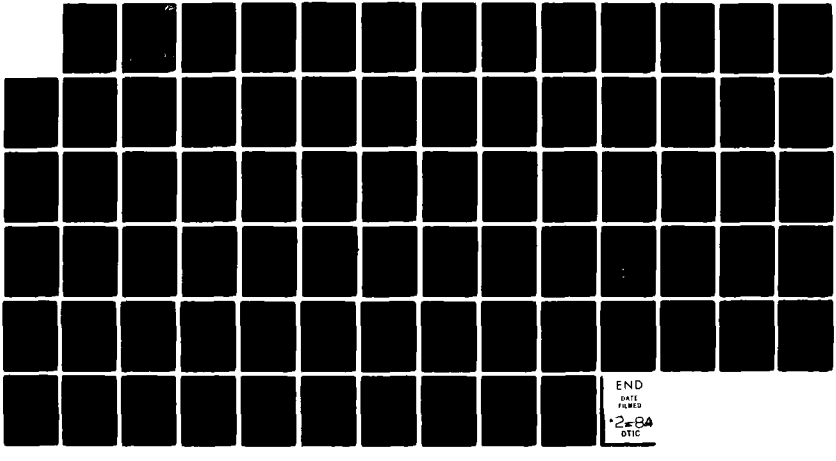
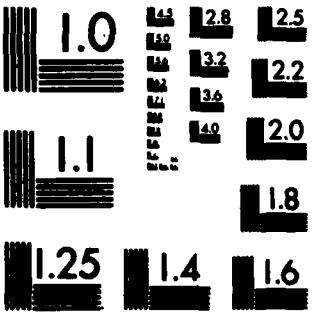
AD-A136 552 PRINCETON VLSI PROJECT(U) PRINCETON UNIV NJ DEPT OF 1/1
ELECTRICAL ENGINEERING AND COMPUTER SCIENCE R J LIPTON
1983 N00014-82-K-0549

UNCLASSIFIED F/G 9/5 NL





Princeton VLSI Project: Semi-Annual Report

Period Ending: April 15, 1983

R. J. Lipton, Principal Investigator

EECS Department
Princeton University

Faculty

B. W. Arden
D. Dobkin
H. Garcia-Molina
A. LaPaugh
K. Steiglitz
J. Valdes

Contract N00014-82-K-0549

Princeton VLSI Project: Semi-Annual Report

R. J. Lipton

1. Introduction

There are three major aspects to our project. The first concerns the development of ALI2 which is a procedural language 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. ALI

2.1. ALI2 [Kalin, Valdes, Vijayan]

An almost complete version of ALI2 became operational at the beginning of March. This version implements most of the language features described in the accompanying *"Language Overview"* and is currently being used to layout a number of circuits.

The language has now a small group of local users which includes people not involved directly on the language design and implementation. The first circuits designed with ALI2 will probably be sent to fabrication before the end of April.

The number of users will soon increase substantially, when the students in our VLSI design course begin work in earnest on their course projects. Most of our current efforts are directed towards making the system stable enough for them as well as for release to users outside Princeton.

On the language itself, the only unimplemented features are those related to *completeness checking* and direct interface to simulators. The code to implement completeness checking will be relatively short and straightforward to implement. It has not been implemented yet because we judged more important to have a version of the language publicly available before the end of the semester. We have not yet selected the simulator to which ALI2 will be interfaced, but the interface should not present great problems. We expect to complete the implementation of these two features during the summer.

The initial experience of ALI2 users has been generally positive, in spite of the fact that they have carried most of the burden of testing and debugging of the language translator and run time support. We are confident that the current system will be completely stable very soon. The efficiency and convenience of use of the language have been those that we expected at the time we designed it.

Future work on ALI2 will primarily involve the completion of the current version as described earlier. A longer term effort would involve the implementation of a new version of the language that incorporates the experience gained using the current one. Such an effort is contingent on the level of use of ALI2 and the general reception from its users, and it is not likely to be initiated for another six months or so.

On a slightly different but related topic, we intend to research constraint based, low level, layout specifications as a possible intermediate description level between CIF and high level languages. Two of the main lesson we have extracted from the ALI2 project is the usefulness of such descriptions and the pressing need for well defined intermediate representation of that nature.

2.2. Graphics Engine [Dobkin, Valdes]

The design of the graphics engine has progressed to the design of our initial line drawing chips. New algorithms for line drawing have been developed which are more efficient than Bresenham's algorithm and which are especially suited for parallel implementation. We are currently in the process of implementing one such algorithm in ALI2.

Our progress to date has been the design of an Htree circuit to do 16 bit wide comparisons along with the controlling logic for the chip. We expect to have completed a similar Htree for addition within the next few weeks. At that point, an initial version of the line drawing circuit will be ready for fabrication. The structure of this circuit is very non VLSI-ish. Long data paths have been permitted in ordered to allow us to do a piecemeal design and adjust to the design environment. A new design is in process which will overcome these difficulties.

The new design is expected to have area 1/3 less than the current design and to have short paths which will probably allow a faster clock. Furthermore, the replication of data and computations in the new circuit will be minimal.

During the design phases so far, we have had an opportunity to evaluate the ALI tools and to serve as "friendly debuggers" of these tools. Our experiences have been positive. The procedural nature of the tools has allowed us to delay design decisions which we would have had to make earlier in a graphics oriented language. The cost in area has not been great. Our estimates are that a similar design in a graphics based language would have required 70 or 80% of the current area. Indeed, these numbers are borne out by comparisons with portions of the circuit which had previously been designed using ICARUS.

2.3. Clay [Lipton, Lucas, North]

In parallel with ALI2 another effort has started on a related layout language that is called **Clay**. This language has many of the same features as ALI2 but is implemented as a package in the programming language C. This approach has allowed us to get a version of Clay up very quickly. Both ALI2 and Clay share essentially the same constraint based view of the layout process. Clay however will allow us to experiment more easily with a variety of new ideas more quickly. Thus, Clay has a feature that allows a designer to link several different pieces of layout and still keep them all flexible. Currently a number of our PhD students are using Clay to design a very simple processor that we call *Prism*.

2.4. Local Algorithms for Large Layout Problems [A. Huang]

While the usual algorithms for topologically sorting large graphs (which arise in VLSI layout problems) use asymptotically linear time and space, in practical situations they perform badly because of page thrashing. This comes about because memory references are frequently made to information on pages outside of fast memory. We are studying the construction of algorithms for topological sort which have good behavior in this respect; preliminary results show that the number of page faults can be bounded by a constant (which depends however on the geometric constraints of the layout problems) times the number of pages. This should make practical more efficient ways of laying out very large integrated circuits.

3. Census

There are several projects underway in the area of census and related architecture.

3.1. Unbounded Fan-In Circuits [Lipton]

Work continues on our investigation of the power of such circuits. We have recently been able to further characterize the power of such circuits.

3.2. Maximising Throughput with Latching [P. Cappello, A. LaPaugh, K. Steiglitz]

In many applications of custom chips, especially for digital signal processing tasks such as convolution, filtering, and Fourier transformation, high throughput is very important, and a relatively large delay is tolerable. If a circuit has stages in which there is large delay because the combinational logic is many levels deep (an array multiplier is an example), the throughput can be increased by introducing intermediate levels of latching. This costs area and clocking delay. The tradeoff between extra latching circuitry and clocking on the one hand, and increased throughput on the other, was studied using analytical model. The results provide guidelines for

the choice of latching density, using the product of area and period as a performance measure. Significant increases in throughput are predicted for situations like a bit-parallel array multiplier.

3.3. Massive Memory Machines [Garcia-Molina, Lipton, Valdes]

The size of VLSI circuits being designed is growing at a fast rate, and there are predictions of circuits with as many as one hundred million transistors by the mid nineties. The tools to design these circuits will be limited, not by the speed of central processing units, but by the speed at which random accesses can be made to memory. The reason for this is that most tools contain algorithms with good asymptotic running times, but reference memory in unpredictable ways. Thus, a very fast "super-computer" which relies on a disk paging mechanism to access a large circuit will easily be outperformed by a slower machine which can hold all the necessary data in its main memory.

We are investigating the feasibility of such a *Massive Memory Machine*. There also appears to be a need for such a machine in other fields like databases and artificial intelligence, and the cost for a machine with one billion bytes of main memory seems comparable to that of current super-computers. The design of a gigabyte computer is not a simple task, and novel architectures for it are being pursued. Some preliminary results on this topic can be found in an attached report.

4. Testing [Arden, LaPaugh, Lipton]

This project concerns itself with all aspects of testing. Currently Arden is working with the Siemens group on *first silicon* testing at Munich.

4.1. Bipartite Testing

Work continues on refining our bipartite approach to circuit testing. This approach exploits very structured design for testability. It has three components: the use of combinational circuits which are bipartite, the use of a controllable logic gate (*nand* or *nor*), and the use of logic to observe internal values. It achieves 100% single stuck-at fault coverage and the detection of many multiple faults using under twenty test vectors. A simple transformation of an arbitrary circuit into a bipartite one exists. It may double the number of gates in a circuit, but will never do worse. LaPaugh recently presented this method to the IEEE workshop on VLSI circuit testing.

Initially, the approach was developed for nMOS circuits. We are now investigating the extension of this approach to CMOS and to designs which use steering logic as well as gates. We are also continuing development of a special latch which will allow a large class of sequential circuits to be tested by this method without the addition of scan paths. We are also beginning to work with Siemens on developing actual devices. These devices will be used to build circuits to test our method. Many implementation issues remain which will be explored through the design of these circuits.

4.2. Testing of Regular Arrays [Vergis]

We are looking at conditions that guarantee that one- and two-dimensional bilateral arrays of combinational cells are testable. New conditions have been derived which are more general than previously known. The work is being generalized towards combinatorial cells (with internal states). The question of having tests which take time linear in the number of cells is also being studied. The results should be useful in testing such regular structures as systolic arrays for digital filters and convolution.

5. Papers (See Attached)

Optimal Choice of Intermediate Latching to Maximize Throughput in VLSI Circuits †

Peter R. Cappello ††
Andrea LaPaugh, and Kenneth Steiglitz

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

ABSTRACT

In many computational tasks, especially in signal processing, it is the throughput that is important, rather than the latency, or delay. If a special-purpose VLSI chip is designed for a particular signal processing task, such as FIR filtering, for example, the maximum clock rate, and hence throughput, is determined by the depth of the combinational logic between registers and the time required for the distribution and operation of the clock. If the combinational logic is sufficiently deep (in bit-parallel circuits, for example), the throughput can be increased by inserting intermediate stages of clocked latches. This is at the expense of increased area and delay to operate and clock the intermediate registers. Roughly speaking, the strategy amounts to using more of the chip area to store information useful for pipelining.

This paper investigates the optimal tradeoff between the degree of intermediate latching and cost, using the measure AP, where A is the chip area and P is the period (the reciprocal of throughput). We derive expressions for the time and area before and after intermediate latching, using the Mead-Conway model, both for the cases of on-chip and off-chip clock drivers. The results show that significant reductions in AP-product (reciprocal of throughput-per-unit-area) can be achieved by intermediate latching in many typical signal processing applications, for a wide range of circuit parameters. The array multiplier is used as an example.

† 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. A preliminary version of this paper was presented at the 19[illegible] IEEE International Conference on Acoustics, Speech, and Signal Processing, Boston, MA, April 14-16, 1983

†† Peter R Cappello is now with the Department of Computer Science, University of California, Santa Barbara, CA 93106

1. Introduction

When certain tasks are implemented with special-purpose VLSI chips, it is often the *period P* (time between successive outputs) that is crucial, rather than the latency or delay *T*. This is especially true in signal processing, where typical tasks such as filtering and Discrete Fourier Transformation often have high volume requirements and relatively lax delay requirements. Recent work has described bit-serial and bit-parallel VLSI architectures that do in fact allow the period to be equal to the clock period (see, for example, [2,4-9,12]). In [5,7] a class of these circuits is called *completely pipelined*. In this paper we take up a different question, that of inserting intermediate stages of latching so as to maximize the rate at which the clock can run without a disproportionate blowup in area requirements. We will use the criterion of minimizing the *AP*-product, where *A* is the area of the VLSI circuit and *P* is the period. The *AP*-product can be thought of as the reciprocal of throughput-per-unit-area, and a completely-pipelined circuit optimal with respect to this criterion can be claimed to make best use of chip area. Leiserson and Saxe [14] treat the related problem of redistributing latches so as to decrease period, but they do not consider area or clocking penalties.

We assume that the circuits we discuss are designed along the lines described by Mead and Conway [1]: typically that a two-phase clock is used to transfer information between *registers* (or *latches*), and that these registers are separated by combinational logic. The following sections are devoted to modeling the time and area requirements of the latches, the combinational logic, and the clock driver. We then consider the overall circuit and investigate the optimal choice of the amount of latching for the two cases of on-chip and off-chip clock drivers. While the assumptions made about first-order circuit behavior pertain to nMOS technology, the analysis technique uses dimensionless parameterization and is applicable to any situations with deep combinational logic -- typically bit-parallel circuits. A representative tradeoff curve is shown for an example.

2. Clock Timing

We will adopt a version of the two-phase clocking system described by C. L. Seitz in Chapter 7 of [1], a typical stage of which is shown in Fig. 1. Fig. 2 shows the corresponding timing diagram: First, we must drive the Phase 1 clock signal ϕ_1 high, taking time t_{clock} (the *clock driver* time). We then need a minimum time t_{delay} (the *delay* time) to charge the input stage of the combinational logic. Phase 1 must then go low (taking time t_{clock}), and Phase 2 must then go high (also taking time t_{clock}). We must insure that there is a minimum time t_{12} during which both clocks are low; otherwise we run the risk that skew between the clock phases will cause both clocks to be on at the same time. This brings us up to the point where the combinational logic has already started to work.

The input values propagate through the combinational logic, taking some time t_{logic}. This time includes the time during which ϕ_1 is brought down and ϕ_2 is

brought up. The time t_{logic} will ordinarily dominate the clock-interchange time, but in general we need to set the time for this operation to

$$t = max(t_{logic}, 2t_{clock} + t_{12})$$

where for safe operation of the circuit t_{logic} must of course be taken as the *maximum* delay time of the combinational logic.

We next need to transfer the output values of the preceding logic stage to the input of the latch whose output is controlled by ϕ_1; that is, ϕ_2 must remain on for a minimum charging time t_{set} (the *preset* time). The ϕ_2 clock signal must then be brought down (taking another clock driver time t_{clock}, and another dead time (t_{21}) provided to insure non-overlap of clocks in case of clock skew.

The *minimum* period P of the circuit is therefore

$$P = 2t_{clock} + t_{delay} + t_{set} + t_{21} + max(t_{logic}, 2t_{clock} + t_{12})$$

To be more accurate, we might want to take into account the fact that the up-going and down-going clock waveforms are not completely symmetric; but the term t_{clock} can be taken to represent the average of the up- and down-going clock times in a single driver. In a multistage driver the stages alternate up and down, and we can take t_{clock} to be the sum of the averages of the up- and down-going times along the driving chain.

3. Latch Time and Space

We next want to express the time delay of the latches in terms of basic units that are determined by the technology. For this purpose, we consider the nMOS inverter with a minimum size pulldown and a pullup/pulldown ratio of 4 to be the *basic* cell, with area A, pulldown gate capacitance C, effective pulldown resistance R, and pulldown time (*transit time*) τ when driving the input of an equal size inverter. We refer to such a cell in what follows as a *minimal inverter.*

Now inverters in the latches drive pass transistors, so the discussion in [1] shows that we should choose a pullup/pulldown ratio of 8. The time required for the second inverter to charge its load is therefore approximated by the following RC constant:

$$t_{delay} = (R_1 + R_{pass})(C_{load} + C_{pass})$$

where the R's and C's are shown in Fig. 3. Assuming that the pass transistors are minimum size, $R_{pass} = R$ and $C_{pass} = C$. Also assuming that the capacitative load (input to the combinational logic) is minimal, we get

$$t_{delay} = 2(R_1/R + 1)\tau$$

$$= 2(L_1/W_1 + 1)\tau$$

where from now on we express resistance in terms of the length-to-width ratio of the transistor:

$$R_1 = (L_1/W_1)R$$

If the pullup/pulldown ratio of the latches is taken to be 8 (as mentioned above), we can write the normalized delay time as

$$t_{delay}/\tau = 2(8r + 1)$$

where $r = L_2/W_2$ is the size of the latch pulldown. When $r = 1/2$ the pulldown transistor of the latch inverter will be twice as wide as the corresponding transistor of the minimal inverter, but the pullup/pulldown ratio is 8, not 4, so the pullup transistor will then be the same length as in the minimal inverter. The area of such a latch inverter with $r = 1/2$ will be only a little larger than that of a minimal inverter, perhaps about 25% larger. The choice of $r = 1/2$ thus speeds up the latch without much area penalty, and we will use this value in this paper, although it could be kept as a parameter.

Using a similar argument based on RC charging times, the preset time is

$$t_{set}/\tau = (8r + 1)(1/r + 1)$$

The $1/r$ term comes from the input capacitance of the second inverter, which loads the first inverter. To see this, write

$$C_{load} = (L_2W_2/LW)C = (W_2/L_2)C = (1/r)C$$

where $L_2 = L = W$ are minimum size.

The latching area is easy to write down. Assuming that the pass transistors are the same size as minimal inverters, and that the latches have area $1.25A$, each two-phase latch requires normalized area

$$A_{latch}/A = 2(1.25 + 1) = 4.5$$

4. Combinational Logic Time and Space

We want a fairly general model for the combinational logic that is sandwiched between the latches; such logic may be built from NAND and NOR gates, pass transistors, or some combination of the two. We will assume that the typical logic stage is a uniform array of $n \times k$ *logical elements*, each of which has an area A_{elem} and a delay τ_{elem}, where

$$A_{elem} = \alpha A$$

and

$$\tau_{elem} = \beta\tau$$

This array will be thought of as n rows by k columns, with a maximum delay path from left to right of k elements. Since logic stages are not usually so uniform, the α and β parameters must represent *average* values for the combinational logic. If gates are built out of inverters and coupled directly, for example, β will

generally be determined by the fan-out factor of the logic and the size of the inverters. An average fan-out factor of 3 using gates (with a pullup/pulldown ratio of 4) will result in $\beta \approx 12$, because we must allow for the worst case in the propagation of logic, where all signals are up-going. To reduce this to a value closer to that of a minimal inverter, we expect to increase the area to, say, twice that of a minimal inverter. Thus we can take values of $\alpha = 2$ and $\beta = 4$-12 as typical of combinational logic implemented with arrays of gates. We should also note that the value of α should be selected to reflect the space per logical element required for power and ground lines.

We will assume that the nominal circuit has one typical logic stage between a pair of two-phase latches, and we then consider the insertion of $(m-1)$ latches equally-spaced in the combinational logic, $m \geq 1$. The case $m = 1$ then represents the original situation. We assume the latches can be made to "fit" well; that is, that the combinational logic is arranged regularly enough so that stages can be pushed apart and columns of latches inserted. The total time required for the logic is therefore

$$t_{logic}/\tau = \beta(k/m)$$

and the area

$$A_{logic}/A = \alpha d k^2$$

where $d = n/k$ is the height-to-width ratio of the original logic block, another dimensionless parameter, usually assumed to be 1.

5. On-Chip Clock Driver Time and Space

If we use an on-chip clock driver, we want to use a multi-stage version as described in [1], since the driver will have a large capacitative load, especially if there is an appreciable amount of intermediate latching introduced. We assume that clock distribution is on metal, so that propagation delay along the wires is small. Each stage is assumed to have a pulldown f times the size of the preceding, so if there are S stages driving Y pass transistors, each with minimal capacitance C,

$$f = Y^{1/S}$$

If we start the clock driving with a minimal inverter, the normalized delay of such a driver is approximately

$$t_{drive}/\tau = 2.5fS$$

The factor of 2.5 results from averaging the pullup time of 4τ and pulldown time τ along the inverter chain. (If we do not insist that S is an integer, and we minimize this delay with respect to f, we get the value $f = e$ [1]. But S is an integer.)

This estimate for delay assumes that we insist on a globally-synchronized

clock -- that the clock signals at the input of the driver can be used anywhere else without concern for synchronization. Caraiscos and Liu [11] have pointed out that the rise and fall times of the clock waveforms may be much smaller than the absolute delay, and that using a local clock may allow higher throughput, at the expense of using local clock signals that must be made synchronous with the signal itself at different points on and off the chip. Sending the clock along with the signal will incur other costs, of course. (For a discussion of the virtues of a globally-synchronized clock in signal processing, see [10]). The analysis in this paper is conservative in the sense that the resulting degree of latching and increase in throughput is on the low side. (We can avoid the area and delay penalty incurred by using an on-chip clock driver by moving the clock driver off-chip. That case will be discussed in more detail in Section 7.)

We must also consider the area contribution of the clock driver in relation to the rest of the circuit. The normalized area of the driver is

$$A_{drive}/A = \sum_{i=0}^{S-1} f^i = (Y-1)/(f-1)$$

Next we look at the overall time and space requirements of the circuit.

6. Optimization of *AP*-Product with an On-Chip Clock Driver

We can now write the total minimum normalized period $P/\tau = p$ in terms of our parameters as follows:

$$p = 5fS + 25 + \tau_{21} + \max(\beta k/m\ ,\ 5fS + \tau_{12})$$

where as above

$$f^S = Y = (m+1)n = \textit{number of lines driven}$$

and $\tau_{12} = t_{12}/\tau$, $\tau_{21} = t_{21}/\tau$. Similarly, the total normalized area $Area/A = a$ is

$$a = 2(Y-1)/(f-1) + 4.5Y + \alpha kn$$

where the factor of 2 accounts for the fact that we must have two drivers, one for each phase. (These can be combined to some extent, but the total area is still nearly twice that of a single driver.)

We now have the function $ap(m,S)$, where m and S are discrete parameters. The number of stages is never much larger than $\ln Y$, since the optimal choice of f is usually around e. In most cases of interest, therefore, it suffices to take the minimum of ap for $S = 1,\ldots,16$, producing what we call $ap(m,e)$:

$$ap(m,e) = \min_S ap(m,S)$$

The range of m is certainly between 1 and k, so the optimal choice of m can be determined simply by

$$ap(e,e) = \min_m ap(m,e)$$

The *gain* G in AP-product achieved by latching is therefore

$$G = ap(1,*)/ap(*,*)$$

7. The Case of an Off-Chip Clock Driver

As mentioned in Section 5, if we allow the clock driver to be off-chip, we can drive the larger capacitive loads incurred by extra latching with essentially no penalty in clock delay or driver area. The normalized period and area can then be written

$$p = 2\tau_{clock} + 25. + \tau_{21} + \max(\beta k/m\ ,\ 2\tau_{clock} + \tau_{12})$$

$$a = 4.5Y + \alpha kn$$

where we have assumed some delay of $\tau_{clock} = t_{clock}/\tau$ for the clock rise and fall times. The ap-product is therefore a function of only one unknown parameter, m.

