New Jersey Institute of Technology
Digital Commons @ NJIT

Theses | Electronic Theses and Dissertations

9-30-1990

A transputer based distributed graphics display

Ramana V. Kattula
New Jersey Institute of Technology

Follow this and additional works at: https://digitalcommons.njit.edu/theses

Part of the Electrical and Electronics Commons

Recommended Citation

Kattula, Ramana V., "A transputer based distributed graphics display" (1990). *Theses*. 2781.
https://digitalcommons.njit.edu/theses/2781

This Thesis is brought to you for free and open access by the Electronic Theses and Dissertations at Digital Commons @ NJIT. It has been accepted for inclusion in Theses by an authorized administrator of Digital Commons @ NJIT. For more information, please contact digitalcommons@njit.edu.

Copyright Warning & Restrictions

The copyright law of the United States (Title 17, United States Code) governs the making of photocopies or other reproductions of copyrighted material.

Under certain conditions specified in the law, libraries and archives are authorized to furnish a photocopy or other reproduction. One of these specified conditions is that the photocopy or reproduction is not to be "used for any purpose other than private study, scholarship, or research." If a, user makes a request for, or later uses, a photocopy or reproduction for purposes in excess of "fair use" that user may be liable for copyright infringement,

This institution reserves the right to refuse to accept a copying order if, in its judgment, fulfillment of the order would involve violation of copyright law.

Please Note: The author retains the copyright while the New Jersey Institute of Technology reserves the right to distribute this thesis or dissertation

Printing note: If you do not wish to print this page, then select "Pages from: first page # to: last page #" on the print dialog screen

NJIT
New Jersey's Science & Technology University

The Van Houten library has removed some of the personal information and all signatures from the approval page and biographical sketches of theses and dissertations in order to protect the identity of NJIT graduates and faculty.

ABSTRACT

Title of Thesis: A Transputer Based Distributed Graphics Display

Ramana V. Kattula, Master of Science, 1990

Thesis directed by: Dr. John Carpinelli, Assistant Professor

The design of a transputer based graphics display is discussed. The problems associated with the single processor graphics systems are described. A solution based on multiple transputers is proposed. The transputer based graphics display described is a general purpose graphics system which can achieve any required compute performance and drawing performance without using special hardware. The display resolution and color depth can be enhanced as per the user requirements with little or no change in hardware.

A Transputer Based Distributed Graphics Display

by

Ramana V. Kattula

Thesis submitted to the Faculty of the Graduate School of the New Jersey Institute of Technology in partial fulfillment of the requirements for the degree of Master of Science in Electrical Engineering

1990

APPROVAL SHEET

Title of Thesis: A Transputer Based Distributed Graphics Display

Name of Candidate: Ramana V. Kattula
Master of Science in Electrical Engineering, 1990

Thesis and Abstract Approved: ______________________ 9/6/90

Dr. John Carpinelli — Date
Assistant Professor
Department of Electrical and Computer Engineering

______________________ 9/6/90

Dr. Anthony Robbi — Date
Associate Professor
Department of Electrical and Computer Engineering

______________________ 9-6-90

Dr. Edwin Hou — Date
Assistant Professor
Department of Electrical and Computer Engineering

VITA

Name: Ramana V. Kattula

Permanent address:

Degree and date to be conferred: Master of Science in Electrical Engineering, 1990.

Date of birth:

Place of birth:

Secondary education: Hyderabad Public School. Hyderabad. INDIA.

Collegiate institutions attended	Dates	Degree	Date of Degree
N. J. Institute of Technology	9/88-8/90	M.S.E.E.	October 1990
Osmania University	08/84-5/88	B.E.(E.C.E.)	May 1988

Major: Electrical Engineering.

Contents

List of Figures iv

List of Tables vi

1 Introduction 1

1.1 *OCCAM* 1
1.2 System design 2
1.2.1 Programming 3
1.2.2 Hardware 3
1.2.3 Programmable components 3
1.3 System architecture 4
1.3.1 Communication links 4
1.3.2 Local memory 4
1.4 Communication 5

2 Transputer architecture 8

2.1 Sequential processing 9
2.2 Instructions 10
2.2.1 Direct functions 11
2.2.2 Prefix function 11
2.2.3 Indirect functions 13
2.3 Support for concurrency 13
2.4 Communications 15
2.4.1 Internal channel communication 16
2.4.2 External channel communication 17
2.5 Timer 17
2.6 Alternative 18

2.7 Floating point instructions 19
2.8 Floating point unit design 19
2.9 Graphics capability 20

3 Hardware design considerations 21
3.1 Designing with the IMS T800 memory interface 21
3.1.1 Memory interface timing 22
3.1.2 Early and late write 25
3.1.3 Refresh 25
3.1.4 Wait states and extra cycles 26
3.1.5 Setting the memory interface configuration 27
3.2 Basic considerations in memory design 28
3.2.1 Minimum memory interface cycle time 28
3.3 Debugging memory systems 29
3.3.1 Peeking and poking 29
3.4 Connecting INMOS links 30
3.4.1 Introduction 30
3.4.2 Link operation 31
3.4.3 Electrical Considerations 32

4 Distributed graphics display 34
4.1 System performance 34
4.2 Parallel graphics system 35
4.2.1 Introduction 35
4.2.2 Transputer modules (TRAMs) 37
4.2.3 Graphics TRAMs 38
4.3 Serial port TRAM 39
4.3.1 Introduction 39
4.3.2 Random access port 45
4.3.3 Serial access port 49
4.4 Display TRAM 53
4.4.1 Introduction 53
4.5 System configuration 59
4.5.1 Driving the frame store 59
4.5.2 Frame store configurations 59

5 Conclusions 62

Bibliography 64

List of Figures

1.1 A node of four transputers 2
1.2 Links communicating between processes 5
1.3 Link protocol 6

2.1 Transputer interconnections 9
2.2 Registers 10
2.3 Instruction operand register 12
2.4 Linked process list 14
2.5 Floating point unit block diagram 20

3.1 Memory interface 22
3.2 T800 memory map 23
3.3 The configurable strobes 25
3.4 Link connection 32

4.1 Spatial distribution 36
4.2 Chronological distribution 36
4.3 Objective distribution 36
4.4 Characteristic distribution 37
4.5 Connecting graphic TRAMs 40
4.6 Serial port TRAM block diagram 41
4.7 Memory map 42
4.8 Multiplex arrangements with dynamic RAMs 47
4.9 Serial interface block diagram 49
4.10 Address generation scheme 51
4.11 Display TRAMs 54
4.12 Pixel channels 56
4.13 8 bit mode 57

4.14 24 bit mode 58
4.15 High resolution 24 bit mode 58
4.16 High resolution 24 bit display 61

List of Tables

3.1 Tstates of memory cycle 23
3.2 Parameters for typical Dynamic RAMs 29

Chapter 1

Introduction

A transputer is a microcomputer with its own local memory and with links for connecting one transputer to another. A typical member of the transputer family is a single chip containing a processor, memory and communication links which provide point to point connections between transputers [5]. In addition, each transputer product contains special circuitry and interface adapting it to a particular use.

A transputer can be used in a single processsor system or in networks. Transputer networks are high performance concurrent systems and can be easily constructed using point-to-point communication.

1.1 *OCCAM*

Transputers can be programmed using almost all high level languages used today and are designed so that the compiled programs will be efficient. Where it is required to exploit concurrency, but still to use standard languages, *occam* can be used as a harness to link modules written in selected languages. To

gain the most benefit from the transputer architecture, the whole system can be programmed in *occam* [5].

1.2 System design

The transputer architecture simplifies system design by using processes as building blocks. Figure 1.1 shows the interconnection of four transputers forming a node.

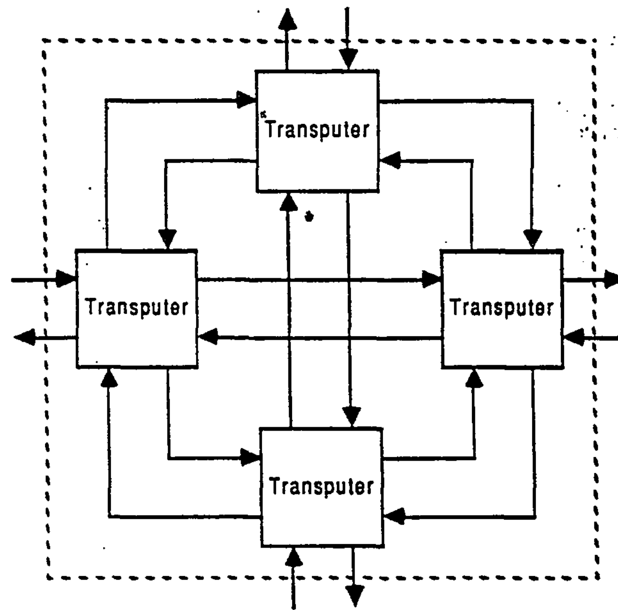

Figure 1.1: A node of four transputers

1.2.1 Programming

The software building block is the process. A system is designed by interconnecting a set of processes. Each process can be considered an independent unit by itself. It communicates with other processes through point-to-point channels. A process is completely characterized by the messages it sends and receives. There is no need for a synchronization mechanism for communication between processes since it is synchronized internally. The design of the system is heirarchical and at any level of design we are concerned with only a small set of processes.

1.2.2 Hardware

The individual software building blocks are each implemented in hardware. The hardware process is a transputer executing an occam program. Each hardware process can be easily designed and compiled. Its internal structure is hidden and it communicates with other processes through its links.

1.2.3 Programmable components

A transputer can be programmed using occam to perform a specific function. Once it is programmed it can be considered a black box. For improving the performance, some processes can be hard-wired. A system can be designed using a combination of software processes, programmed transputers and hardware processes. That system can again be a part of a larger system.

1.3 System architecture

1.3.1 Communication links

Transputers use point-to-point communication links. All transputers have one or more standard links which can be connected to links of other transputers. This gives the advantage of making networks of any size and structure possible.

The advantages of point-to-point communication links over multiprocessor buses are:

- The communication capability is not limited by the number of transputers in the system.
- There is no capacitive load penalty as transputers are added to a system.
- The communication bandwidth is not saturated by an increase in system size. The greater the number of transputers in the system, the higher is the total communication bandwidth, since regardless of the system size the connections are short and local.

1.3.2 Local memory

Each transputer has its own local memory which is used by the transputer for the process it executes. When using a number of conventional processors to form a network the total memory bandwidth is limited, but the memory bandwidth in a transputer system is proportional to the number of transputers in the system. This means that the memory interfaces are not shared and are

not linked with the communications interface; this enhances the speed of access and provides high bandwidth with a minimum of external components.

1.4 Communication

For the communication to be synchronized, each message must be acknowledged. So a link should consist of one unidirectional signal wire for each direction of communication. Figure 1.2 shows links communicating between processes of two transputers.

