JPL PUBLICATION 84-97

PIFEX: An Advanced Programmable Pipelined-Image Processor

Donald B. Gennery
Brian Wilcox

February 1, 1985

NASA
National Aeronautics and Space Administration

Jet Propulsion Laboratory
California Institute of Technology
Pasadena, California

JPL PUBLICATION 84-97

PIFEX: An Advanced Programmable Pipelined-Image Processor

Donald B. Gennery
Brian Wilcox

February 1, 1985

National Aeronautics and
Space Administration

Jet Propulsion Laboratory
California Institute of Technology
Pasadena, California

The research described in this publication was carried out by the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration

ABSTRACT

PIFEX is a pipelined-image processor being built in the JPL Robotics Lab. It will operate on digitized raster-scanned images (at 60 frames per second for images up to about 300 by 400 and at lesser rates for larger images), performing a variety of operations simultaneously under program control. It thus is a powerful, flexible tool for image processing and low-level computer vision. It also has applications in other two-dimensional problems such as route planning for obstacle avoidance and the numerical solution of two-dimensional partial differential equations (although its low numerical precision limits its use in the latter field). The concept and design of PIFEX are described herein, and some examples of its use are given.

CONTENTS

LIST OF ILLUSTRATIONS

1. THE PROBLEM

Computer vision requires an enormous amount of computing. This seems to be especially true for the low-level portions of the task, in which the data are still in the form of an image. Often thousands of fundamental operations must be performed for each pixel (picture element), and typically there are around a hundred thousand pixels per image. Real-time processing at a rate of 30 or 60 images per second therefore may require a processing speed of around 10^{10} operations per second. Conventional computer architectures (of the Von Neumann type) are not currently capable of approaching these speeds. The fastest Von Neumann computers are two or more orders of magnitude too slow for typical problems in real-time computer vision.

The solution to this large speed deficit is generally thought to be some form of parallel processing, so that a large number of computational elements operating simultaneously can achieve the necessary rates. (For reviews of parallel processors see [1] and [2].) There are several ways in which the necessary parallelism can be achieved.

One way is to use a multiple-instruction-stream multiple-data-stream (MIMD) system, which consists of many Von Neumann machines operating on different parts of the same problem and communicating their results to each other. Such a multiprocessor system may be appropriate for the high-level portion of powerful future vision programs. However, for the low-level portions of the vision task, such a system is not cost-effective. This is because low-level vision tasks contain computations that are performed almost identically over the entire image, and it is wasteful to use the full power of general-purpose processors to do these repetitive tasks.

Another type of parallel computer is the single-instruction-stream multiple-data-stream (SIMD) system. In such a system arithmetic units for each portion of the picture (perhaps each pixel) perform the same operations simultaneously under the control of a master processor. If there is an arithmetic unit for each pixel, such a system is fairly convenient to use and is very fast. However, the cost is high. For example, the Massively Parallel Processor [3] (which was built for NASA by Goodyear Aerospace and is possibly

the most powerful computer of this type so far) contains 16,384 arithmetic units, which occupy 2048 chips, and costs several million dollars. It is arranged as a 128-by-128 array, and for example can add the elements of one 12-bit array to those of another 12-bit array in 3.7 microseconds, which corresponds to 4.4×10^9 operations per second.

Another approach is a pipelined-image processor, which processes the pixels sequentially as they are scanned (usually, but not necessarily, at the normal video rate). The parallelism can then be built into the device so that it performs more than one arithmetic operation for each pixel. (Some of these operations can be done simultaneously on corresponding pixels in parallel data paths, and some can be done in a pipelined fashion in which one operation is being done on one pixel while the next operation is being done on the previous pixel, which already has had the first operation performed on it.) Also, no time is spent decoding instructions while this processing is going on, because the same operations are performed over and over, at least for one frame time, and no access time for the data is needed. This type of system can be far less expensive than an SIMD system, because it requires a number of processing elements depending on the number of steps in the algorithm instead of depending on the size of the image, and the former is usually a few orders of magnitude less than the latter. It usually is not as fast as an SIMD system, but it can process an entire image in one frame time (normally 1/30 second or 1/60 second), and thus it is suitable for most real-time applications. (If the number of steps in the algorithm exceeds the number of processing elements, separate passes can be made to complete the algorithm. This requires extra frame times and perhaps additional time for reprogramming the device.)

Pipelined-image processors have been built in the past. (Some of them are mentioned in Section 2.) However, they are very restricted in the kind of computations that they can do. They do not include the full range of desired computations, and what they do include often is not fully programmable. Furthermore, their computational power falls short of what is needed for many tasks. What is desired is a programmable system that will perform elaborate computations whose exact nature is not fixed in the hardware and that can handle multiple images.

The main problems in designing such a system are choosing a set of fundamental operations that are sufficiently general and that can be implemented in the desired quantity at a reasonable cost, and finding a practical way of interconnecting these operators that allows sufficiently general programmability. These problems are discussed further in Section 3.

PIFEX will be a programmable pipelined-image processor meeting the above criteria. A moderate-sized PIFEX costing less than a hundred thousand dollars will be able to perform about 10^{10} 12-bit operations per second. However, no algorithm can utilize this computational power with 100% efficiency, because of a lack of a perfect match between the nature of the algorithm and the architecture of PIFEX. In fact, very simple algorithms would have a very low efficiency if running alone on PIFEX, because a computation requires one complete frame time no matter how small it is. However, several small algorithms can run simultaneously. (Some sample algorithms are given in Section 9.)

2. HISTORY

Two concepts which are important in PIFEX are cellular computers and pipelined-image processors.

A cellular computer is based on the concept of a cellular automaton. A cellular automaton consists of an array of cells, a finite set of permissible states for a cell, and a transition function, which is a set of rules for determining the new state of each cell as a function of the old states of itself and its neighbors. The cellular automaton then operates in discrete time steps, changing the states of the cells at each step. A cellular computer is similar to a cellular automaton, except that the transition function can be different on different steps, according to a program. Other differences may exist in particular cellular computers. (The array is usually two-dimensional.) For more information see [4] and [5].

An early cellular computer was the Golay Processor [6] at the Perkin-Elmer Corporation. It used a hexagonal array, with neighbors consisting of the six nearest neighbors to each cell. Basically, the states of the cells were binary, and thus the transition function was Boolean. However, it contained more than one array, and these could interact with each other. This could produce the effect of having more than two states per cell. It also used the concept of subfields, which will be described in Section 8. Such early devices (and many since) were implemented as SIMD machines. Gennery and Jordan [7] at RCA described a device similar to the Golay processor but based on a rectangular array, with eight neighbors for each cell. They devised a convenient language for programming it, but the device itself was only simulated on a conventional computer and was not implemented in hardware.

A common image processing operation is convolution. A convolver forms a linear combination of the pixel values over a neighborhood according to given weights (constant over the image) and uses the result for the new center pixel value. Since in practice the pixels have only a finite number of bits and thus a finite number of states, this can be considered to be a cellular computation. Such computations are often performed in conventional computers,

they can be programmed from more elementary instructions in a cellular computer, or a cellular computer may contain hardware convolvers as single stages (as in some of the examples below).

The idea of a pipelined-image processor (described in Section 1) apparently occurred independently to many people. One of the present authors (Gennery) thought of such a device in the mid 1970's when he was at Stanford University. This device would have a set of programmable functions that it could apply to the neighbors (in a fairly large neighborhood) of each pixel on each pass through the device in order to compute the pixel values resulting from the pass. However, no details were worked out. Meanwhile, the Cytocomputer [8] was being developed at the Environmental Research Institute of Michigan. Each stage of the Cytocomputer performs one iteration of a cellular computation on a rectangular array, with eight neighbors per cell. Eight-bit pixels are used. The new value of each cell is determined by a nonlinear process under program control. Successive stages perform successive iterations in one pass. Also, IMFEX [9] was being developed at JPL. IMFEX implements a simple edge detector similar to the Sobel operator. It consists of two 3-by-3 convolvers in parallel to determine the two components of the gradient, circuitry to determine the sum of the absolute values of the two components and (with a 3-by-3 operator) to locate the ridge (one-dimensional maxima) of its values, and a programmable 3-by-3 Boolean operator using the resulting thresholded values. Only the Boolean operator is programmable, by means of a look-up table. More recently several other devices of this type have been developed, for example the PIPE chip [10] at Texas Instruments. It is a 3-by-3 convolver in one integrated circuit. The convolver weights are stored in EPROM instead of RAM, and thus are not conveniently programmable, and the circuit operates at about one megahertz.

Since the above pipelined-image processors all operate on some neighborhood, they all require buffers as part of their circuitry, in order to store the data that is needed to cover the neighborhood at any instant as the pixels flow by. Since they all operate on two-dimensional arrays, they require line buffers. With 3-by-3 neighborhoods, two line buffers are needed per stage, plus storage for a few extra pixels on the current line, and a delay of one line plus a few pixels is introduced per stage. (In this regard,

IMFEX is considered to have three stages.)

In 1980, Gennery came to JPL and subsequently became familiar with IMFEX. In 1982, discussions with others at JPL concerning the capabilities of VLSI (very large scale integration) circuits led him to the concept of combining convolvers, various hardware arithmetic functions, and nonlinear neighborhood operators in a flexible switching arrangement, as described in Section 3. In May of 1983 the other author (Wilcox) thought of the concepts of the modular card and the use of interpolated table lookup for the arithmetic functions, also described in Section 3, in order to reduce the switching requirements. Meanwhile, starting in August of 1982 as a project in a VLSI design course, Wilcox designed a preliminary version of a VLSI 3-by-3 convolver chip with fully programmable weights. In September of 1983, NASA decided to fund the development of a device based on the modular concept and the table-lookup functions. In October of 1983 Gennery decided on preliminary specifications for a general nonlinear neighborhood operator, which essentially is a generalization of the concepts in his 1973 RCA paper and is similar to the Cytocomputer stage. The original concept for the topological connections of the modules was as a plane or a cylinder with a horizontal axis, where the main data flow is considered to be from left to right. However, in February of 1984, Gennery thought of making it a torus or a cylinder with a vertical axis, so that widely differing algorithms could be efficiently mapped onto the device. Discussions between Gennery and Wilcox continuing through March of 1984 refined all of these concepts to produce the versions described briefly at the end of Section 3 and in detail in Sections 4, 5, 6, 7, and 8.