With these changes in a and p, the same methodology applies – a numerical example will be given in the next section. Note, however, that now the optimal value of m will occur roughly near the breakpoint where $\beta k/m = 2\tau_{clock} + \tau_{12}$, and that these times are both highly uncertain and small in size. The analysis in this case is therefore much less reliable, and much more sensitive to unmodeled effects such as propagation delay, than in the on-chip clock driver case.

8. Numerical Examples

We now give some typical numerical results. For this purpose, we consider a 16-bit array multiplier, implemented by an array of full adders, as described , for example, in [2]. We also assume that the full adders are implemented with gates; each full adder will then be about 3 gates deep. The carry propagation will require an array that has a maximum depth of 2×16, so altogether the combinational logic will have $k \approx 100$. (This is consistent with the value of "113 gate delays" given in [3].) Say that each gate takes about double the area of a minimal inverter ($\alpha \approx 2$, optimistic for area, and hence pessimistic for our purposes), and that, as discussed in Section 4, $\beta \approx 6$. The array is roughly square, so that $d \approx 1$. Finally, we will assume that clock skew is not an important problem, and take $\tau_{12} = \tau_{21} = 4$.

Fig. 4 shows a plot of normalized period $p(m,*)/p(1,*)$; normalized area $a(m,*)/a(1,*)$; and normalized AP-product $ap(m,*)/ap(1,*)$; vs. m. The period as a function of m decreases sharply (roughly as $1/m$) until the combinational logic time is dominated by the clock-swapping and dead time (that is, until $t_{logic} \approx 2t_{clock} + t_{12}$). After this point the clock-driving time will determine the minimum clock period and it no longer pays to increase m, because the area will increase with no payoff in speed. The minimum value of period occurs close to the minimum value of AP-product. Thus, in theory, the period can be decreased somewhat from its value when the AP-product is minimized, at a slight cost in

area. In practice the optimal values are almost always nearly equal, and sometimes identical because of the discreteness of the parameters m and S.

Fig. 5 shows a plot of gain G in AP-product vs. the depth of combinational logic k, for the values $\alpha = 2$ and $\beta = 4,6,8,12$. The graph shows significant gains in AP-product (more than 2) over the unlatched case when $k \geq 50$ and $\beta \geq 6$. Even when the gates are as fast as a minimal inverter (worst-case delay factor $\beta = 4$) there is an AP-product gain of 2.2 when $k = 100$. Note that a larger value of α would only improve the gain.

We conclude by looking at the actual numerical values of the minimum clock periods and areas involved in this analysis. Taking the $k = 100$, $\alpha = 2$, $\beta = 6$ case above for a hypothetical 16-bit array multiplier, and assuming $\tau = .3$ nsec for current technology, we get a period of $P = 210$ nsec with no intermediate latching, and an optimal period of $P = 66$ nsec with $m = 6$ (5 intermediate latching stages).

The area before latching is $2.11 \times 10^4 A$, which at $\lambda = 1.5\mu$ (3μ line width) and a $225\lambda^2$ inverter is about 10.7 mm^2. After the intermediate latching, the area becomes 12.1 mm^2; certainly a modest increase in area for about a three-fold increase in speed.

The preceding example assumed an on-chip clock driver. When we use an off-chip clock driver at presumed small cost, as discussed in Section 7, we naturally get much faster solutions. In this example, the optimal value of period with the parameters of Section 7 and $\tau_{clock} = 4$ (assuming a very sharp clock rise- and fall-time), minimizing AP-product, is 18 nsec, compared with the unlatched value of 191 nsec. The area goes from 10.6 mm^2 with no latching to 16.5 mm^2 with latching. This large increase in area reflects a corresponding increase in the density of latching: 26 ($m = 27$) latching stages are introduced. We emphasize that in the case of an off-chip clock driver, the numerical values of the parameters t_{12} and t_{clock} are very uncertain and the optimal values of period, area, and latching density are sensitive to these parameters. The large predicted speedups in possible clock rate may not be realizable in practice.

9. Conclusions

We have modeled the timing of a generic pipelinable VLSI circuit in which there are combinational logic stages separated by latching stages driven by two-phase clocks. An array multiplier is typical of such a configuration. We then investigated the effect of introducing intermediate latching stages, especially the tradeoff between increased throughput and increased area. Expressions were derived for area and minimum clock period, normalized in terms of minimal inverter area and delay, and we showed that optimal choices of the number of clock driver stages (S), and the number of intermediate latching stages ($m - 1$), can be made by simple enumeration.

The numerical results illustrate the choice of latching density in a typical signal processing application. According to our model, a 16-bit array multiplier

with gate logic and an on-chip multistage clock driver can be clocked about 3 times faster with about a 13% increase in area using 5 intermediate latching stages. This decrease in period is also accompanied by an increase in the latency, or delay, of the multiplier.

Higher throughput can be achieved with an off-chip clock driver, but the parameters in that case are less well known, and at such speeds the model becomes less reliable.

Much more work needs to be done on detailed modeling of the timing of such VLSI circuits if we are to achieve maximum throughput rates in applications like signal processing. Future work will attempt to refine our model, along the lines of [13] as an example. We also need to study propagation delay, which was assumed to be relatively small in the examples (4 times the minimal inverter gate delay τ for clock distribution, a reasonable assumption if the clock lines are metal, for example). Another important set of interesting problems concerns the study of the way algorithms, topologies, and layouts interact with the timing problems considered here. Recent work on completely-pipelined or bit-level systolic arrays is a start in that direction (see, for example, [2,4-9,12]).

10. Acknowledgements

We are indebted to C. Caraiscos and B. Liu for valuable comments on the manuscript.

11. References

1. C. Mead and Conway, L.,*Introduction to VLSI Systems*, Addison-Wesley Publishing Co. Menlo Park, Ca., 1980.
2. 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, & Computers*, Nov., 1981.
3. Bötcher, K., A. Lacroix, M. Talmi, D. Wesseling, "Integrated Floating Point Signal Processor," *Proc. 1982 IEEE Int. Conf. Acoustics, Speech, and Signal Processing*, Paris, May 1982, pp. 1088-91.
4. Cappello, P. R. and K. Steiglitz, "Digital Signal Processing Applications of Systolic Algorithms," *CMU Conference on VLSI Systems and Computations*, H.T. Kung, Bob Sproull, and Guy Steele (eds.), Computer Science Press, Rockville, Md., 1981.
5. Cappello, P. R. and K. Steiglitz, "Bit-Level Fixed-Flow Architectures for Signal Processing," *Proc. 1982 IEEE Int. Conf. on Circuits and Computers*, Sept. 29 - Oct. 1, 1982.
6. 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 (to appear).

7. Cappello, P. R. and K. Steiglitz, "Completely Pipelined Architectures for Digital Signal Processing," *IEEE Trans. on Acoustics, Speech, and Signal Procs.*, Vol. ASSP-31, No. 4, August 1983 (to appear).

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

9. P. B. Denyer 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. (*Proceedings of the First International Conference on Very Large Scale Integration*, University of Edinburgh, August 18-21, 1981.)

10. 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.)

11. C. Caraiscos and B. Liu, private communication.

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

13. Penfield, P. Jr. and J. Rubinstein, "Signal Delay in RC Tree Networks," *Proc. of the Second California Institute of Technology Conference on VLSI*, 1981.

14. Leiserson, C. E. and J. B. Saxe, "Optimizing Synchronous Systems," *Proc. 22nd Annual Symp. on Foundations of Computer Science*, IEEE, October 28-30, 1981.

Figure Captions

Fig. 1 Two-phase clocked latches between stages of combinational logic.

Fig. 2 Clock-timing diagram.

Fig. 3 Details of the clocked latches, showing pullup and pulldown effective resistances and capacitances.

Fig. 4 Normalized period, area, and AP-product vs. m for $\alpha = 2$, $\beta = 6$, $k = 100$. The parameter $(m-1)$ is the number of intermediate latching stages.

Fig. 5 Gain in AP-product vs. combinational logic depth k for $\beta = 4, 6, 8, 12$. The parameter β is the delay of a combinational logic element, normalized in terms of that of a minimal inverter.

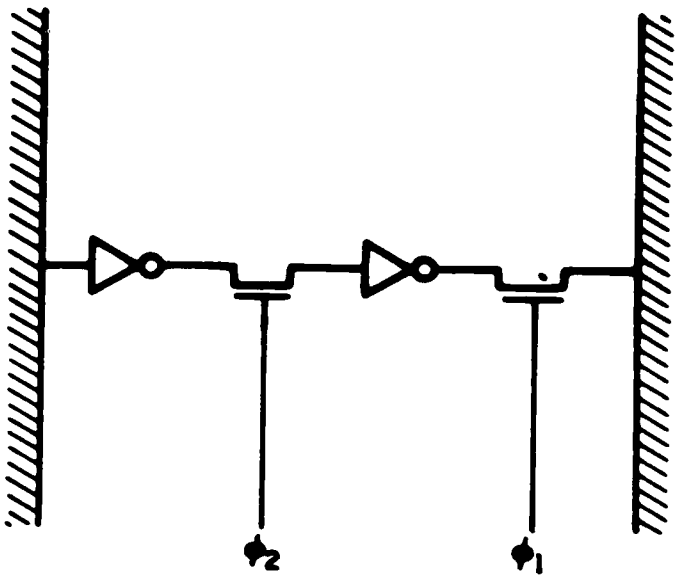


Fig. 1
Cappiello et al

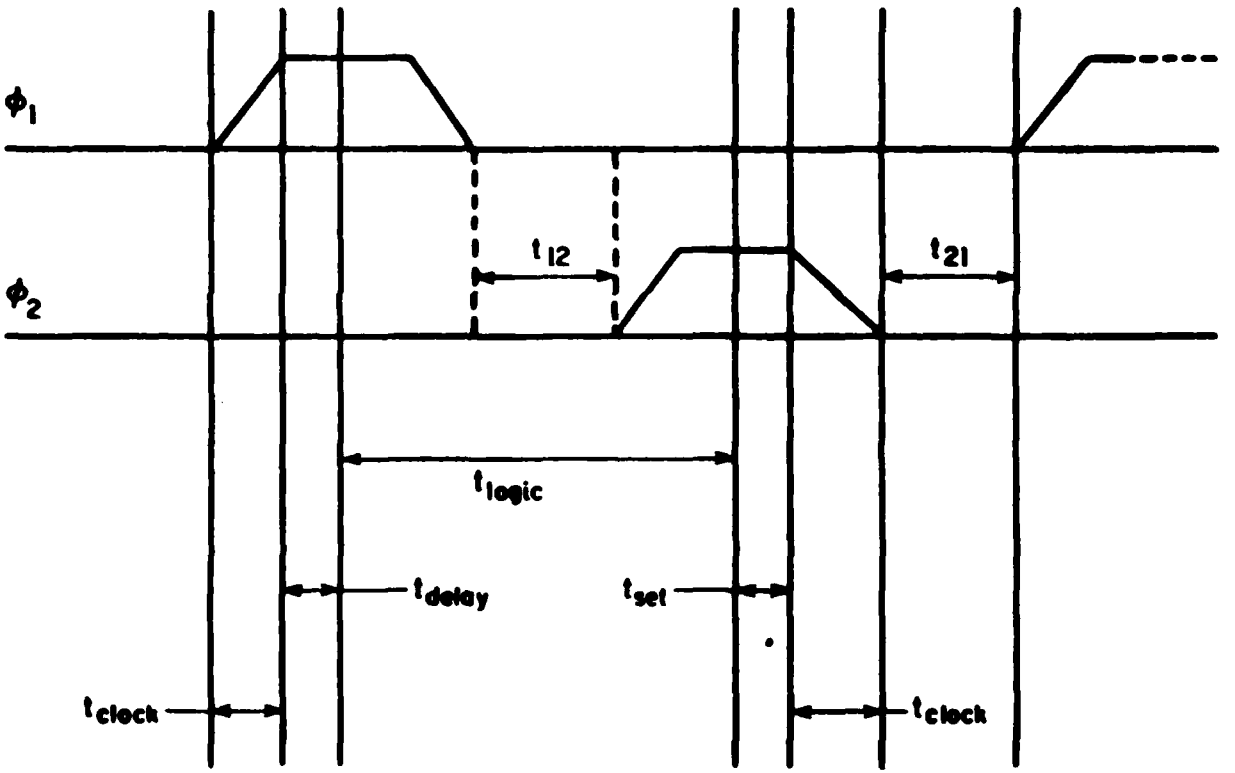


Fig 2

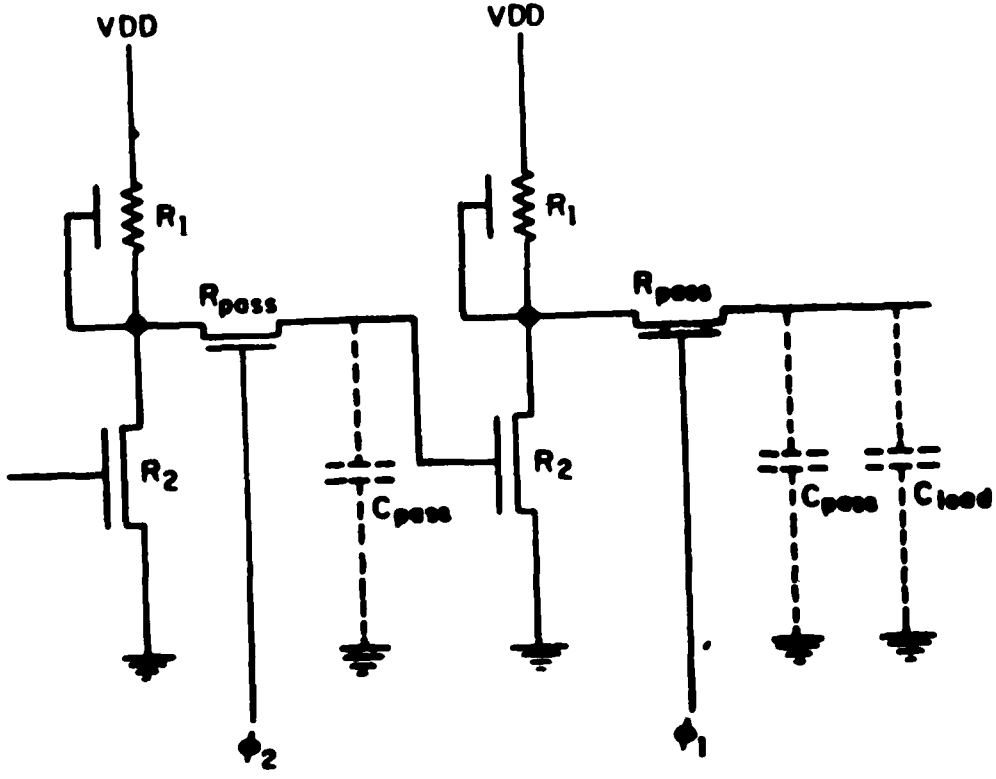


Fig. 3

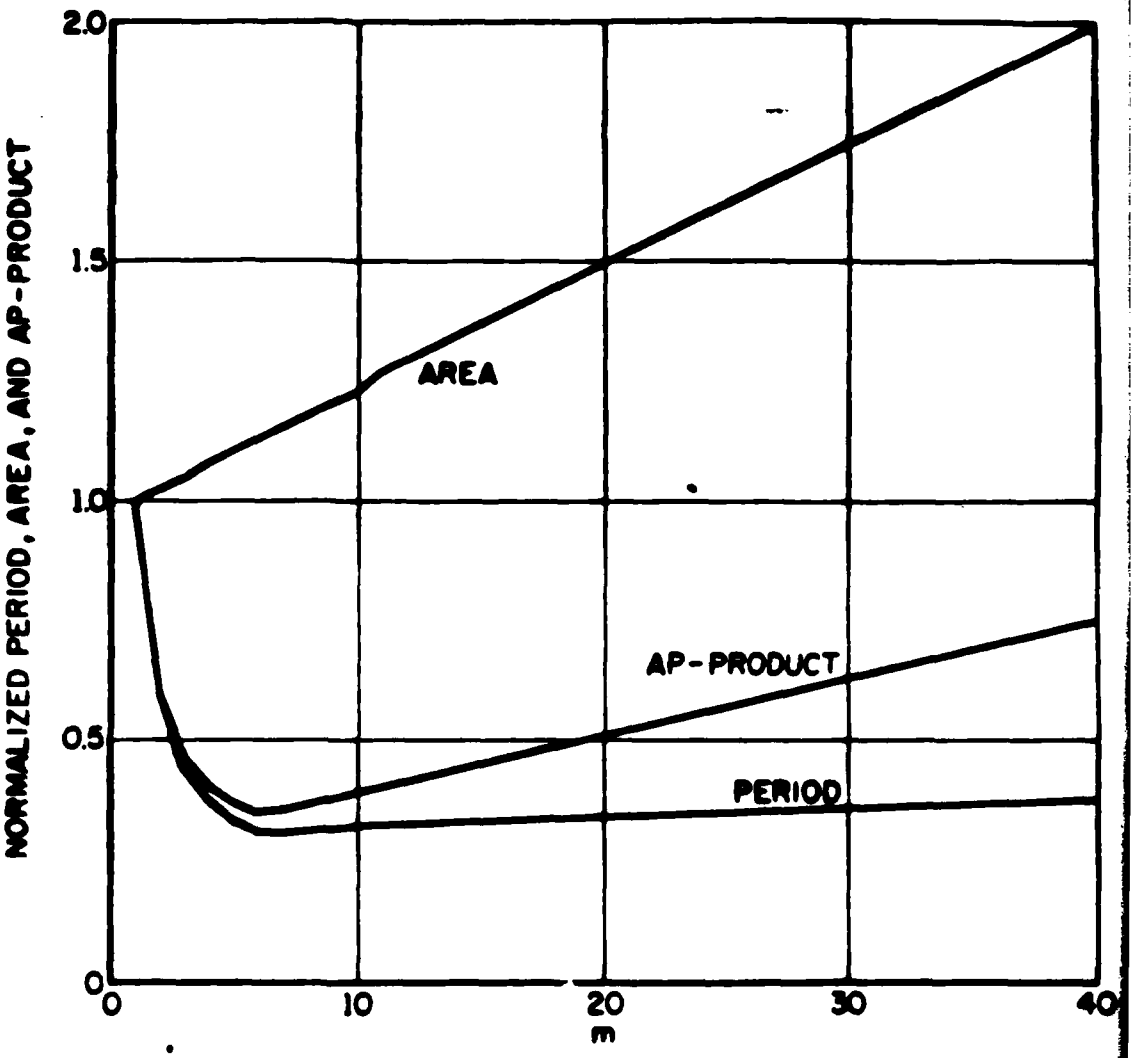


Fig. 4
Coppolo

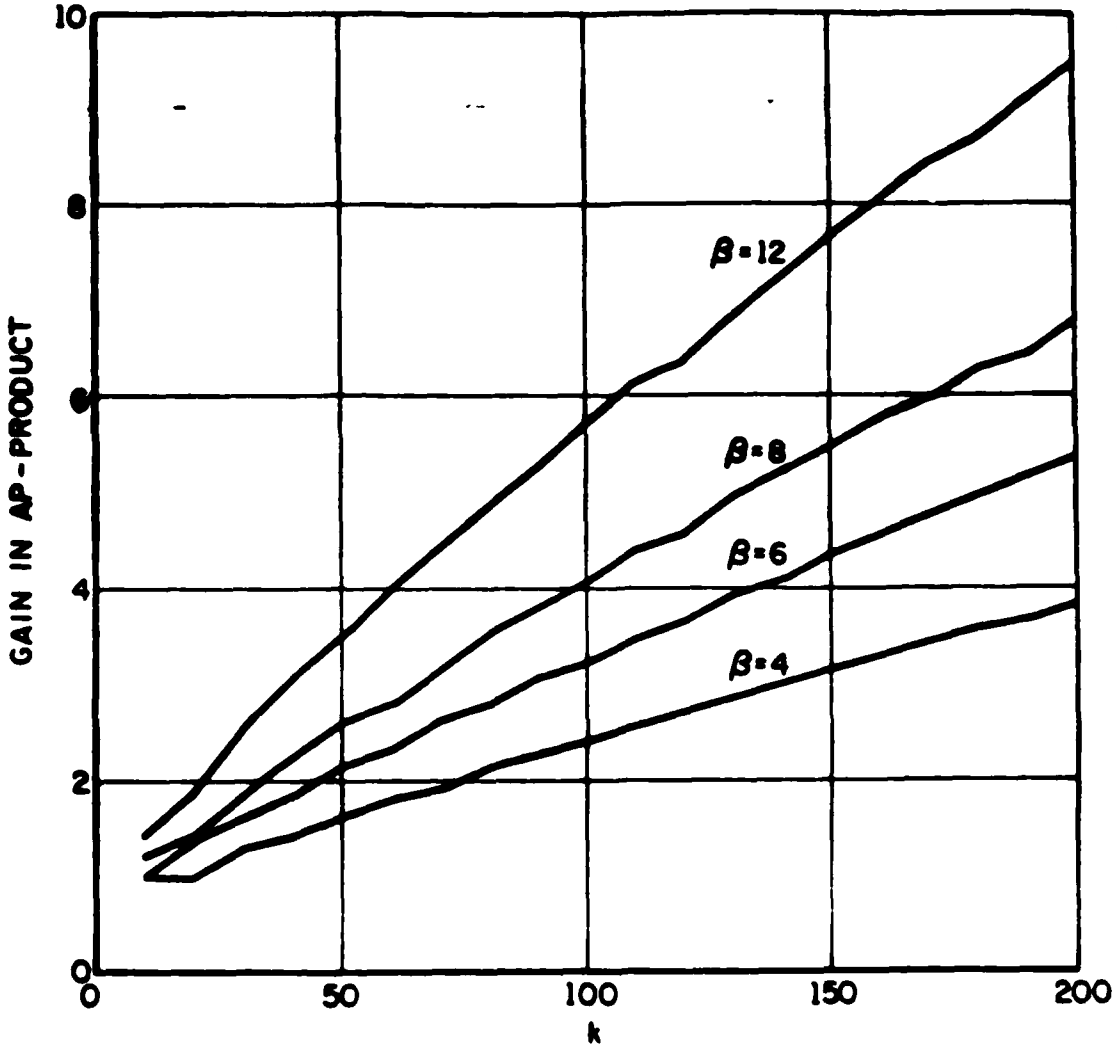


Fig. 5
Coppelle et a

ALI2 Documentation and Implementation Guide

Language Overview

Version 4
February 25, 1982

J. Valdes, R.L. Kelin

1. Introduction

This document is a reference manual for ALI2, a Pascal-based[†] procedural language for describing VLSI layouts. Several revisions have recently been made to the language, so the current document may differ slightly from previous versions.