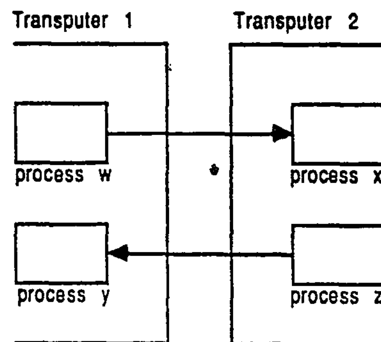

Figure 1.2: Links communicating between processes

A link between two transputers is implemented by connecting a link interface on one transputer to a link interface on the other by two one-directional lines which carry data serially. The two signal wires of the link can be used to provide two occam channels, one in each direction. This is accomplished by a simple protocol. Each line carries data and also control information. The protocol provides the synchronized communication of occam. The use of the protocol providing for the transmission of an arbitrary sequence of bytes allows

transputers of different wordlengths to be connected. Figure 1.3 below shows a link protocol [4].

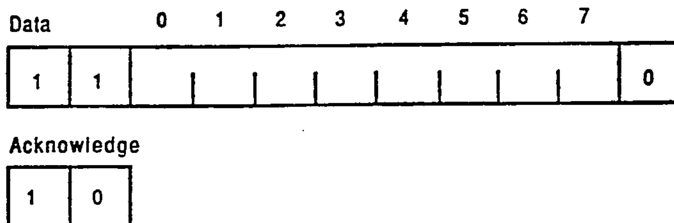

Figure 1.3: Link protocol

The messages between transputers are transmitted as a sequence of single byte communications which requires just a single byte buffer in the receiving transputer to ensure that no information is lost. Each message is transferred as a start bit followed by a one bit followed by eight data bits and then a stop bit. Once the message is transmitted, the sender waits for an acknowledge from the receiving transputer. This consists of a start bit followed by a zero bit. The acknowledge implies both that a message byte was received and the receiving link is ready to receive another byte. The sending link schedules the next transmission only after it receives an acknowledge for the previous message.

The data bytes and acknowledges are multiplexed on each signal line. If there is room to buffer more than one message, the acknowledge can be transmitted as soon as the data byte reception starts. As a consequence, transmis-

sion can be continuous, with no delays between data bytes.

The links make the design of systems simple. Board layout of two wire connections is easy and is area efficient. All transputers support a standard communications frequency of 10 Mbits/sec, regardless of their performance. Hence it is possible to directly connect processors of different performance.

Communication between links is not sensitive to clock phase. So systems clocked independently can still communicate as long as the frequency of the communications is the same. The transputer family includes a number of link adapter devices to connect links to non–transputer devices.

Chapter 2

Transputer architecture

A transputer consists of a processor, memory and communications system connected by a 32-bit bus. The bus is also connected to the external memory interface, so that additional local memory can be used. The floating point transputers also have a on-chip floating point unit. The block diagram of figure 2.1 indicates how the major blocks of the transputer are interconnected.

The CPU in the transputers contains three registers, A, B and C, used for integer and address arithmetic; they form a hardware stack. When a value is loaded into the stack, B is pushed into C and A into B, before loading the new value into A. Retrieving a value from A pops B into A and C into B. The Floating Point Unit (FPU), similarly, has three registers to evaluate arithmetic operations, called AF, BF and CF. When values are loaded or retrieved, they push and pop same way as the A, B and C registers.

The address of floating point values are formed on the CPU stack, and the CPU controls the transfer of the values between address memory locations and the FPU stack. The word length of the CPU is independent of that of

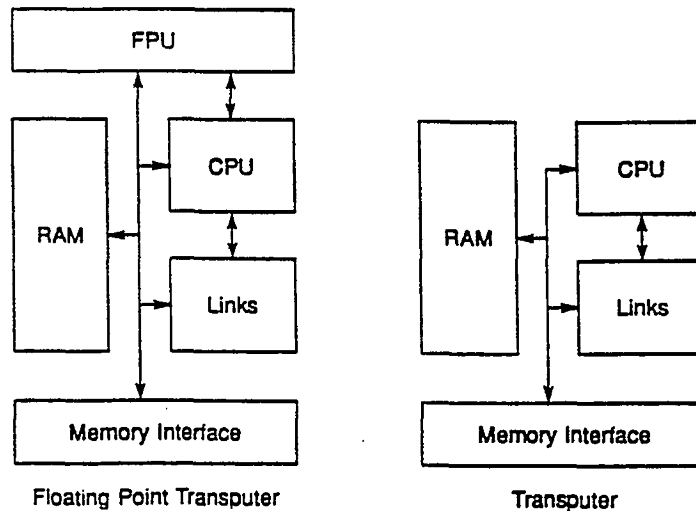

Figure 2.1: Transputer interconnections

the FPU, as the CPU stack is used only to hold the addresses of floating point values. Hence it is possible to use the same FPU together with a 16-bit CPU.

2.1 Sequential processing

The availability of fast on-chip memory makes having large numbers of registers for the processor unnecessary. The CPU has six 32-bit registers for the execution of a sequential process. The fewer registers with a simple instruction set enable the processor to have relatively simple data-paths and control logic. The six registers are:

- The workspace pointer which points to an area of storage where local variables are kept.

- The instruction pointer which points to the next instruction to be executed.

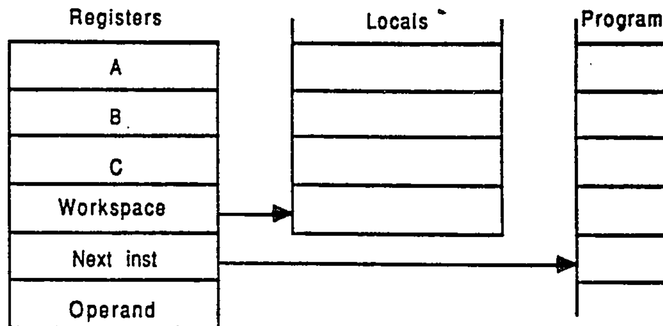

Figure 2.2: Registers

- The operand register which is used in the formation of instruction operands.
- The A, B and C registers which form an evaluation stack, and are the sources and destinations for most arithmetic and logical operations.

A schematic diagram of the registers is shown in figure 2.2. Expressions are evaluated on the evaluation stack, and instructions refer to the stack implicity. As an example, the *add* instruction adds the two top values in the stack and places the result on the top of the stack. The use of a stack removes the need for instructions to respecify the location of their operands. The hardware has no protection to prevent more than three values being stored in the stack.

2.2 Instructions

The instruction set is designed for simple and efficient compilation. It contains a small number of instructions, all with the same format. The microcode can be used for transputers of different word lengths, as the instruction set is independent of the processor word length. Each instruction consists of a single byte divided into two 4-bit parts. The four most significant bits of the byte

are a function code, and the four least significant bits are a data value.

2.2.1 Direct functions

The representation of instructions gives sixteen possible combinations for functions with sixteen data values (0 to 15) for each function. These functions include:

load constant	*add constant*	
load local	*store local*	*load local pointer*
load non-local	*store non-local*	*call*

The most common operations in a program are the loading of small literal values, and the loading and storing of one of a small number of variables. The *load constant* instruction enables values between 0 and 15 to be loaded with a single byte instruction. The *load local* and *store local* instructions access loactions in memory relative to the workspace pointer. The first sixteen locations can be accessed using a single byte instruction.

The *load non-local* and *store non-local* instructions behave similarly, except that they access locations in memory relative to the A register. Compact sequences of these instructions allow efficient access to data structures and provide for simple implementations of the static links or displays used in the implementation of block structured programming languages such as *occam.*

2.2.2 Prefix function

Two other instructions of the total sixteen function codes allow the operand of any instruction to be extended in length. They are:

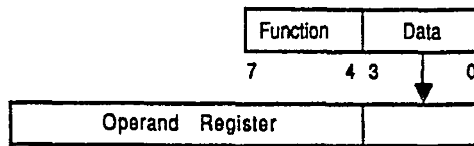

Figure 2.3: Instruction operand register

Prefix *negative prefix*

To execute any instruction the four data bits are loaded into the four least significant bits of the 32-bit operand register. The execution of an instruction (other than prefix instruction) ends by clearing the operand register, which makes it ready for the next instruction.

The prefix instruction loads its four data bits into the operand register, and then shifts the operand register up four places. The *negative prefix* instruction is similar, except that it compliments the operand register before shifting it up. Consequently operands can be extended to any length up to the length of the operand register by a sequence of prefix instructions. In particular, operands in the range -256 to 255 can be represented using one prefix instruction. Figure 2.3 shows the instruction operand register.

The use of prefix instructions has certain beneficial consequences. First, they are decoded and executed in the same way as every other instruction, which simplifies and speeds instruction decoding. Second, they simplify language compilation by providing a completely uniform way of allowing any instruction to take an operand of any size. Third, they allow operands to be represented in a form independent of the processor word length.

2.2.3 Indirect functions

The last function code, *operate*, causes its operand to be interpreted as an operation on the values held in the evaluation stack. This enables the programmer to encode 16 such operations in a single byte instruction. Just like that of any other operand, the *operate* instruction's operand can be extended using the *prefix* instruction. Hence, there are an infinite number of operations possible by the instruction representation.

The encoding of the indirect functions is chosen so that the most frequently occurring operations are represented without the use of a prefix instruction. These include arithmetic, logical and comparision operations such as:

add *exclusive or* *greater than*

Less frequently occuring operations have encodings which require a single prefix operation. The 32-bit transputer IMS T800 has additional instructions which load into, operate on, and store from the floating point register stack. It also contains new instructions which support color graphics, pattern recognition and the implementation of error correcting codes.

2.3 Support for concurrency

The processor in the transputer provides efficient support for the *occam* model of concurrency and communication. It has a microcoded scheduler which enables any number of concurrent processes to be executed together, sharing the processor time. This removes the need for a software kernel. The proces-

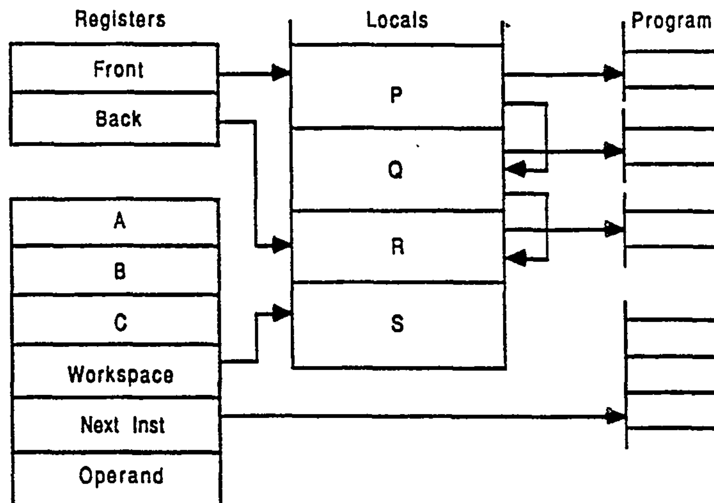

Figure 2.4: Linked process list

sor does not need to support the dynamic allocation of storage as the *occam* compiler is able to perform the allocation of space to concurrent processes.

