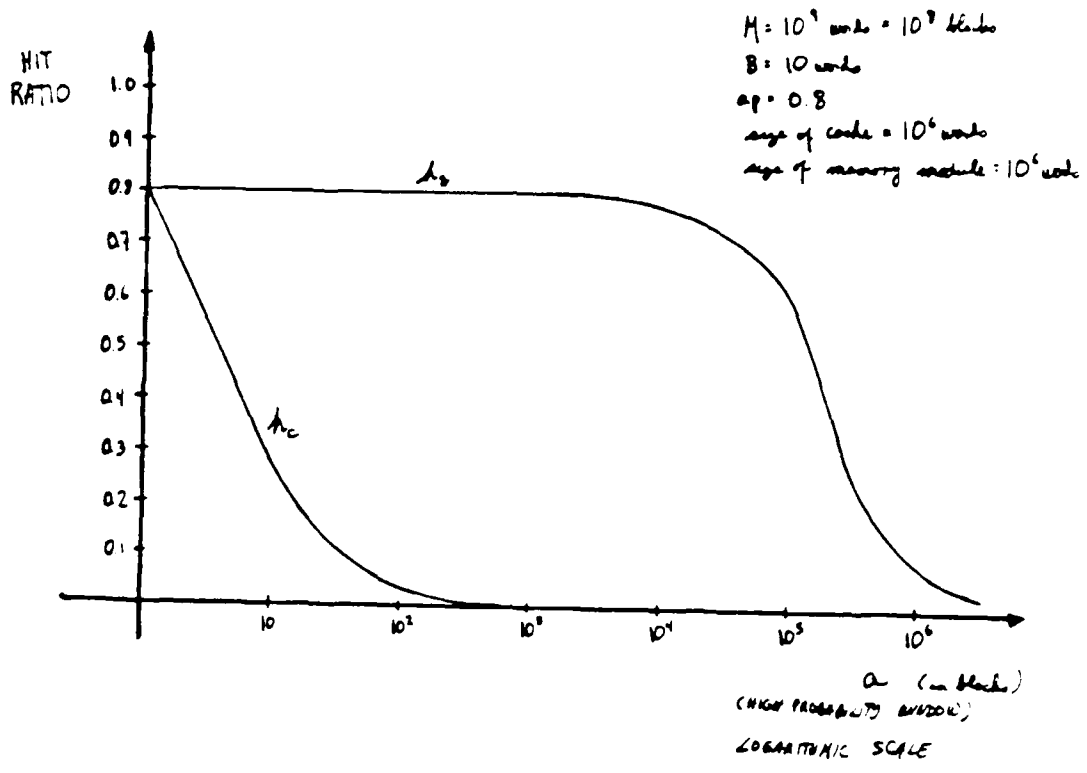


Fig. 6: *Hit Ratios for ESP and Cache*

continue to rely on the system bus. This is because the cache mechanism retrieves data from memory in very small units, on the order of a few words. The improvement will be, roughly, the ratio of system bus access times to local bus times.

The programs that will use a MMM, as we argued in Section 1, are memory intensive ones, programs that cause a virtual memory system to thrash. Thus we expect these programs to operate in the range of localities where the ESP mechanism does pay off.

([Garc83] presents more results, and also considers other probability distributions. The trends obtained are similar to what we have presented here.)

3.3. System and Local Bus Access Times.

The performance improvements of an ESP MMM over a conventional architecture depend on the value of the system bus access time, D, and the local bus time, d. If we can implement a system bus with D small compared to the cycle time of the processor(s), then cutting this time by a factor of two or more may not be important. Similarly, if d is not significantly lower than D (as we have assumed so far), then the gains of the ESP mechanism will be limited.

The values of d and D depend on the hardware used to implement the MMM, as well as on the size of the memory. Thus, it is difficult to reach any definitive conclusions. However, we can discuss two implementation scenarios where certainly D is significant as compared to the cycle time, and where d is orders of magnitude less than D. In both of these cases, the ESP MMM performs very well.