The syntax of the language will be presented by way of syntax diagrams. An effort has been made to minimize the number of diagrams and shorten the accompanying text by (a) including more semantic information in the syntax diagrams than is customary, and (b) making reference where possible to syntactic entities of the Pascal syntax given in [1] (which has been included as an appendix to this document). Semantic content which is not evident from the syntax diagrams will be explained in the text.

The ALI2 design goals and the problems that ALI2 addresses are described in detail in [2].

2. General description of ALI2

ALI2 programs are compiled by first translating the ALI2 statements into standard Pascal [3], and then compiling the Pascal statements into linkable binary object files. Partly as a consequence of this arrangement and partly for aesthetic reasons, ALI2 programs look very much like Pascal programs. Many features of ALI2 have syntax and semantics very similar, if not identical, to those of corresponding Pascal constructs. For the sake of brevity, this document takes knowledge of Pascal for granted. The syntax and semantics of ALI2 will be described in terms of the syntax and semantics of Pascal whenever possible.

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

[†]Based on UCB Pascal under UNIX.

The ALI2 system is proprietary to Princeton University, Princeton, New Jersey. Its development was supported in part by DARPA under grant N00014-82-K-0549.

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.

The entire layout generated by an ALI2 program is itself actually an instance of a single cell defined by the program. The body of an ALI2 program is simply the statement that instantiates the cell definition. Just as typical Pascal main programs will include numerous calls to constituent procedures and functions, the definition of the main ALI2 cell will typically include the instantiation of numerous component cells.

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. Wires which are connected to only one cell will not appear in the layout produced by the program.

If both endpoints of a wire are interior to a cell C3 (i.e., the wire connects two cells C1 and C2 internal to cell C3), then the wire is said to be *local* to cell C3. Such wires will be declared as local *wire variables* in the definition of cell C3. Wires which run from a boundary of cell C3 to the outside boundary of another cell internal to C3 (e.g., cell C1) are instances of formal parameters. These wires will be declared in the heading of the definition of cell C3. Thus, ALI2 wires may be obtained by (a) declaring local wire variables in a cell which is to use those wires for internal cell connections, and (b) declaring formal parameters for connecting wires that end on the boundary of a cell to one or more of its internal cells. Fig. 1 illustrates these relationships.

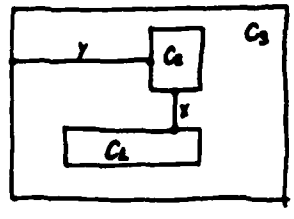


Fig. 1
C1, *C2* and *C3* are cell instances; *s* is local to *C3*;
y is an instance of a formal parameter in the definition of *C3*.

In addition to the cell bounding boxes which are automatically created by ALI2 for each instantiated cell, a user may explicitly create and manipulate other boxes. These user-defined bounding boxes are used to enclose rectangular areas of a layout that are to be considered as a

unit during certain operations performed while the ALI2 program is being executed. ALI2 will permit wires to pierce these boxes, and generally delegates all responsibility for their manipulation to the user.

In the remainder of this document, we will describe ALI2 by discussing in turn the general form of an ALI2 program, the type structure of ALI2, the facilities for cell definition and instantiation, the statements specific to ALI2, and finally the predefined cells, procedures and functions of the language.

3. General form of ALI2 Programs

The general form of the units that the ALI2 system manipulates is given by the syntax diagrams of figs. 2 and 3. These units are of two types: *programs* and *modules*. ALI2 *programs*, when compiled and run, produce layouts. *Modules* can be separately compiled and linked to other modules or programs but they are purely declarative and cannot be run by themselves. (for the details of how these two units are actually handled by the ALI2 system, see [4] and [5]).

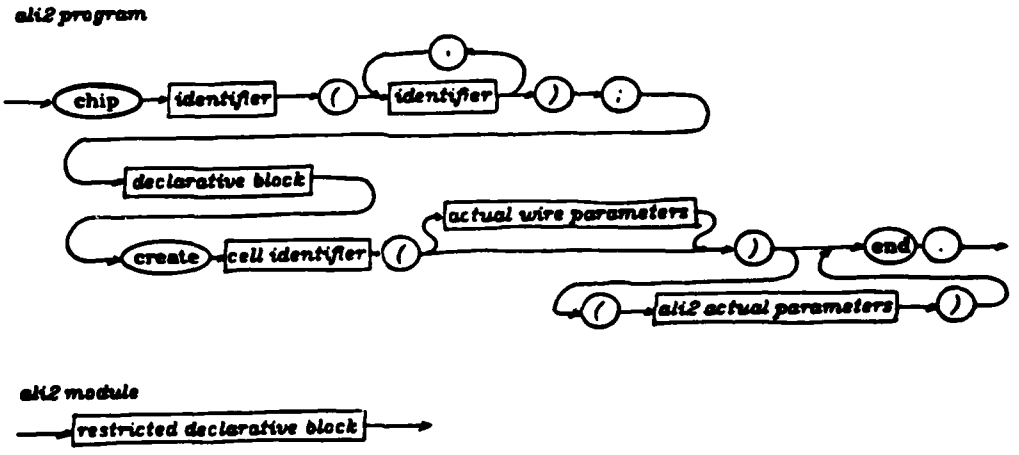


Fig. 2
The general form of ALI2 programs and modules

An ALI2 program consists of a series of cell declarations and the instantiation of a single cell. Each cell declaration consists of a *header* and a *body*. The header specifies the formal parameters, in effect defining the cell boundary. The body describes the local objects and contains statements indicating how to create an instance of the cell when actual parameters are provided to be connected to the formal ones.

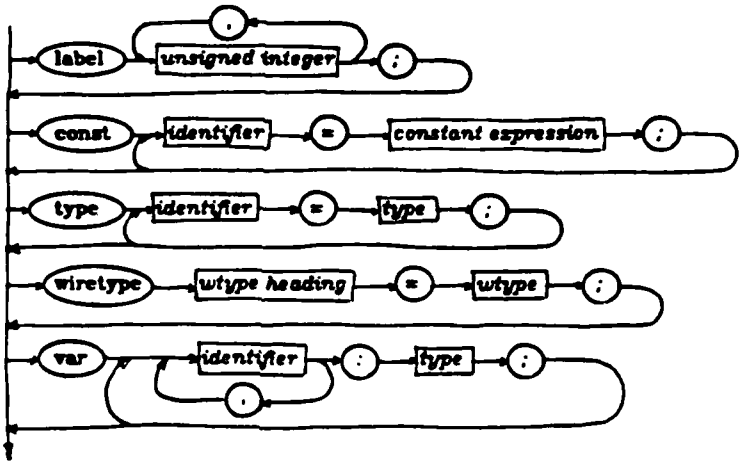
The relationships between wires and boxes that can be expressed in an ALI2 cell definition are mostly metric free: no actual sizes or positions may be specified. The sizes and positions of the wires of the layout produced by the program are determined by a simple process that minimizes the final layout area while preserving the relationships between the wires stated in the program [4].

Note that declarations of ALI2 objects -- *cell* declarations and declarations under the keywords *wiretype*, *wirevar* and *box* -- are completely separate from the declarations of Pascal objects. The declarations of Pascal objects have exactly the same syntax and semantics as in Pascal. The

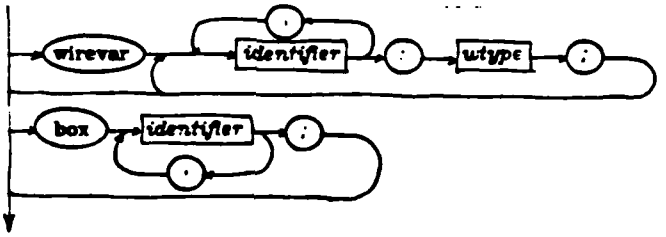
declarative block

restricted declarative block

general declarations



object declarations



PFC declarations

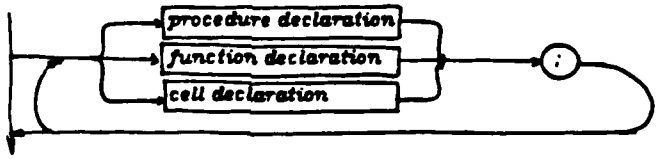


Fig. 3
The syntax of declarative blocks

only exception is that ALI2 permits the right hand sides of constant declarations to include expressions that are evaluated at compile-time (see fig. 4 for the syntax of such expressions).

It is important to notice also that no wires or boxes may be declared in a module. This is done in order to guarantee that each wire in a layout is either local to a cell instance, or an instance of a formal parameter.

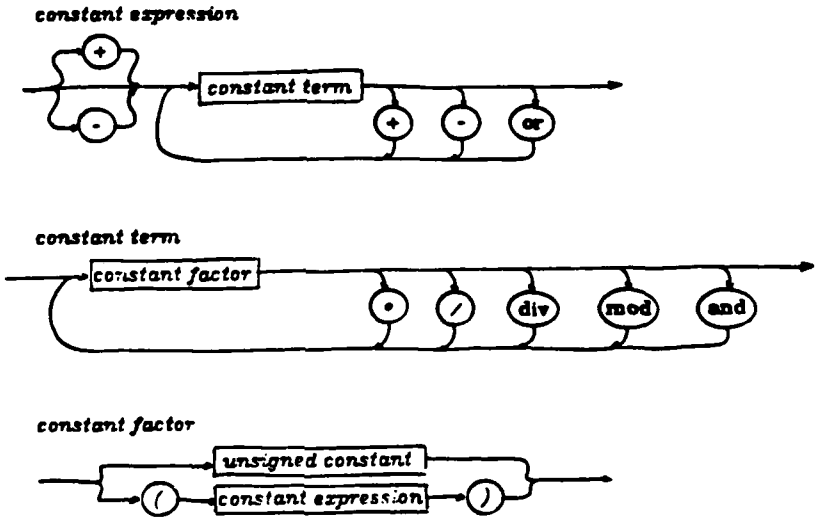


Fig. 4
The syntax of the expressions allowed on the right hand side of constant declarations

ALI2 inherits Pascal's type structure almost intact. Thus enumerated types, subrange types, arrays, records, pointers, files etc. are all ALI2 types. The only exception is the *set* type, which does not exist in ALI2.

All of Pascal's predefined identifiers exist in ALI2 as well: types (*integer, boolean...*), constants (*maxint, true...*), files (*input, output*), procedures (*put, get, write...*), functions (*abs, sqrt...*) and directives (*forward, external...*). All of these behave in an ALI2 program as they would in a Pascal program.

The header of an ALI2 program includes a list of identifiers. They are interpreted, as they are in Pascal, to be a list of logical file names used by the program. As in Pascal, each of these identifiers except *input* and *output* must be declared as variables of type *file* in the declarative part of the program.

Although it is not shown in the syntax of fig.2, all Pascal statements are also ALI2 statements; the only exception is the *with* statement, which has been omitted for a mixture of aesthetic and pragmatic considerations.†

†The deviations from standard Pascal just mentioned are small enough so that no great problems should arise from their existence. It is important to note that object files containing translated ALI2 programs will be totally compatible with those generated by the Pascal compiler [4]. Thus, Pascal fragments that use the disallowed features extensively can be compiled separately and loaded with translated ALI2 programs, minimizing further

4. Predefined Pascal types

ALI2 inherits all the predefined types of Pascal. ALI2 has several additional predefined Pascal-like types: *wireorientation*, *orientationchange*, *directionofseparation*, *layer*, *wirelayer* and *signal*. All of them are enumerated scalar types except *wirelayer*, which is a subrange of *layer*. As we will see in the detailed description that follows, these types are identical in all respects to any other enumerated predefined type with the exception of *signal*, which differs in a minor way.

The type *wireorientation* is defined as

wireorientation = (nullorient, vertical, horizontal)

The composition of this type reflects the fact that ALI2 wires can lie only in one of two orientations so that only "Manhattan" layouts can be expressed in ALI2. The orientation of a particular wire will be determined by the way it is used in the program (i.e., at run time), with all wires having *nullorient* as their initial orientation. The run time system of ALI2 will be responsible for assigning orientations to wires and checking that no inconsistent use of wires with respect to their orientation occurs.

The type *orientationchange* includes all the operations of the dihedral group – the only rigid plane motions that map a wire orientation to a wire orientation. It is defined as follows:

orientationchange =
(nullchange, rotated90, rotated180, rotated270, flipped0, flipped45, flipped90, flipped135)

Changes of orientation are useful when instantiating cells. They permit the creation of instances of the same cell declaration that can be mapped into one another by rigid plane motion, thus avoiding the need for multiple definitions.

The type *directionofseparation* consists of the eight directions along which ALI2 objects can be separated (plus the null value for this type). It is defined as follows:

directionofseparation = (nulldir, ttob, btot, ltor, rtol, tltobr, brtotl, trtobl, bltotr)

The symbols that belong to the type *layer* depend on the process to which the ALI2 system is targeted. For the current system (nMOS as described in [6]), its definition is the following:

layer = (nulllayer, metal, poly, diff, cut, impl, glass, virtual)

In general, this type will include the names of all the physical layers of the process being used plus the identifiers *virtual* (the layer on which boxes may be imagined to lie) and *nulllayer* (a null value for this type that will be used exclusively by the ALI2 run-time system.

The type *wirelayer* is the subrange of *layer* that contains the layers on which wires are normally constructed. It is defined as:

wirelayer = metal .. diff

The type *signal* is also an enumerated scalar, but it has some special characteristics. Unless redeclared by the user, the type contains only the value *nullsignal*. Users can redeclare the type as an enumerated type, but if they do so, the identifier *nullsignal* must be among the values of the

---

the chances that these absences will hurt the ALI2 user in a serious way.

type because this identifier is used in a special manner by the ALI2 runtime system.

Here is a valid signal type definition:

signal = (nullsignal, power, ground, datain, dataout)

5. Wire types

This section describes the type structure of ALI2 wires. The syntax of the wire type declarations is given in fig. 5. The semantic content of these declarations will be described in detail in the next two sections. We will examine the simple wire types first and then consider the composite wire types.

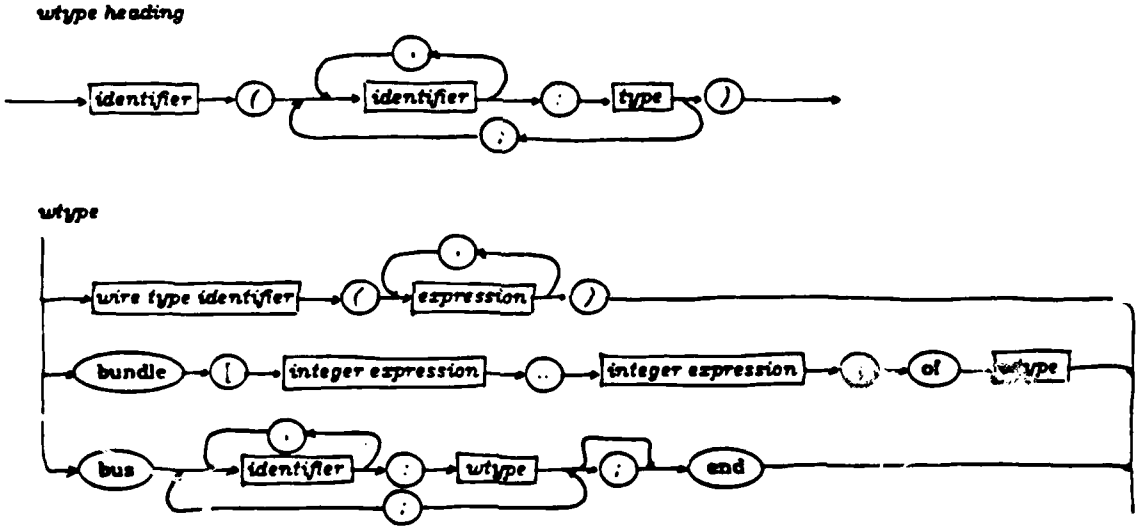


Fig. 5
The syntax of the right hand side of wire type declarations

5.1. Simple wire types

All wires in ALI2 are of similar type. This type, however, is a *parametric type*, a concept which is not part of Pascal and therefore requires explanation here. The ALI2 parametric types are modeled after those described in [7].

Parametric types are designed to make type checking more selective or weaker in certain places without doing away with it altogether. It works particularly well as a way to permit the user to regulate the extent of type checking that is to be performed during procedure or function calls.

The basic idea of a parametric type is that of leaving some characteristics of the elements of the type unspecified. These characteristics become *formal parameters* of the type declaration. When the type identifier is used in the right hand side of the declaration of a new type or variable, each formal parameter in the parametric definition must be provided with a value of the appropriate type as a matching *actual parameter*. Hence, the left hand sides of parametric type definitions look somewhat like procedure headers and the right hand sides like procedure calls. The actual parameters fill out the previously unspecified characteristics of the parametric type.

In the case of ALI2 wires, there are three parameters to the parametric *wire* type: the *layer*, the *width* and the *signal* of the wire.[1] The layer and signal will have to be values of the predefined types *wirelayer* and *signal* respectively. The wire width is an integer expressing the width in hundredths of microns. Width can be expressed in scalable λ units [6] using the predefined constant *lambda* (see examples below).

Other parametric types can be defined from this one by pseudo-calls to the type definition in which actual values for some (or all) of the formal parameters are specified. 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 formal parameter "w"essentially passes over to the right side of declaration to become an actual parameter to the definition of the parametric type "wire". We say that "polywire" is *derived* from "wire".

Similarly, the definition

```
stdpolywire = polywire ( 2*lambda )
```

creates a new type. This new type is different from the two types considered earlier in that it has no parameters. We call this a *bound type*.

Because there is only one predefined parametric type, the global structure of these types will be quite simple: each new type will be derived from a previously existing one by a partial or total instantiation of its parameters. The structure of the types will therefore be a tree of types with the predefined type *wire* at the root.

ALI2 provides a predefined wire constant that belongs to all simple wire types. This wire is called *nullwire* and will be used extensively in connection with cell calls. It is analogous to the predefined constant *nil* used by Pascal in connection with pointer types.

The values used as actual parameters in the right hand side of a parametric type declaration 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 wire types of ALI2 are bound at run time*. This allows for a great deal of flexibility at the cost of some complexity in the run time package.

---

[1] The *orientation* of the wire can be inferred at run time by the way the wire is used and need not be part of the declaration itself

5.2. Composite wire types

As fig. 5 shows, there are three composite wire types in AL12: *bus*, *bundle* and *list*.

The types *bundle* and *bus* 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. Below are some sample definitions of composite objects of these types.

```
data1 ( low, high, width : integer ) = bundle [ low .. high ] of polywire ( width );
data2 = bundle [ 1 .. 100 ] of wire ( metal, 10*lambda, nullsignal );
foo1 ( w1, w2 : integer; l : layer; s : signal ) =    bus
                                                          f1 : polywire ( w1 );
                                                          f2 : wire ( l, w2, s )
                                                      end;
foo2 =  bus
            d1    : data1 ( 10, 20, 2 );
            d2    : data2
        end;
```

The type *bundle* is a parametric type in its own right, since the number of the wires it contains and the values used to access them may be parameters of the type. The type *bus* is parametric only because the types of its components may be parametric.

The type *list* is peculiar to AL12. A list is either the *nulllist* which has no component wires, 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 formal parameters. As we will see later, only formal parameters may be declared to be of type list.

6. Cell declarations

This section describes the syntax and semantics of cell declarations. The *cell* mechanism is very similar in spirit to the procedure facility of Pascal. It permits the users of AL12 to introduce hierarchical information into their programs, and therefore into the layouts they produce. Unlike procedure hierarchies, however, the hierarchy of cell invocations corresponds very closely to features in the program output.

The syntax of a cell declaration is given in fig. 6. The next two subsections examine this syntax and the associated semantics in detail.

6.1. Cell headers

A few syntactically correct cell headers are given below

```
cell shift ( left l : shiftbus; top t : clocks; right r : shiftbus );
cell register ( left l : shiftbus; top t : list; right r : shiftbus ) ( size : integer );
cell silly ( left w : wire; left p : polywire );
cell doit ( ) ( k : tp );
cell generic ( top t1, t2, t3 : wire ) ( cell c ( left l1, l2 : polywire ) );
```

The cell header may include two parameter lists. The formal parameters in the first list must be of wire types, while no formal parameters of wire types may be included in the second list. This is done to preserve the strict hierarchy of wires with respect to cell instances.

