Parallelism in Graphics and its Representation in Data-flow Language and Architecture

Abel Ferreira

A Thesis
in
the Department
of
Computer Science

Presented in Partial Fulfillment of the Requirements
for the Degree of Master of Computer Science at
Concordia University
Montréal, Québec, Canada

March 1985

© Abel Ferreira, 1985

ABSTRACT

Parallelism in Graphics and its Repesentation in Data-flow Languages and Architectures

Abel Ferreira

This thesis is a theoretical study of parallelism in graphics and the possibility of using data-flow architecture to support this parallelism. It starts with surveys of graphic systems, data-flow architectures and languages. It demonstrates that graphic parallelism can in effect be supported by data-flow concepts. A data-flow graphic architecture is presented as a result of this integration.

ACKNOWLEDGEMENTS

I received much help during the preparation of this thesis. I wish to thank Matrox Inc. for providing equipment and resources. To D. Lovegrove for her contribution to the illustrations of this document. To my co-workers for their encouragement and fruitful discussions about the ideas presented here. To my wife a word of appreciation for her support and patience, without which this work was not possible. My deepest recognition, however, is to professor Fancott, who patiently allowed my research to proceed in new directions, but guided me with his comments. This document owes much to him.

Contents

1	Introduction	1
2	Graphic Systems Model	5
3	Data-flow Architecture	18
3.1	Data-flow Concept	19
3.2	Data-flow Programs	21
3.3	Data-flow Computers	24
3.3.1	M.I.T. Data-flow Computer	25
3.3.2	DDM1 Utah Data-driven Machine	27
3.3.3	Irvine Data-flow Machine	28
3.3.4	Other Data-flow Computers	29
4	Data-flow Programming Languages	34
4.1	The VAL Programming Language	35
4.2	Data-flow Machine Language	44
5	Graphic Systems and Data-flow Computers	52
5.1	Picture Representation	52
5.2	Parallelism in Graphics	56
5.3	Picture Description with VAL	64
5.4	Data-flow Architecture for Graphics	75

6	Data-flow Graphics Computer	78
6.1	The Operation Units	79
6.1.1	Vector Generator	79
6.1.2	Raster Operators	83
6.1.3	Normaliser	87
6.1.4	Transformer	91
6.1.5	Projection Units	93
6.1.6	Complex Picture Generator	94
6.1.7	Clipper	95
6.1.8	Input Devices	97
6.1.9	ALU	98
6.2	Bit Map Interface	100
6.3	Data-flow Ring Interface	105
6.4	Data-flow Ring Architecture	107
6.4.1	Instruction Cells	107
6.4.2	Arbitration and Distribution Networks	110
6.5	Structure Storage and Processing	115
7	Conclusion	119
	References	121
	Appendix I	132
	Appendix II	135
	Appendix III	138
	Appendix IV	147

Figures

2.1 General System Model 7

2.2 General Graphic System Model 10

2.3 IDS Block 12

2.4 DPU Block Diagram 15

3.1 (a+b)/(c+d) Graph 22

3.2 Gate Operators 23

3.3 M.I.T. Data=flow Computer 26

3.4 uPD7281 30

3.5. M.I.T. with Structure Support 32

5.1 Hierarchical Circuit Design 58

5.2 Schematic Block 60

5.3 Graph of Circuit Block 65

5.4 Data-flow Graphics Computer 77

6.1 Vector Generator Block Diagram 81

6.2 Working Areas 84

6.3 Raster Operator Module 85

6.4 Raster Operator Block Diagram 86

6.5 Window and Viewport 88

6.6 Normaliser Block Diagram 90

6.7 Transformer Block Diagram 92
6.8 ACRTC Block Diagram 95
6.9 Window Clipping 96
6.10 Input Module 98
6.11 ALU Module 100
6.12 Video Signal 101
6.13 Memory Chip Block 102
6.14 Bit Map Interface 103
6.15 Arbitration Timing 104
6.16 Data-flow Interface 106
6.17 Instruction Cells Module 109
6.18 Arbitration Network 111
6.19 Arbitration Module 113
6.20 Distribution Network 115

1

Introduction

Graphic systems are a mandatory component for almost any type of application. If for some, simple graphic routines are enough (business and education), for others a sufficiently high performance graphic system does yet not exist (CAD/CAM, image processing, mapping and animation). The search for such a machine is presently concentrated in the increase of graphics 'intelligence', i.e. the optimization of the speed of devices and the integration of more functions in silicon. Very little has been done, however, in the study of new architectures that could explore other ways of increasing the performance of graphic systems. From our past experience we have realized that the inherent parallelism of graphics operations was not fully exploited. The search for an architecture that best supports this parallelism is the origin of this work.

On our initial literature search we realized the potential of data-flow architectures to support parallelism (their characteristics, the availability of high-level languages, the existence of several prototype machines and the announcement of a data-flow VLSI device dedicated to

image processing). Further research confirmed this possibility and defined the scope of this work: a theoretical study of graphic systems, data-flow computers, data-flow languages and the integration of the two domains in a basic data-flow architecture for graphic systems.

This document starts with a survey of raster scan displays, their evolution and characteristics. Because of the extent of the subject, the analysis was concentrated on details of current graphic architectures to investigate their limitation at handling parallelism in graphics.

Chapter three is another survey, in this case of data-flow architectures and computers. First, the concept of data-flow is explained and compared in its performance to handle parallelism with control-flow architectures. The document continues with the description of data-flow graphs and how they are used to represent data-flow programs. Details of a small graph language are given. This chapter finishes with the operational description of a data-flow computer and an overview of the different types of data-flow architectures.

High-level languages for data-flow architectures is the subject of chapter four. It is a description of VAL, the characteristics of the language, the syntax and semantics of the most representative commands, and the data types and structures. To terminate the overview of this

high-level language we discuss the format of data-flow instructions and give an example of a data-flow machine language program derived from a VAL program sample.

Chapter five brings together this information in an analysis of the integration of graphics and data-flow architectures. It starts with a description of different methods of picture representation and an overview of graphic standards. Parallelism in graphics is the next topic. It is analysed from the application down to the implementation level. An example is used to show the inefficiency of expression of this parallelism in conventional languages. The same example is then described in a VAL program which is analysed in order to illustrate the support it is capable of providing for graphics parallelism. Other issues that refer to the quality of the representation are also studied. The example is also a vehicle to explain VAL instructions, how graphic functions are integrated, and the GKS language binding. Finally the chapter concludes with the presentation of a basic data-flow graphics processor based on the M.I.T. type of data-flow architecture.

The final chapter of this work is a detailed description of the basic architecture presented in the previous chapter. Each section of the computer is described, its function analysed and in most of the cases a

block diagram illustrates a possible implementation. Each operation unit is also presented with a subset of an instruction set whenever possible.

The conclusion of this document refers to areas of further research and analyses the results of this work.

2

Graphic Systems Model

In this chapter an analysis of graphic systems is performed. Because of the considerable extent of the subject the analysis is directed towards the more recent raster scan systems.

The first part is a small historical overview of graphic systems which is followed by a list of the operations usually performed by such systems. Afterwards a description of current graphic systems at the concept level is given with examples of possible implementations .Finally we will show the limitations of these architectures and specify the aspects that can not be improved without major changes in the architecture.

Early graphic systems used display technologies other than raster scan such as storage tubes and stroke writing refresh tubes. The development of high resolution video screens and the decrease in price of solid state memories have made the raster scan display an increasingly attractive alternative to the storage tube display. The new technology brought improvements in graphic capabilities as well as new implementation problems. In the first category,

an example is selective erasing (in storage tubes the whole picture had to be erased and a new one reconstructed every time a part of the picture had to be changed), in the second the aliasing of the picture (oblique lines have a stairway aspect) and the need for solid state memories with very fast access times.

A further aspect in the evolution in graphics systems was triggered by the availability of microprocessors and cost effective dynamic rams. When solid state memory and cpu's where expensive it was the responsibility of the main and, most of the time, only CPU to execute the operating system, control the peripheral devices, perform all graphic transformations and finally convert the image expressed in vectors to a bit map representation that was sent to the screen via specialized hardware.

Cost effective RAM's made it practical to separate the main memory from the video memory (the one that contains the bit map representation of the image). Consequently the main memory was free from the raster scanner accesses that slowed down its throughput due to the frequency of the accesses. The video memory added was a dual ported memory that gave access both to the main CPU and the video scanner. With the availability of denser and less expensive RAM's the video memory was increased both in size (resolution of the screen) and in depth (number of bits per

pixel).

The development of powerful, low cost microprocessors made it possible to use a dedicated CPU for the graphics operations. Therefore the video system was composed not only of the scanner and the video memory but also of a CPU (simple microprocessor, a bit-slice, custom designed VLSI or a pipeline of processors). The availability of this additional processing power reduced the number of tasks to be executed by the main CPU. In fact all the graphic transformations were transferred to the local CPU (the one in the video section), for example the picture bit mapping, rotations, translations, window clipping and others.

Further improvements are possible as we will show later on, but in the following we will describe and analyse the general model of a graphic system [Fole82,Free79].

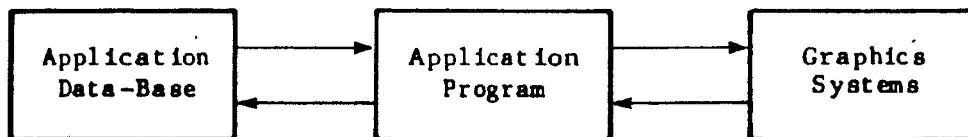

Fig. 2.1: General System Model

The application data-base provides the model and data

structures of the problem to be handled by the system. With the application program the model represented by the data-base can be viewed and operated by the user through the graphic system interface.

The application program has to describe to the graphic system in geometric terms that portion of the model from which the user wants a picture. In the other direction the graphic system provides geometric information to the application program as a result of user interaction. This information is interpreted by the application program in order to modify or add new elements to the data-structured model.

The form, type and amount of information passed between the graphic system and the application program is different from system to system. The analysis of this link can reveal the characteristics of the graphic system.

In earlier systems the information about the contents in the window (the part of the world or model that the user wishes to see) was provided by graphic output primitives such as points, lines, polygons or graphic strings. The coordinates of these primitives had already been transformed to the view-port (the portion of the screen where the window is to be mapped) and device coordinates.

Subsequent systems improved the set of graphic primitives with the addition of new primitives and

attributes, a more comprehensive set of input devices and pick support, more flexibility in the specification of coordinate values and the possibility to call segments of the picture definition. This last feature is equivalent to a CPU subroutine jump/return mechanism and it is a very efficient way to represent a display that has repetitive elements.

Another important advancement was made when the coordinates of the display list (the geometric information passed between the application program and the graphic system) were changed from the device space to the world or model space and support to modeling/viewing transformations was provided. Structured display file capability for traversing an object hierarchy and composing instance transformations were therefore necessary. For 3D systems perspective and projection operations were added to the set of available graphic commands.

The elements mentioned above are integrated in the following block diagram of a general graphic systems model [Fole82].

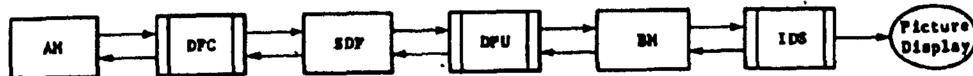

Fig. 2.2: General Graphic System Model

AM (the application model)-Contains a graphical and non-graphical description of an object in a format determined by the application program.

SDF (the structured display file)-Contains a hierarchic description of the graphical representation of the object in integer or floating point world coordinates.

BM (the bit map)-A buffer which holds the scan-converted image.

The logical processors in the system are:

DFC (the display file compiler)-The part of the application program containing the model traverser and calls to the graphics package to map the AM to the SDF.

DPU (the display processing unit)-This unit maps the SDF to the BM.

IDS (image display system)-Reads the bit map and does any color table mapping before generating display monitor signals.

Since most graphic systems approach this model, it is

desirable to have a standard for the display file. Unfortunately a consensus does not yet exist among the different standards available, which makes porting application programs to different graphic systems difficult. Nevertheless there are some manufacturers that use one of the standards available (Core, GKS, GSX and others) with or without modifications. In this work we choose GKS as an example but the work is applicable to any of the other standards.

A visible characteristic of the graphic system model is the pipelining of its elements. To achieve some parallelism and increase throughput this pipelining is used in the physical implementation of the highest performance systems. There are nevertheless certain problems with this type of implementation of a graphic system. The first one is the fact that the pipeline also exists in the reverse direction using the same physical path. The second is that execution times in each element of the pipe as well as in the same element can differ for different operations. Finally, the pipeline does not work with a continuous and constant flow of load. These characteristics do not allow an efficient use of the pipe. For comparison with our architecture presented later, we provide a block diagram of the DPU and IDS of the highest performance systems available today [Door84].

The main function of the IDS is to transfer the image represented in the bit map to the display. The simplest way to do this transfer is to organise the bit map as a one to one representation of the screen and let the IDS scan it in synchronisation with the video rate. Because the memory devices of the bit map cannot match the fast pixel rates of contemporary systems, these are normally accessed in parallel to provide more than one pixel/access. The pixel information is then serialized via shift registers. To add flexibility to the color manipulation, a look up table (LUT) is added between the shift registers and the digital to analog converters. With a LUT it is possible to have multiple planes with dynamic assignment of priority and limited animation [Levo77,Shou79].

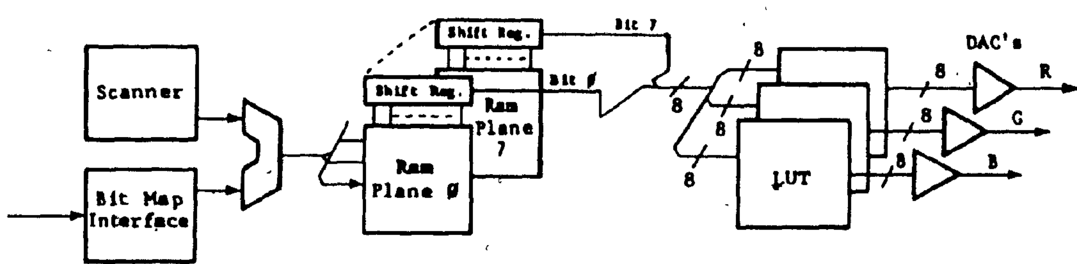

Fig. 2.3: IDS Block Diagram

Another possibility for the IDS is to provide image transformation from the bit map to the screen. Presently

the type of transformation available is scaling and translation. With these features the bit map is normally 'bigger' than the screen and it can contain more than one image. The IDS will scan each of the images to different parts of the screen (called view-surfaces). Each one of the view-surfaces can sometimes be zoomed (by pixel replication) and pan/scrolled independently. A more difficult transformation is rotation which is not available in current systems.

The last function for the IDS is the mixing of video images. This can be done at the DAC level or at the LUT level. The last one is more powerful becaused the displays can be overlyed (mixed) in almost any possible combination depending on how the LUT is programmed.

The DPU in the most complex systems has to perform the following functions:

a-Traverse a hierarchical display list that contains the graphical information about the model represented by the application data-base. This information is provided in master coordinates (the coordinates used in the definition of each object) and normally in floating point format.

b-Perform modelling and instance transformations to construct the world coordinate model.

c-Perform viewing transformations which is a world coordinate to a viewer coordinate transformation.

d-Clip to the window limits in the viewer system.

e-Perform projection transformations.

f-Transform the previous result into a normalised device coordinate space by executing window to view-port transformations with hidden line elimination.

g-Translate the final graphic information into the bit map memory.

In an interactive system the DPU has to give to the application program the objects that are pointed by an input device or just a coordinate when building a new object. In the first case the DPU has to traverse the display list and find out the objects that are intersected by a small window centered around the coordinate given by the input device, after being transformed from device coordinates into world coordinates.

There are several possible implementations of machines that execute all these functions. Depending on cost, technology and other factors the implementations are following a trend where each step of the process is executed by a separate processor. This device is no longer a general purpose CPU but an operation unit dedicated to a specific function, for example, a floating point unit adapted to the graphic resolutions and to perform matrix operations. Another is a unit dedicated to do clipping and in the last stage, the most prolific one, vector

generators, pixel operators and so on. On the other end a dedicated display list traverser can substitute for the general CPU used in the front end for the same function.

A DPU can then be represented by the following block diagram:

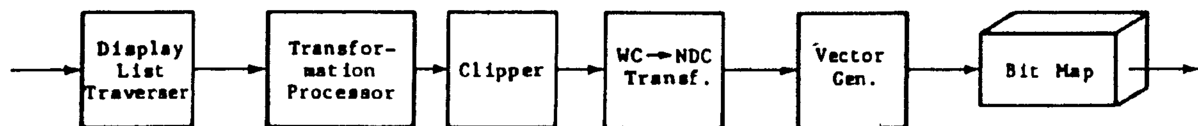

Fig. 2.4: DPU Block Diagram

It has been found that the graphic systems have pertinent characteristics that can be used to improve performance. One is the use of dedicated hardware for specific functions as it was shown in the previous paragraph. Another is the mapping of each step of the pipeline onto a physical component (normally a CPU or dedicated processor) of a particular implementation [Fole82,Door84].

Most of these issues are tied to the evolution of graphic systems and are therefore limited by the way this evolution took place. For example, the pipeline aspect of

the graphic systems was emphasized over other aspects like the inherent parallelism of the display list interpretation and image drawing on a screen. The principal reason for this emphasis was the fact that the first system had only one CPU which had to execute all the phases of the pipeline, each one represented by a separate sub-program. When CPU's and other logic devices become available at lower cost and with more functionality the natural approach was to implement the existant sub-programs in different processors.

In this work what we are proposing to do is to look more deeply into the computer graphic process. The most striking aspect is that many operations in this process can be done in parallel, from the simple case of the vectors representing an object that can be drawn simultaneously, to the case of an hierarchical display list where each object can be transformed and drawn in parallel. Of course this is only possible if we are not restricted to a single processor. With multiprocessors the question becomes how should they be organized?

We have shown the first approach which is a pipeline of processors that achieve parallelism whenever there is a stream of data that requires a fixed sequence of graphic operations which is not always the case. Another approach is a parallel architecture. With this architecture will be

possible to support a single operation applied to multiple data which is the case, for example, of a graphics transformation applied to each object of a display list. The problems associated with this architecture are: The expression of a structured display file in a form suitable to the architecture, the support of sequential operations and the routing of the information flow between the different operation units.

A data-flow architecture is able to support both sequential and parallel operations due to the flexibility of its principle. The flow of information is inherent to the architecture and therefore directly supported by its implementation and finally there are high-level languages that can be used to express a structured display file. The characteristics of a data-flow architecture are presented in general terms in the next sections.

3

Data-flow Architectures

The principles of computer architectures based on the von Neumann organization are the following:

a) a single computing element that includes processor, memory and communications;

b) memory organized in fixed size cells, with linear addressing;

c) control of computation is sequential and centralized.

These principles are reflected in the operation of such architectures. For example the evaluation of an expression such as (a+b)/(c+d) is done by specifying to the machine a sequence of operations that are executed one after the other (Addition, Addition and Division). The operands of each one are available from a specified memory cell and the result is also stored in a memory cell. By giving to the machine the operations to be executed and the correct memory cell where the operands are stored the program is able to calculate the value of the expression. Other consequences of the von Neumann principles are the passive nature of the memory, the cells that store data or

operands are indistinguishable from the cells that store the instructions or operators and parallelism is reduced to the switching of a processor among separate processes or programmer specified decomposition of a program into parallel instruction streams to be executed by separate processors.

3.1 Data-flow Concept

The need to increase the throughput of computers by using the inherent parallelism of some applications has promoted the research of new architectures that have a more natural support for parallel computation. The resulting computers can be broadly classified in two fields: data-flow and demand-driven. The main characteristic of these two types of architectures is the way of controlling their operation. It is no longer a program counter that selects the next operation to be executed but the availability of all operands of an instruction in the first case and the need of a result in the second case. To clarify the two concepts let us consider the evaluation of the expression given above. In a data-flow computer the availability of the four values (a,b,c and d) will trigger the execution of the two additions. Only when the results of these are

available is the division then executed, because only at this time are the operands of this operation available. In a demand-driven computer the need of a result from the expression will trigger the execution of the division, but this one needs the results of the two additions in order to proceed. Finally these can be executed because the results they need are the values of a,b,c and d.