- **Processor and Memory on a Chip**. It will soon be possible to build a reasonable processor with a few megabytes of memory, all on a single VLSI chip. These chips will be ideally suited for the construction of an ESP MMM. The time to access on-chip memory (d) will be very small, since small currents and small distances are involved.

  The limiting factor in this implementation will be the rate at which ESPs can broadcast data out of the chip, into the system bus. However, an optical bus may provide the necessary throughput.

- **Sharing Memory on Existing Computers.** Suppose that we already have an installation with several computers (maybe 2 or 3, maybe 100 or 200) connected via a local area network. The ESP architecture gives us a way to combine these resources into a single MMM, when it is needed. Clearly, local memory access times are significantly less than transmission times over the network, so the ESP is a useful idea. Each existing machine would be provided with an ESP controller, and the network protocols (for MMM operation) would be simplified, e.g., there is no contention, no need for packet headers. (This assumes that while the machines operate as a MMM, the network has no other users.) A program requiring more memory than is available at a single machine (even if it only needs the memory of 3

or 4 other machines) can be sped up considerably. There will be improvements even if its references are totally random, since page faults (with seek, rotational, and substantial data transfer delays) will be replaced by fast (and probably short) network messages.

For some programs it may be possible to implement the ESP mechanism fully in software. If a program has a distribution similar to the one of the previous sub-section, and if a is less than the memory at each computer, then lead changes will be infrequent. A lead change can then be implemented by sending a message with the state (e.g., contents of registers) of the lead machine to the next leader.

4. CONCLUDING REMARKS.

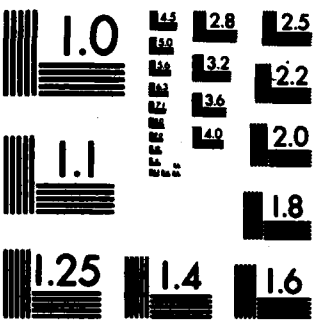
If we look at the ratio of memory size to processor speed of past and present commercial computers, we find that most are within an order of magnitude of one megabyte per MIPS. (The value one megabyte per MIPS is called "Amdahl's constant".) The supercomputers being developed all have ratios well below this value, and are targeted for computationally intensive problems. The machine we proposed here, on the other hand, would have a memory to speed ratio of 100, 1000 or more. We have argued that such a machine would speed up memory bound programs like no other computer could. We also asserted that a massive memory machine having unconventional architecture and features would be more efficient. Yet, in spite of its novel structure, this machine would be simple to program.

We have only sketched the main features of a massive memory machine and the ESP architecture, but of course, there are many other important issues that must be resolved before such a machine can become a reality.

One of these issues is reliability. Fortunately, the ESP architecture appears to be well suited for failure tolera... The state of a computation (e.g., registers and program counter) is replicated at all processors, so if one of them fails, its state can be reconstructed. The processors are connected through a simple linear bus, so it is not difficult to have spare units that can take over when others fail. A major portion of the main memory is not replicated, so if it fails data will be

AD-A136 553 PRINCETON VLSI PROJECT(U) PRINCETON UNIV NJ DEPT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE R J LIPTON 1983 N00014-82-K-0549 2/2

UNCLASSIFIED F/G 9/5 NL



lost. Thus, if this data is important, a secondary copy must be kept. Mechanisms similar to those used in database systems (e.g., logging) can be used to keep the secondary copy up to date. These strategies and mechanisms are discussed *in*[Garc83b].

A second issue is the utilization of the multiple processors in the ESP architecture. Given that they exist, they can also be used for parallel processing. For example, in a database application, the MMM could be divided up for executing a parallel search, then reconstituted. Of course, the programs for the multiple processors will not be as simple as the MMM ones.

Acknowledgments. Several useful ideas and suggestions were made by Jim Gray, Peter Honneyman, Steve North, Peter Weinberger, and Gio Wiederhold.

REFERENCES.