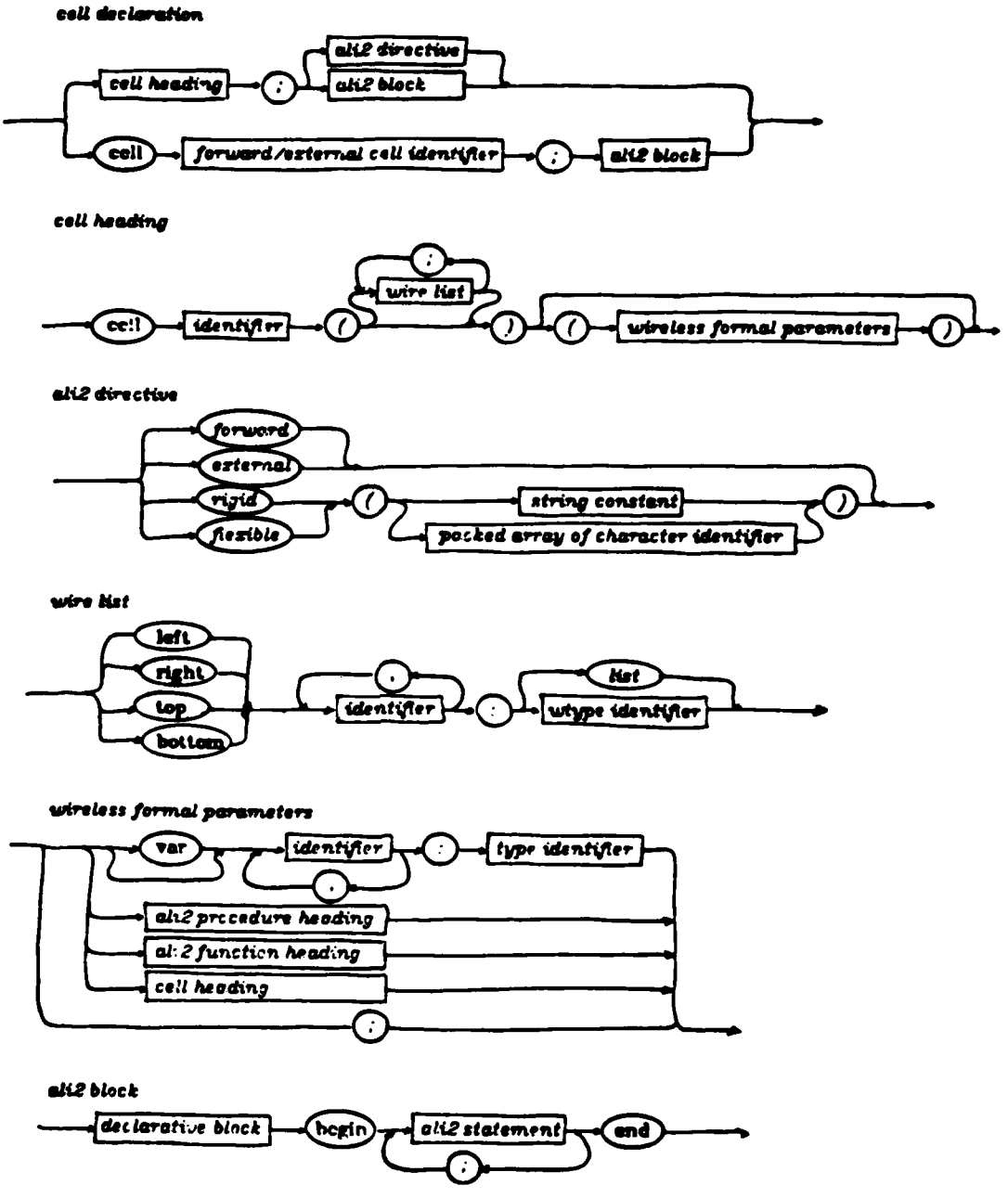


Fig. 8
The syntax of a cell declaration

The formal wire parameter list includes information about where the parameters intersect the cell boundary. Besides the explicit information given by the keywords *left*, *right*, *top* and *bottom*, there is an implicit assumption that the *top and bottom parameters are listed in left to right order* and that the *left and right parameters are listed in top to bottom order*. Thus, the header of a cell definition describes the *boundary* of the cell.

Cells can be declared as formal parameters to other cells by listing a prototype of the cell header in the second of the two parameter lists of the cell definition header. This is the same general manner in which procedures and functions are passed as parameters in Pascal (see [1]).

6.3. Cell bodies

As explained above, the executable portion, or "body", of an AL12 program is a single cell instantiation statement. The body of a cell may be either (i) one or more statements bracketed by the keywords *begin* and *end*, or (ii) a *directive* which specifies that the cell header should be treated in a non-standard way.

The executable part of the cell contains statements which describe the structure and content the cell will have when it is later instantiated. This part may contain any AL12 statement, including recursive instantiations of the cell itself. A complete discussion of the AL12 statements is given in section 9.

The directive *forward* is used exactly as in Pascal to circumvent the AL12 requirement that a procedure, function or cell definition refer only to previously defined cells, functions or procedures.

The directive *external* is intended to permit the separate compilation of files. It is not part of standard Pascal but something similar to it in spirit is found in most recent compilers. Its meaning in AL12 is similar to that of the *forward* directive in that it defers the association of a cell header to a body. It is unlike it in that the external directive adds the cell name to the list of symbols visible to the link editor when the program or module in which the definition is found is translated to a binary file.

For cells defined using the *external* directive, a body can be associated to their header either at compile time (by giving the body as if the header had been listed with a *forward* directive) or at load time (the name of the cell is left unresolved after compilation and the loader will attempt to resolve it later). In the first case, the cell name will be used to resolve references at load time and in the second it will be an outstanding reference which the loader will have to resolve.[†]

Thus, when a file containing the following

```
...
cell z (...) (...); external;
...
cell z; begin ... end;
...
```

is processed by AL12, *z* becomes a symbol visible to the loader which can be used to resolve unbound symbols from other loader files. If the body of the cell had not been given in the same file, *z* would have become a symbol to be resolved by the loader.

[†]Note that AL12 external compilation facilities differ from those of UC Berkeley Pascal in that the header of an external object may be in the same file as its body. This permits type insecurities to creep in by sidestepping the cross-module type checking features of the UC Berkeley UNIX compiler but makes the facility easier to use.

The directives *rigid* and *flexible* indicate that the cell definition is not to be given textually as part of the program: it is to be found in the file named by the string given as argument to the directives. That file must be a "rectangle file" (see [8]) defining a rectangular layout fragment. When a cell defined in this manner is instantiated, the AL12 run time system will attempt to integrate the layout fragment described by the file into the layout defined by the program.

The format of rectangle files includes a description of the elements that touch the boundary of the layout fragment described by the file [8]. The run time system will check that the boundary as described in the file matches the actual parameters given when the cell is instantiated; an error will be generated if a parameter mismatch exists. Note that this check is *not* performed at compile time because it is not always possible to do so. If the two boundaries match, the run time system "connects" each of the actual parameters used in the instantiation to the corresponding element on the boundary of the rectangle file. This connection is performed by abutment if the directive is *rigid*. The directive *flexible* is not currently implemented, but when operative, it will connect the actual and formal parameters using a simple channel router.

7. Wire variables, Box variables, and Formal Parameters

As fig. 3 shows, the format of the declarations of wire variables follows the standard Pascal conventions. Some examples of declarations of wire variables of the types declared as examples in section 6.1 are given below.

```
w1        : foo2;
w2        : foo1 ( 4, 6, metal, power );
w3        : data1 ( 1, 100, 6*lambda );
w4        : bundle [ 1 .. 100 ] of bundle [ 1 .. 100 ] of stdpoly;
w5, w6    : polywire ( 2*lambda );
w7        : stdpolywire;
w8        : bundle [ -10, 10 ] of polywire ( 2*lambda );
```

Each declaration creates a number of objects of the specified type which exist in a certain syntactic scope.

The main operation that the AL12 user will perform with wires is to pass them as actual parameters in cell instantiations. As stated earlier, AL12 expects each wire to be used as an actual parameter in exactly two cell instantiations, i.e., an actual wire parameter connects two cells. Incidentally, AL12 will separate automatically any two cells connected by a wire.

The format of the declarations of box variables is quite different from that of other variable declarations. For instance, the box declaration

```
box  b1, b2, b3;
```

simply states that the identifiers listed may become associated to boxes during the scope of the declaration. If such an association occurs, then the identifier stands for the box associated to it until execution leaves the scope of the definition. No box can be associated to more than one box variable (no aliases) and no box variable can be associated to more than one box (no reassignment).

Box variables are somewhat like labels in a Pascal program, not only in the format of their declaration but also in the way they are used. The only operation that can be performed on a box is to use it in a statement specific to AL12. This shifts responsibility for their manipulation to the AL12 run time system.

There is an important difference between the scope rules for box variables and wire variables and the scope rules for all other variables. Pascal-like variables are governed by the same scope rules used by Pascal, with cells treated in the same way as functions or procedures. Wires and boxes, on the other hand, are only accessible locally: no wire or box can be global to any context. Once again, this is a consequence of our desire to preserve a strict hierarchy of layout elements.

Another important semantic detail is that the type of any ALI2 wire variable has to be *bound* and not parametric. Thus the following wire declaration

```
d3    : polywire;
```

is illegal since it declares *d3* to be a wire variable of a parametric type without giving actual parameters for the formal parameters of the type. Such a wire cannot be created unambiguously (what is the width of *d3*?) and is therefore banned.

Thus wires of parametric types are effectively restricted to appear as formal parameters in cell declarations. In particular, no wire variable other than a formal parameter may be of the type *list*, since it is not a bound type.

7.1. Type checking of wire parameters

By declaring a formal wire parameter to be of parametric type, the user deems acceptable as actual parameters any wires that are of a type derived from that of the formal parameter. These parameters will become bound (i.e., values for the formal parameters in their type definition will be assigned) at run time by inheriting the characteristics of the actual parameters assigned to them at cell instantiation.[†]

The type checking used in the parameter passing mechanism just described allows for a great deal of flexibility. Only certain properties (selected by the user) of the parameters are checked when a wire parameter is passed. All others are inherited by the formal parameter from the actual parameter. As an example, if a cell has the following header

```
cell silly ( top k : polywire );
```

then the following two wires

```
d1  : polywire ( 2*lambda );
d2  : polywire ( 6*lambda );
```

can be passed as actual parameters for *k*. In this example, the layer and signal values of the actual parameters are known to be acceptable at compile time and *k* will inherit the width of the actual parameter of the instantiation which will only be known at run time.

The type *list* is used to declare formal parameters which are aggregates of wires without any further restriction. Thus a cell having the following header

---

[†]The following argument shows that this mechanism will ensure that all wires are bound at run time. Wires accessible at any point in the execution of an ALI2 program must be either (1) declared in the current scope or (2) formal parameters (since no global wires exist in ALI2). In the first case the wires must be of a bound type. In the second case, an actual parameter for each of these wires must have been given when the cell was instantiated or the procedure or function invoked. The actual parameter given must have been bound (by an inductive argument grounded on the fact that wires local to the outermost cell must be bound) and therefore the formal parameter inherited all its unbound characteristics from it and became bound.

```
cell easy ( left n : list );
```

can be called with *any collection of wires* as arguments. AL12 provides a list constructor to permit the user to create wire aggregates to be passed as actual parameters. The expression

```
| x, y, z |
```

means "make a list of each of the wire variables listed and then concatenate these lists in the order given". Because no assignment to variables of type *list* is possible in AL12, this constructor can only be used to build actual parameters for cell instantiations.

The arguments of a list constructor may be wire aggregates such as bus or bundle variables. They are converted into lists by (recursively) taking the elements of the bundle in the order given by its index type and taking the fields of the bus in the order listed in its type declaration.

The constant *nullwire* can be passed as an actual parameter for any formal parameter of a simple type. When used in a list constructor it will be treated as if it had not been listed. Thus, *nullwire* will never be an element of a list. That is,

```
| nullwire | = nulllist   and   | a, nullwire, b | = | a, b |
```

The use of *nullwire* and *nulllist* as actual parameters will be a common phenomenon in AL12, as we will see shortly.

An aside on the general principles guiding type checking in AL12 is perhaps in order. First, notice that since we translate AL12 into Pascal, we are at the mercy of the underlying Pascal compiler for run time type checking on Pascal objects. Second, type checking on wires is restricted to parameter passing, since no assignment to these variables can ever be made.

There are at least two sensible approaches that could have been taken to type checking on wires. The first one -- *strict* checking -- is to require that the formal and actual parameters *be defined by the same type identifier in order to be compatible.* The second one -- the *lenient* method -- requires that the type of the actual parameter be *identical to or derived from* (i.e. defined directly or indirectly in terms of) the type of the formal parameter.

For instance, consider the following declarations:

```
wiretype
     polywire ( w : integer ) = wire ( poly, w, nullsignal );
     poly5 = polywire ( 5 );
wirevar
     pw : polywire ( 5 );
     pw5 : poly5;
cell silly ( top t : polywire );
     begin ... end;
```

If strict checking is performed, *pw5* cannot be an actual parameter of *silly* since the type identifier of *pw5*, namely *poly5*, differs from the type identifier of the formal parameter *t*, namely *polywire*. In the case of lenient checking, *pw* could be passed as a parameter to *silly* since its type has been derived from that of the formal parameter.

AL12 uses the *lenient* approach. The experience with the original ALI language (which used the strict method) convinced us that the extra flexibility afforded by this approach would be helpful. For example, any component of a *list* could be passed as an actual parameter matching a formal parameter of type *wire*.

This conflict is similar to the Pascal issue -- unresolved in the Pascal Report [9] -- of whether the following declarations

```
type tp1 = integer; tp2 = tp1;
var  v1 : tp1;  v2 : tp2;
```

create two variables of the same type or not. Most compilers seem to resolve the matter in a lenient manner by assuming that they do.

7.2. Accessing components of aggregate wire types

The syntax for accessing components of bus, bundle and list variables is given in fig. 7.

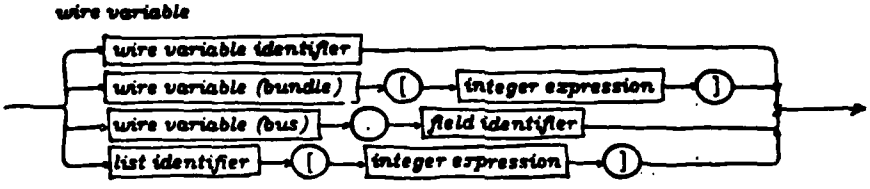


Fig. 7
The syntax for accessing components of aggregate wire types

AL12 simply borrows the Pascal notation for accessing elements of *records* and *arrays* and extends it to *buses* and *bundles* respectively. Here are some examples that use the variable declarations of section 7 to generate legal wire variable names.

```
w1.d1
w1.d1[14]
w2.f1
w3[39]
w4[25][35]
w5
```

In addition AL12 provides two functions that extract the low and high bound of any variable of type *bundle*. This is necessary because *bundles* -- unlike Pascal arrays -- may have sizes that are determined at run time. Thus, given the variable definitions of section 7 and the type definitions of section 6.1,

```
lowindex ( w3 ) = 1
highindex ( w3 ) = 100
```

Accessing of elements of a variable of type *list* is done via a notation similar to array indexing. For instance, if *z* is a variable of type *list*, then

```
z [ 10 ]
```

denotes the tenth element of *z* (the first element is accessed by *z [1]*). All elements of a *list* variable are simple wires.

The value used to index the list can be an arbitrary integer expression which has a value between one and the length of the list. The length of a list may be obtained as follows: if *l* is a list variable,

lengthof (l)

will return an integer whose value is the number of elements in *l*.

8. Procedures and functions

The syntax of the procedure and function declarations in ALI2 is shown in fig. 8.

As these diagrams show, the syntax is identical to that of standard Pascal with additions to (1) permit passing wires and cells as parameters, (2) allow declaration of cells local to the procedure or function, and and (3) allow for separate compilation facilities. Note that functions cannot return wires or boxes as their result.

A fact only partially explicit in the syntax diagrams is that the only wires and boxes accessible inside a procedure or a function are those that are passed in as parameters: the *restricted block* syntax prevents the declaration of wires local to the procedure or function, and no global wires exist in ALI2.

Non-wire parameters may be passed using any of the standard Pascal parameter passing mechanisms (i.e., by reference or by value). Parameters of wire types will always be assumed to be passed *by reference*, so no local copy of an actual parameter is ever created. This parameter passing mechanism for wires and boxes is intended, once again, to preserve the strict hierarchical nature of ALI2 layouts.

The directives *forward* and *external* for procedures and functions work exactly like those of cells described in section 6.2.

9. ALI2 statements

There are only four ALI2 statements that are not Pascal statements. One of them is used to instantiate cell definitions, two others to generate placement information and one to indicate to ALI2 lack of concern about relative placement of pairs of objects. Their syntax is described in fig 9.

In addition, ALI2 has a facility for naming bounding boxes created during execution of a program. The syntax of this facility is also shown in fig. 9.

The rest of this section is divided into five subsections discussing in detail each of these statements and the box naming facility.

9.1. The "ordered" statement

The syntax of the ordered statement should be obvious from the syntax diagram; its semantics however are peculiar to ALI2 and require some elaboration.

During execution of an ALI2 program, the run time system maintains an internal record of the *current bounding box* (which may or may not be associated to a box variable). When the first statement of a cell definition is executed during an instantiation, the current bounding box will become the boundary of the cell instance, and when the instantiation is completed the current bounding box will be whatever it was before the beginning of the instantiation.

Bounding boxes are created automatically on entry to cells or *ordered* statements or by

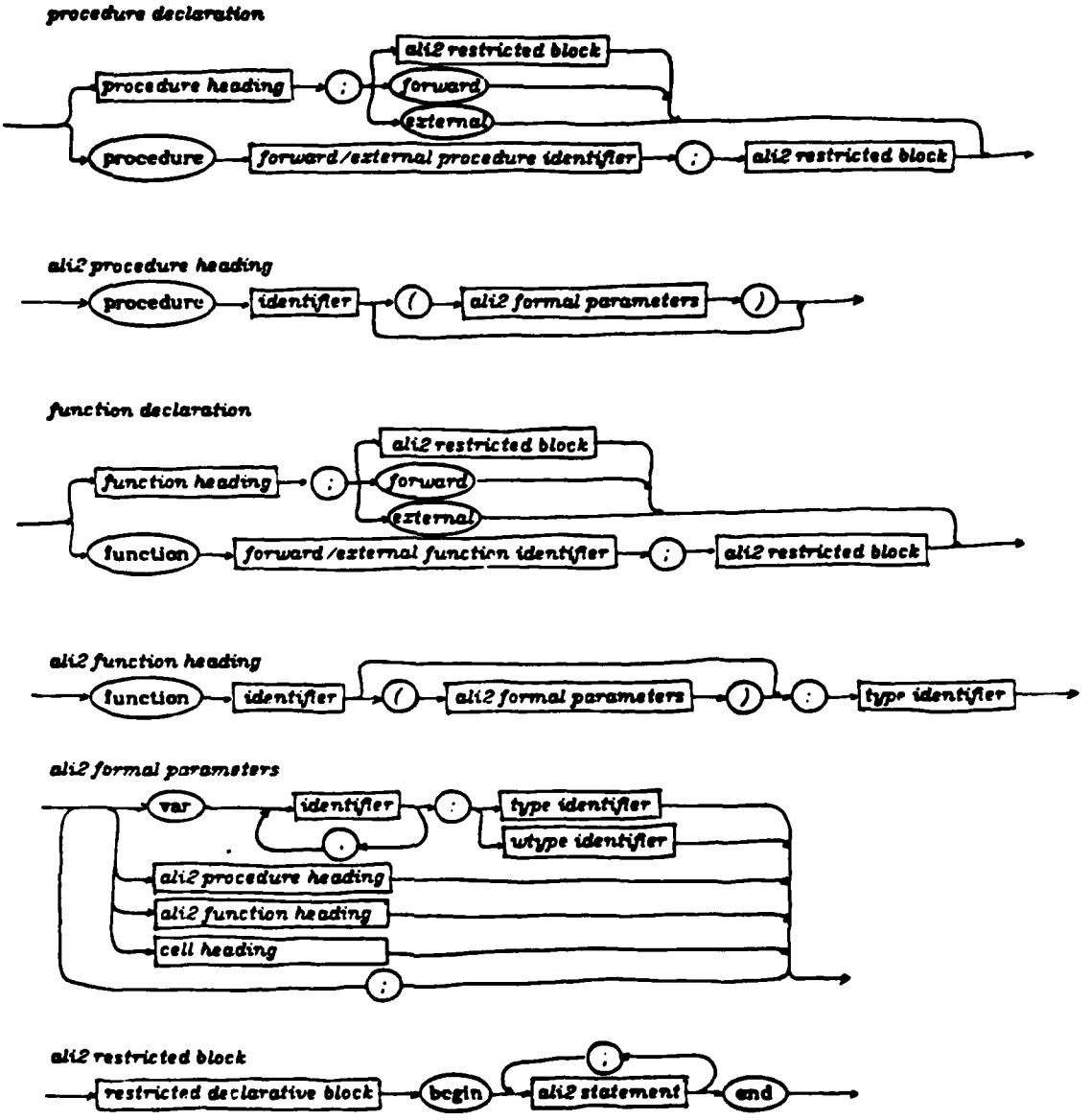


Fig. 8
The syntax of procedures and function declarations

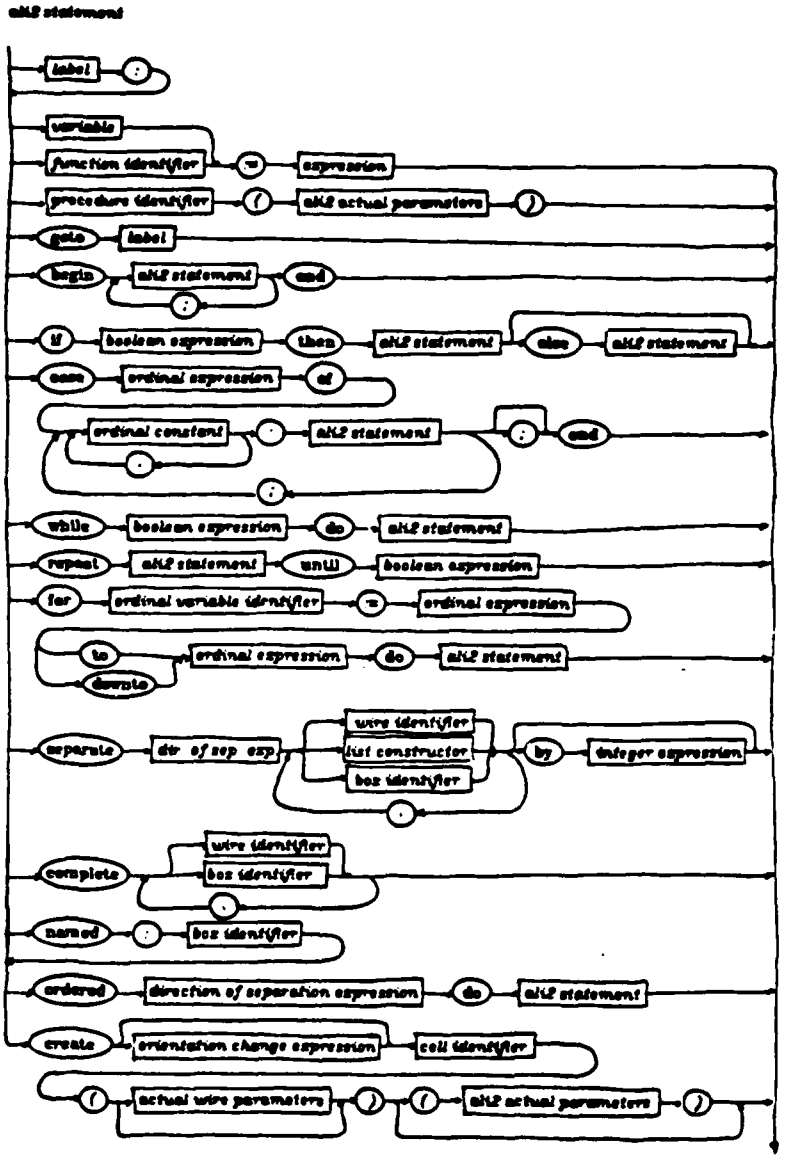


Fig. 9
The syntax of the ALI2 statements

explicit invocation by the user of a predefined procedure (see section 10 for details).