The possibility of executing instructions in parallel and without explicit information from the programmer is one of the main benefits of these architectures and also the main reason for extensive research in the implementation and application of the two new types of architectures.

Before making a more detailed examination of data-flow computers, we will give a brief comparison of the three architectures: control-flow or von Neumann, data-flow and demand-driven [Trel82].

With a control-flow there is a complete control over the sequence of executions but with the disadvantage of having to impose a programming discipline to avoid run-time errors (like the execution of data as a program).

A high degree of implicit parallelism resulting from the execution of instructions as soon as their operands are available is the advantage of data-driven computers. Their disadvantage is the possibility that instructions may waste time waiting for unneeded arguments like for example an if-

then-else which will only use two of its three arguments.

The advantage of demand-driven organizations is the fact that only the instructions whose result is needed are executed and the mechanism of procedure-calling is built in. On the other hand, the disadvantage of such architectures is the wasted effort of propagating demands for the evaluation of expressions such as arithmetic where every instruction always contributes to the final result.

Finally a common advantage of both data-flow and demand-driven programs is that they are free from side effects, making them suitable for distributed implementation.

3.2 Data-flow Programs

In the previous paragraphs we presented the concept of data-flow with a small expression. Now we will expand the concept to more general problems and at the same time introduce the subject of data-flow languages.

The most natural way of representing a data-flow program is using a graph [Denn72]. In it, nodes represent operators and the arcs that connect these nodes represent the operands that flow from one operation to another. The expression given above can be represented by the following

graph:

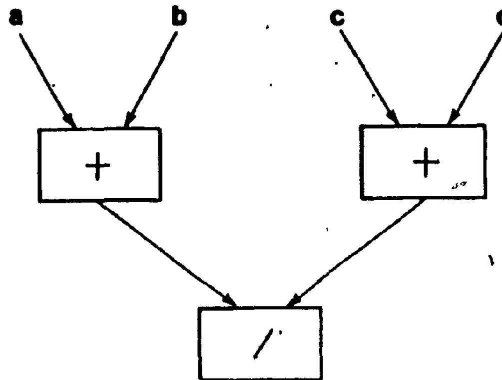

Fig. 3.1: (a+b)/(c+d) Graph

The rules used to write and execute a graph are the following:

a) In an operator node there are as many arcs coming in as operands.

b) The result of the operation is provided at the arc leaving the operator node.

c) Data links are nodes that have only one incoming arc and multiple outgoing arcs.

d) An arc is said to be active when it contains an operand or token of information.

e) An operator node is ready for execution when all its input arcs are active and its output arc is empty.

These rules are enough to construct simple data-flow programs. In order to support more complex problems, the operator set has to be expanded with operators able to

perform decisions. They are the gate operators defined below [Myer81].

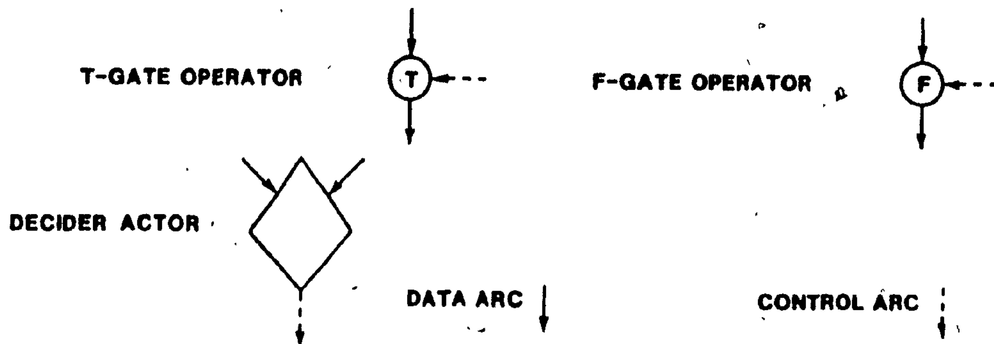

Fig. 3.2: Gate Operators

With these operators we introduce another type of token: the control token as opposed to data tokens for arithmetic operators. The new operators are enabled when their data arcs follow the condition e) above and there is a token on their control arc. Their operation is the following:

a) The decider actor provides a control token (a boolean value) depending on the result of the comparison performed on the two inputs. Examples are <, >, <=, >= or =.

b) The T-gate passes the input token to its output when there is a token on the control arc with the value true. Otherwise no output is produced and both input tokens are 'consumed'.

c) The F-gate acts like the previous actor but with a false value instead.

With this language or similar ones it is possible to represent a problem in a data-flow program [Denn79,Denn80, Myer81] and have a machine that is able to execute it. The construction of such machines is the subject of the next section subject. The problem of expressing data-flow programs in a more workable language will be discussed in chapter four.

3.3 Data-flow Computers

Having described the characteristics of data-flow architectures and how they execute programs, we will present some of the implementations. These are the ones which have resulted from the initial research on data-flow in the Universities that are involved in this field.

The description given here is just a brief overview of the organization of those computers, how they work and what their main features are [Trel82].

3.3.1 M.I.T. Data-flow Computer

The research in the data-flow field done at the M.I.T. has been significant and has formed the basis for other research projects. Their computer is therefore the most representative of the machines of its type [Denn72, Denn79a].

The program organisation follows the data-flow rules expressed above, which do not allow more than one token in each arc. The architecture thus provides for acknowledge signals from the destination operators to the originators. The computer has five modules:

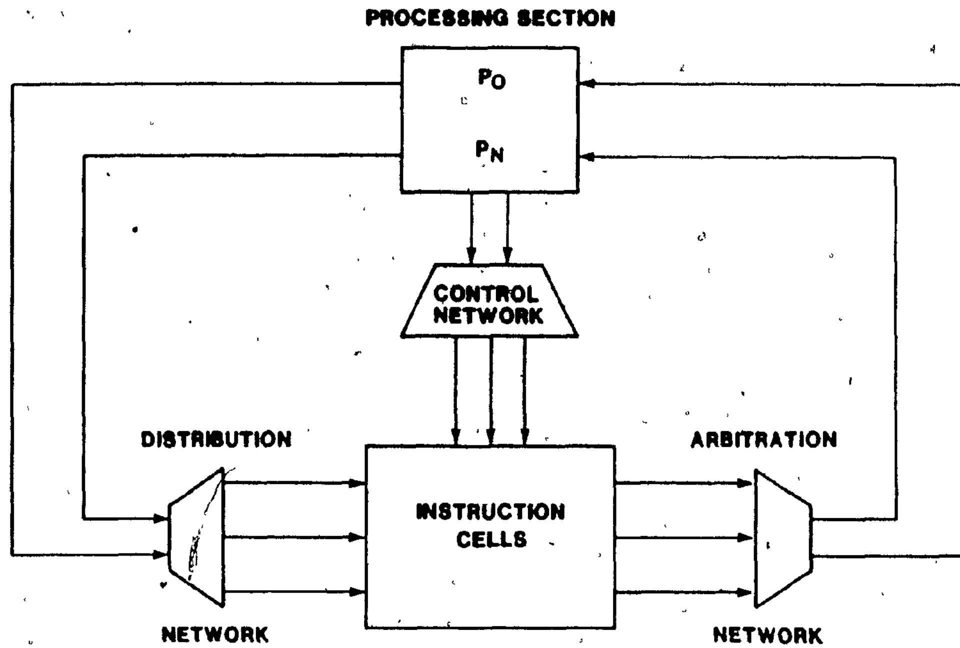

Fig. 3.3: M.I.T. Data-flow Computer

The processing section contains the operation units responsible for the execution of the instructions received from the arbitration network. This network passes information packets, containing instruction codes and operands, from the memory section to the processing section. Two types of results are available from the processing units: data tokens that are directed to the destination instruction cells via the distribution network and control tokens that are sent to the instruction cells through the control network. The first tokens contain input data for the destination cells and the second boolean

values. These only contribute to the triggering of destination instructions and are the result of boolean operators or signals used between instructions to inform of the availability of data arcs. Finally the memory section contains the instruction cells and the logic necessary to detect when an instruction is ready for execution.

3.3.2 DDM1 Utah Data-driven Machine

This computer differs markedly from the previous one with its recursive architecture. It is composed of computing elements (processor memory pairs) where each element is logically recursive and contains other inferior elements. Physically the elements are organized in a tree structure with each element connected to a superior element and up to eight inferior elements [Davi78].

Other characteristics of this computer are:there are no control tokens, as the data tokens provide all communication between instructions, the arcs are viewed as first-in/first-out queues directly supported by the hardware, and finally it is able to recognize locality of reference and therefore reduce the critical problem of system wide communication. This latter aspect is considered important for distributed systems exploiting VLSI.

3.3.3 Irvine Data-flow Machine (Id)

This machine was intended to exploit the potential of VLSI and provide a highly concurrent program organization. The first feature of this architecture is its sophisticated token identification [Arvi77,Arvi81a]. A token identifier consists of the following information: a code block name identifying a particular procedure or loop; a statement number within the code block; an initiation number for the loop; and a context name identifying the activity invoking this procedure or loop. The second feature is the support of data structures, such as arrays , by the inclusion of I structures [Arvi81b]. These structures are sets of components, with each component having an unique identifier and being either a value or an unknown if the component is not yet available.

The Id machine consists of N processing elements and a NxN communication network for routing the result of a processing element to the destination elements. In order to reduce the overhead in communication, the matching unit for tokens for a particular instruction is in the same processing element as is the storage of that instruction and there is a path from each processing element into itself

to route tokens that are destined to the same element that generated it [Arvi80].

3.3.4 Other Data-flow Computers

The previous computers are the most representative of the research being done in this field and all of them have running prototypes. There are, nevertheless, other data-flow computers such as the Manchester data-flow computer [Trel82], the Toulouse LAU system [Plas76], the Newcastle data-control flow computer [Trel78], the Texas Instrument distributed data processor [Corn79] and finally the uPD7281D (ImPP) from NEC Inc. [NECE84], the first commercially available VLSI chip supporting data-flow concepts. The architecture of this IC is a very simple data-flow ring with an addition of input and output controllers used to connect these devices in a chain for complex systems and also to interface with the other devices.

The components of this VLSI data-flow computer are shown in the following illustration [NECE84].

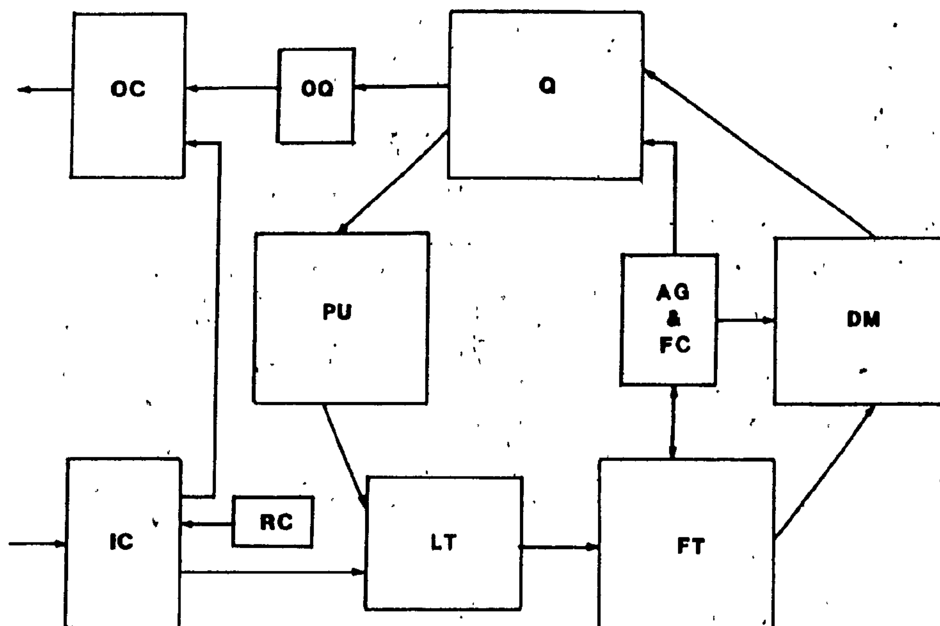

Fig. 3.4: uPD7281 (ImPP) Block Diagram

IC :Input controller. Controls input data tokens and determines whether or not an input data token should be sent to the data-flow ring.

OC :Output controller. Controls output data tokens.

LT :Link table (128 words x 16 bits). Stores instruction parameters.

FT :Function table (64 words x 40 bits). Stores instruction parameters.

DM :Data memory (512 words x 18 bits). Stores constants or temporary data.

Q :Queue (48 words x 60 bits). FIFO queue. Data

queue: 32 words x 60 bits; Generator queue: 16 words x 60 bits.

PU :Processing unit. Executes logical, arithmetic and bit operations.

OQ :Output queue (8 words x 32 bits). FIFO queue for the output tokens.

AG&FC:Address generator and flow controller. Generates addresses for DM and controls the flow of tokens.

RC :Refresh controller. Generates refresh tokens for internal DRAMs.

This VLSI uses a token-based data-flow architecture. a token entered through the Input Controller (IC) is passed on to LT to be processed around the ring as many times as needed. When the processing of a token is finished, it is queued into the Output Queue (OQ) and then outputted via the Output Controller (OC). Before any processing occurs, the host processor down loads the object code into LT and FT of the ImPP by using specially formatted input tokens. At this time, constants may also be sent to DM to be stored. The contents of LT and FT are closely related to a computational data-flow graph. The arcs represent the entries in LT and the nodes represent the entries in FT, where the operation code is logged along with the identification information about the outgoing arc [NECE84].

To finalize this chapter we will introduce an addition

to the M.I.T. data-flow architecture to support structures [Myer81].

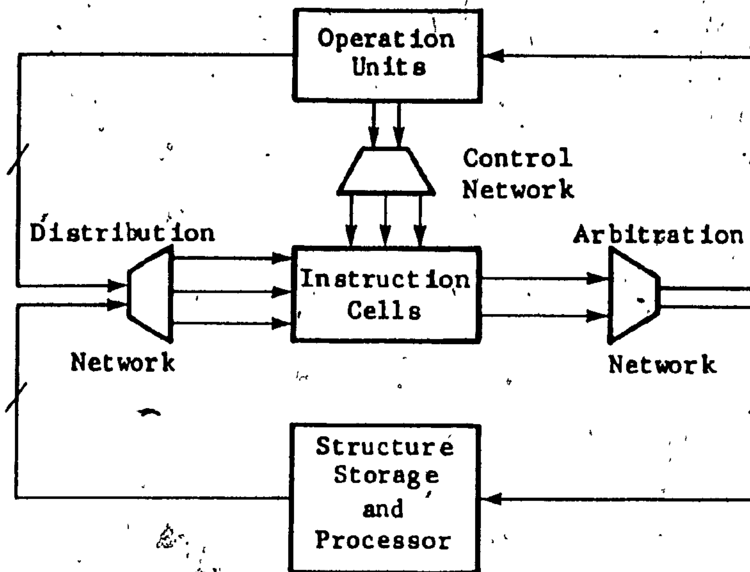

Fig. 3.5: M.I.T. Data-flow with Structure Support

In the architecture shown above the structure storage and processing can be considered in itself another data-flow ring with its own memory cells (holding the structures), processing units to perform operations in the structures such as create, add ..., and the distribution and arbitration networks that will route the tokens between these elements and also between this module and the distribution and arbitration networks of the computer. This

issue and the problems associated with it are analysed in chapter six that proposes a data-flow architecture for graphic systems. In the next chapter we will address the question of data-flow languages and their application to graphics.

4

Data-flow Programming Languages

It is a fact that the usefulness of computers is related to the availability of high-level programming languages, and the same principle applies to data-flow computers. Since the early stages of data-flow research, therefore, equal effort has been put into the development of high-level programming languages for data-flow architectures. The intention of this work was to provide a tool to program and test the new concepts under development and verify the application of data-flow to practical problems.

Two approaches are possible: use available high-level languages and adapt them to the new concepts, or design new languages based on these concepts. The first approach has the advantage of being relatively easy to implement and can profit from a large base of application programs. The second approach lacks this advantage, but on the other hand allows the implementation of the concepts more directly and eliminates most of the problems that are present in standard high-level languages. It is in this domain where research on the new languages has concentrated [Acke79,

Back78, MacL82].

Almost all of the research groups designed their own programming language. Of these many different endeavors the programming language VAL is presently at a more advanced stage, and is used by other research groups in the analysis of language related issues.

4.1 The VAL Programming Language

The design goals for VAL (value-oriented algorithmic language) were: to provide implicit concurrency which is to identify concurrency in algorithms and to map as much concurrency as possible into the data-flow graphs; and to provide the constructions that enable programmers to write correct programs.

The principal feature of VAL is its value orientation reflected in the inexistence of assignment constructions, which are replaced by the concept of expressions [Denn79b, McGr82]. A program in VAL is an expression in the mathematical meaning of the word. Parallelism is implicit and is imposed by the fact that if two operations do not depend on the outcome of each other, then they can execute simultaneously. Another result of the expression orientation of the language is noninterference: once the

values of all inputs are known, execution can not influence the results of any other operations ready to execute. Therefore the features of von Neumann languages, such as the concept of modifying variables and aliasing are eliminated. The principles used in this new language have also been applied in other languages such as LISP, and therefore are not unfamiliar to computer professionals.

In the following paragraphs we provide a brief description of the language in order to support the examples given in this work. More detailed information about the language constructions is available from [Acke79a,Denn79b,McGr82].

The data types of the language are integer, real, boolean,character-string, arrays and record structures. The usual operators can be applied to the types integer, real, boolean and character-string. For arrays the following function invocations are available:

array-adjust	shifts the origin
array-addh,array-addl	adds elements to either end
array-remh,array-reml	deletes elements at either end
array-join	merges two arrays
array-seth,array-setl	sets array bounds
array-limh,array-liml	tests array bounds

and the infix operator || to catenate two arrays. Arrays are created in VAL with expressions that assign an index to

each element. For example,

[1:elem1;2:elem2]

is an expression that creates an array with two elements. The values bound to the two identifiers become the values for the corresponding array locations. In this case there is no concurrency in the construction of the array because the elements are simple values, but in most of the applications they are expressions that can be calculated in parallel. With the functions given above is then possible to add or delete elements to the array.

Record constructions are similar to the arrays in the way they are created. For example,

```
record[xw-min:xlow;xw-max:xmax;
        yw-min:ylow;yw-max:ymax]
```

builds a record with four fields (xw-min, xw-max, yw-min, yw-max). Each field name is followed by an expression representing the value to be entered in the record. As with arrays the expressions on each field can be calculated simultaneously.

VAL is a value oriented language, therefore names used in the language do not refer to memory locations that can be modified as a result of assignments, but are simple binders to a value that are valid for a defined context of the program. This is commonly referred as single-assignment rule: once an identifier is bound to a value, that binding

remains in force for the entire scope of access to that identifier. This rule matches the data-flow concept represented by tokens in the arcs of a graph. Once a token is put on an arc, the same value must be transmitted to all receiving operators.

The language does not contain statements. It is completely based on expressions and functional operators. Besides the standard arithmetic expressions, and the IF-THEN-ELSE there is another basic expression particular to VAL, the LET-IN-ENDLET expression. The language through its compiler is able to recognise the implicit parallelism of such expressions. Two operations can be performed in parallel if none of them depends on the outcome of the other. The LET-IN-ENDLET expression is used to introduce names for expressions because they are common subexpressions of larger expressions. The syntax of this construction is the following:

```
LET
    { <type declaration> := <exp> ; }
IN <exp>
ENDLET
```

The names in type declarations of a LET block are local names meaningful only within the block; these names

must be distinct from each other and may appear free in the expressions.

In quite a number of programs it is possible to carry operations in parallel, for example in the construction of an array. To support this type of calculation with an explicit parallelism, VAL uses the FORALL block.

```
     FORALL <name> IN [ <exp> . <exp> ]
(1)        { <type declaration> := <exp> ; }
           <evaluation> | <construction>
     ENDALL
 <evaluation>::= EVAL  PLUS | TIMES | MAX | MIN | AND | OR
                                   <exp>
 <construction>::= CONSTRUCT <exp>
```

With this expression it is possible to calculate in parallel all the expressions in (1) for all values of <name> in the interval [<exp>.<exp>] and produce a final value or an array. The first is obtained with the EVAL that allows the calculation of a value from the results of all elements of the interval, and the second with the construct that creates a new array and assigns to each index the result of its expression (there is a one to one correspondence between the indexes of the final array and the elements of the interval).