The acronym PIFEX was suggested in September of 1982 by Bob Cunningham and stands for "Programmable Image Feature Extractor," but this is perhaps an unfortunate choice, since feature extraction is only one of PIFEX's capabilities. The name was chosen partly for historical reasons, to show continuity with IMFEX.

Wilcox finished an improved design of his VLSI convolver and the design of the modular card in May of 1984. A wire-wrapped prototype card is expected to be finished in December of 1984, and a demonstration of the

prototype card is expected in January of 1985. The layout of a printed-circuit version of the card is expected in April of 1985, and the production of a PIFEX with about 80 modules is planned for around September of 1985.

3. GENERAL PIFEX CONCEPT

PIFEX will receive one or more images as they are scanned from external image buffers or through A/D converters from TV cameras, perform pipelined operations on them as specified by a control program, and feed the results back into image buffers. In principle, the operations include convolving with various specified functions; arithmetic functions such as addition, subtraction, multiplication, division, square root, maxima, and minima; unary table lookups to map pixel values to new values; and nonlinear neighborhood operations to do such things as thinning, growing, and finding local maxima, minima, ridges, valleys, and zero-crossings. The delays caused by the processing are taken into account by the circuitry that reloads the buffers. If sufficient processing cannot be done in one pass, the results stored in a buffer from a previous pass can be used again, but it is expected that most desired computations can be done in one pass (one frame time plus delays).

Although it would be desirable for some applications to have the ability to convolve the images with fairly large functions, it is impractical at present to fit such a convolver on one chip. Therefore, PIFEX will have only 3-by-3 convolvers (all identical, but with individually programmable weights). However, using several of these in various series and parallel combinations can produce the effect of using larger convolvers. This is practical because of the fact that the desired functions usually are one of two types: small functions (3-by-3) used for such purposes as differentiation, and large functions used for smoothing. The best smoothing function normally is some approximation to the Gaussian function, and the Gaussian function has the advantages that it can be factored into one-dimensional functions and that it can be approximated by convolving several small functions in series. Either fact (especially the latter) allows a good approximation to the two-dimensional Gaussian function of moderate size to be produced by using 3-by-3 convolvers. Any one-dimensional function can be produced easily from these small convolvers. (Although any two-dimensional function can be produced in principle, it is impractical for large functions that do not have special properties such as the Gaussian function.) Although the precision of digitized images usually is only 8 bits, more precision can

be created by smoothing. For this reason and because of the increase in dynamic range produced by some computations, 12 bits are used in the input and output of the convolvers in PIFEX. (The PIFEX convolver will be described in detail in Section 6.)

One approach to the arithmetic functions is to have separate hardware adders, multipliers, etc. Another approach is to use a table lookup which can be programmed for any particular function desired. Each of these approaches has its advantages. The advantages of the separate hardware functions are that they require less circuitry and that they can be reprogrammed quickly by changing the switches that connect them. Several of the functions could fit on one VLSI chip, whereas one table lookup requires many chips (far too many if high precision is needed). The advantages of the table lookup are its uniformity, its greater ease of design (it uses standard memory chips, which are becoming quite cheap), and the fact that any possible function can be programmed into it, whereas with the other approach any function not included in the design (such as trigonometric functions, for example) would have to be approximated by using a considerable number of simpler functions. (Another advantage is that the unary table lookup is not needed, since its purpose is subsumed by the binary table lookup.) The precision needed in the final results of these functions for typical images is about 8 bits. Because of the greater dynamic range produced by such operations as multiplying, intermediate results need greater precision, preferably 16 bits (since only fixed-point computations are practical). However, the table-lookup approach allows a compression of the dynamic range with such operations (perhaps by using the square root or logarithm), so that 12 bits are adequate. A direct binary 12-bit table lookup is impractical, but doing a binary 8-bit lookup with linear interpolation for the low-order 4 bits is reasonable, as described in Section 7.

The nonlinear neighborhood operations are performed by a neighborhood comparison operator which compares the nine pixels in the neighborhood to a fixed threshold or (for the surrounding eight pixels) to the center pixel to produce nine bits of information. Two more bits indicate the subfields. A function of these eleven bits is then used to determine whether the 12-bit output of the operator is the center pixel, the bitwise logical OR of the

surrounding eight pixels, or a function of these eleven bits of information. This process is described in detail in Section 8. By suitably programming the operator, a variety of operations (some of which were mentioned above) can be performed.

The various operators need to be connected through some switching arrangement, so that individual connections can be made to suit individual programmed algorithms. A general arrangement is shown in Figure 1. Delay circuits (which include line buffers) are included so that the delays in different channels to be combined can be equalized (for example, a channel that has gone through a convolver, which has its own line buffers, and one which has not).

It would be desirable if the switches in Figure 1 formed a crossbar switch, so that the operations could be combined in any order and in any combination under program control. However, this would be very difficult for the size of system contemplated. The number of channels in and out of the crossbar switch would be on the order of 200, and they would have perhaps 12 bits each. Therefore, the number of gates would be $200^2 \times 12 = 480{,}000$. Using standard gate chips with four gates each would require 120,000 chips. VLSI technology will soon be able to squeeze 480,000 gates onto a single chip, but since it would need $2 \times 200 \times 12 = 4800$ pins, it would be completely impractical. (The use of a Batcher sorting network [11] instead of a crossbar switch probably is not worthwhile with this relatively small number of channels.) Eventually, it may be possible to put all of PIFEX on one chip (if the separate hardware functions are used), but in the meantime some severe limitations to the switching arrangement are needed (which will cause some sacrifice in the efficiency of utilization of the operators).

One possibility is to group all of the convolvers first, then all of the arithmetic functions, then all of the unary table lookups, if used, and then all of the neighborhood comparison operators. The signals would travel through each of these groups in order, and could recirculate around to pass through the groups again (in the same frame time) to use different members of each group on each pass, so that the three types of operators could be mixed in any order. (Since the order of the groups was chosen to be that in which

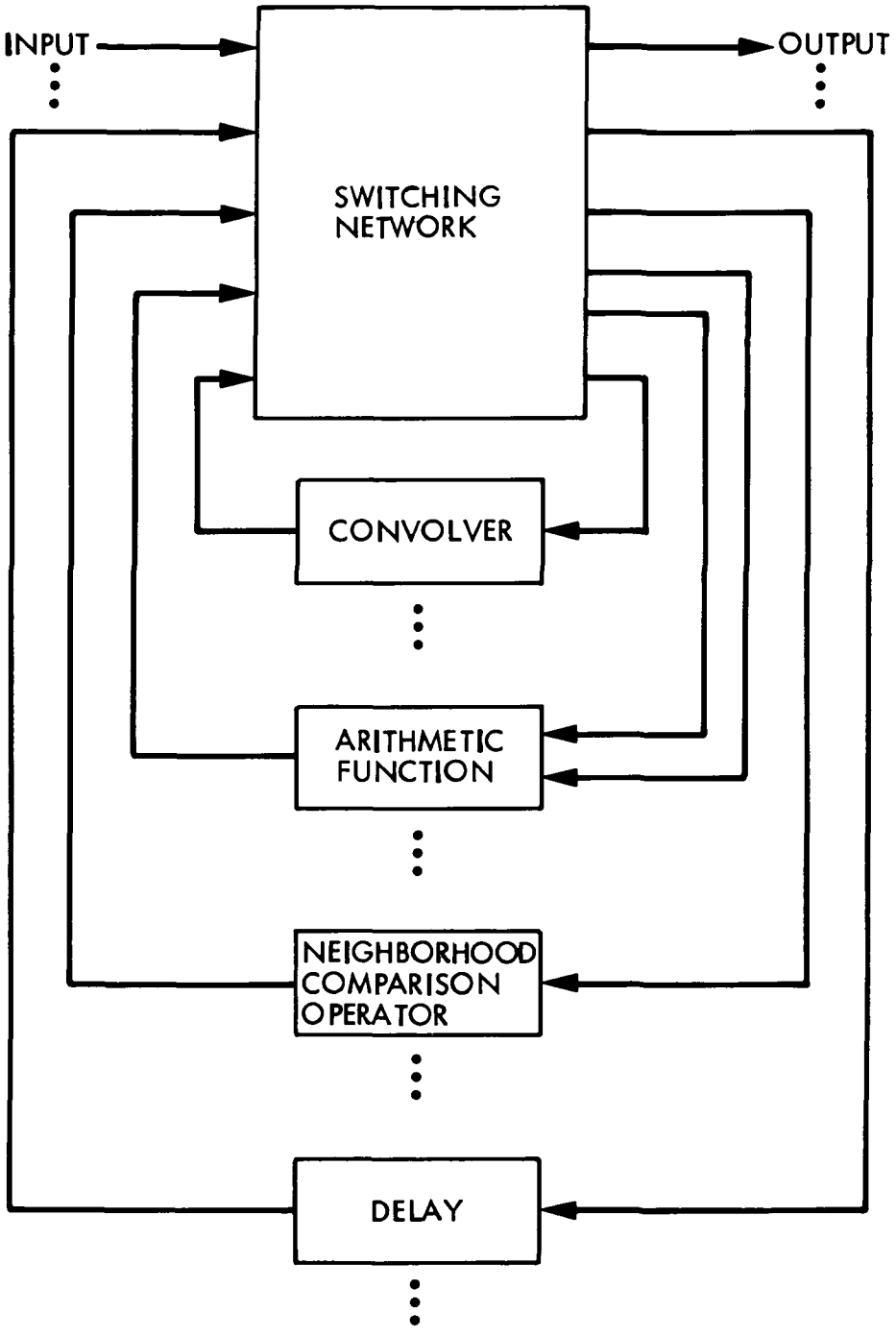


Figure 1. Generalized PIFEX switching concept (many items of each type would be connected to the switching network)

they are usually needed, the recirculation would be minimized.) Within each group, there would be a limited switching circuit. If the separate hardware arithmetic functions are used, the switching problem for this group would be the most severe, because of the fact that there would be several different kinds of circuits.