Any two bounding boxes will either be disjoint or one will contain the other. The execution of an AL12 program therefore creates a tree of bounding boxes, with its root being the bounding box for the cell instance which is the whole layout. In this tree the bounding boxes represented by the descendents of a node *v* will be contained in the bounding box represented by *v*.

The *ordered* statement uses the direction of separation given in its header to place all bounding boxes created in its scope *in the order in which they are created* along the direction given. These boxes will be automatically separated along the direction of separation by an amount equal to the minimum specified for their layers in a *design rule table* available to AL12.

An example of a program and the arrangement of boxes it generates is given in fig. 10 below.

```
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
```

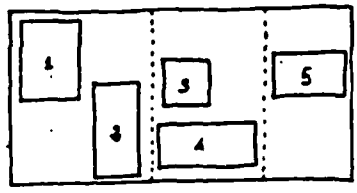


Fig. 10
A program fragment and the arrangement of bounding boxes it produces

9.2. The "separate" statement

The statement

separate *ltor box1, wire1, wire2, wire3, box2, box3;*

will force *box1* to be to the left of *wire1*, *wire1* to be to the left of *wire2*, *wire2* to the left of *wire3* etc. The sizes of the separations that the statement will force between the objects listed will of course depend on their types and will be obtained from the design rule table. AL12 will guarantee that any two objects listed will be separated by the minimum separation required by their layer along the direction specified.[1]

If the statement has the form

separate *ltor box1, wire1, wire2, wire3, box2, box3* **by** *2*lambda;*

then the objects would have been separated by the amount specified. This amount can be an arbitrary expression. If the value of the expression is less than the minimum separation for any two objects in the list, a warning will be issued.

[1] This involves some subtlety because the design rules may not be transitive, i.e., in the example, the layers of *wire1*, *wire2* and *wire3* may be such that separating *wire1* from *wire2* and separating *wire2* from *wire3* does not imply that *wire1* is sufficiently separated from *wire3*.

When a composite object appears in the list of arguments to this command, it is interpreted as if all its simple components had been listed in their natural order (low index to high index for bundles, the order in which the fields are listed in the declaration for buses and the obvious order for lists, all applied recursively).

The direction of separation is interpreted in the obvious way if it is horizontal (*ltor* or *rtol*) or vertical (*ttob* or *btot*). Diagonal directions of separation are interpreted as the conjunction of one horizontal and one vertical direction. The value *nulldir* is meaningless in this statement and its use will produce an error.

Because wire endpoints must always be at cell boundaries, it is meaningless to separate horizontal wires in a horizontal direction or vertical wires in a vertical direction (hence no wires can be separated diagonally). This implies that any wire having *nullorient* as orientation listed in a *separate* statement will be assigned either *vertical* or *horizontal* as orientation.

9.3. The "complete" command[†]

The AL12 system will guarantee that the layouts produced with it are free from design violations without using the standard design rule checking process. In order to do this, the system checks the "logical completeness" of the layout description [10]. This amounts to guaranteeing that any two objects at the same level in the hierarchy of bounding boxes of a layout are (1) either separated (explicitly or through transitivity) by an amount greater than or equal to the minimum separation for their layers prescribed by the design rule table, or (2) explicitly stated to be allowed to be at any distance whatsoever.

The *complete* statement states that its arguments can be as close as necessary without causing any problem. The syntax of the statement is quite simple. For instance,

complete *b1, b2, w1, w2, b3;*

expresses that the user does not care about the separation between any two of the objects listed after the word "complete". If a composite wire variable is given as argument, the above rules are applied to each of its components.

9.4. Cell instantiation

The general syntax of the cell instantiation statement was given in fig. 9; the syntax for the parameter lists of cell instantiations are given in fig. 11. Here are some examples of syntactically correct cell instantiation statements.

create *flipped90 shift (data1, clk, data2);*
create *register (d1, clks, d2) (23);*
create *rotated180 doit ( ) (kk);*

The change in orientation that the user may specify when instantiating a cell is used as follows. The AL12 run time system keeps at all times a *current global orientation*. Calls to certain predefined AL12 procedures have an effect that depends on the current value of the global orientation. By adding a change of orientation to the cell instantiation statement the user specifies that *within the scope of the cell instance, the current orientation should be the dihedral group product of the orientation before the statement and the dihedral group element specified in the statement.* The

[†]The completeness checks have not been implemented yet

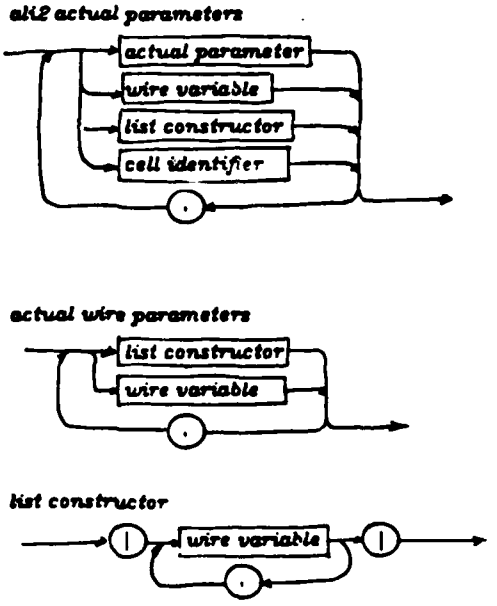


Fig. 11
The syntax of the actual parameter lists
for a cell instantiation statement

global orientation after the statement is completed reverts to its previous value.

Certain placement constraints are automatically generated by the system during cell instantiation.

-1- The actual parameters of the cell are separated: the *top* and *bottom* parameters from left to right and the *left* and *right* parameters from top to bottom.

-2- Cell instances that share an actual parameter (i.e., are connected by a wire) are separated by the minimum separation given in the design rule table for objects having *virtual* as their layer. (This separation can be explicitly overrided as explained in section 10).

The parameter type checking performed for the instantiation is the following. On the wire parameters (those in the first list of arguments), *the type of each actual parameter must be identical to or derived from the type of the corresponding formal parameter.* On the second list of parameters the checking is identical to that performed by Pascal.

Formal wire parameters exist only inside the cell instance, actual parameters exist only outside the cell instance and they are abutted at the cell boundary. A pictorial representation of this is shown in fig. 12.

Note that in the case of cells declared *rigid*, this connection may not be feasible: conditions outside the cell force two actual parameters to be farther apart than the corresponding formal parameters in the cell definition. In that case AL12 will generate an error message during the placement phase [5].

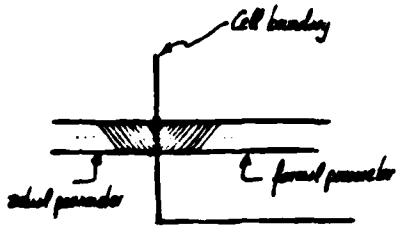


Fig. 12
How wires are passed as cell parameters

This parameter passing schema has the disadvantage of increasing the total number of elements in a layout by oftentimes dividing what at first sight may appear to be one long wire into several small pieces. It guarantees, however, a strict hierarchy among wires of ALI2 layouts, by making it impossible for any of them to straddle a cell boundary.

The parameter passing mechanism for simple wires is extended to composite objects in the obvious manner. In any given instance, the formal parameter will be a composite object having the same general structure and identical components as the corresponding actual parameter.[1] The two objects will be connected by the appropriate method in a component by component manner.

The predefined constants *nullwire* and *nulllist* will be used often as actual parameters in cell instantiation statements. Specifying *nullwire* as an actual parameter in a cell instantiation simply means that the formal parameter will take the value *nullwire*. The use of *nulllist* as an actual parameter for a formal parameter means that the formal parameter becomes a list with no elements. The main purpose of these conventions is to avoid forcing the user to declare wires for the exclusive purpose of passing them as actual parameters to satisfy the type checking mechanisms. This possibility is not extended to formal parameters of the other two composite types: no way of passing a *bundle* or *bus* of dummies is available in ALI2.

9.5. Name tags for bounding boxes

ALI2 contains a facility to attach the name of a box variable to any of the bounding boxes created automatically during execution of a program. The association is performed by tagging the statement that creates the box with the keyword *named* and the box name. The syntax of such a tag is described in fig. 9, and some examples of its use are given below.

```
named n : create flipped90 shift ( data1, clk, data2 )
named it : create rotated180 doit ( ) ( kk )
named newcontext : ordered ltor do ...
```

This facility is akin to Pascal labels except that only statements that create a bounding box can be tagged.

---

[1] In the case of *bit* parameters and *bundle* parameters with parametric bounds this mechanism implies that copies of the actual parameters will have to be made at run time

10. Predefined procedures and functions

We will now describe the predefined procedures and functions of ALI2. They, together with the predefined cells described in the next section, represent the user-visible part of the ALI2 run time system.

The descriptions given here are for general users. Implementation details of the run time system can be found in [11].

---

procedure *crossing (w : wire);*

This procedure informs the ALI2 run time system that the given argument is to appear after the last bounding box created and before the next bounding box along the direction of separation. The argument is enclosed in a pseudo box in order to separate it from other bounding boxes created in the same scope. It is assumed that the current direction of separation is either horizontal (in which case *w* must be a vertical wire) or vertical (*w* must be horizontal).

---

procedure *override (b1, b2 : "any wire or box"; d : directionofseparation; k : integer);*

The purpose of this function is to override the automatic separation features of ALI2 and as such it should be used sparingly. The function has a header that cannot be expressed entirely in ALI2 since it takes as arguments either simple wires or boxes in any combination.

Its behavior is the following. Whatever separation may have been specified for *b1* and *b2* in the direction given by the third argument is replaced by the one specified as the fourth argument. Additionally, completeness checks between the first two arguments will be inhibited.

It should be noted that later separations – generated explicitly by the user or automatically by the system – may undo the effect of this statement. Thus, for it to work properly the user must be somewhat familiar with the inner workings of ALI2.

{ Not yet implemented }
procedure *ckcompleteness (b : "cell bounding box");*

This procedure instructs ALI2 to perform a completeness check on the constraints generated at the top level of the cell instance given as an argument. This check will be performed as soon as the instance is completed during the running of the ALI2 program. Appropriate messages will inform the user of the result. The procedure should be invoked *before* the cell instance is created.

---

function *widthof (w : wire) : integer;*

function *layerof (w : wire) : layer;*

function *signalof (w : wire) : signal;*

function *orientationof (w : wire) : wireorientation;*

The values computed by these functions are the obvious ones. Remember that the width will be in hundredths of microns and that multiplication by *lambda* is needed to convert it to λ units.

---

function *minwidth (k : layer) : integer;*

function *minseparation (k1, k2 : layer) : integer;*

These functions access the design rule tables. The first one gives the minimum thickness of a wire on the specified layer, and the second the minimum distance between objects in the two layers given as arguments such that they will not interact electrically.

---

function *lengthof (z : list) : integer;*

The value returned by *lengthof* is the number of elements in its argument.

---

function *lowindex (z : "any bundle type") : integer;*

function *highindex (z : "any bundle type") : integer;*

These functions compute, respectively, the index of the first element of the bundle and the index of the last element of the bundle. Note that the headers cannot be expressed entirely in ALI2.

11. Predefined cells

There are only four predefined cells in ALI2. All of them are "smart" in that they do a large amount of processing. All of them are "not very smart" in that they will insist that the problems presented to them have a solution with the center lines of all the wires given as parameters meeting at a point. If no such solution is possible -- or if the cell cannot find it -- an error message will result.

Given below are the headers of all four cells and short descriptions of what they do. Note that all four are quite general, and that most users will want to define simpler versions of them to facilitate their repeated invocation.

```
cell sysetransistor ( left gateleft : list;
                      top source : list;
                      right gateright : list;
                      bottom drain : list )
                    ( implanted : boolean );
```

This cell constructs an enhancement mode transistor. The following conditions have to be satisfied by the parameters of this cell. The lists passed as *source* and *drain* have to contain at least one diffusion wire and at least one of the lists *gateleft* and *gateright* must include a polysilicon wire. Each of the lists must contain wires that run on electrically independent layers.[1]

Instantiation of this cell results in the following actions. The source of the transistor will be the diffusion wire in *source*, its drain the diffusion wire in *drain* and its gate the polysilicon wires in *gateleft* and *gateright*, which will be connected together if both exist. The dimensions of the transistor are determined by the maximum of the widths of *source* and *drain* and the minimum of the widths of *gateleft* and *gateright*. The parameter *implanted* tells whether the transistor should be implanted. Any wire parameters having layers other than polysilicon or diffusion will be electrically connected to one another over the transistor. Such a connection will have no electrical interaction with the transistor. If such a connection cannot be built, an appropriate error will be generated.

```
cell syspullup ( left drainleft : list;
                 top source : list;
                 right drainright : list;
                 bottom drainbottom : list )
               ( ratio : integer );
```

This cell constructs a pull-up (depletion mode) transistor. The conditions on the parameters are the following. The *source* must contain at least a diffusion wire and the *drain* at least one wire of any type. Each of the parameters must be composed of wires running on electrically independent layers.

The effect of instantiating this cell is the following. A depletion mode transistor having the specified *ratio* will be created. Its source will be the diffusion wire which is part of *source* and all other wires will be connected to the drain of the transistor. If such a construction is not feasible, an appropriate error message will be generated.

```
cell syscontact ( left l : list;
                  top t : list;
                  right r : list;
                  bottom b : list )
                ( layerbylayer : boolean );
```

[1] "Independent layers" refer to layers whose wires can overlap without an electrical interaction occurring, such as metal and polysilicon in nMOS. From the point of view of ALI2, layers whose minimum separation on the design rule table is zero are assumed to be independent.

This cell generates a contact that electrically connects the given wires. The only conditions on the parameters are two: (1) that they contain at least one wire among them and (2) that it may be possible to connect all the wires in them so that their centerlines meet at a point.

The effect of instantiating the cell is as follows. A region that makes the component wires electrically connected will be created; if the boolean argument is true, the connection will be done so that *wires on independent layers will be connected to each other but not to wires on other layers.* If the connection is not possible -- either in a true sense or because of implementation limitations -- an error will be generated.

```
cell syscrossover ( left   l : wire;
                    top    t : wire;
                    right  r : wire;
                    bottom b : wire );
```

This effect of the cell is to connect the top and bottom wires and the left and right wires separately, with the left-right pair crossing over the top-bottom pair. No constraints are imposed on the parameters of this cell beyond those explicit in the header. Of course, the layers of the wires will in general require that contacts be created to change layers, so that the finished size of the cell may end up being rather large.

12. Final comments

We will close this document by saying a few words about how we view the task of programming in ALI2.

12.1. Conceptual framework

A physical layout is composed exclusively of wires. Thus, wires are the most prominent objects in ALI2. For the purpose of organizing wires into a hierarchy, rectangular boxes have been introduced.

Certain wire arrangements appear very often in VLSI layouts. Although the outward appearance of these arrangements varies widely, they can be classified into a few groups according to their function. ALI2 forces its user to generate these arrangements by calling predefined cells, which try to hide the varied appearances but leave the function visible.

After the arrangements of interacting wires are hidden, a layout becomes simply a matter of routing wires. Until the ALI2 library of routing aids has been completed, wire routing must be handled by the user. The composite types are intended to make this task somewhat simpler: routing aggregates of wires, rather than routing each wire individually, helps to reduce the routing effort. The composite and parametric wire types represent our best efforts to balance the need for flexibility in the type checking with the need for as many consistency checks as possible.

13. Hierarchies in ALI2 programs

An ALI2 program is an object in which several hierarchies coexist. Two of them are familiar to any Pascal programmer: the compile time hierarchy, in which objects (wires, Pascal-like variables, functions, cells, etc.) are defined in terms of one another in a hierarchical fashion, and the run time hierarchy determined in the case of ALI2 by the procedure, function and cell

invocations. The properties of these two hierarchies in ALI2 are almost identical to those of the corresponding hierarchies in many programming languages. The few differences (i.e., the local character of wires) need no further comment.

ALI2 was designed so that programs will, in a natural manner, produce highly hierarchical layouts as output. It is this *output hierarchy*, absent from conventional programs, that is peculiar to ALI2. Because the properties of this hierarchy are unlike those of the more familiar ones, the rest of this section describes them in some detail.

The output hierarchy is determined by the execution of two ALI2 statements: *create* and *ordered*. Each of these statements creates a new *bounding box* in which local layout elements – including other bounding boxes – are enclosed. They define a hierarchy which gives the layout its structure.

The bounding boxes created by each of the two statements differ in a crucial way: while wires may not straddle the boundary of a box generated by a *create* statement (see fig. 11), the same is not true of the boundary of boxes generated by the execution of *ordered* statements. Thus, wires are subject only to the hierarchy defined by cell boundaries.

Note that the box hierarchy is quite different from the run time hierarchy. For instance, wires that are local to a procedure will be inside the current bounding box at the time the procedure is invoked. Two different invocations may produce wires that are in the same box.

All bounding boxes are similar, however, in that they represent a local *context* for the ALI2 programmer. Each box has a *local orientation* with respect to that of the box containing the overall layout, and a *local direction of separation*. The way in which these values are derived for a new bounding box is as follows:

-1- *Create statements.* The local orientation is the (dihedral group) product of the local orientation of the current bounding box at the time the statement is executed and the orientation change specified in the statement. The local direction of separation is *nulldir*.

-2- *Ordered statements.* The local orientation is the same as that of the current bounding box when the statement is executed. The local direction of separation is the one specified in the statement.

Note that the direction of separation inside a given box *is relative to the local orientation*. Therefore if a bounding box has a local orientation which is rotated ninety degrees with respect to the orientation of the outermost box, and its local direction of separation is *ltor*, the result – from the point of view of the outermost box – is that the box has *btot* as its direction of separation.

14. References

[1] L. Atkinson, *Pascal Programming*, John Wiley and Sons, 1980.

[2] R. J. Lipton et. al, "ALI: A Procedural Language to Describe VLSI Lyouts", *Proceedings of the Nineteenth ACM-IEEE Design Automation Conference*, Las Vegas, Nevada, June 1982.

[3] R. L. Kalin, "The ALI2 Front-End: Design and Implementation of Parser/Translator", *ALI2 Documentation and Implementation Guide.*

[4] J. Valdes, "System Overview", *ALI2 Documentation and Implementation Guide.*

[5] S. C. North and G. Vijayan, "ALI2 Solver", *ALI2 Documentation and Implementation Guide.*

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

[7] J. Hennessy and H. Elmquist, "The Design and Implementation of Parametric Types in Pascal", *Software -- Practice and Experience*, vol. 12, pp 169-184 (1982).

[8] S. C. North and G. Vijayan, "ALI2 Runtime File Formats", *ALI2 Documentation and Implementation Guide.*

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

[10] G. Vijayan, "ALI2 Completeness Checker" *ALI2 Documentation and Implementation Guide.*

[11] J. Mata, J. Valdes and G. Vijayan, "The Translation of ALI2 into Pascal and the ALI2 Runtime system", *ALI2 Documentation and Implementation Guide.*

Appendices

Standard Pascal Documentation

Total Stuck-at-Fault Testing by Circuit Transformation*

Andrea S. LaPaugh
Richard J. Lipton

Department of Electrical Engineering and Computer Science
Princeton University

Abstract

We present a new approach to the production testing of VLSI circuits. By using very structured design for testability, we achieve 100% single stuck-at fault coverage with under 20 test vectors and no search. The approach also detects most multiple faults.

1. Introduction

In this paper we present a new approach to the production testing of VLSI circuits. The major features of this approach are that the set of test vectors is both small (less than twenty) and independent of circuit size and that no search for test vectors is required. These features are achieved while guaranteeing the detection of all single stuck-at faults in MOS circuits along with many other faults. Testing can be done without any special test equipment, allowing field testing of VLSI circuits to be done as well as production testing. Thus, major difficulties of current testing methods -- the need for large and expensive searches for test vectors with high coverage and expensive test equipment to apply large sets of test vectors quickly -- are avoided. In fact, the technique is well suited for implementation on a self-testing chip. Little area is needed to store the test vectors and fault coverage is guaranteed rather than probabilistic as in some self-testing strategies.

Our approach is actually the combination of three techniques which could be used independently. There are various penalties associated with each of the techniques. The most severe of these is the requirement that the circuit be put into a special form. There is a purely mechanical transformation for this purpose. Hence, this requirement does not prolong the design time. Also it does not change the depth of the circuit, hence, the circuit's speed is essentially unchanged. Nor does this transformation add many new pins to the circuit. It does, however, increase the size of the circuit. The gate count of the modified circuit can be as much as, but never more than, double the number of gates of the original. The other two techniques add some extra circuitry which may also increase the chip area needed, but not as substantially, and may also increase the delay slightly. While these increases are potentially costly, for many circuits the tremendous advantages of our method will far outweigh its cost. This is likely the case for gate arrays and other semi-custom logic.

2 Technique I: Bipartite Circuits

Our method is based on a special class of combinational circuits. These circuits are special in that they are easy to test for all stuck-at faults and yet they are able to "simulate" other circuits quite efficiently. The circuits can be composed of any type of inverting logic gates, that is any logic gates whose output is 0 when all inputs are 1's and 1 when all inputs are 0's. This includes, of course, any circuits that consist solely of *nor* and *nand* gates. Initially we will discuss only combinational circuits. Sequential circuits will be addressed later.

2.1. Testability of Bipartite Circuits

A combinational circuit C is *bipartite* provided it is possible to two color its gates *black* and *white* so that no wire connects two gates with the same color and each input wire of the circuit C is an input to gates from only one color class. The coloring of gates in a bipartite circuit also defines a corresponding coloring of wires: black (respectively white) wires are input only to black (respectively white) gates. Then outputs of black gates are white wires, outputs of white gates are black wires.

The importance of bipartite circuits stems from the following which we call the *parity principle*.

Parity Principle: *If we set all the black input wires of a bipartite circuit C to the value 1 and all the white input wires of C to the value 0, then all black (respectively white) wires take on the value 1 (respectively 0).*

This principle is the key to the testability of this class of circuits. From the principle, we have the following theorem about controllability of the output wires of gates.