When it is necessary to impose a sequence in the evaluation of expressions, VAL provides a loop construction that follows the value orientation of the language. In this expression values can be transmitted from one pass through the loop to the next. This transmission can only occur by defining loop parameters and then binding new values to them just prior to the beginning of the next pass.

```
FOR
     { <type declaration> := <exp> ; }
DO
     <exp>
ENDFOR
<iter-block>::= ITER <name> := <exp>
                         { ; <name> := <exp> }
```

The loop is initialised by binding names to the values of the expressions defined between FOR and DO. These values are used in the calculation of the expressions in the DO body. Inside this block a conditional clause will decide if the loop terminates or continues. If the result arm selected by the conditional contains an expression, then the loop terminates yielding that expression as its result. Otherwise, the result arm must be an iter-block that initiates another loop pass.

Val functions are similar to functions in other languages with some exceptions: It is possible to generate more than one result from a function and in the function definition the formal parameters name and type all values to be passed on each call. The body of the function is one or more expressions which produces the result or results. These expressions can only access the formal parameters and locally defined names. Because of the single-assignment rule, the body of the function can not rebind the values bound to the formal parameters, therefore there are no side effects. Concurrency is exploited,both in the evaluation of the expressions in the body of the function and in the calculation of the formal parameters values.

```
FUNCTION <name>
      ( <input list> RETURNS <output list> )
      { <type declaration> ; }
      { <function def> ; }
      <exp>
ENDFUN
<input list>::= <type declaration>
                     { , <type declaration> }
<output list>::= <type> { , <type> }
```

These are the major constructions of VAL. A subset of

the language syntax is provided in appendix I.

The following example is a small program that uses some of the VAL constructions. The program does not produce any meaningful result. It is just an example. It takes an array multiplies each element with a scale factor and sums them. The scale factors are different for positive and negative numbers.

```
PROGRAM test
FUNCTION calculate (input:ARRAY[REAL] RETURNS REAL )
    LET
        scale1: REAL :=XWMAX-XWMIN;
        scale2: REAL :=YWMAX-YWMIN
    IN
        FORALL i IN [1..3]
        EVAL PLUS
                IF input[i]<0
                    THEN scale1 * input[i]
                    ELSE scale2 * input[i]
        ENDALL
    ENDLET
ENDFUN
    calculate ([1:-1;2:3;3:-4])
END
```

In this program the final result is:
-scale1 + 3*scale2 + (-4)*scale1 where scale1=XWMAX-XWMIN and scale2=YWMAX-YWMIN (these names identify constants). The size of the array was fixed to three to reduce the complexity of the program representation in machine language, given in the next section. To make it realistic, the first statement of the FORALL expression should be:
FORALL i IN [ARRAY-LIML(input) . ARRAY-LIMH(input)]

To finalize this overview of a data-flow high-level language there are two issues that we want to refer to. The first is the handling of errors or exceptions. In VAL all the data types define error values and the corresponding operators recognize and are able to operate with such values (basically when an operator is not able to perform the task it was assigned it returns the appropriate error value. The second is the list of the language limitations. Bypassing the problems related to implementation the biggest limitations of the language are the lack of recursion, of functional operators (operations that produce functions as their result) and the omission of general input/output facilities [McGr82].

4.2 Data-flow Machine Language

The graph used to explain the operation of a data-flow computer is very close to what we call the machine language of a computer. Each instruction code includes a field where the destination cell addresses are stored (In conventional machines the destination field always points to a memory location where the result is stored). The other fields of the instruction are intimately related to the architecture they support. The ones given here are suitable for an

M.I.T. data-flow type computer. They are adapted from examples given in [Myer81].

4.2.1 The Instruction Cell

The contents of the instruction are grouped in two sets: the header and the operand ports.

		*	*	
HEADER	Operation code		SIG	Dest. addr.
OPERAND PORT a	Gating code	Gate flag	Value flag	Value
OPERAND PORT b	Gating code	Gate flag	Value flag	Value
. . .				
. . .				

* These fields are optional

HEADER.

Operation code-The operation code defines the type of operation to be performed with or upon the values in the value fields. Examples of operations are: ADD,MULTIPLY...

The field next to the operation code gives the number of signals that the cell has to receive before it can be

triggered. These signals are generated by other instruction cells and are used to avoid deadlocks as well as to provide synchronization of different paths in a calculation loop.

SIG-This field indicates that after the execution of the operation a signal is sent to the instruction cell specified by the address in the field.

Dest. addr.-This field gives the location of the instruction cells to where the result of the current instruction has to be sent. Besides the instruction cell address it is also given the location of the operand in the cell.

For special operation codes that can provide more than one result the instruction has more than one group of destination addresses. An example of such instructions are device dependent operations like input which can provide a value and a status.

OPERAND PORTS.

For each operand the following fields are necessary:

Gating code-This field provides a code that defines how and when a value received is going to be used to trigger the instruction cell. The four possible gating codes are:

N-no gating is performed. Any result directed here is placed in the value field (provided that the value

field is empty).

T-a value directed here will be accepted if a true control value (gate flag) has been previously received, or when a true gate flag arrives; otherwise the value is discarded if and when a false gate flag arrives.

F-a value directed here will be accepted if a false control gate has been previously received, or when a false flag arrives; otherwise the value is discarded when a true flag arrives.

C-the value is a constant and is not erased when the instruction is executed.

Gate flag-The gate flag arrives from the control network. There are three possible values

OFF no control packet (gate flag) has been received.

T a true control packet has been received.

F a false control packet has been received.

Gate flags are generated by boolean instructions such as EQUAL,LESS THAN..

Value flag-This field indicates whether a value currently exists in this operand port.

Value-The value field is received from the distribution network.

To complete the description of this machine language we present below the example of the previous section

expressed in operation codes that can be executed by the machine. A full set of these opcodes is given in chapter six. It should be noted that they are rather more 'high level' than the actual opcodes one could expect in a particular machine. They are, however, sufficient to illustrate the concepts analysed in this work.

The design process of any high-level language is not finished until a compiler is available to translate its constructions into a particular machine language. VAL has already compilers for the M.I.T. data-flow computer and other research machines, therefore it is reasonable to assume that the following machine program can be obtained from example 4.1 through a compilation process. The problems of compiler design and optimization are outside of the scope of this work.

To reduce the size of this example we will use a condensed version of the instruction cell:

```
#N   opcode| |SIG  |Destination addresses
     gate code    ****|gate code   ****|
```

The fields that are only used during execution are not shown (gate flag and value flag) and the operands are put together in the same line instead of using a line for each operand port. The *** field contains the value of the

operand if it is a constant or ^# which indicates the instruction cells that can provide a value for this operand. N refers to the instruction cell address.

```
#1    CREATE ARRAY | 3 |SIG   | 4a,8a,12a
      C 3 | C -1 | C 3 | C -4 |
#2    SUBTRACT | 3 |SIG   | 6a,10a,14a
      C XWMAX | C XWMIN |
#3    SUBTRACT | 3 |SIG   | 7a,11a,15a
      C YWMAX | C YWMIN |
#4    ARRAY[] | 3 |SIG 1 | 5a,6b,7b
      N ^1 | C 1 | C 1 |
#5    LESS THAN | 2 |SIG 4 | 6b,7b
      N ^4 | C 0 |
#6    MULTIPLY | 1 |SIG 2,4,5 | 16a
      N ^2 | T ^5,^4 |
#7    MULTIPLY | 1 |SIG 3,4,5 | 16a
      N ^3 | F ^5,^4 |
#8    ARRAY[] | 3 |SIG 1 | 9a,10b,11b
      N ^1 | C 2 | C 1 |
#9    LESS THAN | 2 |SIG 8 |10b,11b
      N ^8 | C 0 |
#10   MULTIPLY | 1 |SIG 2,8,9 | 16b
      N ^2 | T ^9,^8 |
```

```
#11  MULTIPLY | 1 |SIG 3,8,9 | 16b
     N ^3 | F ^9,^8 |
#12  ARRAY[] | 3 |SIG 1 | 13a,14b,15b
     N ^1 | C 3 | C 1 |
#13  LESS THAN | 2 |SIG 12 | 14b,15b
     N ^12 | C 0 |
#14  MULTIPLY | 1 |SIG 2,12,13 | 17b
     N ^2 | T ^13,^12 |
#15  MULTIPLY | 1 |SIG 3,12,13 | 17b
     N ^3 | F ^13,^12 |
#16  ADD | 1 |SIG 6,7,10,11 | 17a
     N ^6,^7 | N ^10,^11 |
#17  ADD |   |SIG 16 |
     N ^16 | N ^14,^15 |
```

Example 4.2

Explanatory notes for this example:

The instruction CREATE ARRAY creates an array in the structure storage unit. The operands for this instruction are: the first one gives the number of elements in the array, and the remaining the elements of the array in increasing order of the index.

The instruction ARRAY[] retrieves from the structure storage the array element that corresponds to the index

value available from the second operand port. The first operand specifies the array and the third specifies, in this case, that only one element is to be retrieved.

The instruction LESS THAN performs the following comparison Operand1 < Operand2. The result is sent to the gate flag field of the destination instructions.

Taking instruction #6 as an example we can say that it performs a multiplication, has to receive one signal (from instruction cell 16 informing that operand port a is ready to receive a value), generates three signals one for each of the instructions that provide an operand (they are 2,4 and 5), the result is sent to instruction 16 operand port a, the first operand is not gated and it is received from instruction 2 and finally the second operand is gated (has to receive a true flag from instruction 5) and the value is received from instruction 4. When in a value field there are more than one originator it can be one of the two cases:

a) The operand is gated and the first originator refers to the instruction originating the gate flag.

b) The originators come from two mutually exclusive branchs of a conditional instruction, which is the case of instructions 16 and 17.

5

Graphic Systems and Data-flow Computers

In the first chapter we described the evolution of graphic systems and the general model for such machines. In the second and third we presented data-flow architectures and languages. In this chapter we will show how graphic systems and data-flow architectures can be matched: First we describe the different methods of picture representation in a graphic system; A graph approach for such representations is then proposed and analysed; Finally, based on the graphic system model and the conclusions about picture representation, we propose a data-flow architecture for graphic systems.

5.1 Picture Representation

Since graphic devices were first integrated into computers it has been necessary to address the problem of picture generation [Berg78,Free79]. One technique was to simply integrate in a program the instructions to the graphic device to draw the picture the programmer wants.

Let's assume, for example, that we have a system consisting of a general purpose CPU and a graphics device only able to write and read a pixel (these actions are given to the graphics hardware via command registers located in the I/O space of the CPU) and we want a routine to draw a line given the start and end points. The program presented in appendix II achieves this using the Bresenham algorithm. With more powerful graphic systems, programs like these are no longer necessary because they are directly performed in the graphic system. Nevertheless, the programs designed to use those systems are quite similar in nature: routines that execute in the main CPU (the host CPU) and utilize the graphic commands. These routines are themselves incorporated in application programs. In summary, a picture is generated via a program that runs on the host CPU and sends commands to the graphic system.

This approach is suitable for systems that do not support all or almost all of the graphic operations (projections, transformation, clipping...) because to generate the desired final picture the host CPU has to be used intensively. In graphic systems that support the above operations it is possible to make a deeper separation between the application and the graphical representation of the model, as is shown in figure 2.2. In order to obtain a picture it is not necessary to embed graphic commands in a

host program but simply provide a list of commands (display list) to the graphic system. The application program in the host builds display lists (normally in main memory) and instructs the graphic system which lists to use and what kind of transformations to apply [Matr84].

The advantage of display lists reside in the reduction of interaction between the host CPU and the graphic system (allowing for more parallelism) and the possibility of having the display lists organized in a data base and easily available to be incorporated in more complex pictures (these display lists are normally referred as segments). It is easier to work with these lists incorporated in the application program than to use a set of routines that not only have to be retrieved from storage but executed by the host CPU (this is in some way equivalent to overlay techniques).

Another advantage of display lists is that they can be considered as functions or routines to be executed by the graphic system. With this extension in the use of the display list some graphic systems are able to support 'instructions' very similar to instructions of general purpose CPUs. For example, 'GOTO' a particular location in the display list and continue interpretation from there, 'CALL' a display list or perform conditional jumps. These instructions give more flexibility to the process of

creating a display list (a picture that contains repetitive elements can include CALLs to a display list that represents that element, instead of repeating the same list for every occurrence).

In both ways of passing information to a graphic system there is a need for standardization. The designers of application packages want their programs to be independent of the particularities of the hardware. There are presently several graphics standard available: the CORE system, GKS, GSX and others. The first two are the most popular. CORE is a set of subroutine calls to be incorporated in FORTRAN programs. An application program only has to link these routines without concern of most of the characteristics of the graphic system. This standard was the first one to be widely used. It was design for vector displays but has since been expanded for raster scan systems [Fole79]. CORE supports three dimensional graphics but does not support segments. GKS is a more recent standard and therefore incorporates new concepts such as workstations, segments and a multitude of input devices and input modes [GKS84,Hopg83,Inter81]. Unfortunately it does not support 3D, however the expansion to include three dimensional coordinates is not difficult and a proposal for such extension already exists. GKS is also a set of routines that can be bound to any programming language. The

support of display lists that are called metafiles in GKS are still under discussion. In this work we will use GKS nomenclature for our examples, but the use of this standard does not preclude the use of the ideas presented here with other standards. A list of GKS 'commands' with a short description is given in appendix III.

5.2 Parallelism in Graphics

Most applications that use a graphic interface for both output and input use a structured approach to 'build' the pictures that support the application program. This approach is used in order to improve the system capabilities and throughput.

For example, a package to automate PCB design will have a data-base that contains all the electronic components supported [Inter84]. For each component, the data-base must have logical information (logic operation for simulation, type of input and output...), electrical information (propagation delays, voltage and current levels for each input/output...), mechanical information (body dimensions, clearance area...), and graphic information (schematic symbol, pad layout...). A final PCB design will include a components list, a connections list, a schematic

diagram, components placement, connections routing, simulation vectors and others. Let's consider the schematic diagram which is basically graphic information.

In large circuits, the design is partitioned in logical blocks to improve the design phase (each block can be done by different people at the same time), the verification phase (the compliance with the product specification is easier to verify), and the simulation phase. The layout and artwork design is facilitated and blocks of the design can be used in other circuits. A design structure like the following is therefore quite common.

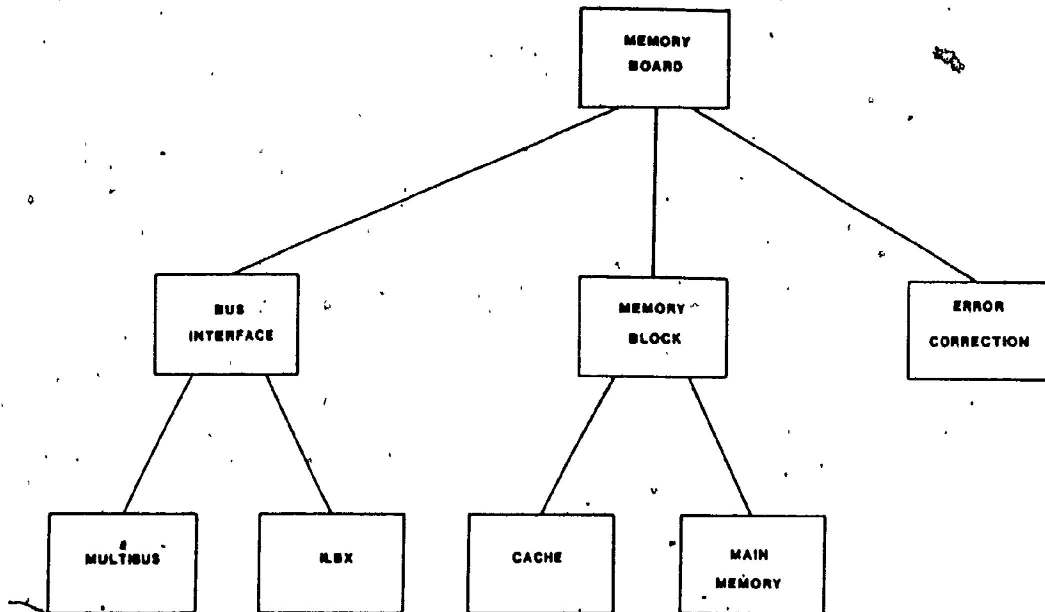

Fig. 5.1: Hierarchical Circuit Design

The schematic of this circuit has the same structure. Let's consider now how the graphic information for the schematic is organized. It is assumed that all logic symbols are already available, that they can be referenced by the device number, and that they have their coordinates defined relative to the origin (for completeness, the position of the origin relative to the symbol is the position of the bottom left corner of a hypothetical rectangle that encircles the symbol. The same principle applies for block references). The graphic information for each block of the schematic is: a transformation matrix

that gives the position in world coordinates, the scaling factor and angle of rotation (very unlikely); a viewport array that defines the portion of the world space that should be displayed; the viewport to window transformation; the viewport priority level (used during a pick operation when the pick device points to an area of the screen with overlapped viewports); the blocks or devices that belong to it; a list of vectors that interconnect the different components and a list of strings that represent the labels in the schematic.

For illustration purposes let the following circuit be a block of a more complex circuit:

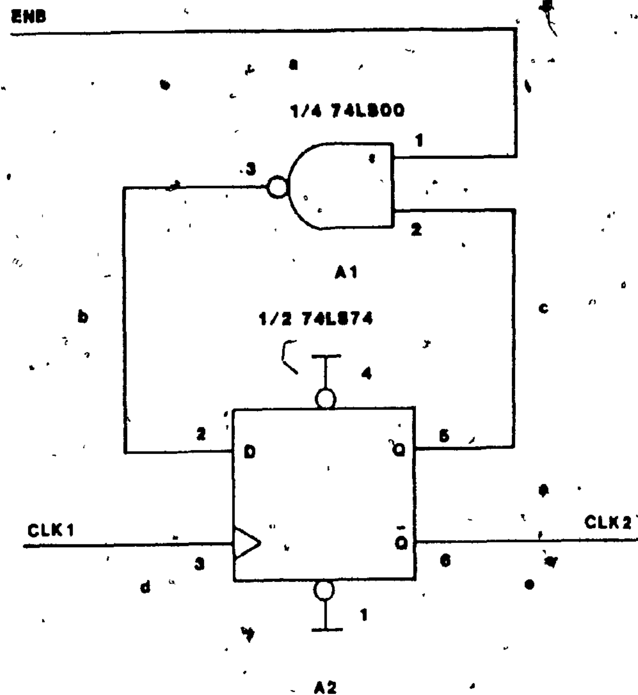

Fig. 5.2: Schematic Block

The lower case letters besides the connection are there just for reference.

The graphic information of this block will contain:

a) A transformation matrix to be applied to all components of the block and the priority of the segment.

b) A window and viewport coordinates.

c) Five polylines (see appendix III), one for each of the connections (a to e).

b) Seven strings, one for each of the labels.

c) Two symbol references. Each one with its own ('local') transformation matrix to position it in the block

space.

Each of the symbols in this picture are represented respectively by:

The NAND gate.

a) One polyline for the symbol body.

b) One circle for the output pin (use the GKS generalized output primitive).

c) Three strings, each of them a single number for the pin numbers.

The D FLIP-FLOP.

a) Two polylines. One for the square body and the other for the small clock input triangle.

b) Two other polylines to draw the +5V connections to the set and reset pins.

c) Two circles, for the set and reset inputs.

d) Nine strings, for the three letters and six numbers of the symbol.

This description can be put more formally in the following 'pseudo' FORTRAN program (no restrictions are followed for variable names, variable declarations are not given, and instead of CALL POLYLINE we just write POLYLINE)

OPEN SEGMENT (clock block)

defines the subsequent calls as part of a block.

SET SEGMENT TRANS (clock block,block matrix)

sets the transformation matrix to be used on all outputs included in this block.

SET ...

SET ... calls to set all the parameters window, viewport and text attributes.

EVALUATE (...,nand) evaluates the transformation matrix for the NAND gate.

EVALUATE (...,dff) evaluates the transformation matrix for the D FLIP-FLOP.

SET SEGMENT TRANS (74LS00,nand)

draws the NAND gate after transformation. The segment for the NANd gate is already defined.