One way of arranging the separate hardware arithmetic functions in the above method would be to have several identical VLSI chips, each with a few of each operator and a few input and output lines. Each chip might contain the following: six multipliers, five adder-subtracters, one divider, one square root extractor, and possibly one rectangular-to-polar converter (perhaps instead of the square root extractor), and five delay lines for equalization of delays. Some shift operators might also be needed, but these could be combined with the multipliers, so that the specified portion of the double-length product could be used in each case, under program control. The limiting factor seems to be the number of pins on the chip. The chip will not be very useful unless there are at least five input channels and one output channel. But then the number of bits must be limited to 10 for input and output of the chip if the number of pins is 64. Another approach is to multiplex the input and output, by feeding the bits at twice the normal rate. Then there could be 6 input channels and 3 output channels, if 12 bits are used. Alternately, a larger package with more pins could be used. The chip could contain a crossbar switch that can connect the inputs, outputs, and operators in all possible combinations. For the numbers used above this means about a 25-by-35 switch, perhaps 16 bits wide. (However, it may be desirable to use less than a full crossbar switch, in which case the loss in efficiency caused by the lack of generality in switching may more than be made up for by the greater number of functions that this would leave room for on the chip.) These chips (perhaps about ten of them) would be richly interconnected in a pattern allowing for flexibility in programming.

Although some approach such as the above may be pursued further in the future, for the near term it was decided to use the table-lookup method for the arithmetic functions, which will be referred to hereafter as binary functions, since they each have two inputs. (Unary functions such as the square root can of course be implemented by simply ignoring one input.) The

switching method chosen for the near term uses a standard module containing two convolvers, one binary function, one neighborhood comparison operator, and switches for module input and ouput selection. (The module thus is similar to IMFEX, but more general.) The exact arrangement of the module is described in Section 4. (At first, each module will consist of one circuit card, but more compact circuits may be devised later.) The modules are connected in a regular pattern described in detail in Section 5, in which each of the two ouputs from each module branches to the inputs of several different modules.

Even though the chosen approach results in a physically larger device (and perhaps greater cost if produced in quantity) than the previous approach mentioned above, it has the advantages of quicker and less expensive development (because of the need for fewer types of complicated custom VLSI chips), ease of computing arbitrary functions (because of the generality of the table-lookup functions), and easy growth to a more powerful system (because of the modular concept with the regular interconnection pattern).

4. PIFEX MODULE

The economic advantages of having many identical units connected in a regular way outweighs (in the near term, at least) the inefficient use of extra hardware in each unit. Thus it is desirable to have general, programmable elements of each basic type (convolvers, binary functions, and neighborhood comparison operators) in each module. Since the binary function needs two data paths, the smallest possible such module would have two inputs feeding a binary function, which in turn feeds a convolver and then a neighborhood operator. However, examination of simple algorithms (e.g., edge detection) suggests that two convolvers (one for each input) ahead of the binary function is much more useful than having a convolver after the function. This is especially true since the convolvers are relatively inexpensive compared to the amount of memory needed for the large look-up table used in the current approach to the binary functions, so that adding a convolver does not greatly increase the cost of the modular unit. The two inputs to the module will be denoted A and B.

The interconnection of the modular units requires some flexibility in the routing from one stage of the pipeline to the next. If one visualizes columns of modular units, all synchronized, as being a stage of the pipeline, then a flexible interconnection scheme to the next column requires that the output of a module in some row be routable to some other row in the next column. A connection to all possible rows need not be provided, only enough to allow a rich interconnection architecture for a wide variety of possible algorithms. (The adopted approach is described in Section 5.) Assuming that the modules are plugged into a passive backplane which provides only connecting wires (as is the common practice in both general-purpose and special purpose electronic equipment and computers), then some input selection mechanism must be added to the modular card. Since we deduced above that each module should have two input channels (to the two convolvers), an input selector should choose which of several inputs should be fed to each of the two convolvers. Selectors built from standard logic circuits are available for 2, 4, 8, or 16 alternatives. Since a 12-bit data path has been chosen, the number of wires to be selected is 12 times 2 inputs times the number of alternative inputs to the selector. For 8 alternatives, this is 192 wires,

which is a reasonable maximum for signal input to a single circuit board. Furthermore, examination of possible algorithms indicates that a choice among 8 inputs provides sufficient flexibility to make efficient use of each module in the array.

The outputs of the two convolvers are hardwired to the two inputs of the binary function, its ouput is hardwired to the input of the neighborhood comparision operator, and its output forms one output of the module, called the A output. (The convolvers and the neighborhood comparison operator include the line buffers needed to enable them to operate on 3-by-3 windows.) To allow a significant increase in programming flexibility at a minor cost in terms of increased hardware, an additional output path is provided, called the B ouput, which passes any one of the intermediate values produced in the module (inputs to the convolvers as selected by the input switches, the convolver outputs, the binary function output, or the neighborhood comparison operator output). The pipeline delay (including extra line buffers) is equalized on this output to match the A output. (This is an important feature of the modular concept; there is automatic delay equalization as the signals propagate through the stages of the pipeline, unlike the situation where all the various functions are connected via a switching network.)

The above description of a module is summarized in Figure 2. Detailed descriptions of the convolvers, the binary function, and the neighborhood comparison operator may be found in Sections 6, 7, and 8, respectively.

All data paths between modules and between operators and functions within a module consist of 12 bits each. In order to have one more bit of precision when negative numbers are not needed, each operator in each module can treat these quantities either as unsigned integers or as positive or negative two's-complement integers, as specified by the programming information. (Since the binary functions are completely programmable, the representation in these is up to the programmer.)

In cases where not all of the operations in a particular module are needed, it is of course possible to program any of them to be identity operations. For a convolver, this means using unity for the center weight and

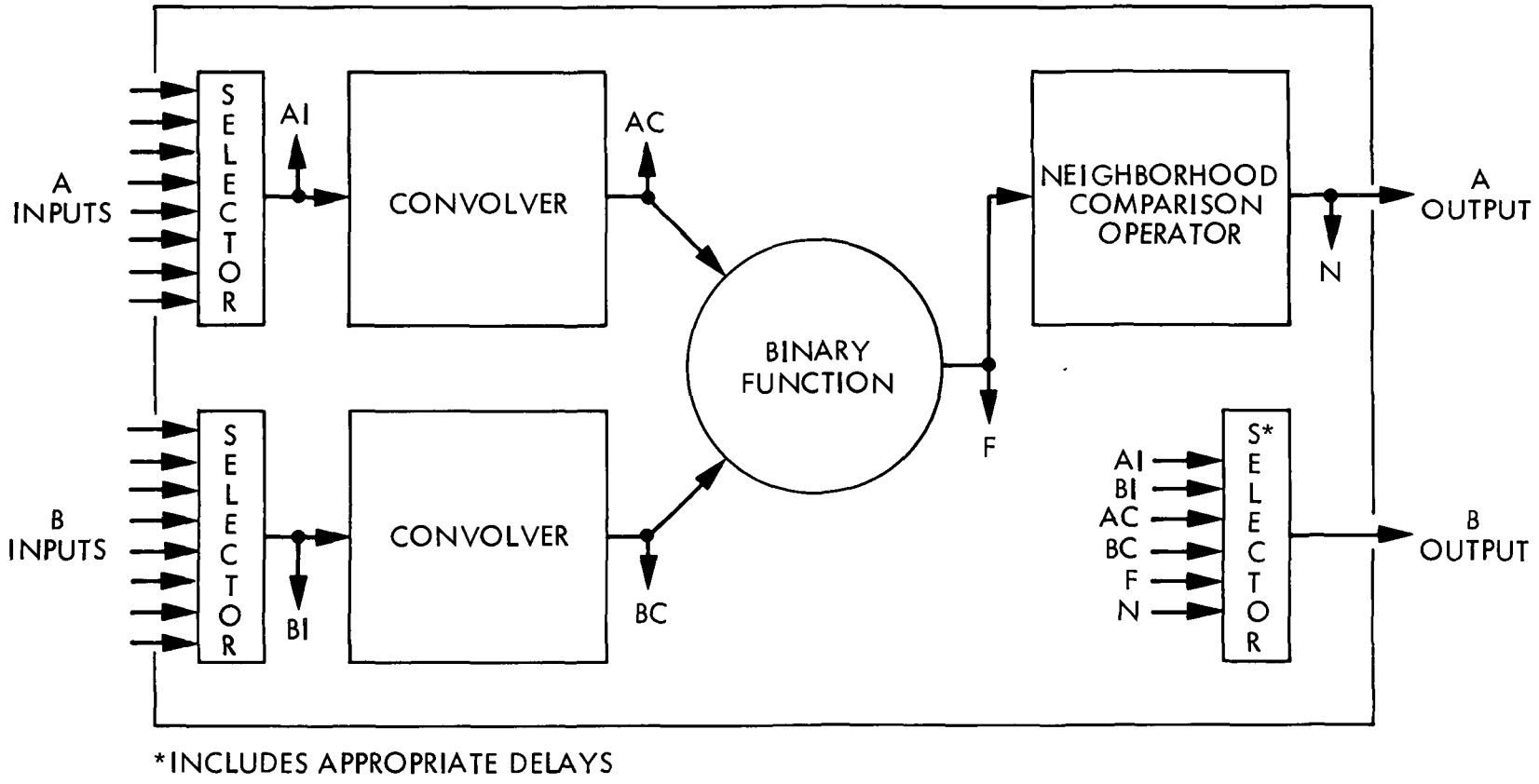


Figure 2. PIFEX module

zero for the others, for the binary function it means having the contents of the table lookup be the same as one of its inputs (with unity slope for interpolation), and for the neighborhood comparison operator it means passing the center value in all cases.