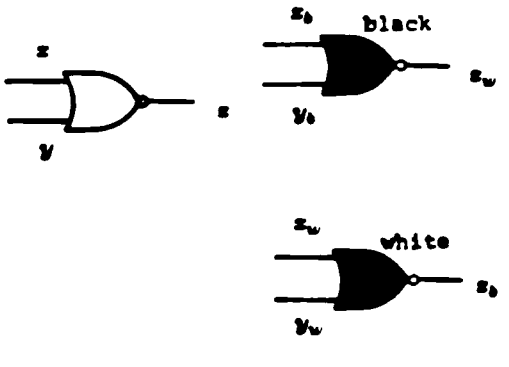
* Supported by DARPA #N00014-82-K-0549

Theorem 1: ***Given a bipartite inverting logic circuit, only two input vectors are required to force the output node of all gates to the values 0 and 1.***

2.2. Universality of Bipartite Circuits

Since it is easy to see that not all circuits are bipartite, we will now show how to transform any combinational circuit into an equivalent bipartite one. This transformation will at most double the number of gates. However, even better, it is a "local" transformation and thus will only increase the area of an integrated circuit layout by a factor of at most four (two in each direction). It will also not increase the depth of the circuit at all. This is our main advantage over past techniques for 100% fault coverage [Ha74, Sa74]. We first present a simple method that always doubles the number of gates. We then discuss avoiding actually doubling the number of gates.

Given any inverting logic combinational circuit, make two copies of each gate of the circuit, label one copy of each gate "white" and one copy "black" (see Figure 1a and 1b). Also, make two copies of each wire between gates: one copy goes from the black copy of the first gate to the white copy of the second gate -- a white wire, the other goes from the white copy of the first gate to the black copy of the second gate -- a black wire. Input wires and output wires are doubled in like fashion so that each black (respectively white) gate has only black (respectively white) input wires and white (respectively black) output wires. The resulting circuit has exactly twice the number of gates of the original and is bipartite. Note, that the fanout and depth of the transformed circuit is the same as that of the original.



(a) original gate (b) transformed "gate"

Figure 1.

The transformation doubles the number of input and output wires, an undesirable effect if the number of pins must also double. The new circuit will simulate the original if the black and white versions of each input agree. For test operation, we want the black and white versions of each input to be complements of each other. To do this with the same number of input pins, we use a special input cell. It is a piece of combinational logic with inputs x and *mode* and outputs x_b and x_w. When *mode* is 0, the input pad drives x_b and x_w so that $x = x_b = x_w$; when *mode* is 1, the input pad drives x_b and x_w so that $x = x_b = \bar{x}_w$.

It is even simpler to save output pads. Since in normal operation the black and white versions of each input wire agree, the black and white versions of each wire in the entire circuit agree. Therefore, only one version of each output wire need be connected to an output pad. By eliminating redundant outputs and any wires and gates that compute results used only by the eliminated outputs, we can obtain a bipartite circuit with less than twice the number of gates. In general, it is not computationally feasible to optimally choose which output of each pair is eliminated so that the resulting circuit is minimum size (it is NP-complete [Ga82]). However, heuristics can be used in an attempt to avoid doubling the number of gates.

3. Technique II: Control of transistor faults

The parity principle for bipartite circuits is a very powerful one. It allows us to easily set all the wires of a circuit to both 0 and 1. If we could examine each wire, say with a scanning electron microscope (SEM) [Ki82], then we could detect any wire stuck-at fault. However, such an approach is not powerful enough to also detect transistor stuck-at faults: a transistor can be stuck without any wire also being stuck. The problem is that our test vectors for a bipartite circuit do not guarantee that all combinations of input values will occur at any gate. In fact, just the opposite is true: the bipartite test strategy guarantees that all "healthy" inputs to a gate will be the *same* value.

Our second technique is the introduction of a controllable gate which when used in a bipartite circuit allows the detection of both wire and transistor stuck-at faults. We must further restrict the type of logic gates used in the circuit to achieve this. We present a controllable *n*MOS *nor* gate which is a modification of the standard *n*MOS *nor* [Me80]. We have also designed a controllable *nand*, but the two cannot be used in the same circuit if our simple test strategy is to be achieved. Thus the original bipartite circuit must contain only *nor* gates or only *nand* gates. (Inverters are achieved by using a *nor* or *nand* with both inputs the same.) The gates are designed so that all stuck-at transistor faults can be forced to occur even assuming the normal inputs to a gate always take on the same value when the circuit is under test.

Given a bipartite circuit of *nor* gates, each gate is replaced by the circuit displayed in Figure 2. We

have added two new global control lines C1 and C2. These are inputs to every gate of the circuit. Clearly, when C1 = C2 = 1 this gate computes the *nor* of x and y; hence, in this case the new circuit simulates the old one. Also, for all settings of $C1$ and $C2$ except both 0, the gate is inverting with respect to the values of x and y. Therefore, a bipartite circuit of such gates will retain the parity principle for all values of $C1$ and $C2$ except both 0. On $C1=C2=0$, the outputs of all gates should be 1.

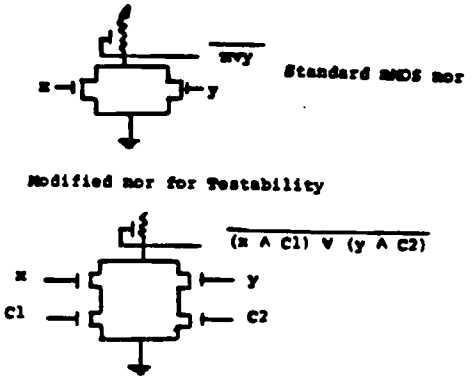


Figure 2.

There are five active transistors in the controllable *nor*. For each individual stuck-at fault for each of these, we give settings for $C1$, $C2$, x, and y with $x=y$ such that this fault causes an incorrect output. We first consider the pulldown transistors

(i) **The transistor controlled by $C1$ (respectively $C2$) is stuck-on**: Set $C1=C2=0$ and $x=y=1$. Since the transistor controlled by C1 (respectively C2) is on, the output is incorrectly pulled down to 0.

(ii) **The transistor controlled by C1 (respectively C2) is stuck-off**: Set $C1=1$, $C2=0$, (respectively $C1=0$, $C2=1$) and $x=y=1$. Since the transistor controlled by C1 (respectively C2) is stuck off, the output is incorrectly pulled up to 1.

(iii) **The transistor controlled by x (respectively y) is stuck-on**: Set $C1=1$, $C2=0$, (respectively $C1=0$, $C2=1$) and $x=y=0$.

(iv) **The transistor controlled by x (respectively y) is stuck off**: Set $C1=1$, $C2=0$, (respectively $C1=0$, $C2=1$) and $x=y=1$.

The pullup transistor is depletion mode with its gate tied to its source. Therefore, if it should always be on. A stuck-off fault can be detected by setting $C1=0$ and $C2=0$. The output should be pulled high but will not be if the pullup is stuck off. Note however that in this case if the fault occurs the output is left floating. Therefore, it must have previously had the value 0 to be sure such a fault is detected. Appropriate ordering of the tests above can insure this.

4. Bipartite circuits with special gates

We now show how to cause any single stuck-at fault to cause an incorrect gate output in a bipartite circuit consisting of controllable *nor* gates. This will give us controllability of all single stuck-at faults. The following table gives the test procedure.

Input Settings				Event Check For	
black inputs	white inputs	C_1	C_2	all white wires	all black wires
1	0	1	0	1	0
0	1	1	0	0	1
1	0	0	1	1	0
0	1	0	1	0	1
1	1	0	0	1	1

Note that three events, i.e. states of all black and white wires, must be detected. How we detect these events is the subject of the next section. If all events are as expected, then we *accept* the circuit; otherwise, we *reject* it. Recall that C1 = C2 = 1 is "normal mode"; in this mode the circuit simulates the original. It is interesting to note that we do not need this setting while we are testing for stuck-at faults.

Theorem 2: *If a bipartite circuit of controllable nor gates is fault free, then it is accepted by the test procedure. If the circuit has a wire or transistor stuck-at fault, then it is rejected as long as there is at most one fault per gate.*

We now wish to tie off one remaining loose end -- faults in the pad cells. First, since we drive the outputs to 0 and 1, any faults in the output pads are easily detected. Thus, it only remains to detect faults in the input cells. The easiest way to handle this is to introduce a fourth event: all black and white input wires are equal to 0. Note, this event only concerns itself with the input wires. Now all possible values of input wires are observed using some event.

5. Technique III: Observing Events

Clearly, if we could probe internal nodes with say a SEM, then events would be easy to detect. However, we wish to avoid using such equipment and hence will add additional logic to the circuit solely to detect the four events we must observe. This additional logic is quite simple and takes up relatively little area. Such observation logic could be added to observe any set of events but the amount of logic needed is proportional to the number of *distinct* events observed. Thus the key to our success is the need to observe only four

different events during our testing sequence.

Any event really consists of two sets of wires: all wires in one set must take on the value 0 and all wires in the other set must take on the value 1. Clearly, we can do this detection simply by using large fan-in *or* gates for the 0-valued set and large fan-in *and* gates for the 1-valued set. The area needed for these gates is quite small. Each gate could be physically distributed through the circuit. As a practical matter, we would use a tree of reasonable size fan-in gates to avoid great delay penalties. It is important to note that the speed of this circuit is not critical since it only affects how fast testing can be done. When only a small number of test vectors are used, as in our method, the total testing time is very small. It is also important to note that if a single fault occurs in the total circuit, we may call the "real" part of the circuit faulty when it is good because of faulty observation logic, but we will not call it good if it is faulty. Many multiple faults will also be detected. In particular, a fault in a transistor of a large fan-in gate will mask a fault in the "real" circuit only if the fault in the "real" circuit must be detected through the bad transistor.

6. Sequential Circuits

We have presented our method for combinational circuits. Of course, it can be used with scan path to test sequential circuits. It is also possible to use our method to directly test sequential circuits without any scan path. Just as we require a modified gate and input pad, this extension requires a modified latch. With this latch, state setting can be done in one step rather than by shifting. Thus even for sequential circuits, our method requires constant time.

7. Conclusions

We have presented a new way to do production testing of MOS combinational circuits which is the combination of three techniques: the use of bipartite circuits, the use of a controllable gate, and the use of observation logic to detect a small number of events. This approach trades off real estate for easy of testability. A critical question that must be studied in the future is this tradeoff. Since increased real estate decreases yield, our tradeoff can be put another way: is it better to have a high yield circuit with low fault coverage or a lower yield circuit with high fault coverage? We claim no definitive answer here, but believe in the area of low volume circuits such as gate arrays our methods may be quite important.

The seriousness of the real estate/testability tradeoff is dependent on the actual real estate increase when our method is applied to actual circuits. There are classes of circuits which are naturally bipartite, for example PLA's and precharged (dom[illegible] CMOS [Ho83]. The implementational details of the modified gates and circuitry for observation are also important in determining the costs of our technique. We have focussed here on the basic techniques. Some work has been done on more efficient implementations, but optimal design of the test circuitry remains an important issue. Another important area for further investigation is the effect of our approach on the placement and routing of gate arrays. Since our transformation is local, we expect that the usual placement and routing algorithms can be modified to take advantage of this.

To control the cost of our method, it may be possible to give the logic designer more feedback during the design process. One of the nice features of our method is that the test for being bipartite is "linear time" and so can be done very quickly. Thus, we can constantly advise the designer on the current "cost" in extra gates of his design. In this way the designer will perhaps be able to make intelligent choices about logic alternatives in a way that will aid the circuit's testability.

Acknowledgements

We would like to thank Stuart Daniels and Ken Anderson of Siemens Corporation for essential comments during the development of this method.

References

[Be82] Beresford, R., "Technology Update: Semiconductors," *Electronics*, Vol. 55, No. 21, Oct. 20, 1982, pp. 118-125.

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

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

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

[Ho83] Hodges, D., Jackson, H., *Analysis and Design of Digital Integrated Circuits*, McGraw-H[illegible] (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.

[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 Transactions on Computers*, C-23, No. 5, May 1974, pp. 552-554.

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

A Massive Memory Machine Architecture

Hector Garcia-Molina
Richard J. Lipton
Jacobo Valdes
Department of Electrical Engineering and Computer Science
Princeton University

1. The basic idea

This paper argues the case for a machine with *billions* of bytes of primary storage. Our main thesis is that such a machine is justified by the importance of certain applications in which *memory bound computations* occur naturally: VLSI design, AI and data bases, to name just three. For these computations, a classic Von Neumann machine with a relatively slow (1 to 10 MIPS) processor and massive amounts of physical memory, would vastly outperform all supercomputers currently being researched and would be, in addition, far easier to program.

2. Impact of proposed supercomputers on memory bound computations

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.

There are many applications for which such a machine would be limited by its disk to memory transfer rates. For example, consider a database application in which the size of the data is four gigabytes and the access pattern to records essentially random. A machine with one hundred megabytes of memory can be expected to generate a page fault in just about *every* access to a new record, 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 lets us assume that the supercomputer is "infinitely fast" while the MMM runs only at one MIP. Of course the supercomputer will vastly out perform the MMM on compute bound tasks. However, assume that the supercomputer creates a page fault every f instructions on some large task. Then on this task the MMM still computes at its one MIP rate while the supercomputer is reduced to computing at about $100f$ instructions a second (we assume that the supercomputer's disks are capable of about *100* page faults 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.

3. An obvious solution?

One can argue that the MMM problem has not been investigated because it has an obvious solution, namely connecting all the memory desired to the chosen processor by a very long bus. This is, however, a far more problematic proposition than it seems. 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.

Commercial busses are simply unequal to the task. Those that are part of complete computer systems are usually very well matched to the overall system design and can only support an

inadequate amount of memory. Standard busses hardly fare any better; most have a limit on the number of devices attached to them of only a few tens, while we need an order of magnitude more.

The design of a special purpose bus to support that many devices is no trivial matter either. There are two factors that will adversely affect the access times on a bus with many devices. First, the capacitance effects may cause significant delays. Second, it may be virtually impossible to operate all the devices synchronously: The asynchrony will in turn lead to more complex bus protocols and, once again, to greater access times.

The truth is that the "obvious" way to build a MMM dissolves rapidly into a host of difficult questions about bus behavior and machine architecture.

4. Architectural solutions to the MMM problem

We will now describe in some detail an architecture for a MMM called the *Ghost Machine*. Our intention is not so much to present a "conclusive solution", but to demonstrate that clever architectures may lead to better MMMs than brute force methods.

A schematic description of the *Ghost Machine* is shown in fig. 1.

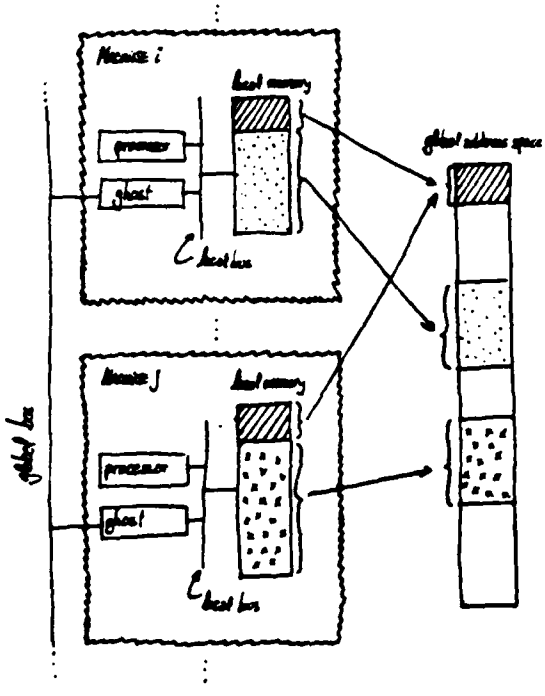


Fig. 1: *The Ghost Machine*

The machine 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 bus. The gateway of each machine to the global bus is a *ghost* device connected both to the

system bus and the local bus.

The individual processors share the same address space. This address space is distributed among the local address spaces as follows (see fig. 1). 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. A *ghost* device connected to each local bus is responsible for servicing requests that involve non local addresses.

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.

Suppose now that a reference to an address outside the shared subspace occurs for the first time. The address involved will be mapped to the local memory of some machine m, so the processor of that machine gets an immediate response via its local bus; the *ghost* of m – realizing that a reference to a non shared address has occurred – reads the response off the bus and broadcasts it over the global bus; The *ghosts* of all the other machines – realizing that a non local reference has been generated – wait *for the next datum* to appear on the global bus and use it as a response to their requests.

During this operation, processor m "takes the lead", i.e., gets ahead of all others. It will continue execution ahead of the rest as long as the common program generated requests for addresses that are local to m. Meanwhile, all other processes continue execution at the same rate as m, with their *ghosts* supplying the data they need by reading it from the global bus. These "trailing" processors, will be behind the leader by an amount of time equal to the one-way delay time from *ghost* to *ghost* through the global bus.

When a reference to an address local to another machine, n, occurs, the *ghost* of m will wait until the next data arrives on the global bus (sent by n's *ghost*) and use it as the response. Now – as long as references local to n continue to occur – all machines will execute at their full rate with n slightly ahead of the rest and furnishing them with the data they need through the global bus.

A simple example of this behavior is depicted in fig. 2 below.

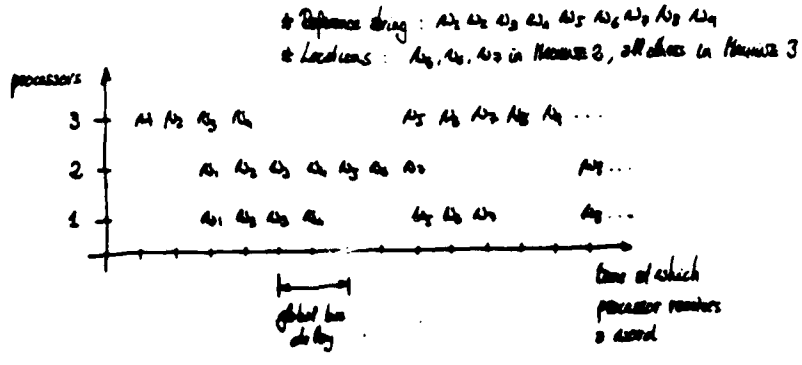


Fig. 2: *Execution in a Ghost Machine*

This solution has the following advantages:

-1- The local machines are conventional architectures. They may be used independently when the MMM is not needed.

-2- The global architecture can be easily made transparent to the user program and the task of distributing the global address space to the spaces of the individual machines can be relegated to a sophisticated loader. The *exact same* programs could run on a conventional Von Neumann machine (assuming some virtual memory management) and in the *Ghost Machine*. Hence the *Ghost machine* will be no harder to program than a conventional machine.

-3- Memory references in the *Ghost machine* can be serviced in half the time (or less) that they would require in a conventional MMM. In a conventional machine, the address must be transmitted on the global bus and the referenced datum must be transmitted back. In a *Ghost machine* data will either be available through the faster local bus or be provided by the *ghost*. The data needed by the *ghost* appears without being requested, avoiding address transmission delays. If the requested word causes a "lead change", the word will be available in approximately the time needed to broadcast a value over the global bus. If no lead change occurs, the delays will be even shorter.

-4- The rate of execution – except for pauses for "lead changes" – of the whole machine is the same as that of each of the individual machines. The "lead change" pauses will last approximately as much as the time needed to broadcast a value on the global bus. This time should be much smaller than the time required to service a page fault on an ordinary machine. (Lead changes will also be considerably less frequent than page faults).

-5- The *Ghost machine* will reward "locality of reference" by minimizing "lead changes" in programs that exhibit it (the fewer the lead changes, the faster the *Ghost machine* will execute). Locality in this context, however, has a wider meaning, as any two references local to the lead machine are equivalent, and any one machine may have as much as sixty megabytes of local storage.

Obviously, these gains come in exchange for duplicated processors and memory.

TESTABILITY CONDITIONS FOR BILATERAL ARRAYS OF COMBINATIONAL CELLS [†]

Anastasios Vergis and *Kenneth Steiglitz*

Department of Electrical Engineering and Computer Science
Princeton University
Princeton, NJ 08544

ABSTRACT

Two new sets of conditions are derived that make one-dimensional bilateral arrays of combinational cells testable for single faults. The test sequences are preset and, in the worst case, grow quadratically with the size of the array. The basic cell can operate either 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. A straightforward generalization for the two-dimensional case is made, a systolic array used for matrix multiplication is presented as an example for this case.

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. Digital systems of iterative arrays have also been suggested in the past in several places in the literature, (references can be found in [2]-[7]) mainly for realization of Boolean functions in asynchronous mode. 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. Fault detection in

[†] 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.

maps the one-dimensional synchronous bilateral array into a two-dimensional asynchronous unilateral array. Figure 2b shows the inputs and outputs of a cell in the space-time transformation according to the notation described above.

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

We also define $g_R(r_1/r_2, z, l) = r'_1/r'_2$ if $g_R(r_1, z, l) = r'_1$ and $g_R(r_2, z, l) = r'_2$ *and* $r_1 \neq r_2$, $r'_1 \neq r'_2$. We define similarly $g_R(r, z, l_1/l_2) = l'_1/l'_2$, and similarly for g_l.

Another definition is: $g_R(r/R_0, z, l) = r'/R'_0$ if $g_R(r, z, l) = r'$, and $g_R(R_0, z, l) \subseteq R'_0$. Similarly for $g_R(r, z, l/L_0)$ and for g_L. (Note that $g_R(r/R_0, z, l)$ is not uniquely defined, since R'_0 may be any superset of $g_R(R_0, z, l)$.)

For simplicity r/R_r will sometimes be written as $r/*$.

3. ONE-DIMENSIONAL ARRAYS

Gray and Thompson [9] derived the following sufficient condition G for testability:

There exists an $a \in Z$ such that for every $r \in R$, for every $l \in L$

G1: $g_R(r, a, l) = \mu(r)$ and

G2: $g_L(r, a, l) = \nu(l)$

where μ is a permutation of R (independent of l) and ν is a permutation of L (independent of r).

Note that the right output depends only on the left input, and the left output depends only on the right input. Fig. 3 illustrates the above condition.

Let V be the following conditions:

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

unilateral arrays has been studied extensively [1]-[7], [10]. Results for bilateral arrays have appeared in Gray and Thompson [9] (for one-dimensional arrays of combinational cells) and Sung [8] (for two-dimensional arrays of sequential cells). However, the sufficient conditions for testability derived there appear more restrictive than necessary. In this paper we first examine one-dimensional arrays; two sets of sufficient conditions for testability are derived, which improve upon the condition reported in [9]. Testing time, however, in the worst case increases from linear to quadratic (in the number of cells). A straightforward generalization to the two-dimensional case is made next.

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 p be the total number of cells in the array.

Let R be the set of right-moving signals, L the set of left-moving signals and Z the set of vertical cell inputs. (The absence of vertical cell inputs is equivalent to the case $|Z|=1$.)

Let $g_R: R \times Z \times L \rightarrow R$ be the right-moving signal mapping, and $g_L: R \times Z \times L \rightarrow L$ 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, z, 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. 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.

We say that an array is *testable* if any input combination can be applied to any cell of the array and any fault can be propagated to an observable output of the array ([1]).