SET SEGMENT TRANS (74LS74,dff)

same as above for the D flip-flop.

POLYLINE (4,XDa,YDa) draws line a. XDa and YDa are arrays with respectively the X and Y coordinates for line a.

POLYLINE (4,XDb,YDb) line b

POLYLINE (4,XDc,YDc) line c

POLYLINE (2,XDd,YDd) line d

```
POLYLINE (2,XDe,YDe)      line e
TEXT (X1,Y1,'ENB')        writes string 'ENB' starting at
                          the X1,Y1 position.
TEXT (X2,Y2,'1/4 74LS00')
TEXT (X3,Y3,'A1')
TEXT (X4,Y4,'1/2 74LS74')
TEXT (X5,Y5,'CLK1')
TEXT (X6,Y6,'CLK2')
TEXT (X7,Y7,'A2')
```

When executing this program the host CPU will invoke the GKS subroutines which will pass the appropriate commands to the graphic system. The order in which the picture will appear in the screen is the order given here for GKS calls. First the NAND symbol is drawn followed by the D FLIP-FLOP, the connection lines, and finally the schematic labels. It is not difficult to realise that there is no reason for this particular order (all the output calls in the previous program can be interchanged in their position). This is merely the result of the inherent parallelism of those calls. All the graphic outputs or all the components of the picture could be drawn simultaneously without affecting the final picture. The only restriction to delay their execution is the availability of the transformation matrices and other attributes. In other words, these values

are the inputs for the graphic functions that can be executed as soon as their inputs are available. If we refer back to previous chapters this is no more than the triggering condition for instruction cells in data-flow computers. Therefore, a data-flow computer can be used in the display processing unit of a graphic system. Before describing such an architecture, we shall show how a high-level data-flow language can be used to describe a picture and utilise at the same time the implicit parallelism in the picture generation process.

5.3 Picture Description with VAL

We have just shown that the data-flow concept can be applied to graphic routines or primitives. If so, they can be represented with a graph, as we can see in the following figure.

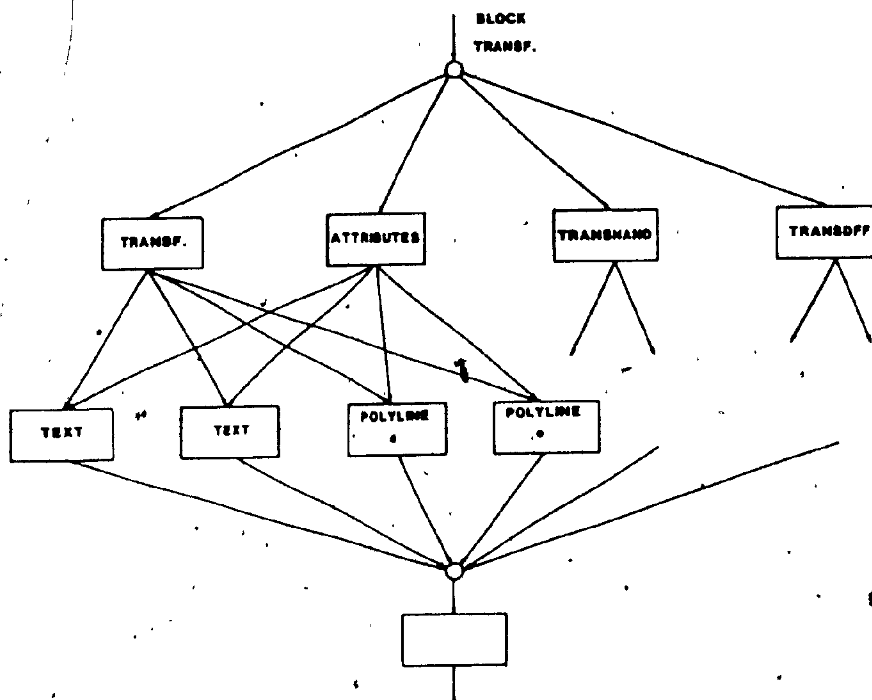

Fig. 5.3: Graph of Circuit Block

The next step is to demonstrate that the program of a graphic system can be done with a high level data-flow language (VAL).

The first issue to be resolved is the support of input/output. One of the reasons why VAL does not support such constructions resides in their lack of conformity with the mathematical definition of a function (the basis for an applicative language). An output does not produce a result, it modifies some global environment, which can be considered a side effect. Both input and output can be invoked simultaneously by independent branches of a graph,

making it impossible to ensure the proper sequencing of output data as well as a coherent sequence of input data.

These considerations are not, however, valid in the case of a graphic display program. First, the proper sequencing of output is irrelevant because of the parallelism in the operations of building a picture. The elements of a picture can be drawn independently as well as the more primitive components of each element. There is, however, an exception to this rule. It is possible to have two pictures that are considered independent but whose bit map images overlap and therefore it is necessary to ensure a deterministic result. This problem can be solved in two ways: a) if the hardware supports segment priority, it is only necessary to have contingent pictures with different priorities, a necessity if picking is supported. The hardware will take care of using the priorities to decide which of the pictures overlaps the other; b) with the graphic functions defined in VAL for graphic representation it is possible to impose a sequence in the graphic outputs. This is discussed in the next paragraphs. Second, the inputs in graphic systems are used to interact with the operator. The characteristics of this type of input are different from input to general programs. It is serial and very slow compared with CPU speeds. It is therefore not necessary to support it with more than one

input function in the graphics program. The problem of inconsequent input data is therefore not a problem in this particular application. Finally, although output functions produce side effects, in our particular case these are outside the program context and can not influence the 'behavior' of any of the programs in the graphic system. The only 'global environment' affected is the operator but this is the purpose of a graphic system.

In order to conform to an existing standard we shall use GKS primitives, but because of VAL's nature certain modifications are necessary.

The SET commands of GKS can be eliminated for the simple reason that they are not functions. They were designed to set a value in a specific memory location or register and all the graphic commands following them will use that value for their operation. For example, SET POLYLINE INDEX (N) sets the polyline color and texture to a certain value and all POLYLINE commands executed after this statement will use that color and texture, until it is modified by another SET POLYLINE INDEX command. Instead of using the SET commands to define values that are then implicitly used by graphic functions we will pass those values directly to the function. This follows the principles used in VAL design, and at the same time makes it easier to write, verify and understand a graphics

program. We will know what the attributes of a graphic function are because they are explicit in the function parameters, instead of being defined somewhere else in the program and being implicitly defined when the function is called.

In terms of the VAL language, a function has to produce an output in order to be usable in the program expressions. If we restrict the graphic functions only to produce graphic outputs, we could not use them in a VAL program without major changes in the language principles. To eliminate this problem, every graphic function will return a boolean value to the program environment. With this value graphic functions can be included in boolean expressions and conditional expressions. Still to be defined is the meaning of the two possible values of the output. Simply expressed, a TRUE value implies that the function was executed successfully, otherwise it was not executed or an error occurred. In summary a picture description is an expression that produces a boolean result with the implications given above.

As an example of a GKS function in VAL, let's consider the polyline function.

```
POLYLINE (window,view,transf,polyindex,
          coordinates,trigger)
```

where

window: is an array defining the window rectangle (two X and Y world coordinates).

view: is another array specifying the normalised coordinates of the viewing rectangle.

transf: is a matrix that defines the scaling, translation and rotation to be applied to the line.

polyindex: defines the color and texture of the line.

coordinates: is a structure with n elements, each one of them specifying the coordinates of a vertex. The order in which they are in the structure defines the polyline to be drawn (V1,V2,V3... implies a line from V1 to V2, a line from V2 to V3 and so on until Vn).

trigger: this is a boolean value. If TRUE the function is executed when all the other inputs are available and provides after execution a boolean value as specified above. If FALSE the function does not provide any graphic output and provides a FALSE value to the other VAL expressions. As with other inputs the function is not executed until this input is received. The main reason for this input is to facilitate the sequencing of graphic functions when this is necessary.

The definition of this and other GKS functions in VAL syntax can be found in appendix IV.

Using the previous constructions we can express the

small schematic of figure 5.2 in a VAL program.

```
FUNCTION clock block (window,view,transf:ARRAY [REAL];
                         trigger:BOOLEAN RETURNS BOOLEAN)
        {The OPEN and CLOSE statements are replaced by a
        function definition. The parameters define what
        are the window and viewport for clipping and
        normalization plus the transformation to apply to
        the all block. The input trigger has the same
        function of the input of the same name in
        POLYLINE given above.}
    LET
        polyindex:INTEGER:=xxx  ;
            {this name identifies the color }
        transfnand:ARRAY [REAL]:= ACCUMULATE
                             (transf,[1:xx;2:xx....]);
            {transfnand identifies the transformation to
            apply to the NAND symbol. It is the result
            of the multiplication of two matrices, the
            transf matrix of the clock block and the
            matrix defining the position of the symbol
            in the block.}
        transfdff:ARRAY [REAL]:= ACCUMULATE
                             (transf,[1:xx;2:xx....]);
            {the same for the D FLIP-FLOP}
```

```
connecta:2DLINE:=[4,1,x1,y1;x2,y2;x3,y3;x4,y4];
     {defines the vertices for connection a.  The
     structure 2DLINE is explained at the end of
     the example.}
...  {repeat the same for the other connections}
textpath:INTEGER:=xxx;
     {defines text path for strings}
...  {provide all the other attributes for text}
labels:BOOLEAN:=TEXT(textpath,...,xt1,yt1,
                           'ENB',TRUE)
     AND TEXT(....,xt2,yt2,'1/4 74LS00',TRUE)
     ...
     AND TEXT(....,xt7,yt7,'A2',TRUE);
     {labels are output to the screen and when
     completed the identifier label will have a
     TRUE value bound to it. The TEXT function
     draws the string specified at the given
     positions. The initial values passed are
     parameters to specify character features. A
     TRUE value is given to the trigger input
     because all these functions are not
     dependent on other graphic functions.}
connections:BOOLEAN:=POLYLINE(window,view,transf,
                      polyindex,connecta,TRUE)
     AND POLYLINE (....,connectb,TRUE)
```

```
...
        AND POLYLINE (....,connecte,TRUE);
        {each of the polyline functions draws a
        connection of the schematic}
    symbols:BOOLEAN:= 74LS00(window,view,
                              transfnand,TRUE)
        AND 74LS74(window,view,transdff,TRUE);
        {the functions 74LS00 and 74LS74 use the
        inputs received here to draw the two
        symbols.}
  IN
    labels AND connections AND symbols
        {this expression simply gives a structure to
        the drawing process. The parallelism
        inherent to these expressions is determined
        by the data dependencies, a normal function
        for a VAL compiler. These data dependencies
        will impose a graph similar to the graph of
        figure 5.3.}
  ENDLET
ENDFUN
```

The following considerations are necessary to complete this chapter. The structure 2DLINE was introduced to reduce the number of references when passing the coordinates of a line. In its place we had to use two arrays, one for the X

coordinates and another for the Y coordinates has specified in GKS. The first element of the structure specifies the number of vertices, the second the priority of the line and the remaining are pairs of X and Y coordinates.

Adapting these concepts for a three dimensional graphic system is not difficult. The coordinates will have three values, the windows will be parallelepipeds and the transformations will include projection. Some of these operations are discussed in the next chapter.

Improved error processing could be added by modification of the trigger type. Boolean types only have two values. In the graphic functions, we use values of this type as outputs (see above). It is advantageous if instead of this type we define a type with one data value and one or more error values. These could be used to trigger certain actions that will correct or overcome the error. In this case the AND operator used in the graphic expressions will be substituted by a new operator capable of performing such actions. The application of this new type has to be studied to verify its real usefulness.

None of the graphic functions used in the example have any interdependence. If in a graphic program it is necessary to ensure that one function is executed only after another one, it can be easily expressed with the following constructions:

```
POLYLINE (.......,POLYLINE (.....,TRUE)
or
    LET
    ....
    seq:BOOLEAN:=POLYLINE (...,TRUE) AND TEXT (...,TRUE);
    .....
    seq1:BOOLEAN:=POLYLINE (...,seq) AND POLYLINE(...,seq)
```

both expressions use standard VAL data dependency to impose sequencing. Therefore the sequence of graphic expressions can be ensured through a variety of formats.

Of all the graphic functions, the functions related to graphic input provide more than one result. For example a REQUEST LOCATOR provides status, a normalization array and a X,Y pair of coordinates. These multiple results are easily accommodated by the language.

```
    status:INTEGER,
    norm:ARRAY [REAL],
    xpos,ypos:REAL
            := REQUEST LOCATOR (,....);
```

which is a standard approach for VAL to handle multiple results from functions. Each of the results of the function

is bound to the respective name of the type declaration. In the body of the function, the expressions that give the four results must be in the same order that the names given above.

5.4 Data-flow Architecture for Graphics

In the previous sections we have shown how a graphics program can be written in VAL. It remains to be shown how the graphic functions are expressed. Their implementation is dependent on the characteristics of the graphic system. If this has very limited graphic hardware primitives, these functions are quite complex programs because most of the operations have to be executed in a general purpose operation unit. If, however, the graphic system is able to support most of the graphic operations, these functions are merely a translation of parameters. We have demonstrated that the parallelism of graphics can be expressed in a data-flow language using a graphics standard to maintain hardware independence. It follows that this parallelism can be exploited through the use of data-flow architecture. It is our intention, in the following, to propose an architecture based on a data-flow organization. We have chosen, in our design, to propose implementation of most of

the graphic operations in hardware, as this approach is the most direct way to achieve a powerful, special purpose architecture. Some of these units, in fact, are already implemented in recent commercial VLSI devices [BLTC84, Clar80,NECE84,VLSI84], and others are suitable candidates.

Because VAL was designed and used on a M.I.T. data-flow computer we will base our architecture in this computer. Starting from a computer that supports structures as shown in figure 3.5, we shall assume that the operation units are not only standard ALU's but also specialized devices able to execute graphic functions such as transformation, projection, clipping and operate on the bit map such as vector generator, raster operator and so on. A general block diagram including the specialized operation units is shown in the following figure.

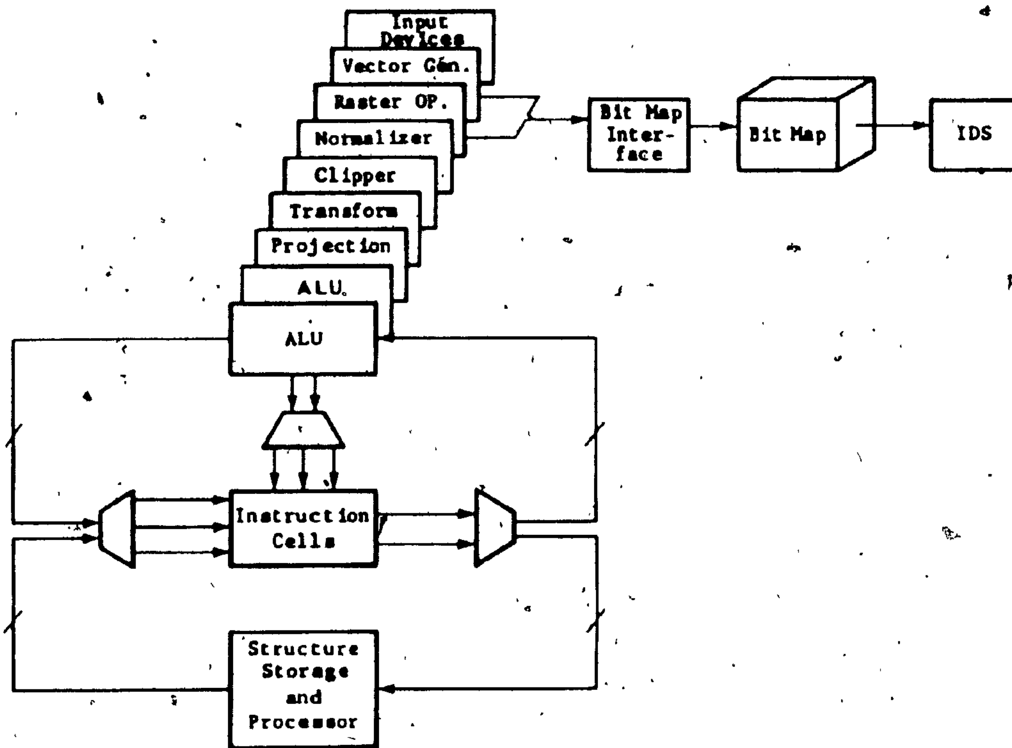

Fig. 5.4: Data-flow Graphics Computer

How these units can be integrated in the standard data-flow architecture and the modifications to be introduced are the subject of the next chapter.

6

Data-flow Graphics Computer

If we analyse the block diagram of the architecture given in figure 5.4 it is possible to recognize:

a) A data-flow ring that includes the instruction cells, the distribution networks for both data and signal, the arbitration network and the operation units.

b) A ring where the structures are stored and processed.

c) A set of operation units, each one with its specific functionality.

d) The diagram of a bit map and image display system.

If we refer to figure 2.4 the items described in a,b and c are the constituents of the DPU (display processing unit).

To describe in more detail this architecture we will start by the operation units, followed by the bit map interface and finally the data-flow ring, structure storage and processing. Because this architecture is intended for graphic applications we will concentrate on the particularities of the machine that are directly related to graphics and give only a general description of the possible implementations of the data-flow ring and

structure handling.

6.1 The Operation Units.

The operation units given in the diagram of figure 5.4 are very diversified in their functionality and in their implementation. The following paragraph will provide for each unit a description of its functionality, its set of inputs and outputs, a subset of the instructions it can execute with a suggested format and syntax, a diagram of an implementation, and a description of its operation (the interfaces to the bit map and to the data-flow ring for all operation units where applicable are described at paragraph 6.2 and 6.3 respectively).

6.1.1-Vector Generator

The units of this type are responsible for scan converting a vector into a set of pixels in the bit map. This conversion is normally based on the Bresenham's algorithm [Fole82,Suen79].

The input provided to these units is the instruction described below and the output is a signal (implying

completion) sent to the destination instruction cells. The command accepted by the vector generator is:

MULINE |SIG x,x...|PAR1 |PAR2 |

Where PAR1 is a digit specifying color and texture and PAR2 a pointer to a 2DLINE structure in the structure storage (if the data-flow implementation chooses to pass directly a structure to the operation units, the PAR2 should be substitued by the values of the 2DLINE structure). The 2DLINE structure starts with a number of vertices (n) followed by the segment priority, and by n pairs of X and Y coordinates, one for each vertex of the line.

With this command, the vector generator converts each segment into a set of pixels in the bit map that starts and ends at the specified end points.

The following implementation of such units is one of many possibilities. In this one the main objective besides the implementation of the previous command is to be able to integrate the hardware into a gate array or custom VLSI.