The initial version of PIFEX will process data at rates of up to 8,000,000 pixels per second (in each of the parallel paths through the module interconnection network). It will handle images with up to slightly more than 2000 pixels per line. (The latter limit is determined by the size of the line buffers. There is no limit on the number of lines in the picture.)

PIFEX will be programmed from the host computer (a Digital Equipment Corporation VAX) through a DR-11W direct memory access (DMA) interface. This will permit each binary function (which uses a quarter-megabyte lookup table) to be programmed in under 200 milliseconds. The binary functions in all PIFEX modules which are to be loaded with the same function can be programmed simultaneously in two 64K 16-bit DMA transfers. This is a great benefit, since a PIFEX array with several hundred modules would likely have only a dozen or so different binary functions, so that the programming time is a few seconds rather than many tens of seconds. The lookup table for each different neighborhood comparison operator is similarly programmed in a 2K DMA transfer. The convolver weights, input selection, and other miscellaneous data are programmed in an 80-word DMA burst for each module needing reprogramming.

5. INTERCONNECTION ARCHITECTURE

The modules described in Section 4 are connected in a two-dimensional pattern chosen so that the switch selections inside the modules produce flexibility in programming.

The outputs from the modules in each column in the pattern are connected to the inputs of modules in the next column, so that the main data flow is considered to be from left to right. In this way, synchronism is achieved, since all of the modules in a given column (except for the wrap-around of rows discussed below) are processing corresponding pixels at the same time. Different rows of modules correspond to parallel data paths, but these different paths can communicate with each other because of the branching of the connections from one column to the next.

In the branching patterns considered, the A outputs from modules in one column connect only to A inputs in the next column, and B outputs connect only to B inputs. (Although this restriction is not necessary, it seems to be convenient.) Originally, a fanout of four was tried, with the number of rows upwards from output to input being -1, 0, 2, 7 for A and -5, 0, 1, 3 for B. An alternate pattern considered was -1, 0, 2, 6 for A and -2, 0, 1, 2 for B. The former of these, at least, works fairly well for simple algorithms, but a fanout of only four is quite restrictive for more complicated algorithms. Therefore, it was decided to use a fanout of eight. That is, each A or B output branches to eight different inputs, and thus the A and B inputs on a module each receive eight signals, one of which is selected in the module, as described in Section 4. The fanout pattern chosen, at least initially, is -3, -1, 0, 1, 2, 3, 4, 8 for A and -2, -1, 0, 1, 2, 3, 5, 9 for B. This pattern is shown in Figure 3. (Notice that these fanout patterns are biased upwards. This helps to take advantage of the toroidal topology described below.)

Each column is considered to wrap around to form a loop, and the fanout pattern shown in Figure 3 is cycled invariantly around the loops. This feature is convenient for algorithms that just barely fit, since crossing the boundary that otherwise would exist at the top and bottom may help in making the necessary connections. More importantly, each row also wraps around to

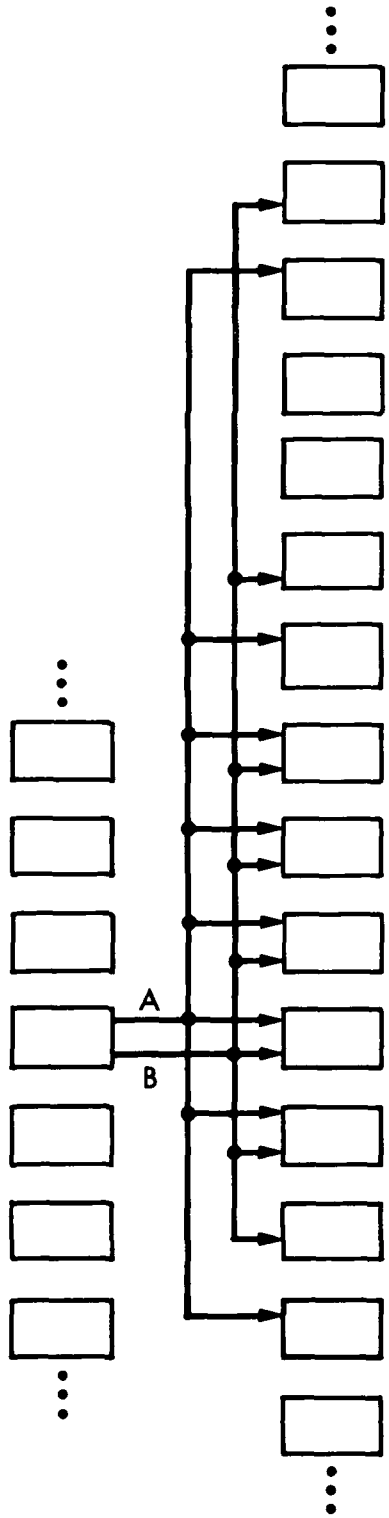


Figure 3. Interconnection of modules (the connections from any particular module in one column to the modules in the next column are shown)

form a loop. The fanout pattern continues cyclically around these loops also, except that after one particular column there are switches that can break each connection between the output of a module and the fanout to the next column, so that outputs can be extracted here and inputs can be inserted. This row wrap-around feature is important for efficient coding of algorithms that vary greatly in the width and length of data paths that they require, since an algorithm that requires a long path can spiral around several times, using only as many parallel data paths at any point as it needs. The upward bias of the fanout helps in spiraling the paths upwards in order to avoid collisions. A simple example of this is shown in Figure 6 in Section 9. (Since the pixels have been delayed by different amounts on different times around, ordinarily data from these different paths should not be combined with each other.) Because of the column wrap-around in addition to the row wrap-around, the upward spiraling can easily be made to cause the output to occur at the same rows as the input, thus avoiding the waste of modules that otherwise could occur.

The two wrap-around features combined cause the the interconnections of the modules in PIFEX to have the topology of a torus. There is a cut around the torus at one place to allow inputs (from image buffers or TV cameras) and outputs (to image buffers) to be switched in, under control of the host computer. Figure 4 shows two representations of the torus, which are topologically equivalent (since turning the torus inside-out changes one into the other).

It is planned that the initial version of PIFEX will have 5 columns and about 16 rows. (Thus it would be possible to code algorithms that vary from requiring a data path 16 modules wide and 5 modules long to requiring a data path one module wide and 80 modules long, without having to use separate passes through PIFEX on separate frame times.)

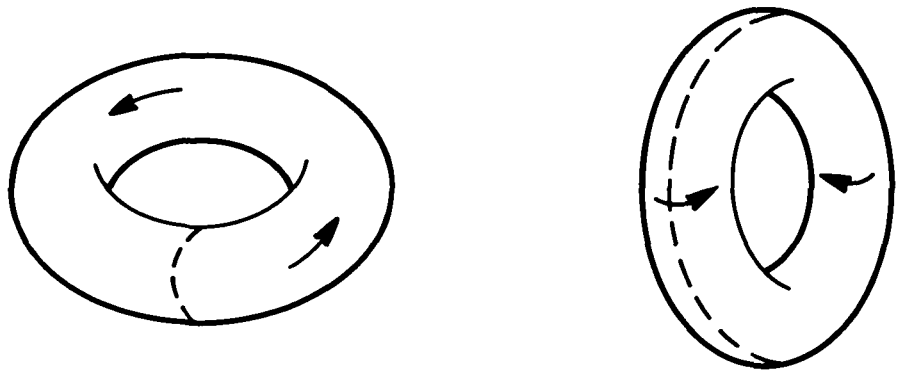


Figure 4. Two representations of interconnection topology

6. CONVOLVER

The convolver produces the sum-of-products of nine fixed weights times the corresponding pixel values in a 3-by-3 window. Examination of the typical weights used in low-level vision algorithms indicates that small positive or negative integers are most commonly used, with the ratio of the smallest to the largest weight being usually less than 20. This means that a six-bit (including sign) weight will be adequate, since this can represent integer values from -31 to 31. To prevent the 12-bit data path from overflowing, it is also necessary to scale the output of the convolver in some variable way (since all large positive weights will produce a much larger result than a mix of small positive and negative weights). Scaling is most easily accomplished in hardware by shifting the data one or more bits, i.e., dividing by some power of two. This is an essential feature of the convolver implemented for PIFEX.

Multiplying a 12-bit quantity by the five (unsigned) bits of each weight produces a result having 17 significant bits. Adding nine of these 17-bit quantities together can produce a result having as many as 21 significant bits. Since only 12 of these bits can be output (to maintain a constant data path throughout the pipeline), it seems excessive to compute the result accurately to all 21 bits. However, somewhat more than 12 bits must be retained in intermediate stages of the convolver, since it is common to take derivatives of heavily smoothed data, which involves subtracting quantities which are nearly equal. To preserve 12 significant bits of result when subtracting quantities differing only by 10% or so requires a 16-bit internal data path. To ensure validity of the least significant bit of the output an additional bit of low significance is also needed internally to the convolver. Thus the convolver is designed with a 17-bit internal data path.