At any time, a concurrent process may be

active	-	*being executed*
	-	*on a list waiting to be executed*
inactive	-	*ready to input*
	-	*ready to output*
	-	*waiting until a specified time*

The scheduler operates in such a way that inactive processes do not consume any processor time. The active processes waiting to be executed are held on a list. This is a linked list of process workspaces, implemented using two registers, one of which points to the first process on the list, the other to the last. In figure 2.4, S is executing, and P, Q and R are active, awaiting execution.

A process is executed until it has to wait for an input or output, or has to

wait for the timer. When a process does stop for such a reason, its instruction pointer is saved in its workspace and the next process is taken from the list. The process switch times are very small, since little state information needs to be saved.

The processor provides a number of special operations to support the process model. These include

start process *end process*

When a parallel construct is executed, *start process* instructions are used to create the necessary concurrent processes. A *start process* instruction creates a new process by adding a new workspace to the end of the scheduling list, enabling the new concurrent process to be executed.

The correct termination of a parallel construct is assured by use of the *end process* instruction. This uses a workspace location as a counter of the components of the parallel construct which have yet to terminate. The counter is initialized to the number of components before the processes are started. Each component ends with an *end process* instruction which decrements and tests the counter. For all but the last component, the counter is non zero and the component is descheduled. For the last component, the counter is zero and the component continues.

2.4 Communications

Processes communicate through channels. As noted earlier, *occam* communication is point-to-point, synchronized and unbuffered. So a channel does not

need a process queue, message queue or message buffer. If the two communicating processes are on the same transputer, the communication is implemented by a single word in memory. If they are on two different transputers, the communication is implemented by point-to-point links. The processor provides a number of operations to support message passing. The most important are:

input message *output message*

These messages determine from the address of the channel whether the channel is internal or external. This eliminates the need for two different instructions for hard and soft channels and allows a process to be written and compiled without the knowledge of where its channels are connected. Communication can take place only when both sending and receiving processes are ready. So the process which first becomes ready must wait until the other is also ready.

A process performs an input or output by loading the evaluation stack with a pointer to a message, the address of a channel, and a count of the number of bytes to be transferred, and then executing an *input message* or an *output message* instruction.

2.4.1 Internal channel communication

At any given time, an internal channel either holds the identity of a process, or holds the special value *empty* [5]. The channel is initialized to empty before it is used. When a message is passed using the channel, the identity of the first process to become ready is stored in the channel, and the processor starts

to execute the next process from the scheduling list. When the second process to use the channel becomes ready, the message is copied, the waiting process is added to the scheduling list, and the channel is reset to its initial state. It does not matter whether the inputting or outputting process becomes ready first.

2.4.2 External channel communication

When a message has to be passed through an external channnel, the job is assigned to an autonomous link interface by the processor and the process is descheduled. When the transfer is complete the process is rescheduled by the processor. This way the processor can continue with other processes while messages are being transferred.

A link interface uses three registers:

- a pointer to a process workspace
- a pointer to a message
- a count of bytes in the message

2.5 Timer

The clock contained in the transputer has a period of one microsecond. The current value of the processor timer can be read by executing a *read timer* instruction. A process can be programmed to execute after a specified time has been reached; this is accomplished by the *timer input* instruction. A time

has to be specified for the instruction. If the current time is already past the specified time, then the instruction has no effect. If the time is in the future then the process is descheduled until the specified time is reached and then rescheduled.

2.6 Alternative

The *occam* alternative construct enables a process to wait for input from any one of a number of channels, or until a specified time occurs. This requires special instructions, as the normal *input* instruction deschedules a process until a specified channel becomes ready, or until a specified time is reached. The instructions are:

enable channel *disable channel*
enable timer *disable timer*
alternative wait

The alternative is implemented by 'enabling' the channel input or timer input specified in each of its components. The 'alternative wait' is then used to deschedule the process if none of the channel or timer inputs are ready. The process will be rescheduled when any one of them becomes ready. The channel and timer inputs are then 'disabled'. The 'disable' instructions are also designed to select the component of the alternative to be executed; the first component found to be ready is executed.

2.7 Floating point instructions

The core of the floating point instruction set includes simple load, store and arithmetic instructions. The transfer of operands between the transputer's memory and the floating point evaluation stack is done by means of floating point load and store instructions. There are two groups of such instructions, one each for single length and double length operands. We discuss only the double length instructions here, but there exist corresponding single length instructions.

The CPU computes the address of the floating point operand on its stack and the operand is loaded onto the FPU's stack from the addressed memory location. Operands in the FPU's stack are tagged with their length. This tag will be set when the operand is loaded or computed. The tags enable us to reduce the number of instructions required. So a *floating add* will do the work of both the *floating add single* and *floating add double* instructions. The FPU and CPU operate concurrently in a floating point transputer, so the CPU can perform an address calculation while the FPU performs a floating point calculation. This improves the performance in real time applications which access arrays heavily.

2.8 Floating point unit design

The floating point unit contains a fast normalizing shifter. Multiplication is performed three-bits per cycle, and division is performed two-bits per cycle.

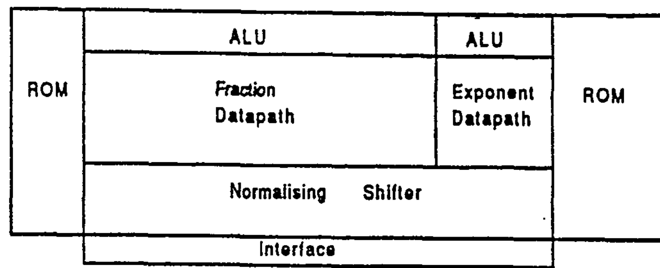

Figure 2.5: Floating point unit block diagram

Figure 2.5 shows the block diagram of the FPU.

The data paths contain registers and shift paths. The fraction data path is 59 bits wide, and the exponent data path is 13 bits wide. The normalizing shifter interfaces to both the fraction data path and the exponent data path. This is because the data to be shifted will come from the fraction data path while the magnitude of the shift is associated with the exponent data path.

2.9 Graphics capability

The fast block move instructions of the *occam* make transputers efficient in graphics applications using byte-per-pixel color displays. The block move on the transputer is designed to saturate the memory bandwidth, moving any number of bytes from any byte boundary in memory to any other byte boundary using the smallest possible number of word read and write operations. The instructions for the graphics achieve the speed of the simple move instruction, enabling a 1 million pixel screen to be drawn many times per second.

Chapter 3

Hardware design considerations

3.1 Designing with the IMS T800 memory interface

The IMS T800 has a configurable memory interface and allows easy interfacing of a variety of memory types with a minimum of extra components. The interface directly supports DRAMs, SRAMs, ROMs and memory mapped peripherals [9].

The transputer has a 32-bit multiplexed bus for data and address with a linear address space of 4 Gbytes. The interface connections include four byte write strobes, a read strobe, a refresh strobe, five configurable strobes, a wait input, a memory configuration input, a bus request input and a bus grant output. Figure 3.1 shows the inputs and outputs associated with the memory interface. With this flexible arrangement, a variety of memory timing controls can be obtained with little external hardware.

All RAM appears to the IMS T800 as 2^{32} bytes mapped as 32 bit words

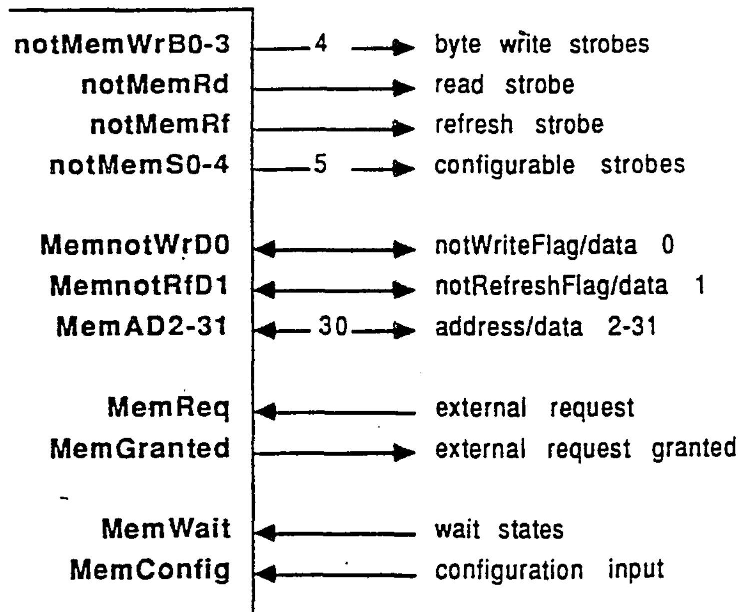

Figure 3.1: Memory interface

in a linear signed address space. Addresses run from $80000000 through $FFFFFFFF to $7FFFFFFF. But the *occam* map starts at $0 and is organized as words. The T800 has MemStart at $80000070 and the start of external memory at $80001000 [1].

As shown in Figure 3.2, the T800 has 4Kbytes of on-chip RAM at addresses $80000000 to $80000FFF. Therefore, the memory space from $80000000 to $FFFFFFFF can be used for RAM and $00000000 to $7FFFFFFF for ROM and I/O.

3.1.1 Memory interface timing

The memory interface cycle has six timing states, called **Tstates**. The Tstates have the functions described in table 3.1. The duration of each state can be

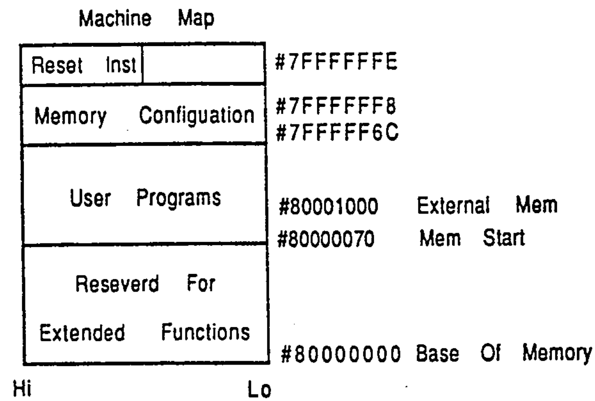

Figure 3.2: T800 memory map

Tstate	Description
T1	address setup time before address valid strobe
T2	address hold time after address valid strobe
T3	read cycle tristate/write cycle data setup
T4	extended for wait states
T5	read or write data
T6	end tristate/data hold

Table 3.1: Tstates of memory cycle

configured according to the memory devices used and can be from one to four **Tm** periods. One Tm period is half the processor cycle time, i.e., half the period of **ProcClockOut**. T4 can be extended by adding wait states in the form of additional Tms. A0 and A1 are not output with the rest of the address. During a write cycle, byte and half-word (16 bit data) addressing are achieved by the four write byte strobes (**notMemWrB**); only the write byte strobes corresponding to the bytes to be written are active. During a read cycle, this is achieved by internally selecting the bytes to be read.

Thus the two lowest order address lines are not needed. The two lowest order data lines are not multiplexed with address lines but, during the address period, are used to give early indication of the type of cycle which will follow:

MemnotWrD0 is low during **T1**, and **T2** of a write cycle.