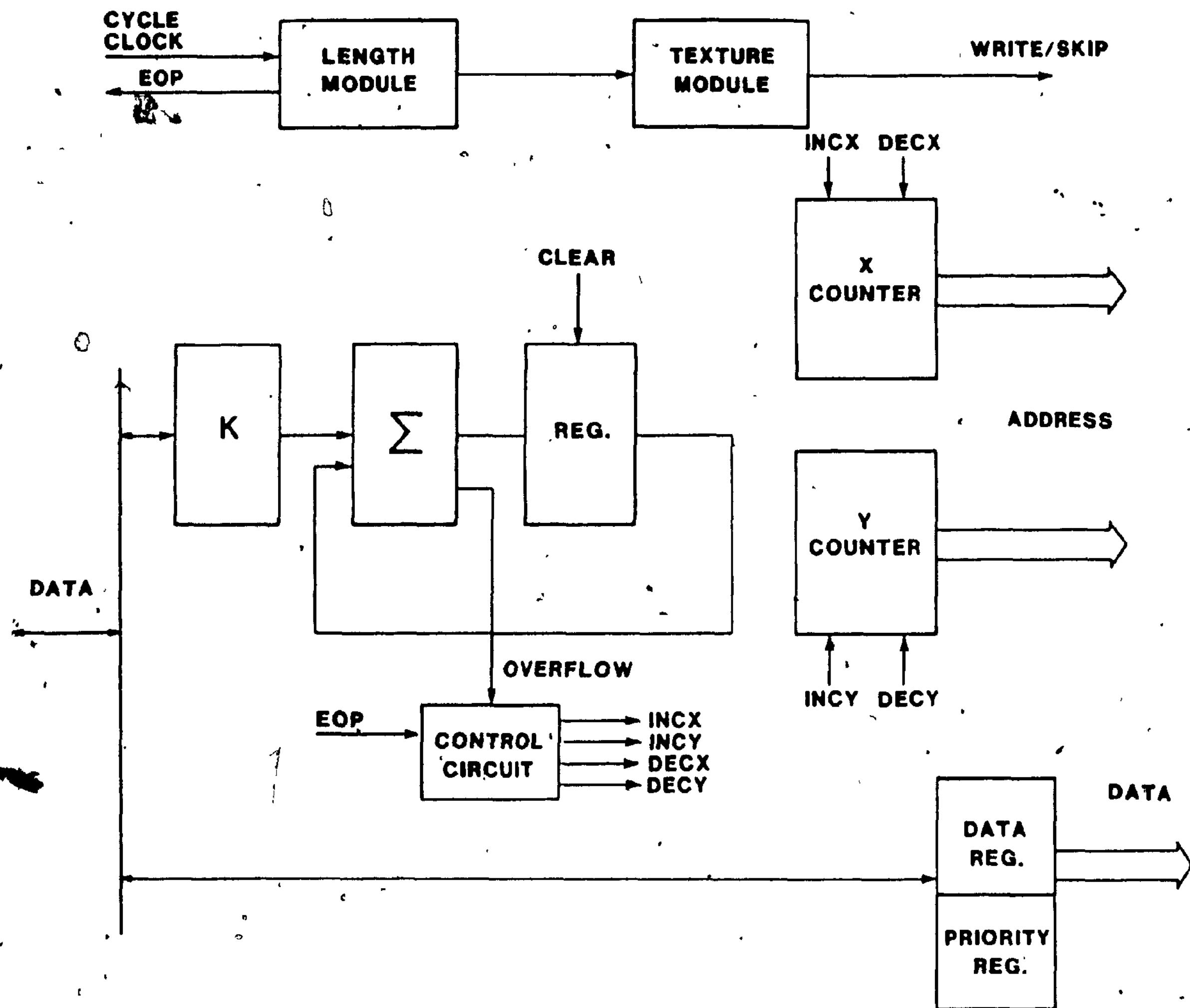

Fig. 6.1: Vector Generator Block Diagram

The operation of the circuit represented by the previous block diagram is the following: The length module controls the length of the operation that is equivalent to the length of the line and the texture module specifies when a pixel should be written or when a skip should be performed. This allows the support of dotted or dashed lines. The main blocks of the circuit are the X and Y

counters, the K and data registers and finally the adder, multiplexer and control circuit. Before a line is drawn the X and Y registers are loaded with the coordinates of the first dot, the data register with the color of the pixel and its segment priority, the length register with the number of pixels on the line, the texture register with a sequence of ones and zeroes that represent the desired texture, and the K register with the slope of the line. At every cycle of the line generation process, the length register is decremented, the K value added to the previous results via the feedback loop on the adder, and the X and Y registers incremented according to the following rule: if there is no overflow in the addition only X is incremented otherwise both registers are incremented. This rule is adapted to a K greater than one by interchanging X with Y. By selecting increment or decrement operations, the circuit can write lines in any direction. The texture register is circulated with every cycle and the bit presented at the output specifies that a write is to be performed if equal to one or no write operation if it is a zero. When the control circuit detects that the length register is zero the process is terminated.

6.1.2-Raster Operators

These units manipulate raster display data in the bit map (or display memory) by means of user selected primitive operations. The raster operators can modify or combine rectangular areas to provide the complete generality needed to scroll screens, manipulate windows or 'paint' characters. The output data sent to the bit map is generated according to the function specified by the instruction and of data present in the source and destination pixels and a pattern register. Because these are three boolean operands (source,destination and pattern) there are 256 possible functions. Of these we will provide the most common subset.

As for the vector generators, the input of these units are the instructions and the output a completion signal.

RCOPY |SIG x,x... |X0 |Y0 |X1 |Y1 |X2 |Y2 |

This command asks the operation unit to simply copy the contents of the bit map in the rectangle

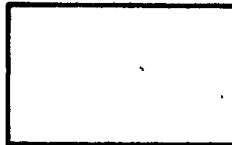

Fig. 6.2: Working Areas

to a similar area of the bit map that has the top left corner located at (X2,Y2).

RCOMPL |SIG x,x... |X0 |Y0 |X1 |Y1 |

This command complements the contents of the bit map rectangle defined by the two vertices (X0,Y0) and (X1,Y1) as shown in the previous figure.

The next three commands are equivalent with the exception of the boolean operation applied to each pixel of the source and destination rectangles. The result is written back to the destination rectangle.

ROR |SIG x,x... |X0 |Y0 |X1 |Y1 |X2 |Y2 |

RAND

RXOR

The following diagram describes a possible implementation of a raster operator unit. It is based on a Rasterop chip designed by Silicon Compilers [VLSI84].

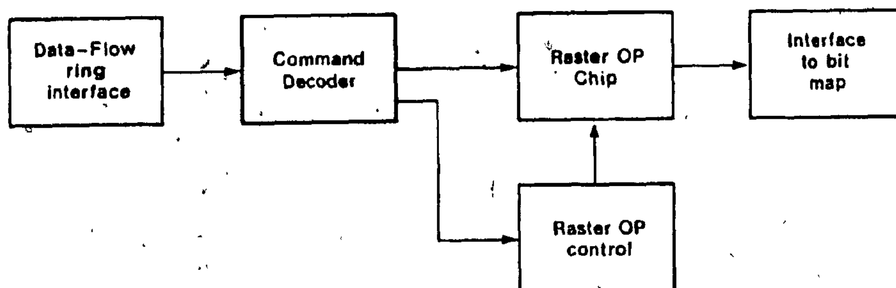

Fig. 6.3: Raster Operator Module

The data-flow ring interface passes to the command decoder the instruction to be executed by the unit. Furthermore it sets up the Rasterop chip by loading the internal registers of the Rasterop chip by strobing the chip select, write and four address lines for each register to be loaded. These registers remain fixed for the duration of the Rasterop operation.

The command decoder loads the registers of the Rasterop control with the vertices available from the instruction fields. With this information the Rasterop control reads 16 pixels of display memory to the Rasterop chip source registers. It also reads 16 pixels of the area to be updated to the destination register. Finally the

output of the Rasterop chip is send back to the display memory. The Rasterop control is also responsible for repeating this procedure for each 16 pixel cell of the display area defined in the instruction fields.

A very brief description of the Rasterop chip is given here. More information is available from [VLSI84].

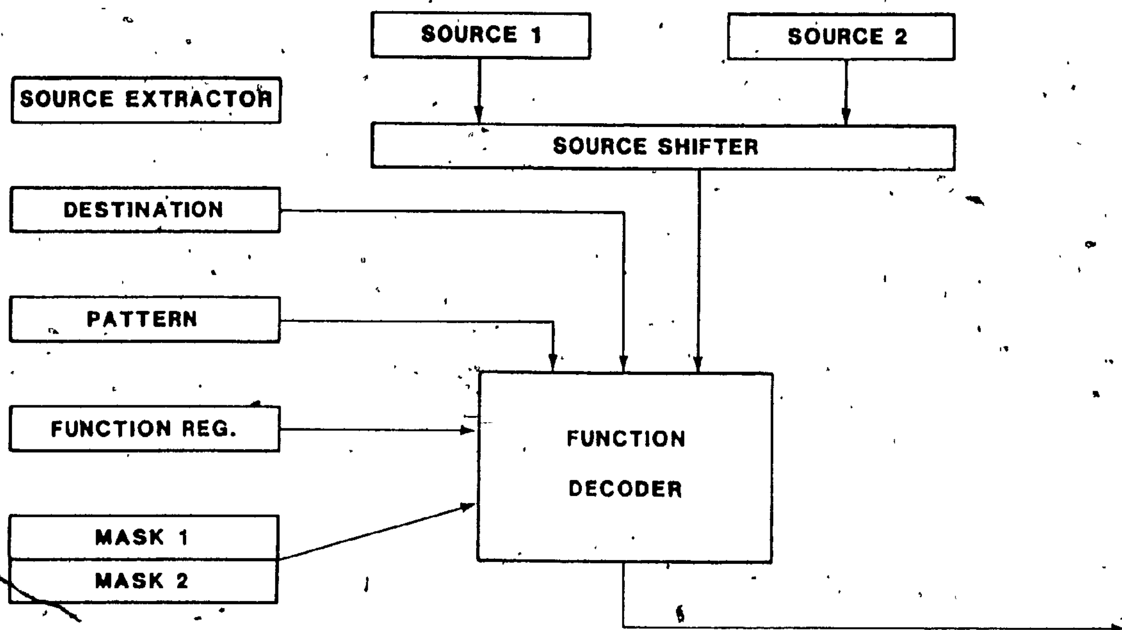

Fig. 6.4: Rasterop Block Diagram

The two source registers receive the data from two consecutive 16 pixel words. The source extractor receives information from the shift register and decoder to decide which pixel to provide to the function decoder (this allows the alignment at the pixel level of the source field to the destination field). The pattern and function registers provide the function decoder with the necessary information that together with the data from the source and destination registers will give the final output. Mask1 and Mask2 registers allow certain pixels of the destination register to be written back without change.

We have been referring to pixels instead of bits but the Rasterop chip works only with a one bit plane. In order to work with all bit planes of each pixel, this operation unit needs an appropriate number of Rasterop chips.

6.1.3-Normalizer

This unit transforms world coordinates into device coordinates. This transformation is done through a linear conversion given below.

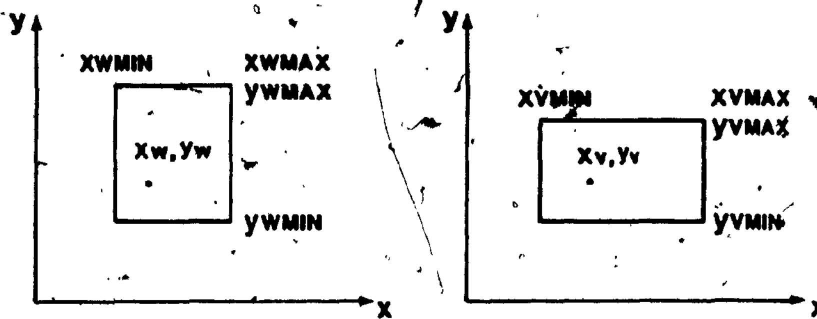

Fig. 6.5: Window and Viewport

XV= XVMIN + (XW-XWMIN) * (XVMAX-XVMIN) / (XWMAX-XWMIN)

YV= YVMIN + (YW-YWMIN) * (YVMAX-YVMIN) / (YWMAX-YWMIN)

Units of this type receive one instruction as input and provide a normalized set of coordinates as output.

The only instruction accepted by these units is the following:

NORM |x,x... |PAR1 |PAR2 |

where PAR1 is a pointer to an array with eight elements (four x,y pairs) that define the window and the viewport. PAR2 points to a 2DLINE structure (the same comment given for the vector generators applies here). It is quite possible that for a large number of vertices the normalization array is the same . This means that for each instruction the array has to be passed to the operation

unit. An improvement to this situation is to 'store' the normalising array in the operation unit. This action goes against the data-flow principles. A compromise is to implement the following protocol: PAR1 is a label or instance of a structure in the structure storage. Every time the structure is loaded the label is changed. In the operation units, if the instance is the same as the current array in the unit it just carries on with the execution. If not, the unit requests a copy of the array from the structure storage. This technique can be used for parameters that are frequently used in consecutive operations without the penalty of repeatedly loading the same information. In addition the protocol of copying maintains the principles of data-flow architecture.

A possible implementation of such units is:

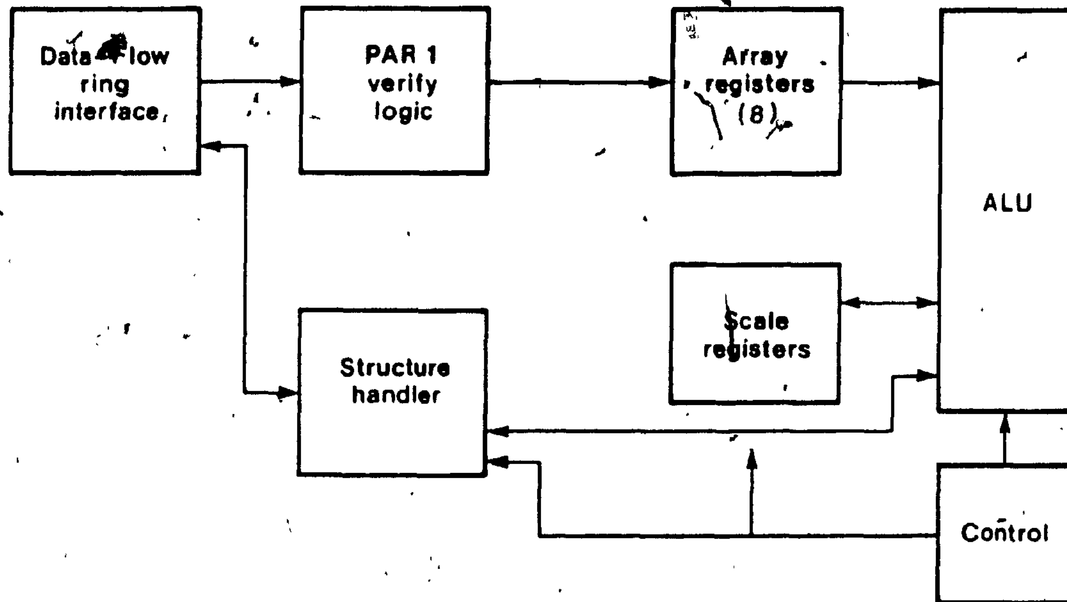

Fig. 6.6: Normalizer Block Diagram

The PAR1 verify logic block is only available if we implement the instance labeling described above. The array registers contain the four x,y pairs defining the window and viewports. The scale registers contain the ratios (XVMAX-XVMIN)/(XWMAX-XWMIN) and (YVMAX-YVMIN)/(YWMAX-YWMIN). These registers are updated every time the normalization array is changed. The structure handler gets x,y pairs from the 2DLINE structure and feeds them to the ALU. The results from the ALU are send to the same block that constructs a 2DLINE structure for output. Even if not explicit in the diagram, the ALU section can be made in-

such a way to support simultaneous calculations for the x,y pair of values.

6.1.4-Transformer

These units are basically matrix multipliers. They are responsible for applying to a vertex the transformation matrix that represents rotation, translation and scaling.

As in the previous operation units, these receive a transformation matrix and graphic structures as input and provide a transformed line structure to the subsequent instruction cells .

The units are able to support 2D or 3D transformations. Two commands are therefore available:

2DTRANS |x,x... |PAR1 |PAR2 |

and

3DTRANS |x,x... |PAR1 |PAR2 |

In both commands PAR1 receives the transformation matrix and PAR2 the graphic structures. The same considerations given for the normaliser units apply here.

The world coordinates presented to the tranformer units are homogeneous, therefore the transformation

matrices are square matrices of 3*3 and 4*4 for 2D and 3D respectively. The contents of the matrices and their implication in the efficiency of the execution are not discussed here. More information about this matter can be found in [Roge76].

The block diagram of a transformer unit is represented in the following figure.

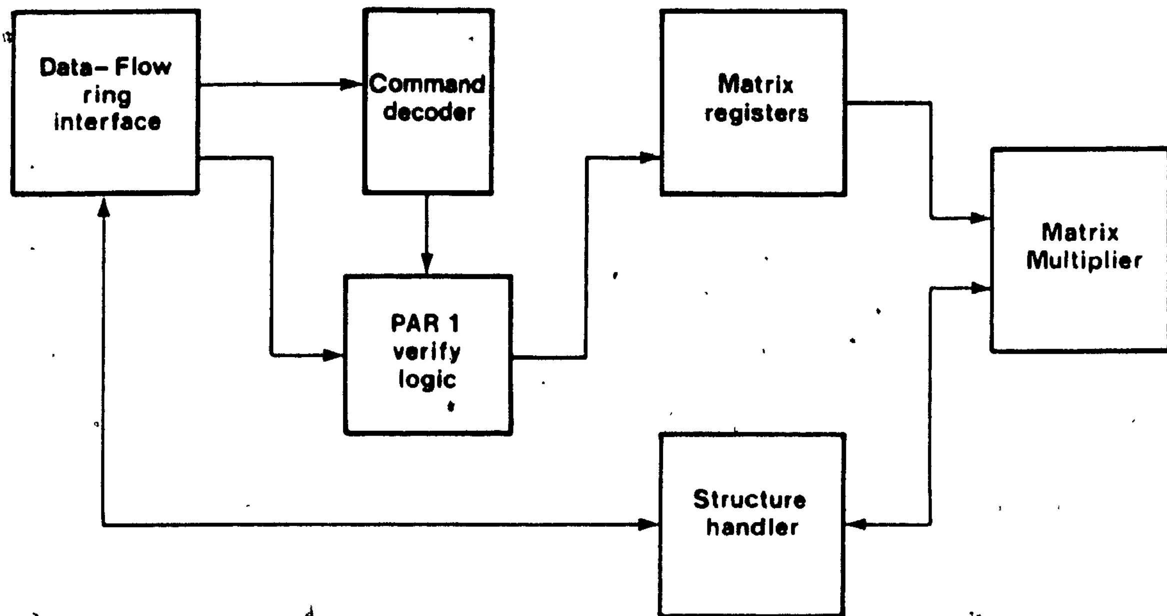

Fig. 6.7 : Transformer Block Diagram

The blocks in this picture have the same functionality as similar blocks in other units. Only the matrix multiplier has not yet been presented .There are several

ways to implement this block:

a) With one ALU and several registers, perform the multiplication and addition to generate the elements of the output matrix.

b) Have four ALU's in such a way that a row and column multiplication can be done in one pass. In this case the operations to generate one coordinate are done simultaneously.

c) To pipeline the arithmetic operations with the structure handler.

6.1.5-Projection Units

The main difference between this type of unit and the transformers is the meaning of the elements of the matrix. In some cases the projection and transformation matrices can be combined in one. A detailed description of the mathematics of planar geometric projections is given in [Carl78,Roge76].

PROJECT |x,x:.. |PAR1 |PAR2 |

is the command accepted by these units and the parameters are the same as the ones for 2DTRANS or 3DTRANS.

6.1.6-Complex Picture Generator

The vector generator units are the simplest case of dedicated circuitry to scan convert pictures. Not all graphics are composed of just straight lines. It is useful to rely on hardware to generate more complex pictures such as circles, ellipses respective arcs and filled polygons. These functions are available today in VLSI chips like the ACRTC from Hitachi [Hita84]. If chips like these are used as operation units in the architecture they will be responsible for such commands as:

```
CRCL |SIG x,x... |XO |YO |R |
ELPS |SIG x,x... |XO |YO |A |B |DX |
....
```

where XO,YO are the coordinates of the center point for both the circle and the ellipse; R the radius of the circle; A and B represent the ratio X**2/Y**2 on the elipse equation and DX the maximum displacement in the x coordinates from the ellipse center.

In order to integrate this chip in a data-flow ring the interface circuit can be organised as shown in the

following block diagram.

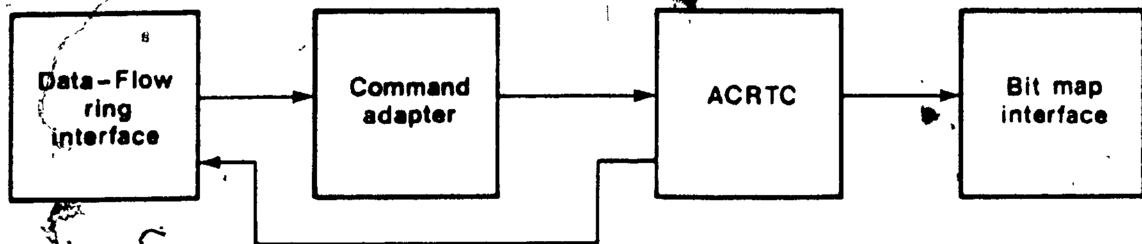

Fig. 6.8: ACRTC Block Diagram

The command adapter block is responsible for the conversion of the command code and parameters to the ones accepted by the ACRTC chip. It also generates the signal to be sent to the destination instruction cells.