The scaling of the result mentioned above is accomplished in two stages: the input pixel values to the convolver may be shifted down in significance, allowing more room for carry overflow when large positive weights are used, and the 12-bit output from the 17-bit data path may be shifted up to allow for cancellation when subtracting nearly equal quantities. As discussed above, adding nine 12-by-5 multiplies can produce a

21-bit result. However, needing all nine weights near the maximum value of 31 is very unlikely, since they could all be divided by two and the result scaled to produce nearly identical results. Thus it is reasonable to assume that the sum of the nine weights can always be kept to somewhat under the maximum 279 (9 times 31). If the sum of the weights is kept under 256 (9% less than 279), overflow into the 21st bit can be avoided. This means that shifting the input pixel values down in significance by up to three bits permits the 17-bit data path to accomodate the most significant bits of the 20-bit result. Thus the convolver design calls for a programmable shift of from zero to three bits in the input data. The shift of the output 12-bit data path with respect to the internal 17-bit data path is similarly programmable from zero to three, so that when subtracting nearly equal quantities more significant bits are preserved.

The convolver is being implemented in custom VLSI circuitry. (The two complete scan lines that must be stored in order to cover the 3-by-3 window are stored in line buffers external to the convolver chip.) The implementation of the convolver (except for the line buffers) in one chip is highly desirable because each convolver requires nine multiplies and adds, which would be excessively cumbersome if done in standard components. Custom VLSI implementation allows the 12-by-5-bit multiplies to be implemented without wasteful use of hardware (standard parts exist for multiplying by four or eight-bit quantities, but not five), and it permits the internal 17-bit data path to be realized directly (standard parts come in multiples of four or six bits).

Custom VLSI is easiest to design when the circuit to be implemented is a regular, repeated structure. Accumulation of successive multiplications can be accomplished most straightforwardly (although not in the fastest manner, nor in the least amount of hardware) by repeated shifting and adding. This means that a 17-by-5 array of one-bit full adder circuits can do the 12-by-5 multiply involved in each of the nine positions in a 3-by-3 pixel window. Thus nine 17-by-5 arrays, for a total of 765 full adders, are needed for the convolver chip. Seventeen-bit latches are used between the nine arrays to store the intermediate accumulated results, as the nine multiplies needed for each pixel are performed on successive clock cycles.

Signed arithmetic is accomplished by the simple expedient of complementing the pixel value prior to multiplication by a negative weight. Since the (one's) complement of the pixel value plus one would produce the negative of that value in two's complement representation, the product of a negative weight and the pixel value is equal to the product of the absolute value of the weight (i.e., strip off sign bit) and the complement of the pixel value plus one. This can be accomplished in the hardware by adding the absolute values of all negative weights together at the beginning (since they are fixed at the time of programming PIFEX), and then adding the pixel values times the positive weights and the complements of the pixel values times the absolute values of the negative weights. Symbolically:

$$\begin{aligned}\sum_{i}^{9} p_i w_i &= \sum_{w_i>0} p_i w_i + \sum_{w_i<0} (-p_i)|w_i| \\ &= \sum_{w_i>0} p_i w_i + \sum_{w_i<0} (\bar{p}_i+1)|w_i| \\ &= \sum_{w_i>0} p_i w_i + \sum_{w_i<0} \bar{p}_i|w_i| + \sum_{w_i<0} |w_i|\end{aligned}$$

where p_i are the pixel values, w_i are the convolver weights, and $\bar{p}_i$ are the one's complements of the pixel values.

In this way a uniform hardware architecture can handle both positive and negative weights. Because this scheme requires that we be able to add an arbitrary number to the sum of the nine multiplies, the convolver can be programmed to add any number to the data stream, which may be useful in PIFEX programming.

Signed pixel values are easily accomodated by replicating the most significant (sign) bit of the 12-bit input value onto all 17 bits of the internal data path. If unsigned pixel values are used, the most significant bit is not replicated. This is determined by a programming bit in the convolver chip.

The current convolver design fits in a 64-pin package (the three 12-bit line buffer inputs, one 12-bit output, plus power, ground, clock and programming pins). As mentioned in Section 4, it is designed to operate at pixel rates up to 8 MHz.

7. BINARY FUNCTION

In Section 3, the need for a programmable binary function (function of two inputs) based on a memory look-up table with linear interpolation was established. The architecture and construction of this device will now be addressed.

The need for a 12-bit data path has been discussed. Given this, the design of the linear interpolation hardware is tightly constrained by commercially available components. There is only one type of memory chip which offers the 120-nanosecond cycle times needed for real-time pipeline processing and the density which makes a single circuit card for the PIFEX module feasible — the 64K CMOS static memory chip. These are organized as 8K by 8; thus blocks of memory using these chips must have an output which is a multiple of 8 bits. Since a 12-bit value plus two slopes must be looked-up in this memory, one has a reasonable choice between a 24-bit-wide table (12 plus two 6-bit slopes), a 32-bit-wide table (12 plus two 10-bit slopes), or a 40 bit-wide table (12 plus two 14-bit slopes). A reasonable compromise between the size of the table and the precision of the result when linear interpolation is inadequate is to interpolate on the least significant 4 bits of each argument. (Also, since 4-bit-wide multiplier chips are available, this choice allows a straightforward implementation with off-the-shelf components, although a custom chip is planned for this.) Thus the look-up is done on the most significant 8 bits of each argument, requiring a total of 16 bits to address the table. A 16-bit value can take on 64K possibilities, so the table must be organized as 64K by 24, 32, or 40. The choice between these three possibilities is determined by the maximum slope which one wants to allow for the stored function. To maintain 1-bit accuracy over the 4-bit interpolation range (16 values), the slope in the lookup table must be able to take on values as low as $1/16$ (a change of one bit over the interpolation range). This means 4 bits to the right of the binary point. Thus the remaining bits in the slope are to the left of the binary point and represent slopes greater than unity. One of these remaining bits must be reserved for the sign of the slope. A 6-bit slope will then have one significant bit to the left of the binary point, for a maximum slope of 2 (actually $1\,15/16$, all ones). A 10-bit slope can represent slopes as high as 32, and a 14-bit slope

can represent values as high as 512. Since mapping 12-bit values onto 12-bit values implies average slopes of about 1 for monotonic functions and 2 for symmetrical functions, a maximum of 4 seems very limiting. On the other hand, a slope of 512 would allow the entire 4096-value range of the 12-bit output to be overflowed well within a single interpolation interval, which seems rather excessive. Thus a 32-bit-wide table is chosen.

These considerations lead to a unique table-look-up design. The 8 most significant bits of each function argument are used to address a 64K by 32 lookup table. Twelve of the output bits represent the function value at each of the 64K values. The remaining 20 bits represent two slopes, each 10-bit signed (two's complement for negative) values, with 4 bits to the right of the binary point. Each of these slopes is multiplied by the four least significant bits of the appropriate function argument, and the two products are added. The four least signifcant bits of the result are discarded, and the remaining portion (10 significant bits plus the duplicated sign bit for two's-complement arithmetic) is added to the 12-bit function value looked up in the table to produce the final interpolated output.

8. NEIGHBORHOOD COMPARISON OPERATOR

The neighborhood comparison operator allows PIFEX to do operations which would be very difficult, if not impossible, within the framework of the two convolvers and the binary function described heretofore. As mentioned in Section 3, these operations include finding peaks, ridges, valleys, etc., as well as region growing, shrinking, and other useful functions.

The operator functions by storing two consecutive scan lines in line buffers and doing the raster-to-window conversion with nine latches, thus making available the nine pixel values of the 3-by-3 window for the comparitors. Using one bit of program control, the eight neighbors are compared to either a programmable constant threshold or to the value of the center pixel, and the center pixel is always compared to the threshold. Another two bits of program determine the sense of the comparison, i.e., greater-than, equal, or less-than. The outputs of the comparitors form a nine-bit value. Two additional bits are appended to this, indicating the evenness or oddness of the raster scan line and indicating the evenness or oddness of the pixel number on that line. The resulting 11-bit number is used to address a 2K-by-14 memory containing arbitrary, preprogrammed information. Two bits of the memory output determine which of the following three 12-bit values becomes the output of the operator: the other 12 bits of the memory table, the center pixel of the window, or the bitwise logical OR of the eight neighbors of the center pixel.

Comparison of two's complement signed data is accomplished by inverting the most significant bit (sign bit) when two's complement representation is used. This makes the positive numbers (sign bit = 0) compare as greater than negative ones (sign bit = 1). One programming bit is used to indicate whether this inversion is to be performed (that is, whether the input is to be interpreted as positive-negative or unsigned integers).

The inclusion (in the table lookup) of the odd-even information about the pixel position is for the purpose of implementing subfields. These two bits allow the definition of four subfields, with different operations being done on each. This feature allows a convenient way of ensuring that a line

(in binary data) is not eliminated when it is desired to thin it to a width of one pixel, as described in [7].

Although only three comparison modes (>, =, <) are implemented, the greater-than-or-equal and less-than-or-equal functions can be achieved by inverting the sense of the lookup table, i.e., $\geq$ is replaced by <, $\leq$ is replaced by >, and the table addresses are complemented before storing the desired lookup table.