MemnotRfD1 is low during **T1** and **T2** of a refresh cycle.

The use of the strobes notMemS0 to notMemS4 depend upon the memory system. The rising edge of notMemS1 and the falling edges of notMemS2 to notMemS4 can be configured to occur from 1 to 31 Tm periods after the start of T2. This is summarized in figure 3.3.

It should be noted that the use of wait states can advance the rising edge of notMemS1 in relation to that of other strobes; care must be taken if this signal is being used for RAS (Row Address Strobe input on the dynamic RAMs) driving DRAMs for which RAS must not be removed before CAS (Column Address Strobe input on the dynamic RAMs).

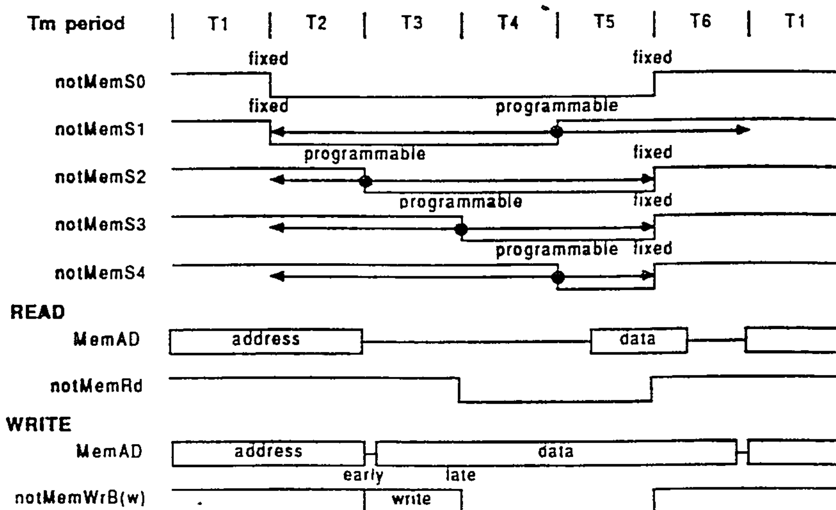

Figure 3.3: The configurable strobes

3.1.2 Early and late write

The notMemWrB strobes can be configured to fall either at the beginning of T3 (early write) or at the beginning of T6. Early write gives a longer set up time for the write strobe but data is only valid on the rising edge of the pulse. For late write, data is also valid on the falling edge of the strobe but the pulse is shorter.

3.1.3 Refresh

The T800 has an on-chip refresh controller and 10 bit refresh address counter and can, therefore, refresh DRAMs of up to 4Mbit capacity without requiring the counter to be extended externally.

Refresh can be configured to be either enabled or disabled. If enabled, the

refresh interval can be configured to be 18, 36, 54 or 72 **ClockIn** periods; if a refresh cycle is due, the current memory cycle is always completed first. The time between refresh cycles is thus almost independent of the transputer speed and the length of memory cycles.

Refresh cycles are flagged by **notMemRf** going low before T1 and remaining low until the end of T6. Refresh is also indicated by **MemnotRfD1** going low during T1 and T2 with the same timing as address signals. The address output during refresh is:

AD0	= MemnotWrD0	high
AD1	= MemnotRfD1	low, to indicate refresh
AD2 - AD11		refresh address
AD12 - AD30		high
AD31		low

During the refresh cycles, the strobes **notMemS0 - notMemS4** are generated as normal.

3.1.4 Wait states and extra cycles

Memory cycles can be extended by wait states. **MemWait** is sampled close to the failing edge of **ProcClok-Out**, prior to, but not at, the end of T4. If it is high, T4 is extended by additional **Tms**. Wait states are added for as long as **MemWait** is held high, T5 proceeds when **MemWait** is low. Note that the internal logic of the memory interface ensures that, if wait states are inserted, T5 always begins on a rising edge of **ProcClockOut**: so the number of wait states inserted will be either always odd or always even, depending on

the memory configuration being used.

Every memory interface cycle must consist of a number of complete cycles of **ProClockOut**: i.e. it must consist of an even number of Tms. If there are an odd number of Tm periods up to and including T6, an extra Tm will be inserted after T6.

3.1.5 Setting the memory interface configuration

A memory interface configuration is specified by a 36 bit word and is fixed at reset time. The T800 has a selection of 13 pre-programmed configurations. If none of these is suitable, a different configuration can be selected by supplying the complement of the configuration word to the T800s **MemConfig** input immediately following reset [9].

A pre-programmed configuration is selected by connecting **MemConfig** to **MemnotWrD0**, **MemnotRfD1**, **MemAD2-MemAD11** or **MemAD31**. Immediately after reset, the T800 takes all of the data lines high and then, beginning with **MemnotWrD0**, they are taken low in sequence. If **MemConfig** goes low when the T800 pulls a particular data line low, the memory interface configuration associated with the data line is used. If, during the scan, **MemConfig** is held low until **MemnotWrD0** goes low, or is connected to **MemAD31**, the slowest memory configuration is used.

After scanning the data lines as described above, the T800 performs 36 read cycles from locations $7FFFFF6C, $7FFFFF70 – $7FFFFFF8. No data is latched off the data bus but, if **MemConfig** is held low until **MemWrD0** is

taken low, each read cycle latches one bit of the (inverted) configuration word on **Memconfig**. Thus, a memory configuration can be supplied by external logic.

Using a pre-programmed configuration has the advantage of requiring no external components: only a connection from **MemConfig** to the appropriate data line. However, selecting an external configuration can also be very economical in component use. If the transputer is booting from ROM, the ROM must occupy the top of the address space. One bit of the memory configuration word can be stored in each of the 36 addresses mentioned above and the only additional hardware required is an inverter connecting the appropriate data line (usually **MemnotWrD0**) to **MemConfig**. **Memconfig** is thus held low until **MemnotWrD0** goes low and is fed with the inverse of the configuration word during the 36 read cycles. Alternatively, the inverted configuration word can be generated from A2-A7 by one sum term of a PAL (Programmable Array Logic).

3.2 Basic considerations in memory design

3.2.1 Minimum memory interface cycle time

The minimum number of processor clock cycles for an external memory access is 3, which occurs when all Tstates are 1 Tm. With a 50 nsec cycle time, this will be 150 nsec.

The most important DRAM parameters to be considered at the start of a memory design are the access and cycle times and the RAS precharge time.

	NEC uPD41256	NEC uP41256-12	Hitachi HM51256-10
Access time	150ns	120ns	100ns
cycle time	260ns	220ns	180ns
RAS precharge	100ns	90ns	70ns

	NMB AAA2800-150	AAA2800-80
Access time	150ns	80ns
cycle time	246ns	151ns
RAS precharge	90ns	65ns

Table 3.2: Parameters for typical Dynamic RAMs

These will be a guide to the fastest timing possible, which is generally a good starting point. Typical values of these times are shown in Table 3.2.

Higher density devices require longer RAS precharge times but, if the memory does not require RAS to remain low until the end of the memory cycle, it can be taken high before the cycle ends.

3.3 Debugging memory systems

3.3.1 Peeking and poking

Transputers can be booted from ROM (**BootFromROM** to Vcc) or from link (**BootFromROM** to ground). When booting from a link, a header byte is expected; if it is in the range 2-255 it should be followed by that number of bytes. These will be placed in memory starting at **MemStart** ($80000070)

and execution will then be transferred to this address. The code executes at low priority and its work space is located immediately above itself. Usually, this code will be a loader, to load the user's program into this transputer and any others, if it is a part of a network.

If the header byte is 0, a 'poke' operation will take place. The 0 byte should be followed by a 4 byte address (AAAA) and 4 bytes of data (DDDD) to be placed at that address:

input: header=0, then A A A A D D D D

If the header byte is a 1, a 'peek' operation will take place. The 1 byte should be followed by a 4 byte address (AAAA). The transputer will then output, on the same link, 4 bytes of data (DDDD) read from that address:

input: header=1, then A A A A

output: D D D D

After both the peek and poke operations, the transputer reverts to awaiting a new header (which could initiate another peek or poke). So, if we have another transputer, it is posssible to test the hardware by poking to the transputer under test to place data in the internal or external memory, and then peeking to read the data back and compare it.

3.4 Connecting INMOS links

3.4.1 Introduction

The INMOS link is fundamental to the concept of the transputer and of *occam*. A link is the hardware implementation of an occam channel; each bidirectional

link provides a pair of occam channels, one in each direction. A link provides serial data communication between two transputer family devices at speeds up to 20Mbits/s [4].

A link between two transputers is implemented by connecting a link interface on one transputer product to a link interface on the other transputer product by two unidirectional signal lines. Each signal line carries data and control information.

Communication through a link involves a simple protocol. This provides the synchronized communication of occam. The use of a protocol providing for the transmission of an arbitrary sequence of bytes allows transputer products of different word length to be connected together.

Electrically, link signals are TTL compatible and as such are a simple means of communication over short distances (< 0.3 meters). Links are designed for local communication. However, it is possible to use them over longer distances although a little more consideraation is needed to ensure reliable operation.

3.4.2 Link operation

An INMOS link between two transputer products consists of two unidirectional signal lines connected to the link interface on each transputer family device, providing point-to-point serial communication, as shown in figure 3.4.

Communication across a link involves a simple protocol. Each message is transmitted as a sequence of single byte communications, requiring only the presence of a single byte buffer in the receiving transputer to ensure that no

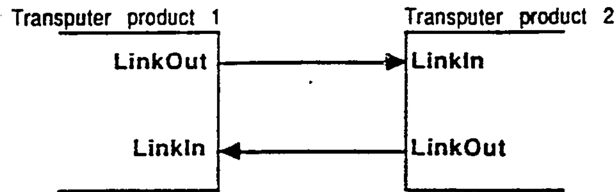

Figure 3.4: Link connection

information is lost.

After transmitting a data byte, the sender waits until an acknowledge is received. This consists of a start bit folowed by a zero bit. The acknowledge signifies both that a process was able to receive the acknowledged byte, and that the receiving link can receive another byte.

Data bytes and acknowledges may be multiplexed down each signal line during duplex communication. In one implementation of the link acknowledges are output on receipt of the full eleven bits of the data packet. The link implementation of the T800 allows ovelapped acknowledges. In this implementation, the acknowledge may be sent immediately on receipt of the start bit and the 'data is to follow' bit, allowing continuous data transmission with no delays between data packets.

3.4.3 Electrical Considerations

Links may be connected very simply over short distances. No engineering is required other than a direct wire connection between LinkOut of one transputer and LinkIn of another. The connection may consist of tracks on a printed circuit board, or a cable.

Over greater distances, certain parameters of the interconnection medium must be taken into account:

- Transmission line effects
- Noise and crosstalk
- Line attenuation
- Pulse dispersion
- Skew
- Propagation delay

A further consideration that applies to all link connections is protection of the link interface from electrostatic discharge.