[Buch78] B. G. Buchanan and E. A. Feigenbaum, "Dendral and Meta-Dendral: Their Applications Dimension", *Artificial Intelligence*, Vol. 11, Num. 1-2, 1978, pp. 5-24.

[Cohe81] J. Cohen, "Garbage Collection of Linked Data Structures", *ACM Computing Surveys*, Vol. 13, Num. 3, September 1981, pp. 341-367.

[Comp80] Special Issue on Supersystems for the 80's, *IEEE Computer*, November 1980.

[Comp81] Special Issue on Arrav Processor Architecture, *IEEE Computer*, September 1981.

[Comp82] Special Issue on Highly Parallel Computing, *IEEE Computer*, January 1982.

[Date81] C. J. Date, *An Introduction to Database Systems*, Addison-Wesley, 1981.

[Evan82] D. J. Evans (Editor), *Parallel Processing Systems*, Cambridge University Press, 1982.

[Flyn72] M. J. Flynn, "Some Computer Organizations and Their Effectiveness", *IEEE Transactions on Computers*, September 1972, pp. 948-960.

[Garc83] H. Garcia-Molina, R. J. Lipton, and J. Valdes, "Analysis of the Massive Memory Architectures", Technical Report 313, Department of Electrical Engineering and Computer Science, Princeton University, May 1983.

[Garc83b] H. Garcia-Molina, R. J. Lipton, and J. Valdes, "A Massive Memory Machine", Technical Report 315, Department of Electrical Engineering and Computer Science, Princeton University, July 1983.

[Gray83] J. Gray, "What Difficulties Are Left in Implementing Database Systems", Invited Talk at *SIGMOD Conference*, San Jose, CA., May 1983.

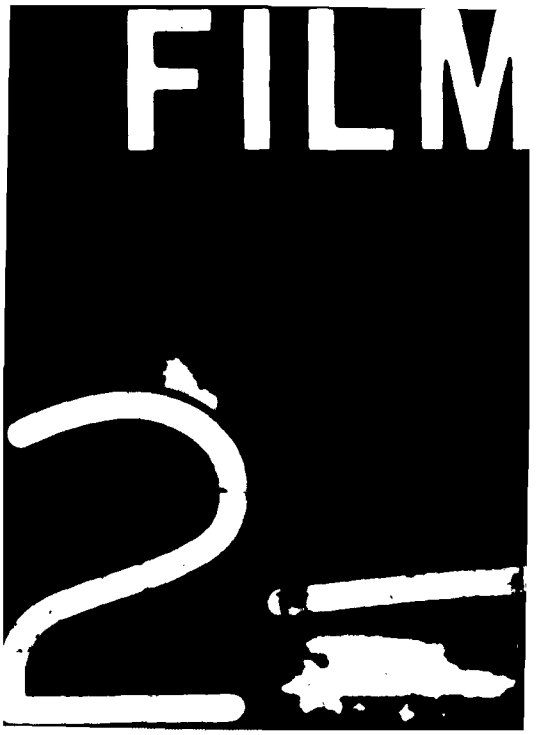
[Kapl73] K. R. Kaplan, R. O. Winder, "Cache-based Computer System," *IEEE Computer*, March, 1973, pp.30-36.

[Knut73] D. E. Knuth, *The Art of Computer Programming; Volume 3: Sorting and Searching*, Addison-Wesley, 1973.

[Lipt82] R. J. Lipton, S. C. North, R. Sedgewick, J. Valdes, and G. Vijayan, "ALI: A Procedural Language to Describe VLSI Layouts", *Proc. Nineteenth ACM-IEEE Design Automation Conference, Las Vegas, Nevada, June 1982, pp. 467-474.*

[Mart71] W. A. Martin and R. J. Fateman, "The MACSYMA System", *Proc. ACM Second Symposium on Symbolic and Algebraic Manipulation*, Los Angeles, CA., 1971, pp. 23-25.

[Mead80] C. Mead and L. Conway, *Introduction to VLSI Systems*, Addison-Wesley, 1980.


FILM