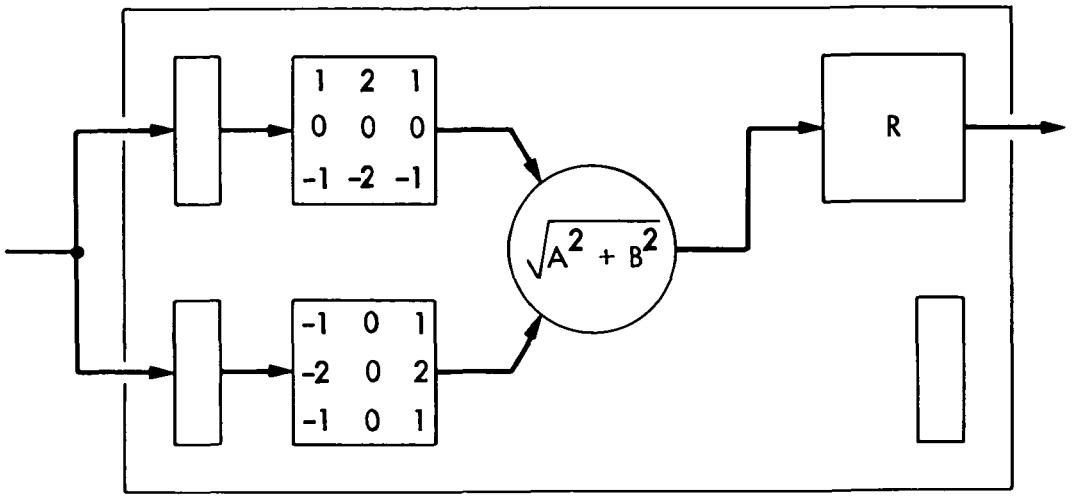
9. EXAMPLES OF USE

In all of the examples shown in this section, details of the programming such as the scale factors of quantities are omitted for simplicity. For example, the weights in the convolvers are shown as integers and the shifts are omitted. In practice, appropriate shifts must be included to prevent fixed-point overflow and to prevent excessive loss of significance in the 12-bit data. (The figures in this section all identify the components in each module according to the layout previously shown in Figure 2.)

Many low-level vision and image-processing algorithms can be performed by PIFEX. These include such things as filtering, detection of edges by various methods, detection of corners and vertices, bridging gaps in lines, shrinking lines, detection of the intersections of lines, comparison of different images to detect changes or motion, simple texture measurement, stereo area correlation, estimation of surface orientation from intensity variations, and estimation of surface color. Some of these, such as complicated vertex detectors or stereo correlation, would require many modules. However, for simplicity only a few algorithms that can be done in a small number of modules will be given here.

First will be the Sobel edge detector [12] plus thinning, which requires only one module, as shown in Figure 5. The two convolvers compute the two components of the gradient (different weights are possible here; the ones shown are for the true Sobel operator), the binary function then computes the magnitude of the gradient, and the neighborhood comparison operator finds the ridges (one-dimensional maxima), so that the resulting nonzero values are only one pixel wide. If the direction of the edge also is wanted, another module in parallel with this one could be used, with the same convolver weights but with the double-argument arctangent as the binary function.

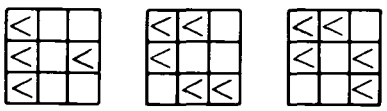
The next example is the computation of the zero crossings of the Laplacian (the trace of the determinant of the matrix of second derivatives, often used in edge detection and stereo correlation [13]) and the Hessian (the determinant of the matrix of second derivatives, sometimes called the Gaussian curvature, often used in corner and vertex detection [14]). These operators



A UPPER INPUT TO FUNCTION (OUTPUT OF A CONVOLVER)

B LOWER INPUT TO FUNCTION (OUTPUT OF B CONVOLVER)

R RIDGE OPERATOR: IF ANY OF THE FOLLOWING PATTERNS OR THEIR ROTATIONS EXISTS:



WHERE < MEANS "LESS THAN THE CENTER" AND BLANK MEANS "DON'T CARE", OUTPUT IS THE CENTER PIXEL; OTHERWISE, OUTPUT IS ZERO
(FOR A THRESHOLDED RIDGE OPERATOR, THE PATTERNS WOULD ALSO INCLUDE "NOT LESS THAN THE THRESHOLD" FOR THE CENTER, AND INSTEAD OF THE CENTER THE OUTPUT COULD BE A SPECIFIED CONSTANT.)

Figure 5. Sobel operator with thinning

are ordinarily applied to smoothed data. Figure 6 shows six iterations of convolving with the averaging function, which results in a fairly good approximation to smoothing with a Gaussian function with a standard deviation of 2 pixels. The Laplacian (L) and the Hessian (H) of the result are then computed, and two neighborhood comparison operators and one arithmetic function are used to find the zero crossings of the Laplacian. (The first neighborhood comparison operator produces approximate zero-crossings; the other two steps force them to be only one pixel wide, by choosing the pixels closest to zero. Better 3-by-3 approximations to the derivative operators are available than the simple ones shown here. Also, if heavier smoothing were done, some of the smoothing should be done after differentation because of the limited precision.)

Now we have an example in the numerical solution of partial differential equations, namely two-dimensional heat flow. The basic equation here is the following:

$$\frac{\partial T}{\partial t} = \frac{1}{c}\nabla\cdot(k\nabla T)$$

where T is the temperature, t is time, k is the thermal conductivity, c is the heat capacity per unit volume, and ∇ is the vector derivative operator. Figure 7 shows a simple way to code this into PIFEX. (The shifts caused by the use of unsymmetrical weights in the convolvers cancel out.) The values k and c, which are constant with respect to time but not with respect to position, are preloaded into image buffers by the host computer. (Actually, it is better to use some nonlinear representation of c, such as 1/c, in order to make better use of the available dynamic range with the fixed-point data. For example, a perfect conducter could be indicated by setting 1/c to zero.) The initial values for T are preloaded into another buffer, but they will be changed by PIFEX. Each pass through the stages shown would correspond to one iteration, with time interval Δt. Several of these could be done on one pass through PIFEX to save time (according to the number of modules that are available), and separate passes (without reprogramming) on consecutive frame times can produce more iterations, with the T values resulting from each pass being available in the external buffer. The k and c values are shown being carried along through additional modules so that they will be available in the

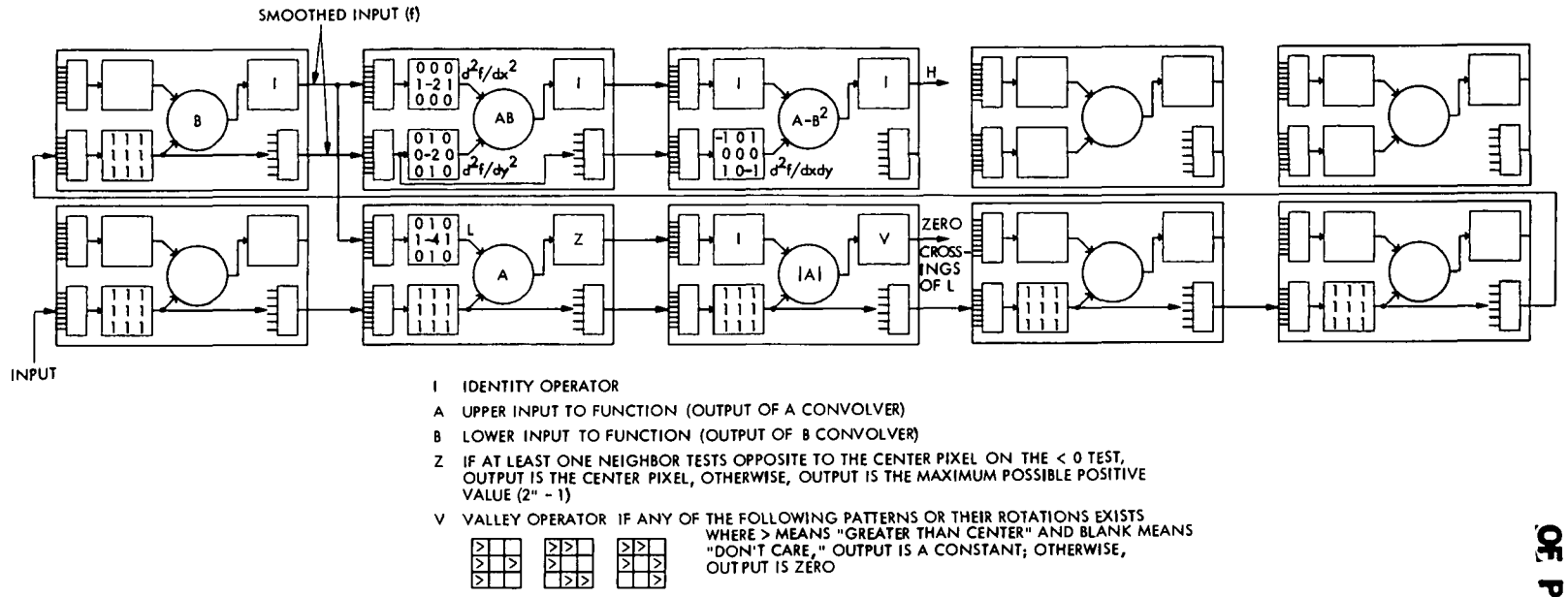
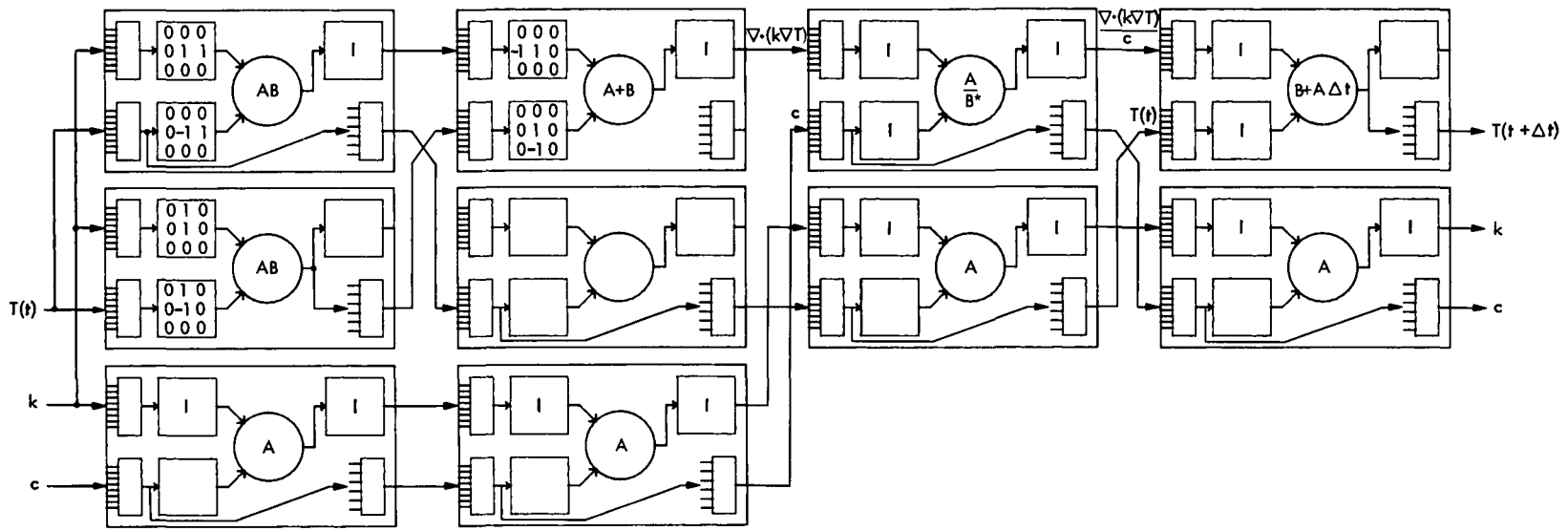