INMOS links are designed to transmit serial data between transputer family devices at speeds up to 20Mbits/s. The signals are TTL compatible and as such are suitable for transmitting data over short distances (up to 30cm) with no engineering except a simple wire connection. Though transmmission line effects come into play at longer distances, we shall not be interested in them.

Chapter 4

Distributed graphics display

4.1 System performance

In many graphics systems, the system performance is reduced by its design aspects. To overcome these performance problems, many systems use custom built hardware, thereby increasing the cost and reducing flexibility.

The solution to these problems calls for a general purpose graphics system with several requirements, as described below:

- **Compute performance:** Any required compute performance desired by the user for his application.

- **Drawing performance:** Any required drawing performance desired by the user for a given application.

- **Display access:** The display scanning must have separate access to the frame store to remove the conflict between the processor and the display scanning hardware.

- **Display resolution and color depth:** Any required display resolution and color depth.
- **Display drivers:** Any required speed of display output. For instance, very high speed device technology may be necessary for a very high resolution display.

4.2 Parallel graphics system

4.2.1 Introduction

Parallel graphics system addresses the problems discussed in the above section. To provide any desired performance requires that the processing task be divided into smaller subtasks and as many processors that are necessary to provide the appropriate performance be used. This allows a system to be built to achieve any drawing bandwidth, with any compute performance. So the problem now is that of distribution and its implementation.

The following are a few methods for distributing processing tasks.

Spatial: The display is broken up into a number of tiles. Each tile is distributed to a different processor or a group of processors. Figure 4.1 illustrates the spatial distribution.

Chronological: This method distributes the entire display to all processors in the system, but only one will display all its data at any one time. Each frame of the display is produced by a processor or a group of processors. Figure 4.2 shows how this distribution is done.

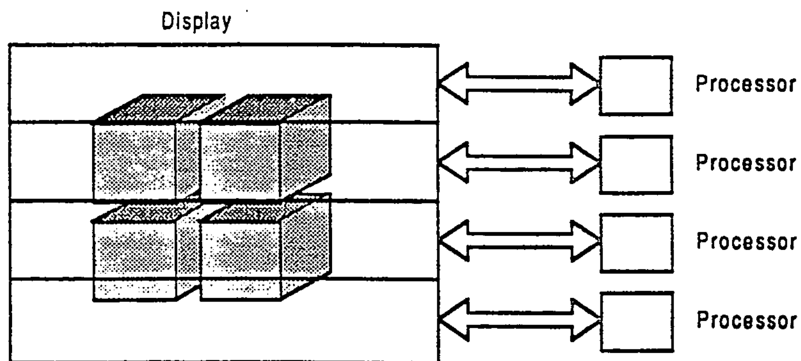

Figure 4.1: Spatial distribution

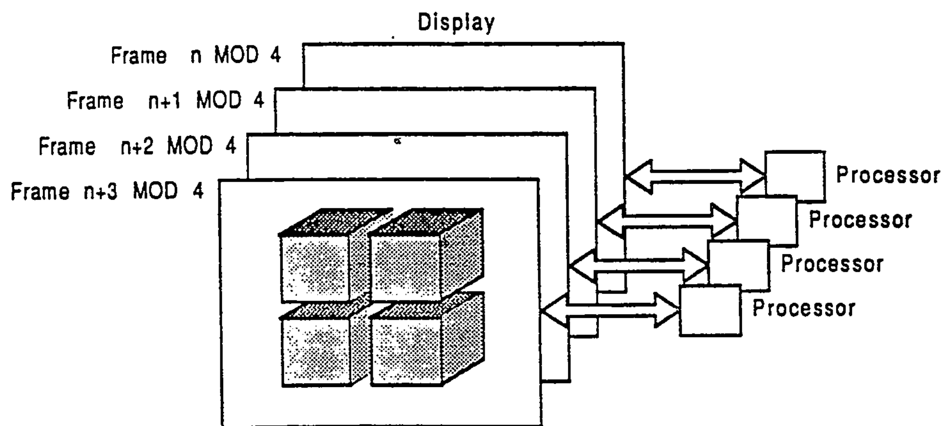

Figure 4.2: Chronological distribution

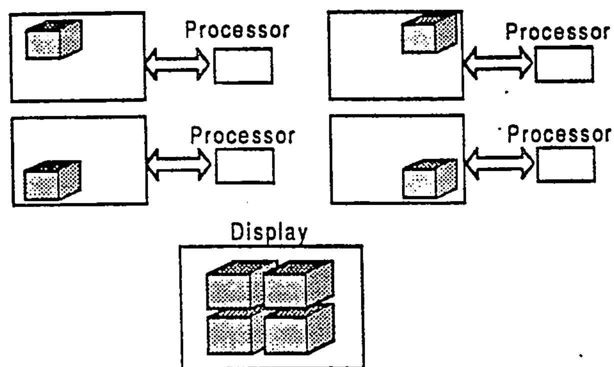

Figure 4.3: Objective distribution

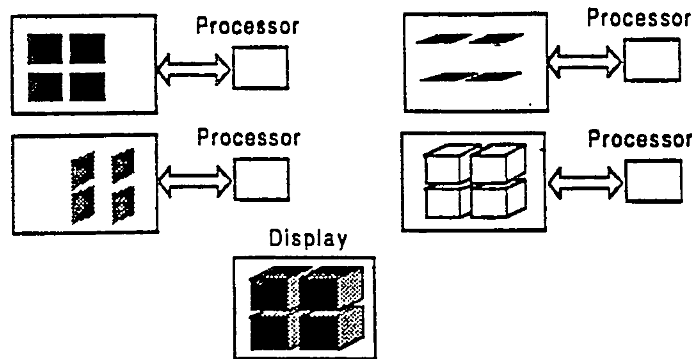

Figure 4.4: Characteristic distribution

Objective: This method distributes different objects in a scene to different processors. This is illustrated in figure 4.3.

Characteristic: This method distributes characteristics of the scene, such as color, to different processors. This is illustrated in figure 4.4.

4.2.2 Transputer modules (TRAMs)

Transputer Modules or **TRAMs** are subassemblies of transputers, a few discrete components, and sometimes some RAM and/or application specific circuitry. All TRAMs:

- Have a standard interface using serial links
- Have a standard pinout
- Have standard sizes
- Are designed to a published specification

These TRAM standards make it very simple for users to build customized TRAMs or motherboards with sockets for TRAMs. The TRAM pinout stan-

dard is independent of:

- Transputer type
- Number of transputers
- Word length of transputer
- Speed of transputer
- Function of the module
- Memory size
- Package
- Implementation (PCB, hybrid, silicon, etc.)

4.2.3 Graphics TRAMs

If the graphical display processors are implemented as modular transputer compute elements, each with transputer, memory and logic to implement special functions, the problem of designing a distributed graphics system becomes much simpler.

To provide the distributed frame store requirements and any display output type, two different TRAMs are necessary.

Serial port TRAM: This contains an IMS T800 and all the logic necessary for a complete frame store. It can be connected to other identical TRAMs so that distribution of the frame store becomes a matter of simple replication

of this TRAM. This is known as the serial port TRAM because of the serial nature of the output data.

Display backend driver TRAM: This contains all the logic necessary to drive a particular display type. This TRAM interfaces directly to, and receives its high speed data from, the serial port TRAM. This TRAM will be known as the **Display Backend** TRAM.

Separation of frame store scanning from the processor address and data bus is achieved on the serial port TRAM using video RAMs. Video RAMs have a separate **serial port** for video data. This allows the frame buffer to be scanned in a serial fashion without causing significant loss of processor access to the RAM, relieving the arbitration problems associated with conventional RAMs.

The serial port TRAM supplies a continuous stream of high speed serial data from the frame store. The **Display Backend** can drive display monitors using this stream of data in a variety of display modes. These TRAMs are connected together by a 60 way ribbon cable, which contains a control bus and a distributed data bus. All serial port TRAMs in the system connect in parallel to this cable. This is illustrated in figure 4.5.

4.3 Serial port TRAM

4.3.1 Introduction

The serial port TRAM consists of:

A transputer: An IMS T800, which maintains the frame store.

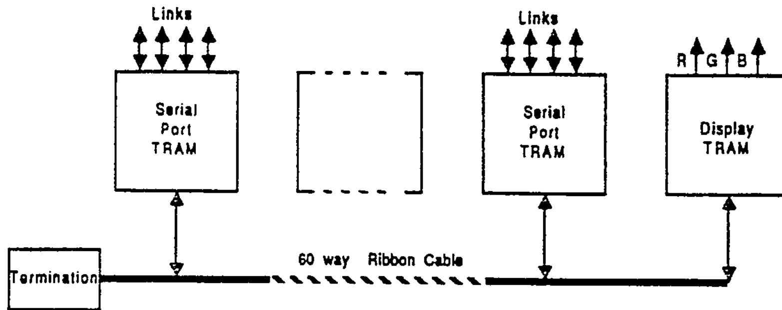

Figure 4.5: Connecting graphic TRAMs

Memory: The standard serial port TRAM contains a total of 2.25 Mbytes of 4 cycle dynamic RAM. Of this 1Mbyte is standard dynamic RAM and 1.25 Mbytes is video RAM.

Video RAM address generator: This controls the VRAM (Video RAM) serial port addressing. It is in turn controlled by the distributed control bus.

Serial bus interface: This is the distributed serial data and control bus interface. It connects the distributed control bus to the various timing components on the TRAM and the VRAM serial data to the distributed data bus.

Figure 4.6 shows the block diagram of the serial port TRAM, outlining some of the blocks described above.

The serial port TRAM can be considered as a transputer with memory, some of which is dual ported video RAM. The VRAM has a serial and a random access port to the frame store. These two ports can be considered as separate entities.

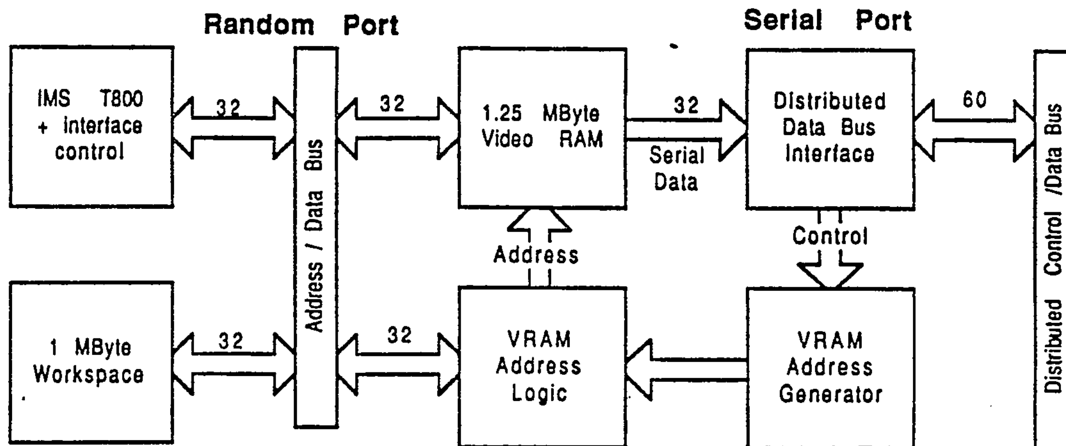

Figure 4.6: Serial port TRAM block diagram

Memory map

From figure 4.7 we see that the video RAM is placed as an extension to the workspace RAM, so that, should the need arise the video RAM can be used as extra workspace RAM.