6.1.7-Clipper

Any picture that is drawn in the screen has to be clipped to a 2D or 3D window. When this operation is performed the output is a different picture from the original, when the output crosses the window boundary. The same picture as the original if it is inside the window, and when it resides completely outside the window the result is an 'empty' picture.

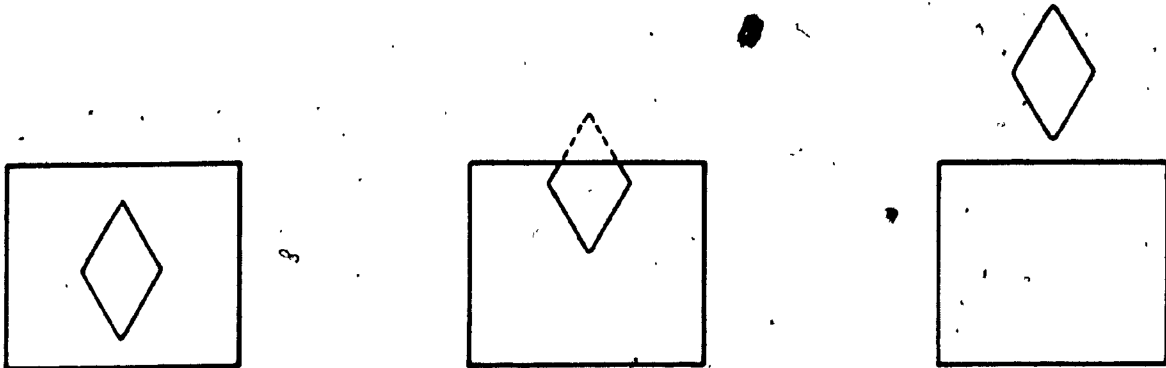

Fig. 6.9: Window Clipping

This last case is supported through a special code of the graphic structures. In order to terminate properly the execution of any picture description and avoid deadlocks, this structure is given to the destination instruction cells.

The two commands accepted by this unit are the following:

2DCLIP |x,x... |PAR1 |PAR2 |

3DCLIP |x,x... |PAR1 |PAR2 |

Where PAR1 defines the clipping window and PAR2 the picture structure to be clipped.

Due to the complexity of this operation it is difficult at the current state of art to have a dedicated piece of hardware that directly implements a clipping

algorithm. The best solution for this type of unit is to have a very fast CPU, ROM, RAM and the interface circuits. Some of the controllers available for digital signal processing can be used in this application [Texa82].

6.1.8-Input Devices

The input devices are responsible for interfacing user input to the system. Due to the diversity of input devices and ways of interfacing them we will consider here the categories defined by GKS for styles of interaction: Request,Sample and Event modes.

In request mode the application program requests an input and then waits for a response from the input device. A similar operation exists with the sample mode, the only difference being that the input device is continously sampled. In the last mode the data received from the input device is put in a queue. The application program first asks for data and if there is some in the queue it gets it through a request (GET command) [Hopg83].

To the unit these modes are implemented by different commands. The output of the unit is a signal for the event mode and the input data for all modes.

REQUEST n	|x,x... |	(request mode)
SAMPLE n	|x,x... |	(sample mode)
AWAIT n	|SIG x |	(event mode)
GET n	|x,x... |	" "

In any of these commands the suffix n is used to select which input device is addressed. This selection can be done by the arbitration circuit or if the devices are grouped by classes, in the interface with the data-flow ring.

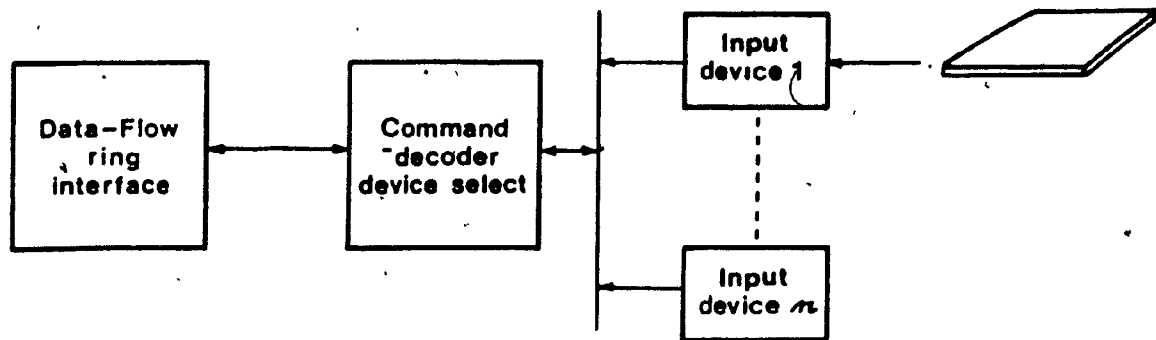

Fig. 6.10: Input Module

The previous diagram is a sketch of the interconnection of input devices to a data-flow ring.

6.1.9-ALUs

These units are responsible for the operations normally executed by the ALU of a general purpose computer. They include the following sub-set:

ADD |x,x... |PAR1 |PAR2 |
SUB |x,x... |PAR1 |PAR2 |
MUL |x,x... |PAR1 |PAR2 |
OR |x,x... |PAR1 |PAR2 |
AND |a,a... |PAR1 |PAR2 |
...

Depending on the requirements of the implementation these units can be simple ALU's or complex circuits including bit-slices or microprocessors with the corresponding mathematical coprocessor. In this case the block diagram of such units will resemble the following:

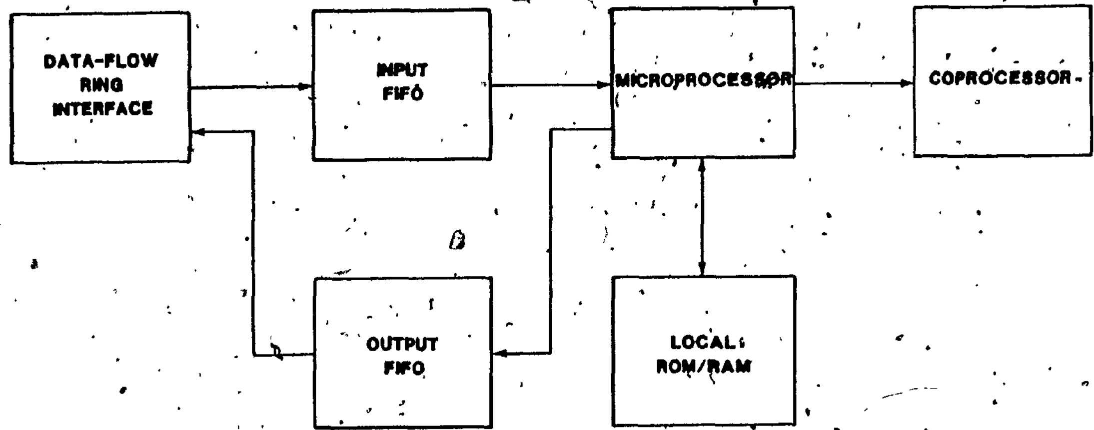

Fig. 6.11: ALU Module

6.2-Bit Map Interface

The units that operate on the bit map have to interface to the video memory. Before we describe how it is done let's look with more detail at the organization of this memory.

In current designs this video memory is accessed by two different circuits: one does a scan of the memory and provides the video output circuit with the contents of each pixel; the other is responsible for the interface between

the video memory and the display processor unit. Another characteristic of the bit map is the organisation imposed by the video bandwidth necessary for the high resolutions used nowadays. This bandwidth does not have a reciprocal in memory accesses, therefore in the same memory cycle more than one pixel data is stored in a shift register (8, 16, 32 or even 64). Another implication of high resolutions is the impossibility of interleaving refresh and DPU accesses (what is called transparent access to the video memory). These are done during the horizontal and vertical blank portions of the video.

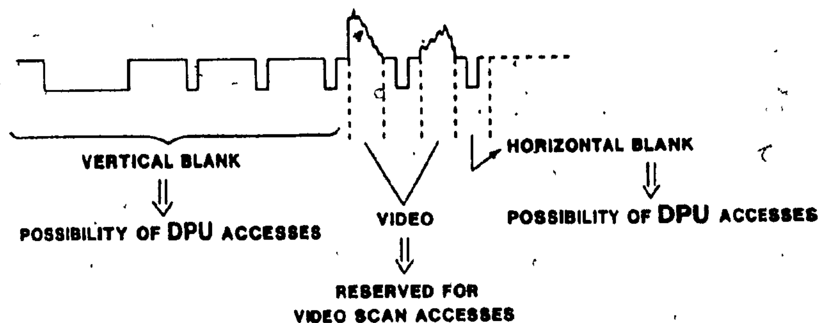

Fig. 6.12: Video Signal

In the above circumstances the DPU accesses to the memory may become a bottleneck and a limitation factor in the performance of the machine. Fortunately there are today a new generation of DRAMs that overcome this problem [Most84,NECE85,Texa84]. In the RAM internal circuit a long

shift register (up to 256 bits) is loaded in one RAS cycle and while its contents are shifted out the memory cells are available for DPU accesses. With this shift register, the number of accesses to refresh the screen are reduced by a factor of 16 in average. The increased availability of memory cycles for DPU accesses allows faster updates and better performances.

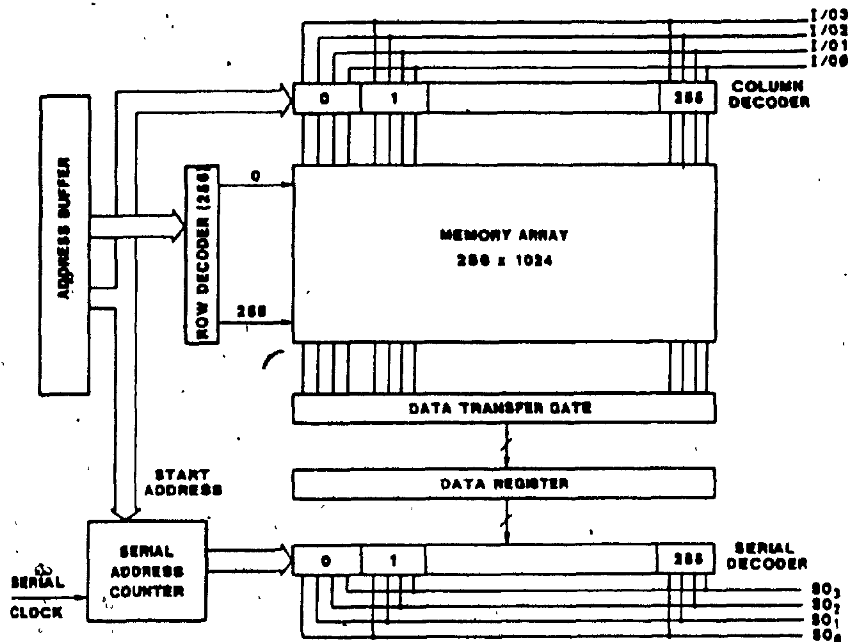

Fig. 6.13: Memory Chip Block Diagram

The time available for bit map accesses has to be distributed among the units that interface with it. This operation is performed by an arbitrator that selects which

of the units has access to the video memory.

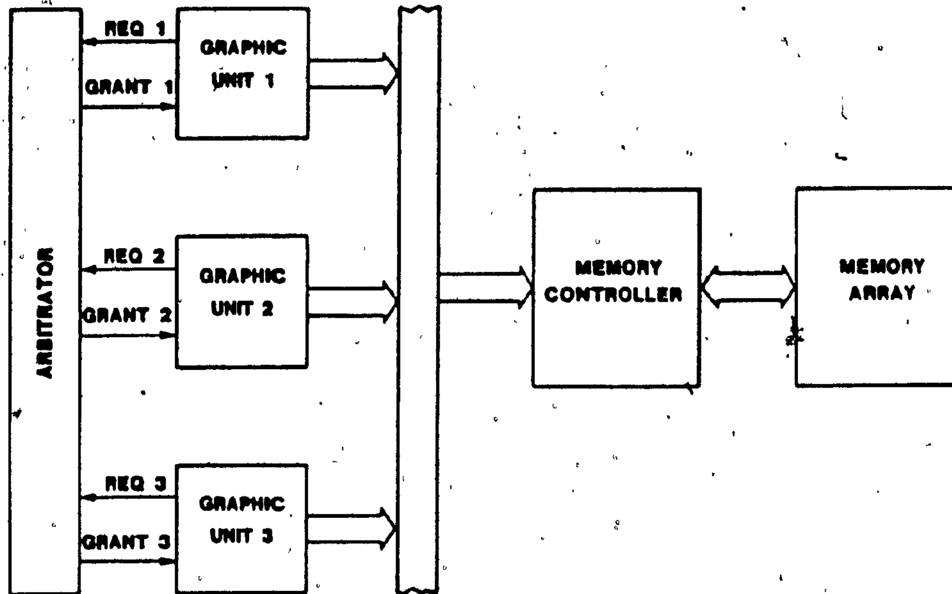

Fig. 6.14: Bit Map Interface

In the previous circuit when a unit has to perform a memory access it sets its request signal. The arbitration between requesting units is done while an access to the memory is performed.

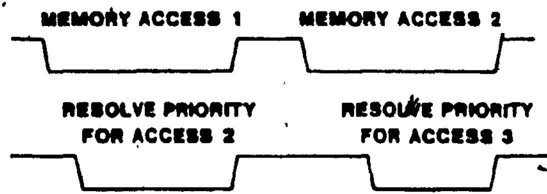

Fig. 6.15: Arbitration Timing

With this pipelining of acceses and arbitration the latency of the circuit is reduced. The arbitration algorithm will depend in the characteristics of the requesting units. If these have the same throughput the circuit should rotate the priority of the access between all the units. If the units have different throughput the arbitration circuit should use a weighted priority.

Each access to the bit map is actually a read modify write cycle. The read portion is used to make available the contents of the z buffer or segment priority. This part of the memory provides for each pixel its Z coordinate in a 3D system or the segment priority in a 2D machine. The value read is compared in the Write Control circuit with the value given by the accessing unit. A lower value in the bit map implies a replace action on the write portion of the cycle. Otherwise a new value is written into the pixel cell.

6.3-Data-flow Ring Interface

The interface of each of the operation units with the data-flow ring is described in a general form and for the case of multiple units i.e. more than one unit of the same kind, as for example, more than one vector generator.

Let's consider first that the arbitration network routes the instruction to the appropriate set of units using the information of the operation code of the instruction. The packet is then absorbed by the group interface circuit. This circuit is responsible for receiving the packet and store it in a temporary FIFO. Every operation unit, after reset or completion of an execution, provides to this circuit a ready signal and stays in a wait mode until it receives a packet, after which it disasserts the ready line. The interface circuit is responsible for choosing between the available units, of which one will receive the instruction packet.

The output part of the operation units have a similar interface. At the end of the execution the results are sent to the control FIFOs or data FIFOs (the number of FIFOs is imposed by the organisation of the distribution network. Because several units can access the same FIFO, each one

has an arbitration circuit that decides which of the units gets the access to this common resource. At the same time the arbitration circuit holds the operation unit in case of FIFO full or unavailability of the common bus to the FIFO.

The block diagram of such a general interface circuit is represented in the following figure:

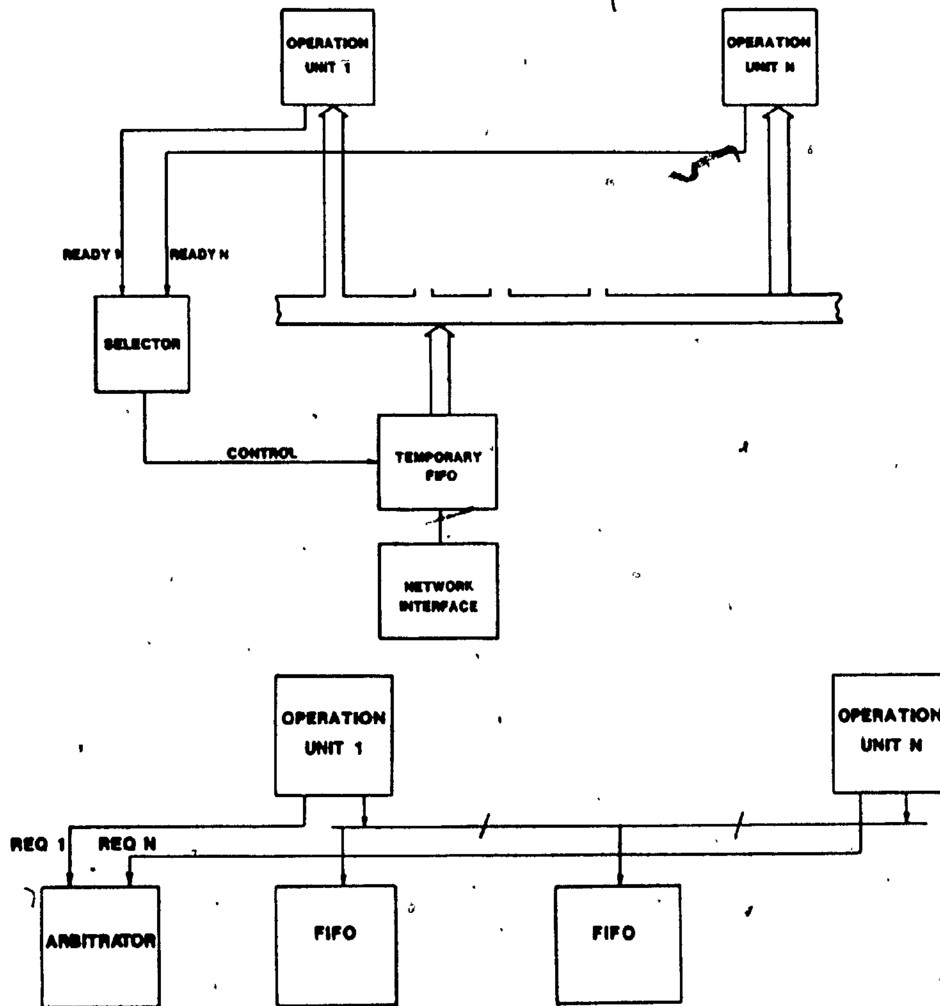

Fig. 6.16: Data-flow Interface

6.4-Data-flow Ring Architecture

All the units described above can be incorporated in any data-flow architecture that supports structures. Here we will propose a block diagram for a suitable architecture based on the requirements imposed by the application and on previous implementations [Denn84,Denn80,Myer81,Rumb77].

6.4.1-Instruction Cells

Referring back to the general block diagram of the architecture (figure 3.5) let's first consider the instruction cell block. The main functions of this block is to store the instructions of a data-flow program, to store the operands of such instructions received from the distribution network, keep track of the number of operands received at each instruction and provide to the arbitration network the instructions that can be executed.

The main characteristic of the implementation we propose here is the existence of two memories. One, the data memory, holds the operation code, the destination addresses and the operand values. The second, the control memory, holds the control information necessary to trigger

the instruction which is: gating codes, gate flag and value flag for each operand, the total number of signals that has to be received before the instruction can be executed, the actual number of received signals, a trigger expression and a pointer to the instruction structure in the other memory. This memory is supported by a control circuit that executes the following algorithm:

a) After program loading or after an instruction packet is send, reset all the flags and the counter of signals received.

b) When a signal is received increase the signal counter and compare to the total number to be received. In case of equality set the signal flag.

c) If an operand is received set the value flag and store the operand in the instruction memory.

d) Set the gating flag when a control code is received and matches the gating code of the operand. Otherwise disregard it.

e) For every signal,gating code or value received calculate the triggering expression which is:

Signal flag AND Value flag 1 AND Value flag 2 AND ... AND Value flag n AND Gate code 1 AND Gate code 2 AND ... AND Gate code n

When this expression is true the instruction is ready for execution.

The following block diagram gives the organisation of the instruction cell block as well as a more detailed overview of the control memory.