Figure 6. Computation of Hessian and zero crossings of Laplacian (using approximate Gaussian smoothing with $\sigma = 2$)



I IDENTITY OPERATOR
A UPPER INPUT TO FUNCTION (OUTPUT OF A CONVOLVER)
B LOWER INPUT TO FUNCTION (OUTPUT OF B CONVOLVER)
* SEE TEXT FOR ALTERNATE REPRESENTATIONS FOR C

Figure 7. Two-dimensional heat flow

modules used in subsequent iterations on this pass; if only one iteration were done in each pass through PIFEX, this would not be necessary. (Many problems involving the numerical solution of partial differential equations would not be suitable for PIFEX because of the low precision and dynamic range caused by its use of 12-bit integers and because of the fact that it operates on two-dimensional data.)

PIFEX can be used for robotic route planning. A modification of an algorithm developed by Witkowski [15], which is tailored for parallel processing, can be implemented on PIFEX. It involves growing in consecutive circles from the beginning and endpoint of the desired route. Each consecutive circle is numbered one greater than the previous ring, and so represents the distance from the center. Obstacles halt the expansion of the rings, which then become distorted as they propagate at constant speed around the obstacles. When the rings from the beginning reach the endpoint (and simultaneously rings from the endpoint reach the beginning, since the distances must be equal), the processing is stopped, and the two sets of ring numbers are added. The sum at the beginning and endpoints are equal to the shortest distance between these points (around the obstacles), and furthermore the best path is identified as the connected points all having this same sum (since moving along the path reduces the distance to one endpoint and increases it by the same amount to the other endpoint).

Several techniques for performing such an algorithm in PIFEX are possible. The one which follows has the advantage that no reprogramming is needed (at least during the main growing phase) no matter how long the path is, and thus it is faster for long paths, even though it requires more modules per stage of growing than some of the other methods. One image buffer is loaded with the obstacles, coded with some special number, and zeros elsewhere. One image buffer is loaded with all zeros except a one for the beginning point of the route. A third image buffer is loaded with all zeros except a one at the end of the route. The growing process is implemented in typical PIFEX stages as depicted in Figure 8. Alternate stages of neighborhood comparison operators are used to do four-neighbor or eight-neighbor growing, thus approximating a circle with an octagon. (Better approximations are possible with some of the other methods.) The neighborhood

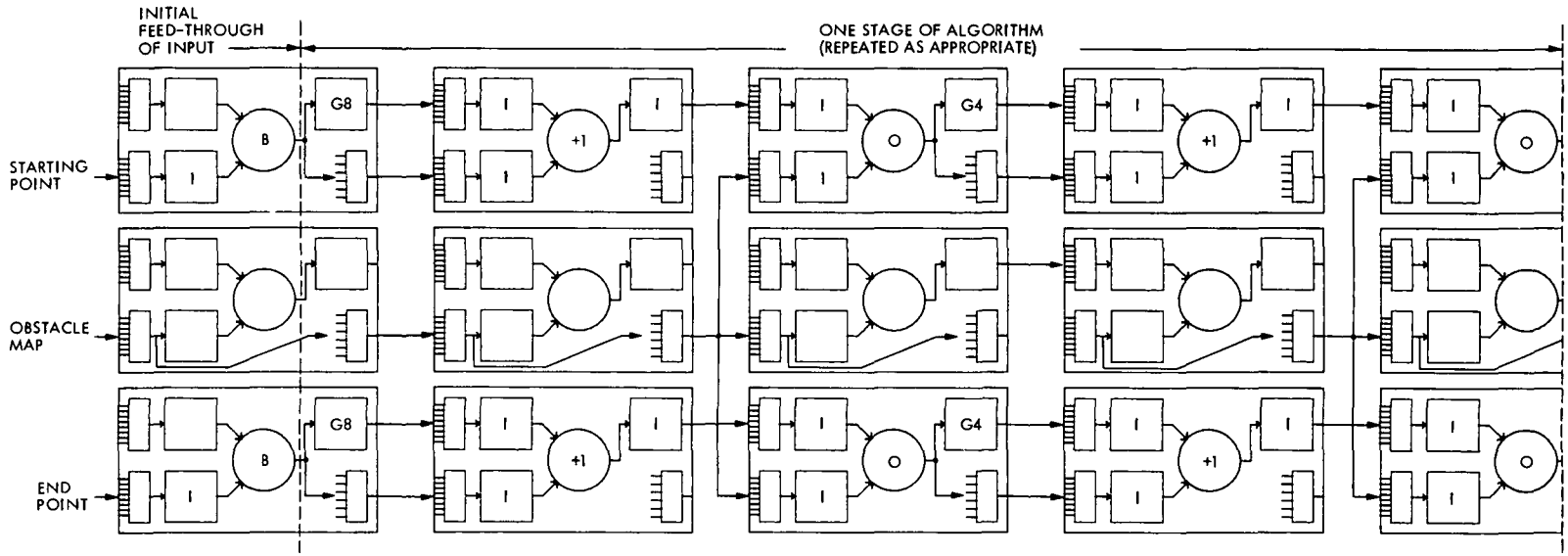


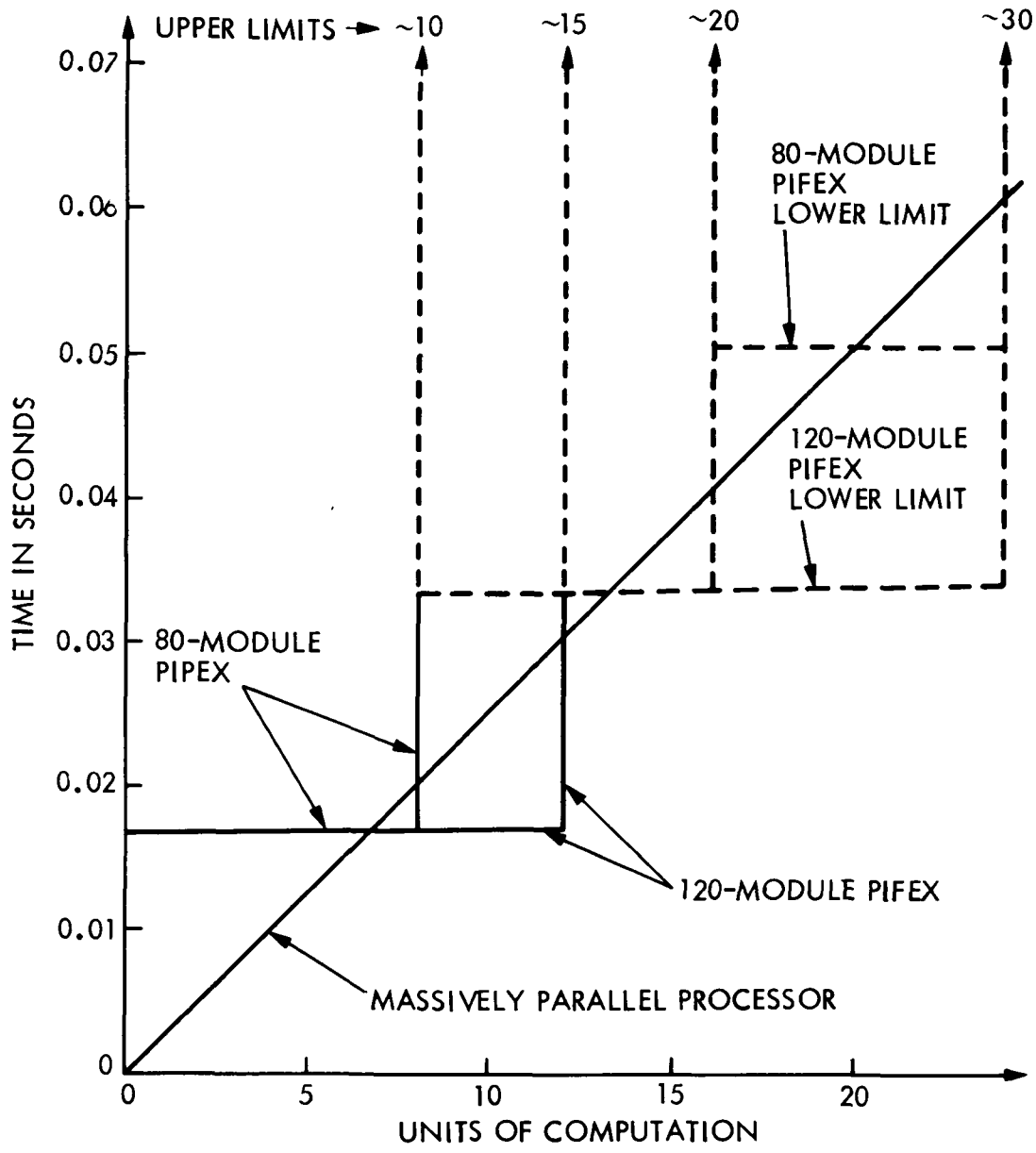
Figure 8. Route planning through obstacles (growth phase)

operator is used to take the logical OR of the neighbors of a given pixel (after comparing all nine pixels in the 3-by-3 window for "greater-than-zero," and in alternate stages coding to ignore corner non-zero neighbors). Since the eight-neighbor test will always be used on even iterations (in terms of the count to be produced by the iteration, which is one greater than the distance), the nonzero values that the OR function sees on these iterations will be two consecutive integers, the larger of which is odd. Thus the OR will produce the larger value, as desired. (On odd iterations, only one nonzero value can be seen by the OR function, since the previous iteration grew according to eight neighbors.) The unmodified value of the pixel is passed to the next stage of the pipeline via the B output, where the binary function passes that value if it is not zero, or if both inputs are zero. If the B input is zero, but the A input (logical OR) is not, then we have the condition that a concentric ring should be grown into the center pixel (i.e., the center is zero but at least one of the neighbors is not). In this case the binary function adds one to the logical OR value. The next binary function (in the next stage) is used to overlay the rings onto the obstacle map, with the function programmed to set that portion of the ring to zero which lies on an obstacle. In this way, the rings are forced to go around obstacles. A parallel set of modules is used for the rings propagating in the other direction.