The video RAM is mapped twice into the decoded memory map so that the special logic modes (marked **Logic Mode**) used in some video RAMs, which need special interfacing cycling, can be used. These special logic modes can be set by writing data to the area of store reserved for this purpose (marked **Logic Set**). Registers which control the serial port addressing and frame synchronization are mapped into the positive address space (marked **System Control**).

Frame store addressing and the video RAM

The serial port TRAMs frame store is designed around the packed pixel architecture. There are two addressing schemes that can be used with video RAMs

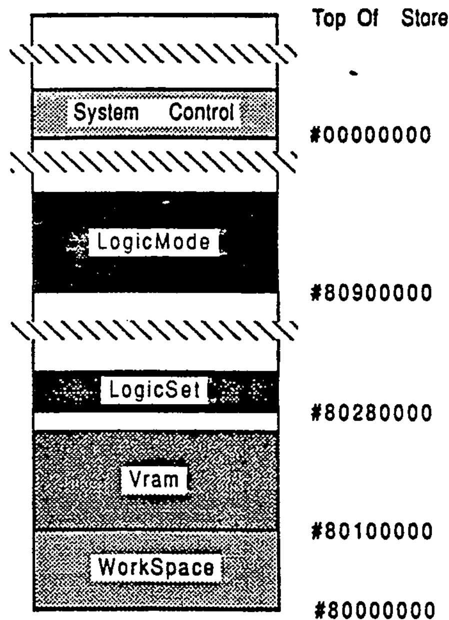

Figure 4.7: Memory map

when using packed pixel architecture.

Memory relative: Data is placed into the frame store with addressing related to the physical addressing of the video RAM. Put simply, the VRAM row and column addresses have a direct relationship with the frame stores X and Y coordinates, but the display can have a different horizontal dimension than the frame store. The maximum width of display is the size of the dual port buffer in the VRAM, i.e., 1024 eight bit pixels.

Display relative: The VRAM row and column addressing have no direct relationship to the frame stores X and Y coordinates. Instead the frame store addressing and the visible display have the same horizontal dimension. This scheme needs the video RAM real time data transfer mechanism, which allows the display horizontal dimension to be longer than the VRAM dual port buffer,

i.e., longer than 1024 eight bit pixels.

The serial port TRAM normally uses the display relative addressing scheme. When interlace is used, which can be set at initialization, it is switched into memory relative mode, and the frame store has a fixed horizontal dimension of 1024 bytes. These methods reduce the logic necessary to construct the address generator.

Pixel mappings

The video RAM can be used for various pixel types and screen sizes. The usage of the frame store entirely depends upon the user software and the backend display TRAM. The following mappings can be the most efficient ones:

8 bit packed pixels: Pixels mapped as bytes, four pixels per word. This allows 256 colors per pixel with a maximum of 1,310,720 pixels. This can be used for high resolution CAD, i.e., one serial port module can produce a 1280 by 1024 by 8 bit display, with an appropriate display backend.

32 bit packed pixels: Pixels can be mapped as 32 bit words, allowing a maximum of 2^{32} colors per pixel. One serial port TRAM can have a total of 327,680 pixels. Applications include any system that needs real color displays.

The method of mapping the frame store to the processor can have a profound effect on the performance of the graphical operations a single IMS T800 can achieve. To achieve most efficient use of the IMS T800 performance, pixels should be mapped as either bytes or 32 bit word data types as this takes advantage of the IMS T800s internal datapath representation.

Double buffered frame store addressing

It is useful, when maximizing performance in some graphic applications such as animation, to have at least two displays mapped onto the frame store. This allows one to be displayed while another is being updated.

To facilitate this, the address of the first pixel at the top left of the display can be preset. This address presetting allows the display to be flipped to alternate areas of the frame store. Flipping the display during frame flyback allows complete frames to be drawn before being displayed. The transputer can be informed of the state of the frame flyback condition so as to synchronize the frame flip to the frame flyback period.

Frame store distribution

The method of frame store distribution can have dramatic effects upon the design of the hardware to implement it. For the special port TRAM the design rests on the specification of the distributed data bus, which consists of a synchronous (clocked) inverted open-collector bus.

The open-collector arrangement allows any serial port TRAM to output data onto the bus at any time without fear of bus contention. This removes any need for any bus arbitration logic, and hence allows arbitrary distribution of screen space among an arbitrary number of serial port TRAMs. Each serial port TRAM has enough memory to be able to address any pixel of the display. Since all serial port TRAMs are synchronized, any one of them can alter the pixel data entry presently on the distributed data bus. If any serial port TRAM

is not responsible for any particular pixel, it simply writes a null (zero) into that location in the frame store.

This distribution technique is simple, and provides the spatial and characteristic distribution methods described earlier. To further enhance the flexibility of this, an output enable control bit is mapped into the IMS T800 address space. Any serial port TRAM output can be switched off completely. This provides the chronological distribution method.

4.3.2 Random access port

Memory cycles

The serial port TRAM has eight different types of memory access:

Internal read/write: This cycle is the fastest. It is internal to the IMS T800 and lasts for a single cycle (50ns on the 20 Mhz transputers).

External read/write: This cycle is the basic external memory cycle. It lasts for four processor cycles (200ns) and consists of a conventional dynamic RAM multiplexed addressed cycle.

Refresh: This is a CAS (Column Address Strobe) before RAS (Row Address Strobe) refresh cycle due to an addressing complication of the video RAMs.

Video update: This cycle is controlled by the video update logic. It allows the video RAM serial port to be updated. The video logic proceeds after gaining control of the data and multiplexed address buses and cycles the video RAM with a serial port update cycle. This cycle only happens

infrequently, when data in the serial port is about to run out of data.

Logic operation set: The logic operation unit available in some video RAMs is activated using a CAS before RAS write cycle. The logic mode remains set until a Reset Logic Mode or another Logic Operation Set Mode is issued.

Logic operation: The Logic Operation cycle is a conventional RAS-CAS cycle but is six cycles long. This cycle needs a special extended RAS pulse, which is generated from a combination of the interface strobes **notMemS1, notMemS2** and **notMemS4.** This cycle is forced to six cycles using the **notMemS4** strobe fed back into the **Wait** input of the IMS T800. This is done as a function of the addressing, and is controlled by programmable array logic.

Serial port control logic: This cycle allows the transputer to access the serial port control logic. All RAMs are disabled in this cycle.

Configuration: The configuration sequence is a conventional external read cycle that is used only after the transputer has just been reset. The configuration data is generated from the configuration PAL using the six least significant unlatched address bits. The configuration data is then latched into a single bit of the decode address latch to hold the data until the end of the cycle.

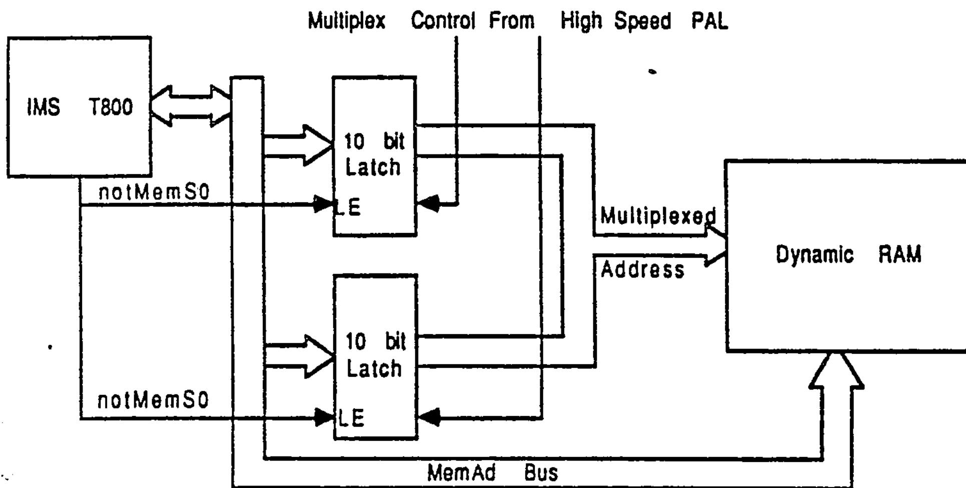

Figure 4.8: Multiplex arrangements with dynamic RAMs

Address latches and multiplexing

Due to the multiplexed address-data bus of the IMS T800 the addresses are only available at the beginning of the external memory cycle. The addresses have to be demultiplexed from the data. This is done using the transputer strobe **notMemS0** driving the latch enable inputs of two ten bit transparent latches. The latches used are high speed CMOS, as these have low propagation delays and have high output drive. Figure 4.8 shows the arrangements.

Due to the multiplexed address bus used with dynamic RAMs, the now demultiplexed transputer addresses have to be multiplexed onto the RAM address bus. To achieve this the output enables of the address latches are controlled from a high speed PAL. The outputs from two latches are connected together.

This control is a function of the transputer memory interface strobes **not-**

MemS2 and **MemGranted**. **MemGranted** is used because the video logic needs to drive the multiplexed address bus during a video update and therefore the multiplexer outputs have to be turned off completely.

Decoding

The top address bits, AD31, AD23-18 and the Configuration data are latched into a separate eight bit transparent latch. These address bits are used for decoding.

The RAM is arranged as:

- A single bank of general workspace RAM arranged as eight 256 Kbit by 4 RAMs.
- Five banks of eight 64 Kbit by 4 (256 Kbit) video RAMs.

The high speed PAL that controls the operation of the address multiplexer also generates four RAS strobes, one for the workspace RAM and three for the video RAM. Pairs of video RAM banks share RAS strobes. The last VRAM bank and the workspace RAM have their own RAS strobe.

The CAS strobes are supplied from another high speed PAL. This essentially is the RAM decoder, having six separate CAS strobes. The decoding is a function of the latched addresses A20-18, A31 and the Option input. The CAS strobes are timed for **notMemS3** on an External Read/Write cycle.