The left, vertical and right inputs of cell j at time t are denoted as $r^j(t)$, $z^j(t)$, $l^j(t)$ 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 2a shows the space-time transformation 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

C2: for every $l \in L$ there exist $l' \in L$, $z \in Z$ such that $g_L(R, z, l') = l$.

O1: for every $r_1, r_2 \in R$ with $r_1 \neq r_2$, there exist $l \in L$, $z \in Z$, such that $g_R(r_1, z, l) \neq g_R(r_2, z, l)$.

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

Figures 4a and 4b illustrate conditions *C1* and *C2*, Figures 5a and 5b illustrate conditions *O1* and *O2*.

Conditions *C1*, *C2* can be thought of as ***controllability*** conditions, and conditions *O1*, *O2* as ***observability*** conditions.

Condition *C1* simply states that if we want to get a particular right output r, all we have to do is to apply vertical input z and left input r', no matter what the right input is (see Fig. 4a). Condition *C2* is the symmetrical version of condition *C1*. Condition *O1* states that if we want to distinguish between right inputs r_1 and r_2, all we have to do is to apply vertical input z and left input l, and observe the right output. Condition *O2* is the symmetrical version of condition *O1*.

V is a broader set of conditions than *G*, and in particular *G1* implies *C1* and *O1*, *G2* implies *C2* and *O2*. We elaborate further on that: Let us construct a digraph $G_R = (R, E_R)$ with node set R; arc $(r_1, r_2) \in E_R$ and is labeled z if and only if $g_R(r_1, z, L) = r_2$. This means that if we apply r_1 as left input and z as vertical input, we get r_2 as right output, no matter what the right input is (Fig. 6). We define in a similar manner the "left" digraph $G_L = (L, E_L)$. Condition *G* states that there exists a label a such that there are exactly $|R|$ arcs of G_R and $|L|$ arcs of G_L labeled a, and these arcs form a set of vertex-disjoint cycles. Conditions *C1* and *C2* state that every node of G_R and G_L has a predecessor. Condition *O1 (O2)* states that for every pair of distinct left (right) inputs r_1, r_2 (l_1, l_2) there exist a vertical and a right (left) input that produce distinct left (right) outputs (Fig. 5).

We can prove now that conditions *V* are in fact sufficient for testability.

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

Proof: Assume we want to test cell j for inputs (r_0, z_0, l_0). First we have to solve the contollability problem, that is we have to apply input (r_0, z_0, l_0) to that cell by controlling the external inputs of the array. Then we have to solve the observability problem, that is we have to propagate the faulty outputs of that cell to the observable outputs of the array; this propagation should be such that the observable outputs are

different from the expected under the presence of faults.

Let $j \geq (p+1)/2$. (The case $j < (p+1)/2$ is treated similarly). These test inputs will be applied at time $t=j$ if the test begins at time $t=1$. Hence $r_0 = r^j(j)$, $l_0 = l^j(j)$, $z_0 = z^j(j)$ (see Fig. 7, shaded cell). First we must make the left input of cell j at time j be $r_0 = r^j(j)$. Condition *C1* guarantees the existence of $r^{j-1}(j-1)$, $z^{j-1}(j-1)$ such that $r^j(j) = g_R(r^{j-1}(j-1), z^{j-1}(j-1), L)$, hence it suffices to apply $r^{j-1}(j-1)$, $z^{j-1}(j-1)$ as left and vertical inputs to cell $j-1$ (at time $j-1$). Inductively, apply $r^{j-2}(j-2)$, $z^{j-2}(j-2)$ to cell $j-2$, etc., until we reach the leftmost cell. Similarly, to apply $l_0 = l^j(j)$ as right input to cell j, we find $l^{j+1}(j-1)$, $z^{j+1}(j-1)$ such that $l^j(j) = g_L(R, z^{j+1}(j-1), l^{j+1}(j-1))$, according to condition *C2*. We proceed inductively in the same way, until we reach the rightmost cell. If $j \geq (p+1)/2$ the first test input to the rightmost cell (p) will be applied at time $2j-p-1$. This way inputs r_0 and l_0 will be applied simultaneously to cell j (at time j). This solves the controllability problem.

Assume that we want to test for the right output fault $r/\hat{r}$; that is, the normal right output is r and we are testing if we get $\hat{r}$ instead. According to our notation $r = r^{j+1}(j+1)$. Let $\hat{r}^{j+1}(j+1) = \hat{r}$. Condition *O1* guarantees the existence of a vertical input $z^{j+1}(j+1)$ and a right input $l^{j+1}(j+1)$ such that $g_R(r^{j+1}(j+1)/\hat{r}^{j+1}(j+1), z^{j+1}(j+1), l^{j+1}(j+1)) = r^{j+2}(j+2)/\hat{r}^{j+2}(j+2)$. $l^{j+1}(j+1)$ is obtained as left output of cell $j+2$ in the same way that $l^j(j)$ was obtained as left output of cell $j+1$ (using condition *C2*). Thus the faulty right output $r/\hat{r}$ is propagated to the right output of cell $j+1$. Inductively, this is propagated to the observable right output of the rightmost cell. Similarly, using condition *O2*, we can simultaneously test for the left output fault $l/\hat{l}$, propagating it to the left output of the leftmost cell. This solves the observability problem.

The above procedure is repeated for every $\hat{r}$ in R_r, $\hat{l}$ in L_l; then we have tested cell j for input (r_0, z_0, l_0). When this is repeated for every (r, z, l) in $R \times Z \times L$ we have tested cell j completely. ■

The testing time shown in Fig. 7 is $p+1$. Hence to test cell j for input (r_0, z_0, l_0) we need $(p+1) \cdot max(|R|, |L|)$ tests, and to test cell j completely we need $(p+1) \cdot max(|R|, |L|) \cdot |R| \cdot |Z| \cdot |L|$ tests, and to test the entire array completely we need $p \cdot (p+1) \cdot max(|R|, |L|) \cdot |R| \cdot |Z| \cdot |L|$ tests.

From Fig. 7 it is clear that if some cell is used for a specific test 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.

For comparison, if condition G holds, testing time is $O(p \cdot |R| \cdot |Z| \cdot |L|)$.

Although condition V is weaker than G, it is still a very strong condition in that it requires the existence of z inputs for which the right-moving signal is independent of the left-moving signal *and* vice versa. However, the very purpose of making the array bilateral suggests dependence of at least one of the right or left-moving signals on both the right and left-moving signals. As an example, consider the case $|Z| = 1$. Then we basically have no z-inputs, and if conditions *C1* and *C2* hold, the array degenerates to two unilateral arrays, one with signal flow from left to right, and one with signal flow from right to left. Anyway, the case might be that for all z, the left output depends on both the right and the left input, so condition *C2* is violated. We derive a set of sufficient conditions for this case. We shall keep the assumption that for some z the right output depends only on the left input. Naturally, this will lead to a more complicated set of conditions.

Let W be the following set of conditions:

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

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

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

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

Figures 8a and 8b illustrate conditions *C1* and *C2*, Figures 9a and 9b illustrate conditions *O1* and *O2*.

Conditions *C1*, *C2* can be thought of as ***controllability*** conditions, and conditions *O1*, *O2* can be thought of as ***observability*** conditions. Condition *C1* (of W) is a stronger version of condition *C1* of V in that it not only requires that if we want to get a particular right output r we can apply vertical input z and left input r', no matter what the right input is, but additionally, if we apply some left input different from r', we get a right output different from r. But still this condition is weaker than *G1*.

Condition *C2* states that if we want l as left output, all we have to do is to apply (r', z, l') (notice that it matters what the left input is, whereas in *C2* of V it does not), but additionally, if instead of r' we apply some other left input different from r', we shall get a right output different from r. Notice that if this additional restriction is removed, condition *C2*

holds trivially if we naturally assume that every output is obtainable for some inputs. Notice also the asymmetry between *C1* and *C2*.

O1 is again a stronger version of condition *O1* of *V*, in that the latter required only the propagation of r_1/r_2 to the right output for some z and l, but *O1* of *W* requires the propagation of $r/*$ for some z, no matter what l is. But still this condition is weaker than *G1*. (In other words *G1* implies *C1* and *O1*).

Finally condition *O2* is the same as condition *O2* of *V*.

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, z that produce two different left outputs. This is a very reasonable set of assumptions.

We can prove now that conditions *W* are in fact sufficient.

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, z_0, l_0). These test inputs will be applied at time $t=2p-j$ if the test begins at time $t=1$. Hence $r_0 = r^j(2p-j)$, $l_0 = l^j(2p-j)$, $z_0 = z^j(2p-j)$ (see Fig. 10, 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 $r^{j-1}(2p-j-1)$, $z^{j-1}(2p-j-1)$ such that $r^j(2p-j) = g_R(r^{j-1}(2p-j-1), z^{j-1}(2p-j-1), L)$; hence it suffices to apply $r^{j-1}(2p-j-1)$, $z^{j-1}(2p-j-1)$ as left and vertical inputs to cell $j-1$ (at time $2p-j-1$). Inductively, apply $r^{j-2}(2p-j-2)$, $z^{j-2}(2p-j-2)$ to cell $j-2$, etc., until we reach the leftmost cell. The difficult part is to apply right input $l_0 = l^j(2p-j)$ to cell j. Condition *C2* guarantees the existence of $r^{j+1}(2p-j-1)$, $l^{j+1}(2p-j-1)$, $z^{j+1}(2p-j-1)$ such that $l^j(2p-j) = g_L(r^{j+1}(2p-j-1), z^{j+1}(2p-j-1), l^{j+1}(2p-j-1))$, and $g_R(r^{j+1}(2p-j-1)/*, z^{j+1}(2p-j-1), l^{j+1}(2p-j-1)) = r^{j+2}(2p-j)/*$. Hence it suffices to apply input $(r^{j+1}(2p-j-1), l^{j+1}(2p-j-1), z^{j+1}(2p-j-1))$ 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 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 $g_R(r'/*, z, l') = r/*$. 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, z_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 $r/*$ in the

right output. Propagation of the error $l/\hat{l}$ to the leftmost output is done in the same way as in the proof of theorem 1, using *O2* and *C1*.

We have not yet discussed the "southeast" portion of Fig. 10, that is the portion below the right-to-left diagonal that passes through the shaded cell. First we have to propagate the fault $r/*$ to the rightmost output. But cell j may fail to function correctly at any previous time, so for instance (see Fig. 10) cell j on input $r^j(2p-2m+j)$ (for some m in $\{j+1, j+2,\ldots,p\}$), 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 $r/*$ to the rightmost output, using condition *O1* we find $z^{j+2}(2p-j)$ such that cell $j+2$ on input $r^{j+2}(2p-j)/*$ outputs $r^{j+3}(2p-j+1)/*$; inductively we propagate this fault to the rightmost output $(r^p(3p-2j-1)/*)$. Similarly, we propagate the fault $r^m(2p-m)/*$ for $m=j+1,j+2,\ldots,p$. Notice that the potential previous fault $r^{j+1}(2p-2m+j+1)/*$ of cell j has been "automatically" propagated to cell m as $r^m(2p-m)/*$ 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 than the expected $r^{p+1}(3p-2m+1)$.

The above procedure is repeated for every $\hat{l}$ in L_l; then we have tested cell j for input (r_0, z_0, l_0). This is repeated for every (r, z, l) in $R \times Z \times L$; then we have tested cell j completely. ■

Remark: If we have already tested cell j for inputs $(r^j(2p-2m+j), z^j(2p-2m+j), L)$ we know that $r^m(2p-m)$ will be the correct input to cell m, so propagation of the fault $r^m(2p-m)$ will not be necessary.

The testing time shown in Fig. 10 is $2p$. Hence to test cell j for input (r_0, z_0, l_0) we need $2p \cdot |L|$ 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.

Again, the obvious pipelining reduces testing time to one-half of the above number of tests.

When $|R|, |Z|, |L|$ are large, the number of tests may be prohibitive. This happens when the basic cell operates at the word level. But in that case it may be possible to drop the assumption that to test a cell

completely we must apply all input combinations; however, we must then of course have some information about the realization of the basic cell. Assume that we have somehow obtained a set of tests that suffice to test the basic cell. Each test is of the form: $(r, z, l), (\hat{r}, \hat{l})$ where (r, z, l) is the input and $(\hat{r}, \hat{l})$ is the faulty output. For example, assume we adopt the stuck-at fault model [14] and a certain line is tested for stuck-at-1 (S-A-1) by applying inputs (r, z, l); the output is $(\hat{r}, \hat{l})$ if this line is indeed S-A-1. Let the number of tests be T. Then, if condition V holds, the number of test inputs is easily seen to be $p \cdot (p+1) \cdot T$ since every cell requires $p+1$ time units for each test t in T. Similarly, if condition W holds, we need $2p^2 \cdot T$ time units.

Parthasarathy and Reddy [10] introduced the notion of one-step testability for one-dimensional 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, z, l) if the number of time units needed to test this cell for input (r, z, 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, z, 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.

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:

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

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

Figures 11a, 11b, illustrate these conditions. Let *OST* be conditions *OST1* and *OST2* together. Let *V–OST* (*W–OST*) be conditions *V* (*W*) and *OST* together.

It is easy to see that *V–OST* is equivalent to conditions *C1* and *C2* of *V*, *OST1* and *OST2*. (*O1* and *O2* of *V* are implied by *OST1* and *OST2*.) If *V–OST* hold, instead of testing for the fault $r/\hat{r}$ for each $\hat{r}$ in $R - \{r\}$ as in the proof of theorem 1, we can test for the fault r/R_r. Thus, to test cell j for input (r_0, z_0, l_0), we only need $p+1$ tests. Therefore, if we want to test all cells for inputs a subset I of $R \times Z \times L$, we need $p \cdot (p+1) \cdot |I|$ tests. (If $Z = R \times Z \times L$ we save a factor of $max(|R|, |L|)$.)

W−OST is equivalent to conditions *C1*, *C2*, *O1* of *W* and *OST2*. (Condition *O1* of *W* is stronger than OST1, that is *O1* of *W* implies *OST1*; *OST2* implies *O2* of *W*.) Similarly as above, by testing for all left output faults simultaneously, we need $2p^2 \cdot |I|$ tests if we want to test all cells for a subset I of $R \times Z \times L$. (If $I = R \times Z \times L$ we saved a factor of $|L|$.)

Remark: 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_R^i, g_L^i for the i-th cell; it suffices to replace the conditions for g_R, g_L by conditions for g_R^i, g_L^i for every i.

Application.

Figure 12 shows the basic cell of a two-way pipeline systolic array used for FIR filtering [11],[12]. 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, $2p^2 \cdot |I|$ tests suffice to test the array.

4. TWO-DIMENSIONAL ARRAYS.

Figure 13a shows a two-dimensional array. The basic cell is shown in Fig. 13b. In addition to g_R, g_L, we now have g_Z which is the vertically-moving signal mapping $R \times Z \times L \to Z$.

Let I be the following "independence' condition:

I: $g_Z(R, z, L) = \mu(z)$, where μ is a permutation of Z.

If condition I holds, the vertically-moving signals become independent of the horizontally-moving signals, hence any z-input sequence can be applied to any row by controlling the z-inputs of the first row. For instance, assume that at time t we want to apply the z-input sequence $z_1, z_2, \ldots, z_p$ to the i-th row. This can be done by applying the z-input sequence $\mu^{-i}(z_1), \mu^{-i}(z_2), \ldots, \mu^{-i}(z_p)$ to the first row at time $t-i$. Hence, if condition I holds, the discussion for the one-dimensional case immediately applies to the two-dimensional case. If, furthermore, μ is the identity permutation, any test inputs required for the one-dimensional array, which is just a row of the two-dimensional array, can be applied to all the rows simultaneously.

Application.

The basic cell of a two-dimensional array shown in Fig. 14 has been proposed in [13] for matrix multiplication. For this cell we have $g_R(r, z, l) = r$, $g_L(r, z, l) = l + z \cdot r$, $g_Z(r, z, l) = z$. Hence condition *I* holds with μ the identity permutation; also conditions *W-OST* hold, therefore if a subset I of $R \times Z \times L$ suffice to test the basic cell, a row of cells can be tested in $p^2 \cdot |I|$ time units. If the array has m rows it can be tested in $p^2 \cdot |I| + m$ time units, since m time units are required for the vertical signals to propagate from the first to the last row.

REFERENCES

[1] W. H. Kautz, "Testing for Faults in Cellular Logic Arrays," in *Proc. 8th Annu. Symp. Switching Automat. Theory*, 1967, pp. 161-174.

[2] P. R. Menon and A. D. Friedman, "Fault Detection in Iterative Logic Arrays," *IEEE Trans. comput.*, vol. C-20, pp. 524-535, 1971.

[3] R. W. Landgraff and S. S. Yau, "Design of Diagnosable Iterative Arrays," *IEEE Trans. Comput.*, vol. C-20, pp. 867-877, 1971.

[4] A. D. Friedman and P.R. Menon, "Fault Location in Iterative Logic Arrays," in *Theory of Machines and Computations*, Z. Kohavi and A. Paz, Eds. New York: Academic, 1971.

[5] A. D. Friedman, "Easily Testable Iterative Systems," *IEEE Trans. Comput.*, vol. C-22, pp. 1061-1064, 1973.

[6] F. J. O. Dias, "Truth-table Verification of an Iterative Logic Array," *IEEE Trans. Comput.*, vol. C-25, pp. 605-613, 1976

[7] B. A. Prasad and F. G. Gray, "Multiple Fault Detection in Arrays of Combinational Cells, *IEEE Trans. Comput.*, vol. C-24, pp. 791-802, 1975.

[8] C. H. Sung, "Testable Sequential Cellular Arrays," *IEEE Trans. Comput.*, vol. C-25, pp. 11-18, Jan. 1976.

[9] 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.

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

[11] 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.

[12] 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.

[13] R. W. Priester, H. J. Whitehouse, K. Bromley, J. B. Clary, "Signal Processing With Systolic Arrays," presented in *Tactical Airborne Distributed Computing and Networks Conference (AGARD)*, held in Roros, Norway, 22-26 June, 1981.