As many stages of the above processing are used as needed to insure that the rings reach the other endpoints. If this requires more modules than are available (which usually would be the case), the images can be recirculated as many times (on successive frame times) as needed through PIFEX (without reprogramming), with the intermediate results being stored in image buffers from one frame time to the next. When the host computer detects (by examining the endpoints in the buffers) that the endpoints have been reached, it stops this processing and reprograms a portion of PIFEX so that a single PIFEX module (not shown in the figure) adds the two sets of rings and finds the shortest route as the points where this function is equal (within one) to its endpoint value. Some more modules can be used to thin this path to a width of one pixel, and the host computer can read the best route from this map in an external image buffer.

Finally, we present, not a specific example, but a comparison of the performance of PIFEX and the performance of the Massively Parallel Processor (MPP) [3] on problems containing a typical mix of operations, chosen so as to be reasonably suitable for both machines. (It is possible to create examples for which one machine or the other is at a great disadvantage. For example, problems needing more precision than 12 bits could not be done on PIFEX, and problems needing no convolutions or neighborhood comparison operators would not utilize PIFEX very effectively. On the other hand, problems needing many general convolutions, elaborate neighborhood comparison operators, or transcendental functions would slow down the MPP considerably.) The particular types of computations used are shown in Figure 9. The dimensions of the image (256 by 384) were chosen to be small multiples of 128, so that the image can be handled efficiently by the MPP, and to have magnitudes such that the common rate of 60 frames per second for PIFEX allows significant vertical and horizontal blanking intervals. (This is more than the normal amount of blanking. By making the blanking intervals very small, PIFEX could use 80 frames per second on images of this size.) The assumption that 10 modules are needed to perform one unit of computation as defined in the figure means that only 30% of the convolvers (6 out of the 20 in 10 modules), 30% of the binary functions (3 out of the 10 in 10 modules), and 10% of the neighborhood comparison operators (1 out of the 10 in 10 modules) are being used (for anything other than identity operations). Experience with complicated algorithms indicate that these figures are typical, although wide variations are possible.

Figure 9 shows the amount of time needed to process an image as described above, as a function of the amount of computation needed. (The times for PIFEX do not include the delay in PIFEX, since this does not affect the rate at which data can be processed.) For the MPP, the time is proportional to the amount of computation, since it is an SIMD machine. (These times are computed assuming that the precision used in the MPP is the same as in PIFEX, as shown in the figure. The time required by the MPP increases as the precision increases.) For PIFEX, the time is constant at one frame time as long as the computation can be done in one pass. However, when the number of modules needed exceeds the number of modules in PIFEX, one or more frame times are needed for extra passes, plus whatever time is needed for



1 UNIT = 5 3-BY-3 SUMS
+1 3-BY-3 GENERAL CONVOLUTION (5-BIT WEIGHTS) WITH SAME DATA
+1 PRODUCT
+1 SUM
+1 SUM OF SQUARES
+1 3-BY-3 COUNT OF THOSE ABOVE THRESHOLD

} OVER ENTIRE IMAGE, WITH 12-BIT DATA

ASSUMED TO USE 10 MODULES IN PIFEX (EFFICIENCIES $\leq$30%)
IMAGE IS 256 BY 384, SCANNED AT 60 Hz IN PIFEX (26% FOR BLANKING)

Figure 9. Times for typical computations

reprogramming PIFEX between passes. The lower limit occurs when no reprogramming is needed (as in the route planning example, Figure 8). The upper limit occurs when everything in PIFEX needs to be reprogrammed with different information. This upper limit is seldom reached, because usually the same common binary functions (such as addition and multiplication) are programmed into many different modules, and the binary functions require the most time for loading the programming information, because of their large table lookup. However, the times required often are far above the lower limit. It can be seen from the figure that for the particular types of computation shown, the performances of PIFEX and the MPP are roughly comparable, as long as the task is not very small and extensive reprogramming of PIFEX is not needed. (The MPP costs between one and two orders of magnitude more than a PIFEX of the sizes considered.)

REFERENCES

[1] Etchells, D. "A Study of Parallel Architectures for Image Understanding Algorithms," in ISG Report 104 (R. Nevatia, editor), pp. 133-176, Univ. So. Cal., Los Angeles, CA, October 19, 1983.

[2] Reeves, A.P. "Parallel Computer Architectures for Image Processing," Computer Vision, Graphics, and Image Processing **25** (1984), pp. 68-88.

[3] Strong, J. "NASA End to End Data System, Massively Parallel Processor," Goddard Space Flight Center, Greenbelt, MD, May 30, 1980.

[4] Codd, E.F. Cellular Automata, Academic Press, New York, 1968.

[5] Rosenfeld, A. Picture Languages, chapter 5, pp. 103-138, Academic Press, New York, 1979.

[6] Golay, M.J.E. "Hexagonal Parallel Pattern Transformations," IEEE Transactions on Computers **C-18** (1969), pp. 733-740.

[7] Gennery, D.B., and Jordan, S.D. "Feature Extraction by Parallel Pattern Transformations Using Square Neighborhood Logic," RCA Engineer, Vol. 19, No. 3 (Oct.-Nov. 1973), pp. 68-74.

[8] Lougheed, R.M, McCubbrey, D.L., and Sternberg, S.R. "Cytocomputers: Architectures for Parallel Image Processing," IEEE Workshop on Picture Data Description and Management, Pacific Grove, CA, August 1980.

[9] Eskenazi, R., and Wilf, J.M. "Low-Level Processing for Real-time Image Processing," JPL Publication 79-79, Jet Propulsion Laboratory, Pasadena, CA, 1979.

[10] Harland, W.L., and Eversole W.L. "Affordable Implementations of Image Processing Algorithms," 26th SPIE conference, San Diego, CA, August 1982.

[11] Moravec, H.P. "Fully Interconnecting Multiple Computers with Pipelined Sorting Nets," IEEE Transactions on Computers C-28 (1979), pp. 795-798.

[12] Duda, R.O., and Hart, P.E. Pattern Recognition and Scene Analysis, J. Wiley and Sons, New York, 1973.

[13] Marr, D., and Hildreth, E. "Theory of Edge Detection," AI Memo No. 518, Artificial Intelligence Laboratory, Mass. Institute of Technology, Cambridge, MA, April 1979.

[14] Dreschler, L., and Nagel, H.-H. "Volumetric Model and 3D-Trajectory of a Moving Car Derived from Monocular TV-Frame Sequences of a Street Scene," Proc. Seventh International Joint Conference on Artificial Intellegence, Vancouver, B.C., August 1981, pp. 692-697.

[15] Witkowski, C.M. "A Parallel Processor Algorithm for Robot Route Planning," Proc. Eighth International Joint Conference on Artificial Intelligence, Karlsruhe, West Germany, August 1983, pp. 827-829.

TECHNICAL REPORT STANDARD TITLE PAGE

1. Report No. 84-97	2. Government Accession No.	3. Recipient's Catalog No.
4. Title and Subtitle PIFEX: An Advanced Programmable Pipelined-Image Processor		5. Report Date February 1, 1985 6. Performing Organization Code
7. Author(s) Donald B. Gennery and Brian Wilcox		8. Performing Organization Report No.
9. Performing Organization Name and Address JET PROPULSION LABORATORY California Institute of Technology 4800 Oak Grove Drive Pasadena, California 91109		10. Work Unit No. 11. Contract or Grant No. NAS7-918 13. Type of Report and Period Covered External Report JPL Publication
12. Sponsoring Agency Name and Address NATIONAL AERONAUTICS AND SPACE ADMINISTRATION Washington, D.C. 20546		14. Sponsoring Agency Code RE156 BK-506-54-65-24-00

15. Supplementary Notes

16. Abstract

PIFEX is a pipelined-image processor being built in the JPL Robotics Lab. It will operate on digitized raster-scanned images (at 60 frames per second for images up to about 300 by 400 and at lesser rates for larger images), performing a variety of operations simultaneously under program control. It thus is a powerful, flexible tool for image processing and low-level computer vision. It also has applications in other two-dimensional problems such as route planning for obstacle avoidance and the numerical solution of two-dimensional partial differential equations (although its low numerical precision limits its use in the latter field). The concept and design of PIFEX are described herein, and some examples of its use are given.

17. Key Words (Selected by Author(s))	18. Distribution Statement
Computer Operations and Hardware Computer Programming and Software Computer Systems	Unclassified; unlimited

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of Pages	22. Price
Unclassified	Unclassified		