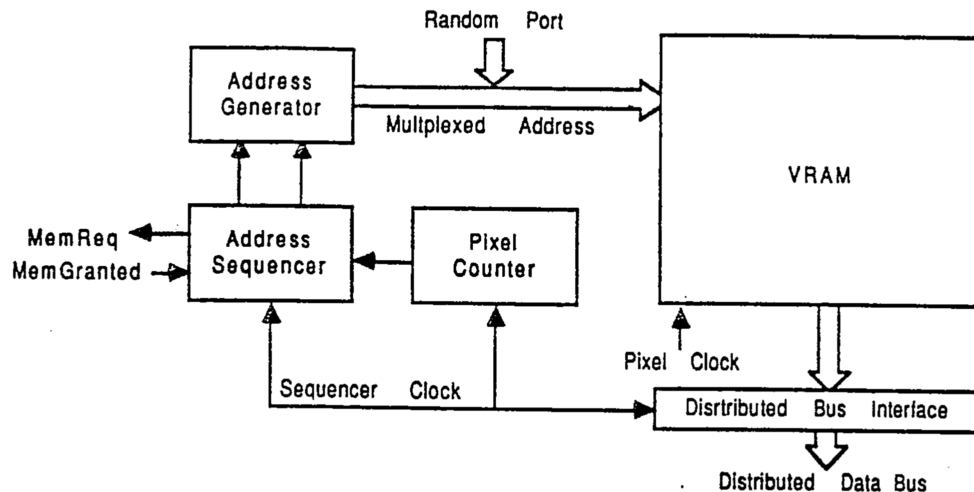

Figure 4.9: Serial interface block diagram

4.3.3 Serial access port

At the heart of the distributed frame store are two clocks which are synchronous. Both clocks are distributed to all serial port TRAMs in the system.

One is known as the sequencer clock and the other is known as the VRAM clock. The VRAM clock is stoppable and is controlled by the display TRAM. It is switched off just before the start of, and switched on just before the end of, the horizontal blanking period. Figure 4.9 shows the block diagram of the interface.

The serial port is built from several distinct groups of logic all synchronized to the previously mentioned clocks:

- **The address generator:** This generates the new serial address for the VRAM during a serial port update. The address generator has tri-state bus drivers connected to the multiplexed address bus of the VRAM.

- **Address sequencer:** This orchestrates control of the address generator

during the update of the serial port. The address sequencer takes over from the transputers memory interface and then cycles the VRAM in a data transfer cycle.

- **Pixel counter:** This starts the sequencer when serial data in the VRAM is about to run out. It is simply a counter that counts the data read out from the serial port, which resets itself immediately after the update occurs.

- **Serial bus interface:** This is the interface to the distributed data and control bus. This interface is clocked using the sequencer clock.

Address generator

The address generator is used when a video update cycle has been initiated. It provides 19 address bits, some of which are presented to the VRAM during a serial port update cycle and some of which are used as decode selectors. These addresses only form the start address for the serial data; subsequent data is accessed by clocking the VRAM. The scheme is illustrated in figure 4.10.

The lower eight bits of the address are fixed but are presettable. This forms the column address to the VRAM during the update cycle. This determines which data appears at the VRAM serial output after the VRAM has been updated.

The next 11 address bits are generated from a preloadable counter that increments just after every update cycle. This address points to the first

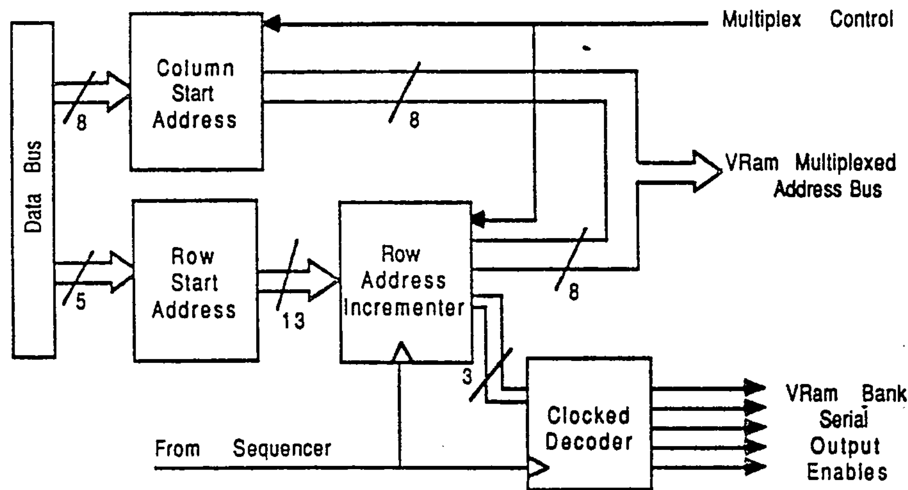

Figure 4.10: Address generation scheme

VRAM row to be accessed after each new frame is started. The lower eight bits from this form the row addresses in the VRAM during the update cycle. The top three bits of the counter are used to control the serial output enables of the five banks of VRAM, as shown in figure 4.10. There is no decoding on the update cycle, i.e., all VRAMs are updated at the same time.

The top five bits in the counter are preloaded from a five bit register which the user can preset so that the display can start from various addresses of the video RAM.

Address sequencer

This logic interfaces the address generator to the VRAM and determines the timing of the serial update control strobes. It arbitrates this update cycle between the address generator and the IMS T800's memory interface logic.

The sequencer is designed to update the serial port without interrupting

the pixel stream. To do this the pixel counter informs the sequencer that the serial data is about to run out. The entire sequencer operation lasts for 31 sequencer clock periods and new data appears at the VRAM serial output after 30 sequencer clock periods.

The sequencer requests the VRAM address bus by asserting **MemReq**. When **MemGranted** is asserted by the transputer, the sequencer cycles the VRAM in a serial port update cycle. This cycle updates the serial port via the random port when the VRAM strobe **DT/OE** is brought high synchronized with the VRAM serial clock.

Pixel counter

The serial port of the VRAM wraps around after 256 clocks. It therefore needs reloading every 256 VRAM clock cycles if data is not to be redisplayed. To implement this, the pixel counter signals to the sequencer when the end of serial data is about to occur. This end of data signal knows that the update will occur 30 clock periods later, so it signals the sequencer early.

A slight complication of the sequencer operation concerns the line flyback period. The sequencer must finish its operation before line flyback occurs, otherwise data destined for the start of the next line will be lost. The pixel counter will not cause an update to occur if an end of line is due, so that the update cannot occur during the line flyback period. The timing of this is critical, as the data which finds its way to the display is pipelined twice before getting to the display. This means the pipeline must be precharged with data

before the display line starts and emptied before the line ends. To this end, the VRAM clock is turned on two clock periods before the start of the line and switched off two clocks before the end of the line.

Distributed control

The serial port TRAM is designed to function as part of a distributed graphics system. For this reason the control necessary to drive the distributed data bus has to be common to all serial port TRAMs in the system. All clocking and control strobes are distributed using parallel terminated transmission lines.

The transmission lines are driven at the source using high speed CMOS logic with high output drive capability. This is terminated with a resistor to ground equal to the characteristic impedance of the transmission cable. All control inputs to the serial port TRAM are short wire stubs which offer little disturbance to the transmission line.

4.4 Display TRAM

4.4.1 Introduction

All display TRAMs have a generic architecture. Figure 4.11 shows the block diagram of the display backed TRAM architecture. It is neither practical nor cost efficient to design a system that is capable of any graphical display output. This particaular display TRAM has been designed so that it can be used in a variety of applications.

The Display Backend TRAM consists of:

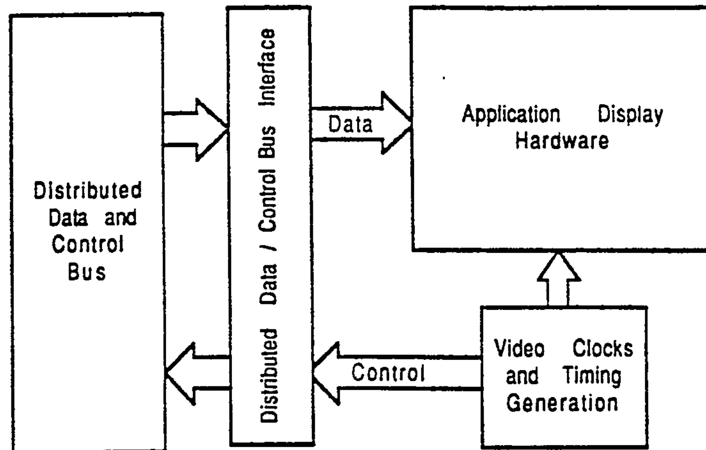

Figure 4.11: Display TRAMs

A transputer link: Communication to this module via at least one INMOS link, as a processor may not be necessary as it is used only for control and initialization of the backend hardware.

Video system clock generator: This provides the video system clock. The video system is timed from this clock.

A video timing generator: From this, all synchronization and system control is derived.

Serial control and data bus interface: This drives the distributed serial control bus and takes data from the distributed data bus.

Application specific display hardware: This hardware produces the application specific output derived from the 32 bit input data.

The display TRAM designed for this system has:

- **A transputer:** The transputer used is the T212. It is used here as a

logic controller to initialize the video timing logic, color look up tables and the mode selection.

- **Distributed control bus interface:** This consists of a few transmission line drivers, distributing the control signals to all the serial port TRAMs.

- **Video clocks and timing generator:** The pixel clocks and video timing generation used to synchronize all serial port TRAMs are controlled by the display TRAM.

- **Three pixel channels:** Each display channel converts 32 bits of input data from three distributed data bus inputs into the analog control signals to drive standard display monitors.

Pixel channels

The display TRAM consists of three independent 8 bit pixel channels, all with common clock and video timing generators. These are depicted in figure 4.12. Each channel has:

- **Premultiplexer:** An eight bit premultiplexer which chooses between eight bits of data from channel 0 or channel 1 and eight bits of data from channel 0 or channel 2. This then maps 24 input bits of channel 0 onto the lowest eight bits of channels 0, 1 and 2.

- **Input latch:** Distributed data bus 32 bit input latch.

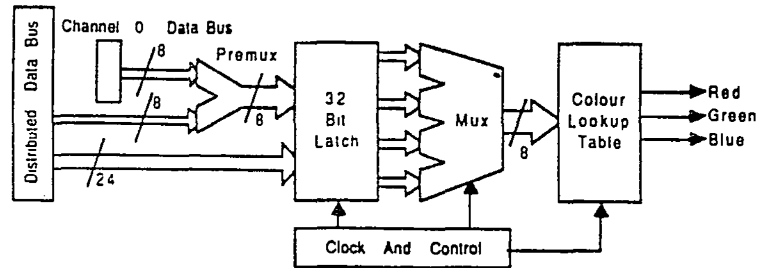

Figure 4.12: Pixel channels

- **Multiplexer:** 32 bit input 4 to 1 multiplexer.

- **CLUT:** 256 location color lookup table.

Display modes

There are three modes that the display TRAM is designed to operate in. They are:

- **8 bit mode:** This mode treats the 32 bit pixel data entering the display TRAM as four 8 bit pixel values. This data is multiplexed to the color lookup table. All three pixel channels operate seperately sharing only the distributed control. This is illustrated in figure 4.13.

- **Low resolution 24 bit mode:** This mode treats the 32 bit pixel data entering the display TRAM as a single 32 bit word of pixel data. The top 8 bits are not used, leaving the lower 24 bits as pixel data. The three pixel channels contribute to the display, one channel per primary color as shown in Figure 4.14.