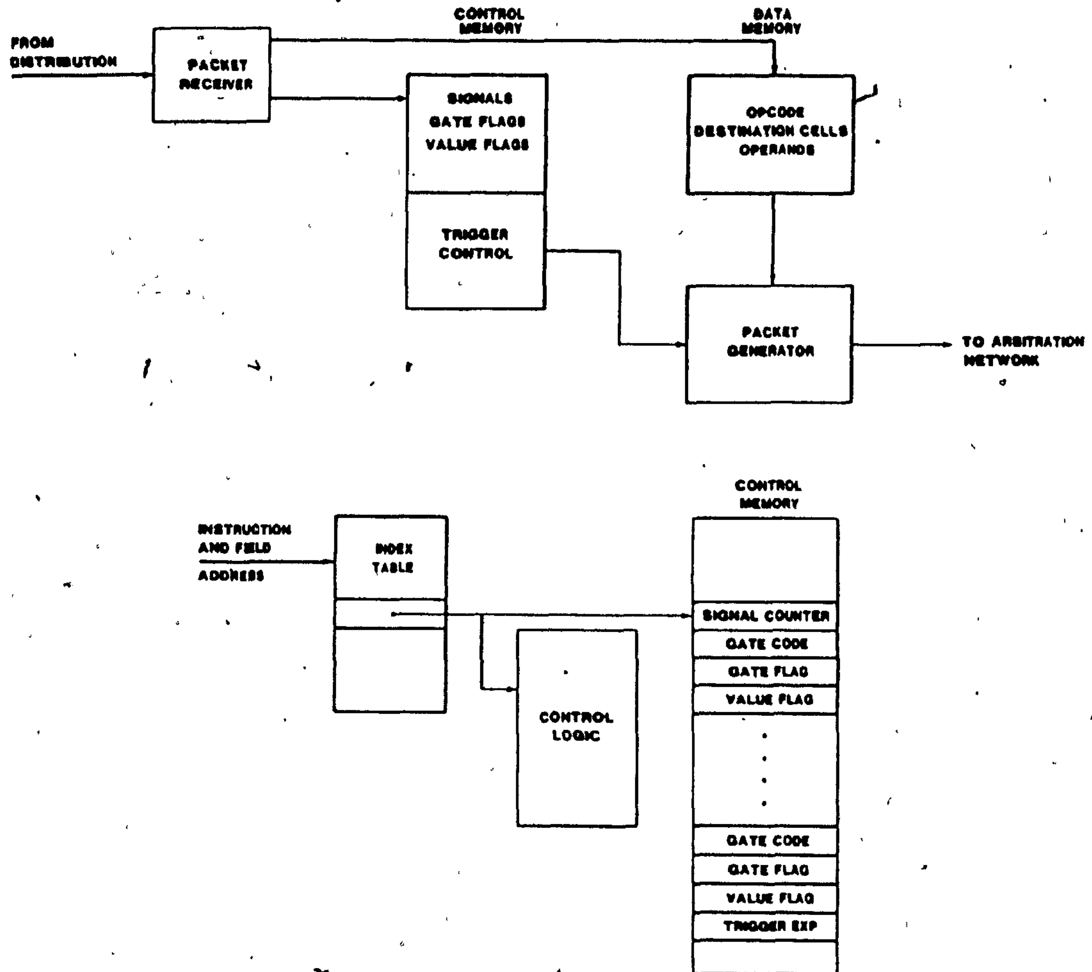

Fig. 6.17: Instruction Cells Module

The packet receiver block is responsible for sending the information received to the appropriate locations in the data and/or control memory. The address of such

locations is maintained in an index table initialised when a program is loaded in the memory. The packet that circulates in the distribution network has the following format: Header, destination address of the instruction cell and the destination field. The header contains the size of the packet and a code specifying if the information is a signal, a gating code, a value or an instruction to be loaded in the instruction memories.

The packet generator circuit is responsible for retrieving the necessary information from the data memory and building a packet that is then send to the arbitration circuit. This packet should have a format similar to the following: Header, operation code, address of destination cells and operand values. The header contains information about the size of the packet and the code that specifies what type of execution units can execute the instruction in the packet.

6.4.2-Arbitration and Distribution Networks

The function of these two blocks is to route the instructions with their operands to the operation units (the arbitration network) and the results from the operation units to the instruction cells (the distribution

network). The throughput of the system is highly dependent on the performance and the degree of parallelism that can be incorporated in these circuits.

A direct interconnection between each instruction cell and each operation unit is impracticable due to the large number of connections necessary. To reduce this number the elements of both blocks are grouped in an hierarchy. The number of levels in this hierarchy will depend on the speed of the interconnection network and the number of units and instruction cells to connect [Myer81].

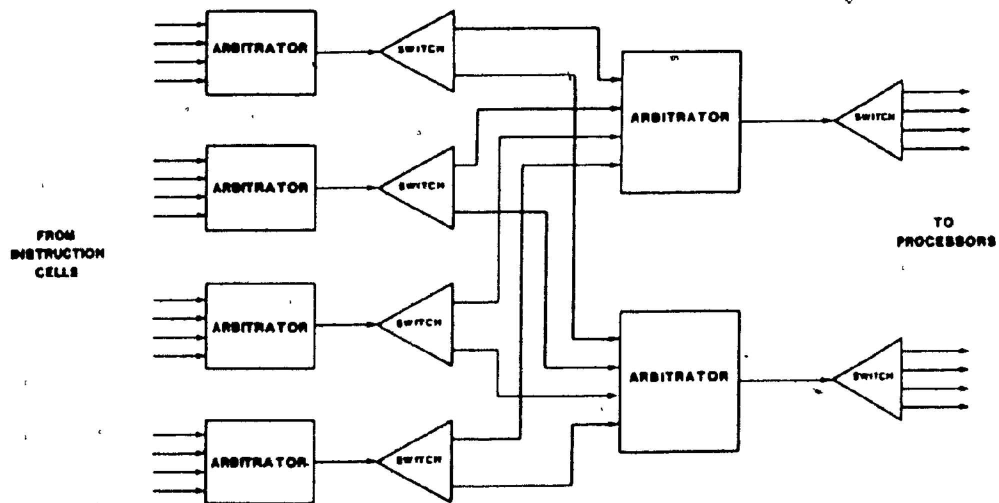

Fig. 6.18: Arbitration Network

When we described the operation units and their interface to the data-flow ring we emphasized the possibility of grouping together units of the same type in order to simplify the interface circuit. At the same time in the description of this interface circuit we implicitly gave the description of an arbitration circuit on the input side and of a distribution circuit on the output side of the execution units.

The same type of circuits can be used to implement the two networks that are the subject of this paragraph. The following diagrams and description explain in more detail the operation of both networks.

a) The arbitration network

The packets generated in the different groups of instruction cells are stored in a FIFO located in the packet generator section of the instruction cell block. The packet generators that are connected to the same channel request the access to that channel via a request signal to an arbitrator. This arbitrator determines which of the packet generators should have access to the arbitrator FIFO. The request arbitration is done at the same time that another packet generator transfers its packet to the FIFO. From this FIFO a switch circuit selects one of the output sections of the arbitration circuit based on information

provided in the header of the packet. Each output section has its one FIFO where packets are stored until they can be passed to the next arbitration unit. The arbitration of these output sections is done in the same way as it was done for the packet generator circuits.

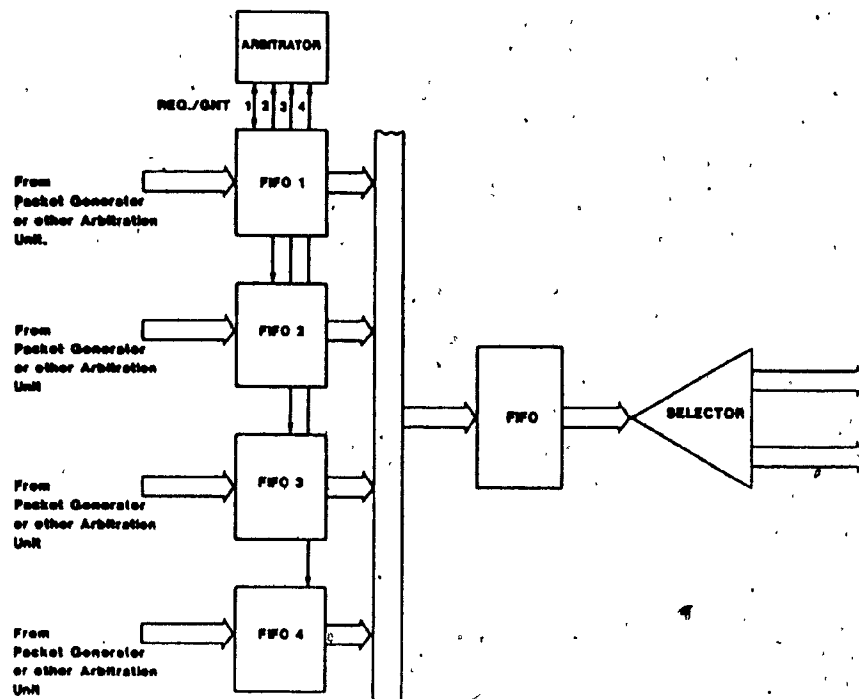

Fig. 6.19: Arbitration Module

The network can be built with such units until they are directly interfaced with the execution units or the structure section of the machine. In a reduced system the output sections of the first arbitrator could interface directly with the execution units interface circuit.

b) The distribution network

The distribution network is similar to the arbitration network in its construction with the only difference in the first stage that is a switching circuit. The results coming out of each operation unit generates one or more packets that are send to a specific FIFO. This selection is based on the address of the destination available in the header of the packet and the grouping of the instruction cells. Each FIFO of each group of operation units that address the same group of instruction cells shares the same communication channel. An arbitrator determines which FIFO will have its packet sent to the next switching circuit.

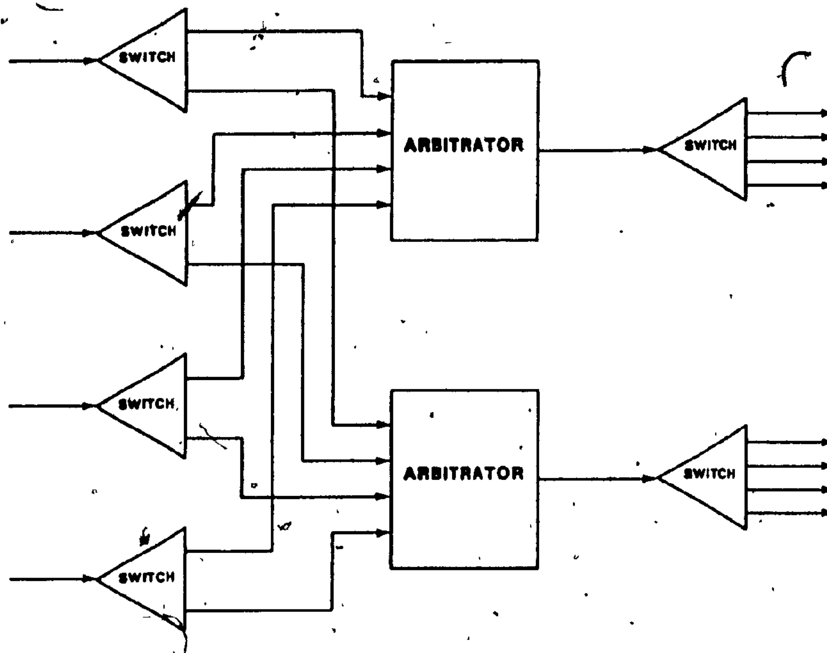

Fig. 6.20: Distribution Network

The process is terminated when the next switching circuit is the one incorporated in the packet receiver circuit of a group of instruction cells.

6.5 Structure Storage and Processing

The operation of this block of the architecture is equivalent to the operation units section. It receives from the arbitration network instructions and operands and after processing, a result is send to the distribution network.

The main characteristic of its instructions is that they operate in data structures [Acke78]. Consider, for example, the case of an array. The operations that can be applied to this structure are: 1-create the array; 2-add an element to it; 3-delete an element; 4-find out the limits of the array; and others as defined in 4.1. For the first one the operands sent with the instruction are the values of the elements of the array. Upon receiving an instruction like this the storage unit will determine a location to store the elements and build the appropriate indexing tables. If the instruction received is a delete instruction, the operand is the index of the element to be deleted and the function of the unit will be to rearrange the links and index tables depending on the way it supports the structures. Another possible and useful instruction not mentioned above is the retrieve instruction. Its function is to read an element whose index is specified in the operand of the instruction. The value is then sent to the distribution network. The following list is a subset of the instruction that can be supported by this block of the architecture.

CREATE ARRAY |SIG x,x... |n |EL1 |EL2 |... |ELn |

The operands for this instruction are: the number of

elements followed by the value of each element in increasing order of their index.

ADD ELEMENT |SIG x,x... |NAME |EL |

NAME is the label or identification of the array and EL the value to be added. The addition is done at the end of the array.

DELETE ELEMENT |SIG x,x... |NAME |INDEX |

The element with the same index of the value INDEX is deleted from the array identified by NAME.

ARRAY[] |x,x... |NAME |INDEX |N |

The values of the N elements of the array NAME starting from INDEX are sent to the distribution network.

Two different structures can be used (2DLINE and 3DLINE). The difference between them and the array is that their elements have two or three values respectively, and the structure has a header that contains the number of coordinates and the priority of the segment to which this structure belongs. The instructions that support these structures are very similar to the ones given above for

arrays.

The matrices that are used for transformations can be represented with a single array or more appropriately by an array of an array. It was mentioned that most of the transformations do not change frequently. It would therefore be more efficient if these operation units stored the transformation array. To support this feature we described the concept of instance and instance labeling in section 6.1.3. Two different instructions would be required to retrieve complete structures: one, the most often used, would provide the structure instance as output; the other would provide all values of the structure, to be used when an operation unit has to update the transformation matrix. The idea of instances it is not new in data-flow architectures [Arvi81,Davi78].

The hardware that implements this structure storage and processing unit is a complex unit, possibly based on a general purpose CPU with special circuits to improve the support of the structure operations. The block diagram for this unit resembles the block diagram of figure 6.11.

7

Conclusion

The architecture presented in the previous chapter is the final element in our study. We have shown that current graphic systems do not take full advantage of graphic parallelism, and that this characteristic matches data-flow principles well enough to allow the use of data-flow languages and data-flow computers in display processing units.

In order to apply the results of this work to the development of a new type of graphic systems, more research has to be done in both fields, graphics and data-flow.

The areas that need further research are: definition of a complete GKS language binding to VAL; study of the implications of using VAL in the design of graphic programs; examination of the modifications to current VAL compilers needed to accommodate the heterogeneous set of operation units; definition and study of the implementation of the operation units dedicated to graphic functions; investigation of the communication problems in the arbitration and distribution networks; study and simulation of the architecture to verify performance characteristics

and finally implementation of a prototype.

References

The following list contains the references consulted while this thesis was being prepared. Only those references marked with an asterisk are actually cited in the text.

Acke78*
Ackerman,W.B. [78]
A Structured Processing Facility for Data Flow Computers
Proc. 1978 Int. Conf. on Parallel Processing

Acke79*
Ackerman,W.B. [79]
Data Flow Languages
Proceedings 1979 Nat. Computer Conf. p1087-1095

Arvi77*
Arvind and Gostelow, K. P. [77]
A computer capable of exchanging processors for time
Proc. IFIP Congress. p849

Arvi78
Arvind,Gostelow,K.P. and Plouffe,W. [78]
The Preliminary Id Report
Tech. Rep. TR 114a Dept. Computer Science Univ. California

Arvi80*
Arvind and Kathail,V. [80]
A processing element for a large multiprocessor dataflow machine
Proc. Int. Conf. Circuits and Computers

Arvi81a*
Arvind and Kathail,V. [81]
A Multiple Processor Data Flow Machine that Supports Generalised Procedures
The 8th Annual Symposium on Computer Architecture. p291

Arvi81b*
Arvind and Thomas,Robert E. [81]
I-Structures:An Efficient Data Structure for Functional Languages
Tech. Rep. TM-178 Dept. Computer Science MIT

Arvi83
Arvind and Iannucci,R.A. [83]
A Critique of Multiprocessing von Neumann Style
The 10th Annual International Symposium on Computer Architecture. p426

Ashc77
Ashcroft,E.A. and Wadge,W.W. [77]
Lucid, a Nonprocedural Language with Iterationd
Commun. ACM Vol 20 Num 7 p519

Ather78
Atherton,P., K. Weiler and D. Greenberg [78]
Polygon shadow generation
SIGGRAPH'78 proceedings p275

Back78*
Backus,J. [78]
Can Programming be liberated from the von Neumann Style? A Functional Style and its Algebra of Programs
Commun. ACM Vol 21 Num 8 p613

Barr74
Barrett,R.C. and B.W. Jordan,Jr. [74]
Scan-conversion Algorithms for a cell organized raster display
Comm. of the ACM, 17(3),March 1974 p157

Bask76
Baskett,F. and L. Shustek [76]
The design of a low-cost video graphics terminal
SIGGRAPH'76 proceedings p235

Bech80
Bechtolsheim,A. and F. BAskett [80]
High-performance raster graphics for microcomputer systems
SIGGRAPH'80 proceedings p43

Berg78*
Bergeron,R.D., P. BONO, and J.D. Foley [78]
Graphics programming using the core system
Computing Surveys 10(4) p389

Blin78
Blinn,J.F. and M.E. NEwell [78]
Clipping using homogeneous coordinates
SIGGRAPH'78 proceedings p245

BLTC84*
BLT Chip [84]
The PMR 96016 Blt chip
Pacific Mountain Research,Inc. data sheet

Boot78
Booth,Kellogg S. [78]
Tutorial:Computer Graphics
IEEE Comp. Society

Burk81
Burkowski,F.J. [81]
A Multi-User Data Flow Architecture
The 8th Annual Symposium on Computer Architecture.
p327

Burk82
Burkowski,F.J. [82]
Instruction Set Design Issues Relating to a Static
Data Flow Computer
The 9th Annual Symposium on Computer Architecture.
p101

Calu82
Caluwaerts,L.J.,DeBacker,J. and Peperstraete J.A. [82]
A Data Flow Architecture with a Paged Memory System
The 9th Annual Symposium on Computer Architecture.
p120

Calu
Caluwaerts,L.J.,DeBacker,J. and Peperstraete ,J. [83]
Implementing Streams on a Data Flow Computer System
With Paged Memory
The 10th Annual International Symposium on Computer
Architecture. p76

Carl78*
Carlbom,I. and J.paciorek [78]
Geometric projection and viewing transformations
Computing Surveys 1(4) p465

Catm78
Catmull,E. [78]
A hidden-surface algorithm with anti-aliasing
SIGGRAPH'78 proceedings p6

Clar80*
Clark,James [80]
A VLSI Geometry Processor for Graphics
Computer Vol 13 Num 7 p59

Comt78
Comte,D.,Durrieu,G.,Gelly,O.,Plas,A., and Syre,J.C. [78]
Parallelism,Control, and Synchronization Expression in a Single Assignment Language
Sigplan Notices Vol 13 Num 1 p25

Corn79*
Cornish,M. [79]
The TI data flow architectures: The power of concurrency for avionics
Proc. 3rd Conf. Digital Avionics Systems. p19

Crow81
Crow,F. [81]
A comparison of antialiasing techniques
IEEE Computer Graphics and Applications 1(1) p40

Davi78*
Davis,A. [78]
The Architecture and System Method of DDM1:A Recursively Structured Data Driven Machine
Computer Architectures News Vol 6 Num 7 p210

Denn72*
Dennis,Jack B. [72]
Data flow schemas
Lecture notes in computer science. p187
Springer-Verlag 1972

Denn75
Dennis,J.B. and Misunas,D.P. [75]
A preliminary architecture for a basic data flow processor
Proc.2nd Int. Symp. Computer Architecture. p126

Denn77
Dennis,Jack B. and Weng,K. K.-S. [77]
Application of Data Flow Computation to the Weather Problem
Tutorial on Parallel Processing. p223

Denn79a*
Dennis,Jack B. [79]
The Varieties of Data Flow Computers
Tutorial on Parallel Processing. p210

Denn79b
Dennis,J.B. and Weng,K.S. [79]
An Abstract implementation for concurrent computation with streams
Proc. 1979 Int. Conf. on Parallel Processing p35

Denn80a*
Dennis,Jack B. [80]
Data Flow Supercomputers
Computer Vol 13 Num 11 p48

Denn80b*
Dennis,J. B.,Boughton,G. Andrew and Leung,Clement K.C.[80]
Building Blocks for Data Flow Prototypes
The 7th Annual Symposium on Computer Architecture.p1

Denn84*
Dennis,J.B.,Gao,[illegible].-R. and Todd,K.W. [84]
Modeling the Weather with a Data Flow Supercomputer
IEEE Transactions on Computers Vol C-33 Num 7 p592

Door84*
Doornink,Douglas J. and Dalrymple,John C. [84]
The Architectural Evolution of a High Performance Graphics Terminal
Computer Graphics and Applications Vol 4 Num 4 p47

Farr83
Farrow,R. [83]
Attribute Grammars and Data Flow Languages
Sigplan Notices Vol 18 Num 6 p28

Fole79*
Foley,J.D., J. Templeman and D. Dastyar [79]
Some raster graphics extensions to the Core System
SIGGRAPH'79 proceedings p15

Fole82*
Foley,James D. and van Dam,Andries [82]
Fundamentals of Interactive Computer Graphics
Addison-Wesley

Free79*
Freeman,Herbert [79]
Tutorial and Selected Readings in Interactive Computer Graphics
IEEE Comp. Society

GKS84*
GKS [84]
Information Processing Systems Graphical Kernel System
Computer Graphics
ACM

Gost79
Gostelow,K.P., and Thomas,R.E. [79]
A view of dataflow
Proc. Nat. Computer Conf. p629

Haml77
Hamlin,G. and C. Gear [77]
Raster-scan hidden surface algorithm techniques
SIGGRAPH'77 proceedings p206

Hita*
Hitachi Ltd. [84]
HD63484 ACRTC Advanced CRT controller

Homm82
Hommes,F. [82]
The Heap/Substitution Concept-An Implementation of Functional Operations on Data Structures for a Reduction Machine
The 9th Annual Symposium on Computer Architecture.p248

Hopg83*
Hopgood,F.R.A. [83]
Introduction to the Graphical Kernel System (GKS)
Academic Press Inc.