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

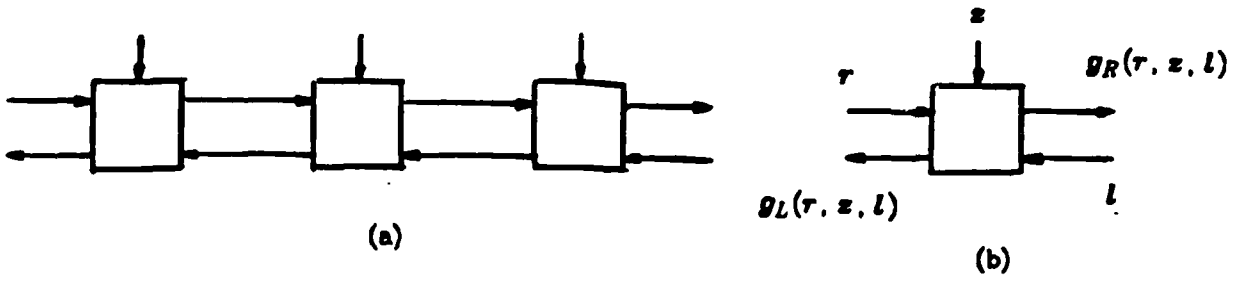


Figure 1
(a): A synchronous bilateral array
(b): The basic cell

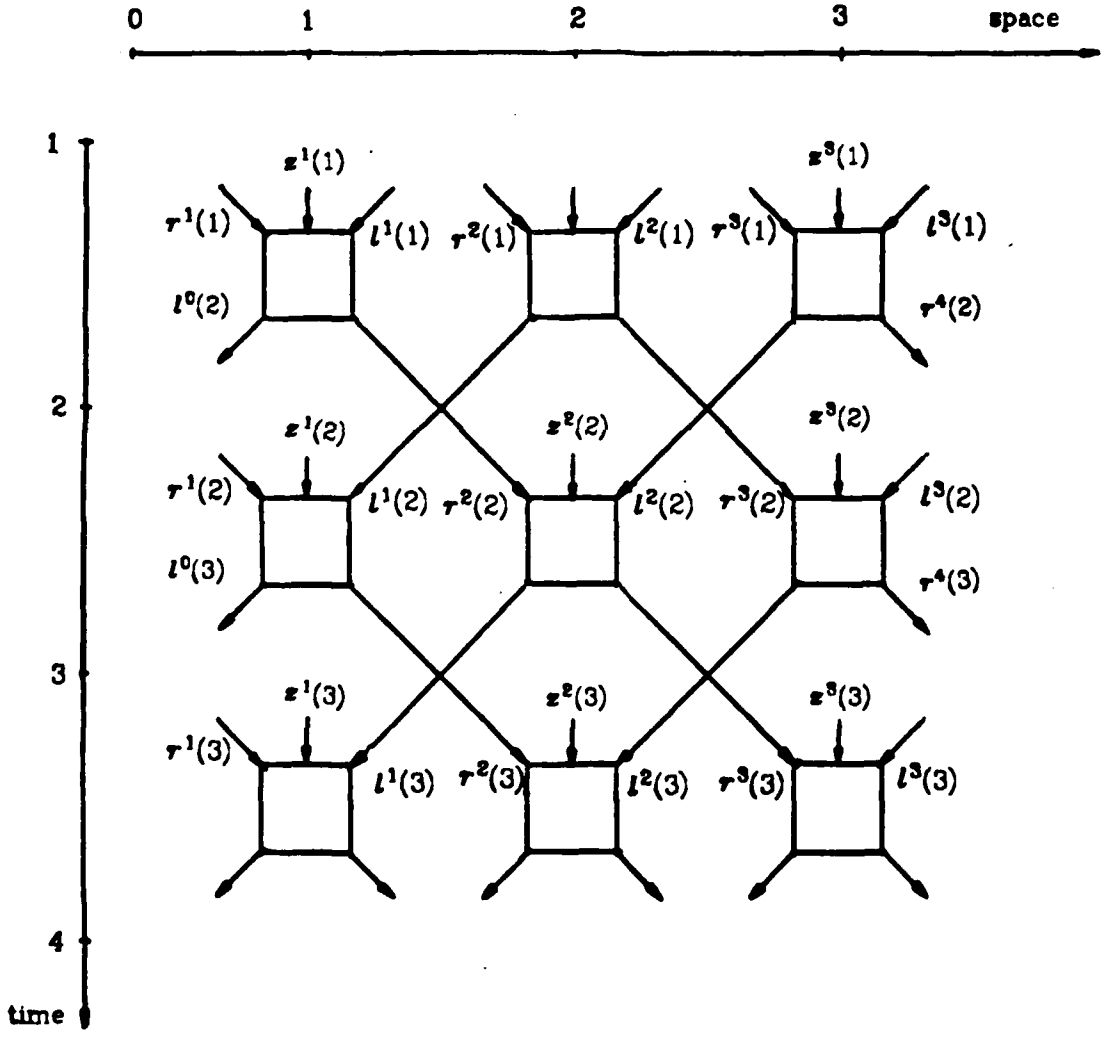


Figure 2a
Time-space transformation into asynchronous unilateral array

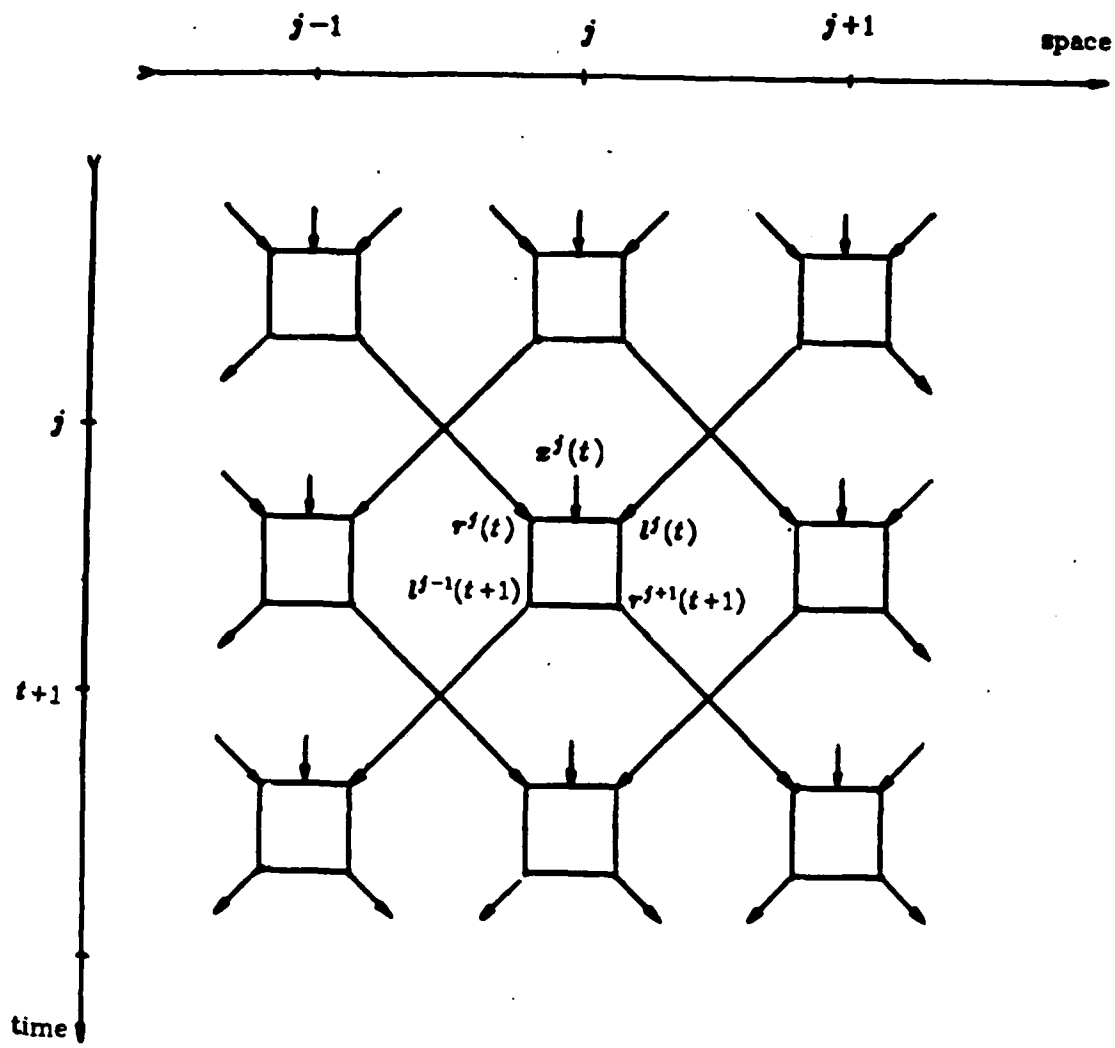


Figure 2b
Inputs at time t and outputs at time $t+1$ of the j-th cell

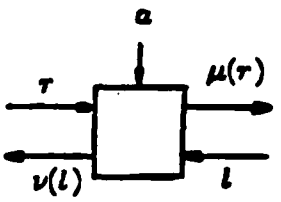


Figure 3
Condition G

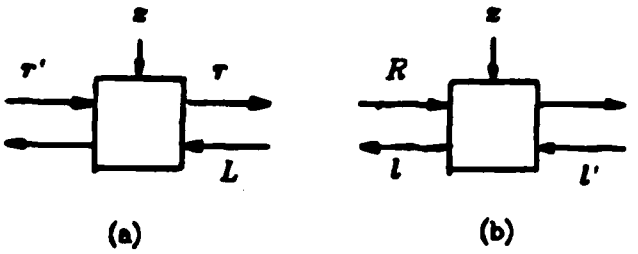


Figure 4
(a): Condition $C1$ of V
(b): Condition $C2$ of V

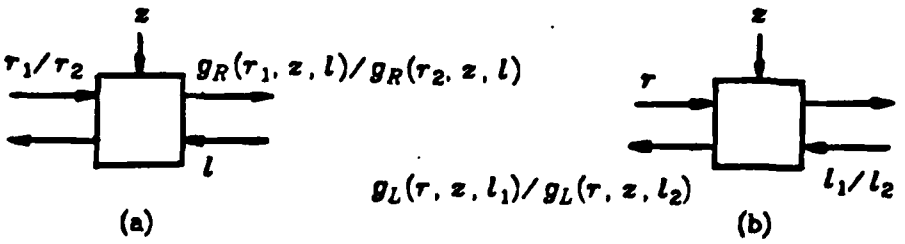


Figure 5
(a): Condition $O1$ of V
(b): Condition $O2$ of V

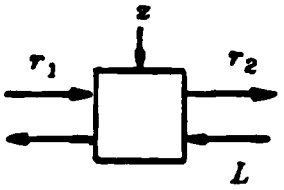


Figure 6
Condition under which arc (τ_1, τ_2) belongs to G_R

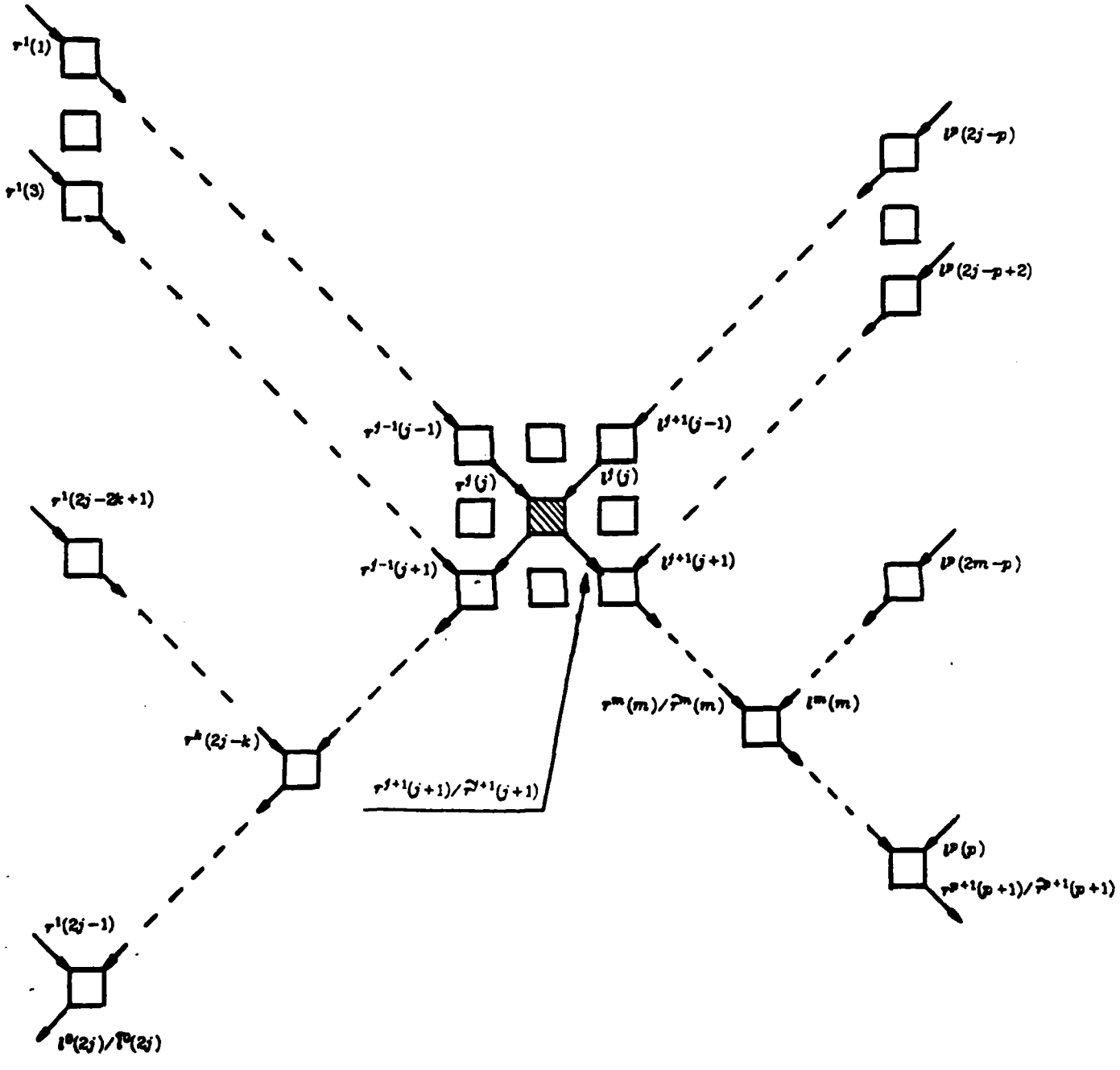


Figure 7
The test described in thm. 1 (vertical inputs are not shown for simplicity)

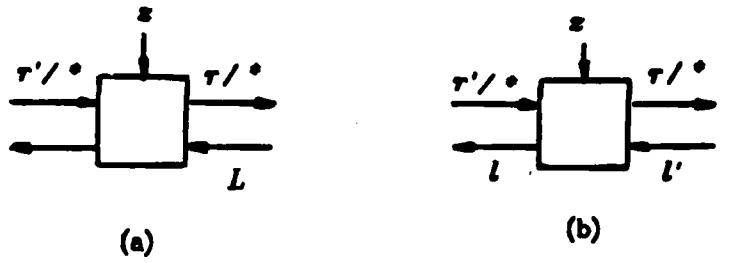


Figure 8
(a): Condition $C1$ of W
(b): Condition $C2$ of W

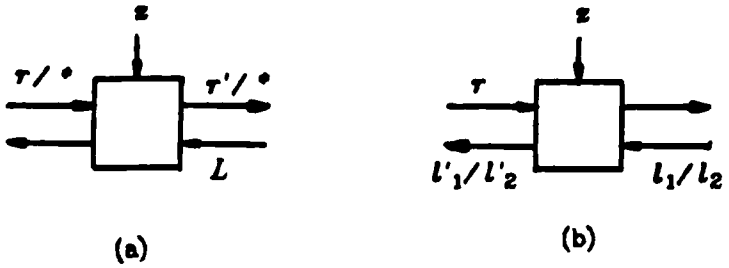


Figure 9
(a): Condition $O1$ of W
(b): Condition $O2$ of W

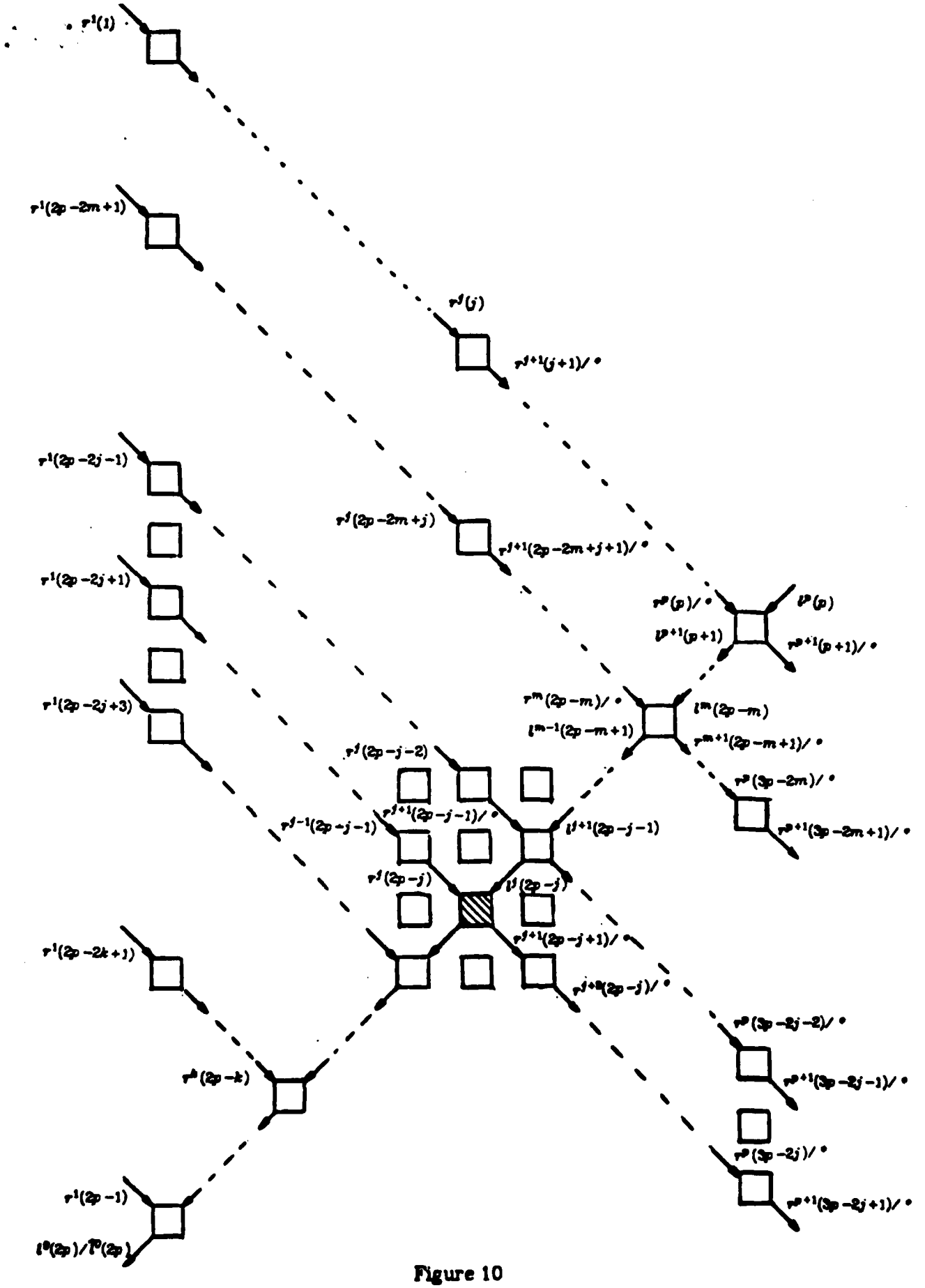


Figure 10

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

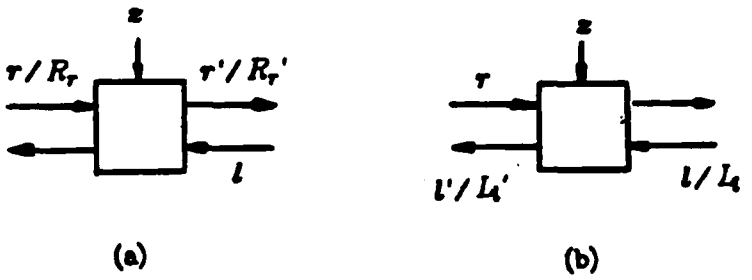


Figure 11
(a): Condition *OST*−1
(b): Condition *OST*−2

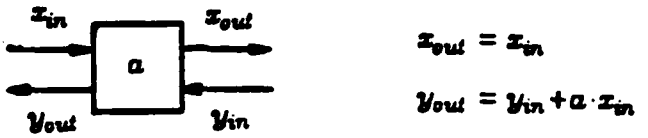


Figure 12
The basic cell of a two-way pipeline systolic array for FIR filtering

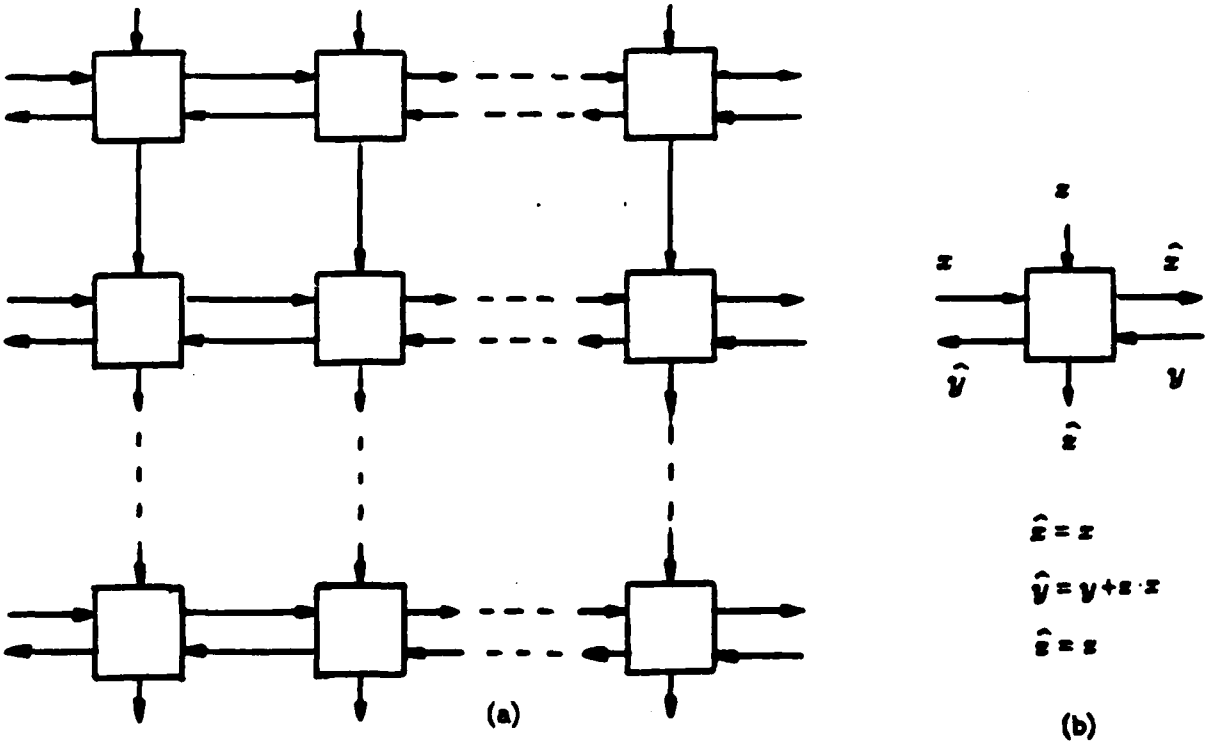
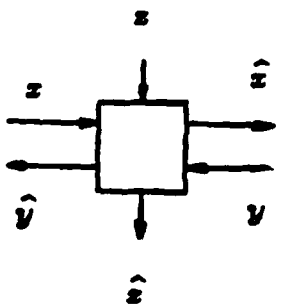


Figure 13
(a): A two-dimensional synchronous bilateral array
(b): The basic cell



$$\hat{z} = z$$
$$\hat{y} = y + z \cdot x$$
$$\hat{x} = x$$

Figure 14
The basic cell of a two-dimensional systolic array for matrix multiplication

[Nils80] N. J. Nilsson, *Principles of Artificial Intelligence*, Tioga Publishing Company, 1980.

[Pohm83] A. V. Pohm, O. P. Agrawal, *High-Speed Memory Systems*, 1983.

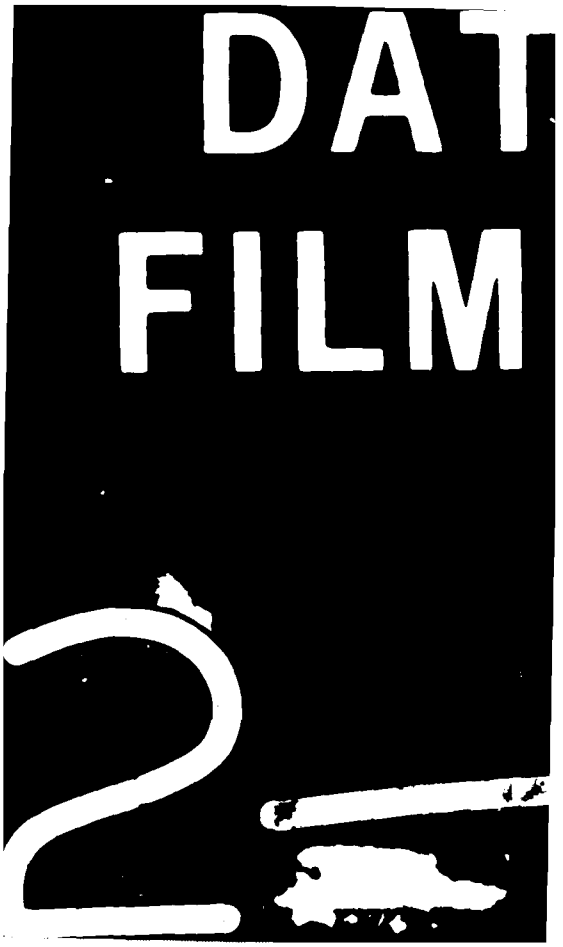
[Siss68] S. S. Sisson, M. J. Flynn, "Addressing patterns and memory handling algorithms," *Proc. AFIPS Fall Joint Computer Conference*, Vol. 33, Part 2, December, 1968, San Francisco, CA., pp. 957-967.

[Smit82] A. J. Smith, "Cache Memories," *ACM Computing Surveys*, Vol. 14, No. 3, September,1982, pp. 473-530.

[Wein83] P. Weinberger, Personal Communication.

[Wied77] G. Wiederhold, *Database Design*, McGraw-Hill, 1977.

[Wins77] P. H. Winston, *Artificial Intelligence*, Addison-Wesley, 1977.


DAT
FILM