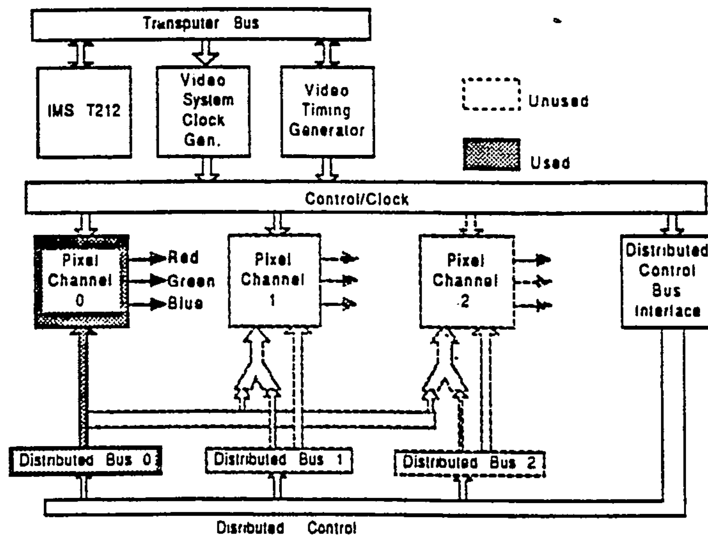

Figure 4.13: 8 bit mode

The 24 bit mode has a different clocking arrangement. Since data is being displayed at the same clock speed (pixel clock) but four times as much data is being used by the display, the input clock speed must be increased, i.e., pixel clock runs at the same speed as the pixel bus. The mode selection can change the clocking arrangements to suit these modes.

- **High resolution 24 bit mode:** This mode is similar to the 8 bit mode, except all three channels are used to provide each of the primary colors as shown in Figure 4.15.

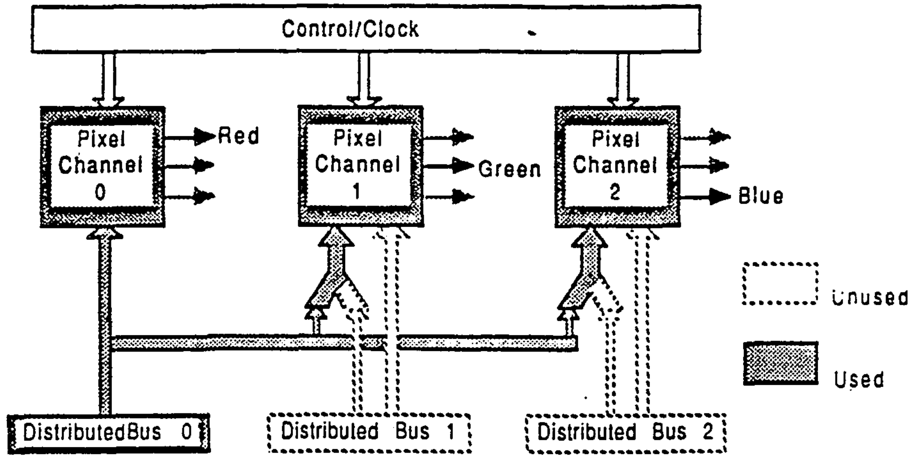

Figure 4.14: 24 bit mode

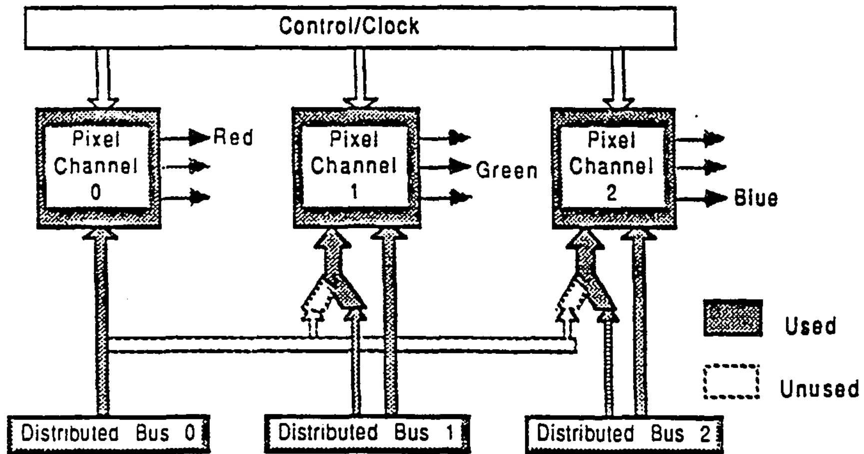

Figure 4.15: High resolution 24 bit mode

4.5 System configuration

4.5.1 Driving the frame store

The serial port TRAM can be used in a varied and non-specific manner, but the techniques fall into several distinct classes.

- **Data generator:** The serial port TRAM receives high level graphical commands from another TRAM and satisfies these commands by generating the drawing data into the frame store. The serial port TRAM becomes a programmable graphical drawing engine.
- **Data sink:** No graphical tasks are executed on the serial port TRAM. The serial port TRAM acts purely as a data sink, receiving data from the serial links and placing this data directly into the frame store. The frame store data is generated elsewhere on other TRAMs with transputers or specific hardware.
- **Data generator and sink:** A mixture of both the above methods.

The performance of the above techniques can be improved by adding more Serial Port TRAMs and distributing the drawing tasks appropriately, thus improving the effective drawing speed or the total serial link bandwidth into the frame store.

4.5.2 Frame store configurations

Using a combination of serial port TRAMs and the display TRAMs many system configurations can be constructed.

Minimal 8-bit display system: The minimal system consists of a single serial port TRAM and is connected as shown in figure 4.5. This minimal system provides all that is necessary for an 8-bit pixel (256 color) graphic display, to a maximum of 1280 by 1024 pixels.

Distributed 8-bit display system: Figure 4.5 shows a distributed 8-bit graphic display system. This distribution provides increased drawing speed and transputer link bandwidth into the frame store.

Minimal low resolution 24 bit display system: The system in figure 4.5 can also be used as a low resolution (maximum of 327,680 pixels) 32 bit pixel system. The display TRAMs premultiplexer is used in this configuration and provides a maximum of 24 bits of output color (8 bits per primary). Each pixel channel is used as a single primary color output.

Distributed low resolution 24 bit display system: The display TRAM in figure 4.5 is set into 24 bit mode as above, but this system provides increased possible drawing and link bandwidth into the frame store as in the distributed 8 bit system, but with more colors.

High resolution 24 bit display system: This system is essentially three separate 8 bit systems. The system is depicted in figure 4.16. This method separates the red, green and blue components into three 8 bit high resolution display channels as in the 8 bit system. It has all the characteristics of the 8 bit system but each of the three pixel channels on the display TRAM operate independently to provide a primary color as in the low resolution 24 bit system.

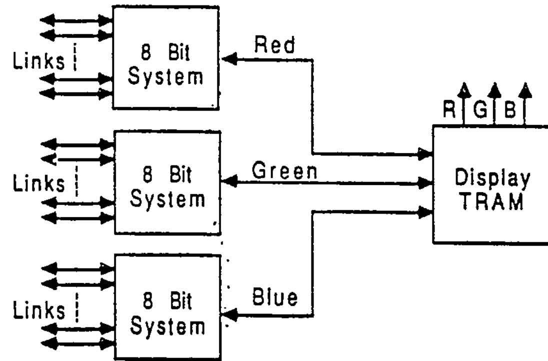

Figure 4.16: High resolution 24 bit display

High resolution distributed 24 bit dislay system: This system is similar to the above described 24 bit system except that each 8 bit pixel channel is distributed in the same way as the 8 bit system.

Chapter 5

Conclusions

This thesis focused on the problems of graphic systems and how they can be resolved using transputers. The needs of a flexible general purpose graphics system with efficient computing performance, drawing performance, etc., were addressed by distribution of the task to several transputers.

It has been shown that the performance of the frame store can be increased using video RAMs instead of using special hardware which makes it special purpose. The flexibility and efficiency in frame store is obtained by mapping the display data directly onto the transputer's address map by the video RAM.

The problems of single processor bus bottlenecks have also been discussed. It is shown how the bottlenecks can be removed by distributing the frame store, and how this distribution is efficient with transputers.

The large amount of processing necessary to perform typical graphical operations cannot be handled efficiently by single processor systems and hence they are slowed down. It is necessary to divide and distribute the processing task as smaller and more manageable tasks in high performance systems. The

control and complexity of such a system is considerably reduced using transputers for processing the distributed tasks, and the distribution of the frame store adds to the efficiency by providing a convenient interface to the display. Once the distribution has been designed and achieved, the task of increasing the system performance is a mere addition of transputers into the system, at the display interface or at the processing end and the system can be run with any desired performance.

The system designed can be easily implemented. The implementation of the system and analysis of the performance measures can be an interesting extension of this thesis. The system performance can be evaluated with respect to the conventional single processing systems. There will be an opportunity to explore the software side for performance of the system, since in parallel systems programming the hardware plays as important a role as designing the hardware itself.

Bibliography

[1] Mark Homewood, David May, David Shepherd and Roger Shepherd. "The IMS T800 Transputer", *IEEE Micro*, vol. 7, no. 5, Oct. 1987, pp. 10-26.

[2] Alan Knowles and Todor Kantchev. "Message passing in a transputer system" *Microprocessors and Microsystems*, vol. 13, no. 2, March 1989, pp. 113-123.

[3] Jean-Daniel Nicoud and Andrew Martin Tyrell. " The Transputer T414 Instruction Set", *IEEE Micro*, vol. 9, no. 3, June 1989, pp. 60-75.

[4] Richard Taylor. "Transputer communication link" *Microprocessors and Microsystems*, vol. 10, no. 4, May 1986, pp. 211-215.

[5] *The Transputer Databook*, INMOS Limited, 1988.

[6] J. D. Nicoud and P. Schweizer. "Multitransputer graphics system" *Microprocessors and Microsystems*, vol. 13, no. 2, March 1989, pp. 88-96.

[7] J. C. Admiraal and C. Pronk. "Distributed store allocation and file management for transputer networks", *Microprocessors and Microsystems*, vol. 14, no. 1, Jan/Feb 1990, pp. 10-15.

[8] Jon Vaughan. "MS-DOS memory resident transputer graphics server", *Microprocessors and Microsystems*, vol. 14, no. 2, March 1990, pp. 83-88.

[9] *The Transputer Applications Notebook: Systems and Performance*, INMOS Limited, 1989.

[10] Peter Croll and George Wilson. "Peripheral handling techniques for the transputer" *Microprocessors and Microsystems*, vol. 13, no. 2, March 1989, pp. 124-128.

[11] Chris Jesshope. "Parallel processing, the transputer and the future" *Microprocessors and Microsystems*, vol. 13, no. 1, Jan/Feb 1989, pp. 33-37.

[12] John Wenler and Dominic Prior. "Solving problems with transputers: background and experience" *Microprocessors and Microsystems*, vol. 13, no. 2, March 1989, pp. 67-78.

[13] C. H. R. Grimsdale. "Distributed operating system for transputers" *Microprocessors and Microsystems*, vol. 13, no. 2, March 1989, pp. 79-87.

[14] Ian Thomas. "Support system for OCCAM objects on transputers" *Microprocessors and Microsystems*, vol. 13, no. 2, March 1989, pp. 129-137.