Inter81*
International Standards organization . [81]
Graphical Kernel System (GKS),Version 6.6

Inter84*
Intergraph Inc.
IEDS User Manual

Kilg81*
Kilgour,A.C. [81]
A hierarchical model of a graphics system
Computer graphics 15(1) p35.

Kish
Kishi,M.,Yasuhara, H. and Kawamura,Y. [83]
DDDP: A Distributed Data Driven Processor
The 10th Annual International Symposium on Computer Architecture. p236

Kron83
Kronlof,K. [83]
Execution Control and Memory Management of a Data Flow Signal Processor
The 10th Annual International Symposium on Computer Architecture. p230

Kuls68
Kulsrud,H.E> [68]
A general purpose graphic language
Comm. of the ACM 11(4) p247

Levo77*
Levoy,M. [77]
A color animation system based on the multiplane technique
SIGGRAPH'77 proceedings p65

Litv82
Litvin,Y. [82]
Parallel Evolution Programming Language for Data Flow Machines
Sigplan Notices Vol 17 Num 11 p50

MacL82*
MacLennan,B.J. [82]
Values and Objects in Programming Languages
Sigplan Notices Vol 17 Num 12 p70

Marc83
Marczynski,R.W. and Milewski,J. [83]
A Data Driven System Based on a Microprogrammed Processor Module
The 10th Annual International Symposium on Computer Architecture. p98

Matr84*
Matrox Inc.
SX-900 Manual

McGr80
McGraw,J. [80]
Data Flow Computing:Software Development
IEEE Transactions on Computers Vol C-29 Num 12 p1095

McGr82*
McGraw,James R. [82]
The VAL Language:Description and Analysis
ACM Transactions on Programming Languages and Systems
Vol 4 Num 1 p44

Mead80*
Mead,C.A., and Conway,L.A. [80]
Introduction to VLSI systems
Addison-Wesley

Mesh84
Meshach,William [84]
Data Flow IC makes Short Work of Though Processing
Chores
Electronic Design p191

Misu79
Misunas,David P. [79]
Report on the Second Workshop on Data Flow Computer
and Program Organization
Dept. Computer Science MIT

Most84*
Mostek Inc.
MK4V64(P/N)-10

Myer81*
Myers,Glenford J. [81]
Data Flow Architectures
Advances in Computer Architecture p463
John Wiley & Sons

NECE84*
NEC Electronics Inc. [84]
uPD7281D Image Pipelined Processor (ImPP)

NECE85*
NEC Electronics Inc. [85]
uPD41264 262,144-bit Dual port dynamic nmos ram

Newm76
Newman,W.M. [76]
Trends in graphic display design
IEEE transactions on computers C-25(12) p1321

Olde84
Oldehoef,A.E. and Jennings,S.F. [84]
Data Flow Resource Managers and Their Synthesis from Open Path Expressions
IEEE Transactions on Software Engineering Vol SE-10 Num 3 p244

Pavl79
Pavlidis,T. [79]
Filling algorithms for raster graphics
Computer graphics and image processing 10(2) p126

Pitt80
Pittewzy,M. and D. Watkinson [80]
Bresenham's algorithm with grey scale
Comm. of the ACM 23(11) p625

Plas76*
Plas,A.,et al. [76]
LAU system architecture:A parallel data driven processor based on single assingment
Proc. 1976 Int. Conf. Parallel Processing. p293

Requ83
Requa,J. E. [83]
The Piecewise Data Flow Architecture Control Flow and Register Management
The 10th Annual International Symposium on Computer Architecture. p84

Roge76*
Rogers,David F. and Adams,J.Alan [76]
Mathematical Elements for Computer Graphics
McGraw-Hill

Rumba77*
Rumbaugh,James [77]
A Data Flow Multiprocessor
IEEE Transactions on Computers Vol C-26 Num 2 p138

Shou79*
Shoup,R. [79]
Color table animation
SIGGRAPH'79 proceedings p8

Shro77

Shroeder,M.A., and Meyer,R.A. [77]
A distributed computer system using a data flow approach
Proc. 1977 Int. Conf. Parallel Processing. p93

Sowa82

Sowa,M. and Murata,T. [82]
A Data Flow Computer Architecture with Program and Token Memories
IEEE Transactions on Computers Vol C-31 Num 9 p820

Srin81

Srini,V.P. [81]
An Architecture for Extended Abstract Data Flow
The 8th Annual Symposium on Computer Architecture.p303

Suen79*

Suenaga,Y.,T. Kamae and T. Kobayashi [79]
A high-speed algorithm for the generation of straight lines and circular arcs
IEEE Transactions on Computers, TC-28(10) p728

Suth74

Sutherland,I.E. and G.W. Hodgman [74]
Reentrant polygon clipping
Comm. of the ACM 17(1) p32

Taka83

Takahashi,N. and Amamiya,M. [83]
A Data Flow Processor Array System:Design and Analysis
The 10th Annual International Symposium on Computer Architecture. p243

Texa82*

Texas Instruments Inc. [82]
TMS32010 High-performance 16/32-bit microcomputers

Texa84*

Texas Instruments Inc. [84]
TMS4161 Multiport video memory

Toko83

Tokoro,M.,Jagannathan,J.R. and Sunahara ,H. [83]
On the Working Set Concept for Data Flow Machines
The 10th Annual International Symposium on Computer Architecture. p90

Tre178*
Treleaven,P.C. [78]
Principle components of a data flow computer
Proc. 1978 Euromicro Symp. p366

Tre182*
Treleaven,Philip C.,Brownbridge,DavidR. and Hopkins,Richard P. [82]
Data Driven and Data Demand Computer Architecture
ACM Computing Surveys Vol 14 Num 1 p93

VLSI84*
VLSI Technology Inc. [84]
VL16160 Raster-op graphics/boolean operator chip

Wats79
Watson,I. and Gurd,J. [79]
A Prototype Data Flow Computer with Token Labelling
Proc. Nat. Computer Conf. New York Vol 48 p623
AFIPS Press

Appendix I

VAL Syntax

The VAL syntax presented here is not complete and possibly incorrect as a language syntax. It should be used as a guide for the examples and explanations used in the text. These rules are an adaptation from a syntax description given in [Denn79b] and examples from [McGr82]. Possibly because the language is experimental, we were unable to find a complete, up to date syntax. The command formats for the graphic functions are given in appendix IV.

```
<program>::= PROGRAM <name> { <function def> } <exp> END
<function def>::= FUNCTION <name>
                  ( <input list> RETURNS <output list> )
                  { <type declaration> ; }
                  { <function def> ; }
                  <exp> ENDFUN
<input list>::= <type declaration> { , <type declaration> }
<type declaration>::= <name> { , <name> } : <type>
<output list>::= <type> { , <type> }
```

```
<exp>::= <primitive opr>
        | <exp> { , <exp> }
        | <let-block exp>
        | <conditional exp>
        | <forall exp>
        | <fordo exp>
        | <iter-block>
        | <application exp>
<primitive opr>::= <exp> <primitive opr> <exp>
                 | <primitive opr> ( <exp> )
                 | <name>
                 | <constant>
<conditional exp>::= IF <exp> THEN <exp> ELSE <exp> ENDIF
<let-block exp>::= LET { <type declaration> := <exp> ; }
                   IN <exp> ENDLET
<forall exp>::= FORALL <name> IN [ <exp> . <exp> ]
                  { <type declaration> := <exp> ; }
                  <evaluation> | <construction>
                ENDALL
<evaluation>::= EVAL PLUS | TIMES | MAX | MIN | AND | OR
                <exp>
<construction>::= CONSTRUCT <exp>
<fordo exp>::= FOR { <type declaration> := <exp> ; }
               DO <exp> ENDFOR
<iter-block>::= ITER <name> := <exp> { ; <name> := <exp> }
```

<application exp>::= <name> (<exp>)

<type>::= INTEGER | REAL | BOOLEAN |CHARACTER-STRING
| STRUCTURE

Appendix II

Examples of Graphic Programs

Two examples are given in this appendix. The first one is a routine to draw a line using the Bresenham algorithm, assuming a very primitive graphic system (the only graphic function used is WRITE-PIXEL). It was extracted from [Fole82]. The second is a very simple example that gives an idea about the format of a display. It was extracted from [Matr84].

The Bresenham's algorithm is attractive because it uses only integer arithmetic. No real variables are used, and hence rounding is not needed. This version works only for lines with slope between 0 and 1, it can be generalized for lines with other slopes.

```
PROCEDURE Bresenham (x1,y1,x2,y2,value:INTEGER);
     VAR dx,dy,incr1,d,x,y,xend:INTEGER;
BEGIN
     dx:=ABS (x2-x1);
     dy:=ABS (y2-y1);
     d:=2*dy-dx;
     incr1:=2*dy;
```

```
        incr2:=2*(dy-dx);
        IF x1 > x2
             THEN BEGIN
                  x:=x2;
                  y:=y2;
                  xend:=x1
                  END
             ELSE BEGIN
                  x:=x1;
                  y:=y1;
                  xend:=x2
                  END
        WRITE-PIXEL (x,y,value);
        WHILE x < xend DO BEGIN
             x:=x+1;
             IF d < 0
                  THEN d:=d+incr1
                  ELSE BEGIN
                       y:=y+1;
                       d:=d+incr2
                       END
                  WRITE-PIXEL (x,y,value)
        END
   END
```

The following list is a display list for a DPU with support of high-level graphic commands. The codes given include the opcode and the parameters.

```
24,A0,00,A0,00b      ;Move absolute to (A0,A0)
80,A0,00             ;Circle with radius=A0
24,E0,01,40,01       ;Move absolute to (1E0,140)
80,50,00             ;Circle with radius=50
```

This display list can be located anywhere in main memory. When instructed the DPU will read it and execute the commands specified above when finished it advises the host CPU that it is ready for another list.

Appendix III

GKS Commands

The following is a subset of the most important commands in VAL. For each one it is given the command name, the inputs and a brief description of the command operation.

Graphical output.

POLYLINE(N,XPTS,YPTS)

N	Number of points
XPTS	X coordinates
YPTS	Y coordinates

Draws N-1 segments joining adjacent points starting with the first point and ending with the last.

SET POLYLINE INDEX(N)

N	Representation index

Selects line representation for all subsequent polylines.

POLYMARKER(N,XPTS,YPTS)

N	Number of points
XPTS	X coordinates
YPTS	Y coordinates

Places a centered marker at each point.

SET POLYMARKER INDEX(N)

N Representation index

Selects marker representation for all subsequent polymarkers.

FILL AREA(N,XPTS,YPTS)

N Number of points

XPTS X coordinates

YPTS Y coordinates

Fills the area delimited by the polyline (N,XPTS,YPTS). If the area is not close the polyline is extended to join the last point to the first point.The polyline is not drawn.

SET FILL AREA INDEX(N)

N Pattern index

Selects fill area pattern for all subsequent fill areas.

TEXT(X,Y,STRING)

X X coordinate

Y Y coordinate

STRING String of characters

Writes string in the position (X,Y).

SET CHARACTER HEIGHT(H)

H Height index

Sets character height for all subsequent text

commands.

SET CHARACTER UP VECTOR(X,Y)

X X component

Y Y component

Defines character orientation and is used as a reference for text path and text alignment.

SET TEXT(PATH)

PATH [RIGHT,LEFT
UP,DOWN]

Defines character position in reference to the up vector direction.

SET TEXT ALIGNMENT(HORIZ,VERT)

HORIZ [LEFT,CENTER,RIGHT]

VERT [TOP,BOTTOM,CAP,
HALF,BASE]

Defines position of the string in relation to the string position point.

SET TEXT INDEX(N)

Defines character quality.

Coordinate systems.

World coordinates(WC)-Cartesian coordinate system used to present graphical output to GKS.

Normalised device coordinate(NDC)-Virtual or normalised

device which has a display surface visible in the range 0 to 1 in both the x and y directions.

Window-Area of the world space to be displayed.

Viewport-Area of the normalized device where a window will be displayed.

SELECT NORMALIZATION TRANSFORMATION(N)

Selects the window to viewport transformation defined by window N and viewport N.

SET WINDOW(N,XWMIN,XWMAX,YWMIN,YWMAX)

Defines window in world coordinates.

SET VIEWPORT(N,XVMIN,XVMAX,YVMIN,YVMAX)

Defines viewport in ND coordinates.

SET CLIPPING INDICATOR(IND)

IND [CLIP,NOCLIP]

Indicates if images will be clipped or not to the viewport boundary.

Segments.

CREATE SEGMENT(ID)

ID Segment identification

Opens a segment. Subsequent calls to output primitive functions will be inserted into the segment and outputed.

CLOSE SEGMENT

Closes a segment. Note-only one segment is allowed to

be open at any time.

DELETE SEGMENT(ID)

ID Segment identification

Deletes the segment identified.

SET SEGMENT TRANSFORMATION(ID,MATRIX)

ID Segment identification

MATRIX 2*3 Transformation matrix

Allows scaling, rotation and translation of the designated segment.

EVALUATE TRANSFORMATION MATRIX

(FX,FY,TX,TY,R,SX,SY,SWITCH,MATRIX)

FX,FY X,Y of fulcrum point

TX,TY Translation vector

R Rotation angle in radians

SX,SY Scaling factor

SWITCH WC or NDC

MATRIX Output matrix

Constructs transformation matrix based on the input parameters.

ACCUMULATE TRANSFORMATION MATRIX

(MATIN,FX,FY,TX,TY,R,SX,SY,SWITCH,MATOUT)

Allows more general transformation matrices to be constructed.

SET VISIBILITY(ID,VIS)

VIS [VISIBLE,INVISIBLE]

Defines if the segment is visible or invisible.

SET HIGHLIGHTING(ID,HIGH)

HIGH [HIGHLIGHTED,NORMAL]

Indicates if the segment should be highlighted or normal.

SET SEGMENT PRIORITY(ID,PRIORITY)

PRIORITY 0< <1.0

Defines the segment priority.

RENAME SEGMENT(OLD,NEW) Rename segment.

Graphical input devices.

There are six classes of input devices.

LOCATOR	Which inputs a position
PICK	Which identifies a displayed object
CHOICE	Which selects from a set of alternatives
VALUATOR	Which inputs a value
STRING	Which inputs a string of characters
STROKE	Which inputs a sequence of (X,Y) positions

In the following commands there are three common parameters:

WS	Workstation identification
DV	Device identification
ST	Device status

REQUEST LOCATOR(WS,DV,ST,NORMTR,
XPOS,YPOS)

Returns a position in world coordinates and a normalisation transformation NORMTR.

SET VIEWPORT INPUT PRIORITY

(TR1,TR2,HILO)

TR1 Viewport number

TR2 " "

HILO [HIGHER,LOWER]

Defines priority of the viewports to allow the return of a NORMTR when two viewports overlap.

REQUEST PICK(WS,DV,ST,SEG,PICKID)

SEG Segment number

PICKID Pick number

Returns a segment identification and the identification assigned for picks (allows to differentiate elements inside a segment).

SET PICK IDENTIFIER(N)

Defines the pick identifier for the subsequent output primitives.

SET DETECTABILITY(ID,DET)

DET [DETECTABLE,UN..]

Defines if a segment is detectable or not.

REQUEST CHOICE(WS,DV,ST,CH)

Returns an integer identifying the choice.

REQUEST VALUATOR(WS,DV,ST,VAL)

Returns a real number.

REQUEST STRING(WS,DV,ST,NCHARS,STR)

Returns a character string and the number of characters entered.

REQUEST STROKE(WS,DV,PTSMAX,ST,NT,
NPTS,X,Y)

Returns a stream of NPT positions with a maximum of PTSMAX with a normalization transformation NT.

The input of graphical devices can be done in three modes:

REQUEST The application program and input processor work alternatively.

SAMPLE Both are active together but the application program is the master.

EVENT Both are active but the input process is the dominant partner.

SET XXXX MODE(WS,DV,MODE,EC)

XXXX Logical device type

MODE [REQUEST,SAMPLE,EVENT]

EC [ECHO,NOECHO]

Sets the mode for a particular device.

SAMPLE YYYY(WS,DV,NT,,,)

Sample functions do not have a status.

AWAIT EVENT(TIMEOUT,WS,DVCLASS,DV)

If the device queue is empty, the application program is put on hold until an event occurs or the timeout is

elapsed. DVCLASS can return NONE or the device class.The device number is given by DV.

GET ZZZZ(output parameters)

Once an event has been recognised a GET command will be used to read the input.

ZZZZ any of the device types.

FLUSH DEVICE EVENTS(WS,DVCLASS,DV)

When called removes all inputs in the queue from the device specified.

Appendix IV

GKS Binding to VAL

The GKS commands described in the previous appendix have to be bound to a particular language by a set of routines. For most languages the binding is a set of routines, one for each GKS command. This is not the case for VAL because of the applicative nature of the language. All the GKS commands used to set attributes and transformations for the graphic systems are passed as parameters to the graphic functions in VAL. The following list provides a subset of a GKS binding for VAL. To simplify, the list parameters that have the same name also have the same type and function, therefore they are only defined once.

POLYLINE (window,view,transf,polyindex,coord,trigger)

In ARRAY[REAL] window	:window coordinates
In ARRAY[REAL] view	:viewport coordinates
In ARRAY[REAL] transf	:transformation matrix
In INTEGER polyindex	:color and line texture
In 2DLINE coord	:coordinates of line vertices and segment priority

In BOOLEAN trigger :general input to control function execution

Returns BOOLEAN :output required by VAL

TEXT (window,view,transf,attrib,x,y,string,trigger)

In ARRAY[INTEGER] attrib :array with text attributes:

1-character quality

2-text path

3-character height

4-alignment horizontal

5-alignment vertical

6-character vector x axis

7-character vector y axis

In REAL x,y :beginning of string

In CHARACTER-STRING string :string to be written

Returns BOOLEAN :output required by VAL

FILL AREA (window,view,transf,areaind,coord,trigger)

In INTEGER areaind :area fill pattern

Returns BOOLEAN :output required by VAL

ACCUMULATE (matrix1,matrix2)

In ARRAY[REAL] matrix1,matrix2 :combines the transformation of the two matrices

Returns ARRAY[REAL] :result matrix

REQUEST LOCATOR (ws,device)

In INTEGER ws	:workstation identification
In INTEGER device	:device identification
Returns INTEGER	:status of device
ARRAY[REAL]	:normalistion to be applied to coordinates
REAL	:X coord. of device position
REAL	:Y coord. of device position

REQUEST PICK (ws,device)

Returns INTEGER	:device status
INTEGER	:segment number
INTEGER	:pick number