The transputer

Colin Whitby-Strevens
INMOS Limited,
Whitefriars, Lewins Mead,
BRISTOL, BS1 2NP, UK

Abstract

The transputer is a programmable VLSI component with communication links for point-to-point connection to other transputers. Occam (*) is a language that enables a multi-transputer system to be described as a collection of processes that operate concurrently and communicate using message passing via named channels.

The INMOS transputer architecture is standardized at the level of the definition of occam (rather than at the level of the definition of an instruction set). The implementation of the first commercially available transputers is illustrated by describing the implementation of occam.

The paper concludes with outline examples of some applications.

1 Introduction

The transputer architecture has been developed to fulfil four main objectives:

> To create a commercial product range that sets new standards in ease of programming and ease of engineering.
>
> To provide the maximum performance to the user.
>
> To exploit future developments in VLSI technology within a compatible family.
>
> To create a programmable component that can be used to build systems with large numbers of concurrent computing elements.

VLSI currently permits 5-10 MIP processors to be manufactured in volume for low prices. There is therefore no economic barrier to the construction of very powerful computer systems containing many processing elements. The challenge is a technical one: how to engineer a system with, say, 1000 processors so as to make the inherent concurrency usable, and how to support the design of applications to take advantage of this amount of concurrency.

(*) occam is a trade mark of the INMOS Group of Companies

In the transputer architecture, the exploitation of a high degree of concurrency is made possible through a decentralized model of computation, in which local computation takes place on local data, and concurrent processes communicate by passing messages on point to point channels. The localized communications architecture also has substantial engineering advantages, described below.

An important design objective of occam and the transputer was to provide the same concurrent programming techniques both for a single transputer and for a network of transputers. Consequently, the features of occam were chosen to ensure an efficient distributed implementation on transputer systems. The concurrent processing mechanisms within the transputer were then designed to match.

The result is that a program ultimately intended for a network of transputers can be compiled and executed efficiently by a single computer used for program development. Once the logical behaviour of the program has been verified, the program may be configured for execution by a single transputer (low cost), or for execution by a network of transputers (high performance), or for a configuration representing a trade-off between these two extremes.

The choice of local processing and communications necessitates a significant change in programming concepts, and new algorithms need to be developed [4]. The study of various applications from this point of view is showing encouraging results ([15], [16], [17], [18], [19], [20], [21], [22]) and illustrative applications are given at the end of this paper.

2 Transputer architecture

2.1 Overview

The architecture of the transputer is defined by reference to occam. Occam provides the model of concurrency and communication for all transputer systems. Defining the architecture at this level leaves open the option of using different processor designs in different transputer products. This allows implementations which are optimized for different purposes. It also allows implementations to evolve with changes in technology, without compromising the standards established by the architecture.

A transputer contains memory, a processor and a number of standard point-to-point communication links which allow direct connection to other transputers. The processing capability may be general purpose, or may be optimized to a specific purpose. The on-chip memory may be extended off chip by a suitable interface.

A transputer may also have special purpose interfaces for connection to specific types of hardware. The separation of

0149-7111/85/0000/0292$01.00

the transputer system interface from other interfaces (eg the memory interface) means that it is possible to optimize the various interfaces individually, simplifying their use and improving their performance.

A system is constructed from a collection of transputers which operate concurrently and communicate through the standard links. Occam formalizes the computational model. It enables such a system to be described as a collection of processes operating concurrently and communicating through named channels.

Transputers directly implement the occam model of a process. Internally, an individual transputer can behave like any occam process within its capability; in particular, it can implement internal concurrency by timesharing processes. Externally, a collection of processes may be configured for a network of transputers. Each transputer executes a component process, and occam channels are allocated to links, which directly implement occam message-passing.

2.2 Occam

Occam [1, 3, 4] enables a system to be described as a collection of concurrent processes, which communicate with each other and with peripheral devices through channels. Occam programs are built from three primitive processes:

`v := e`	assign expression `e` to variable `v`
`c ! e`	output expression `e` to channel `c`
`c ? v`	input from channel `c` to variable `v`

The primitive processes are combined to form constructs. Each construct is introduced by a keyword, followed by a list of the component processes:

SEQuential	components executed one after another
PARallel	components executed together
ALTernative	component first ready is executed

A construct is itself a process, and may be used as a component of another construct.

Conventional sequential programs can be expressed with variables and assignments, combined in sequential constructs. `IF` and `WHILE` constructs are also provided.

Concurrent programs can be expressed with channels, inputs and outputs, which are combined in parallel and alternative constructs.

Each occam channel provides a communication path between two concurrent processes. Communication is synchronized and takes place when both the inputting process and the outputting process are ready. The data to be output is then copied from the outputting process to the inputting process, and both processes continue.

An alternative process may be ready for input from any one of a number of channels. In this case, the input is taken from the channel which is first used for output by another process.

The choice of synchronized communication prevents the loss of data. The choice of unbuffered communication removes the need for any store to be associated with the channel. Copying data from the outputting process to the inputting process is clearly essential for communication between transputers, and it is easy to make copying within a machine fast by use of microcode.

2.2.1 Design correctness

It is necessary to ensure that systems built from transputers, possibly involving hundreds or thousands of concurrent devices, can be designed and programmed effectively.

The design of occam and the transputer architecture has followed two principles to help the designer increase his confidence that his design is correct: simplicity and formality.

Occam has been kept simple, with the aim of making it easy to learn, and easy to use [3].

Formal techniques become much more important when concurrency is involved, as techniques based on exhaustive testing are impracticable. Occam has been designed to have a formal semantics. The way that this was achieved was to define a set of formal properties that the language should possess. These take the form of a number of behaviour-preserving transformations that should be applicable to any occam program [12]. Many semantic issues in the design and development of the language were resolved by reference to this set of properties. Enforcing this discipline has enabled a formal semantics for the language to be developed [13], and has laid the basis for software engineering tools ranging from formal validation to program transformation.

Practical and immediate benefits have been that the language is very self-consistent (which makes life easier for the compiler writer and user alike), that the equivalence of concurrent algorithms can be studied, and that programs can be transformed to have greater or less decentralisation without changing their logical behaviour[4].

2.2.2 Real time

On an individual transputer, a parallel construct may be configured to prioritize its components, and an alternative construct may be configured to prioritize its inputs. A higher priority process always proceeds in preference to a lower priority one.

The equivalent of an interrupt (a high priority process being scheduled in order to respond to an external stimulus) is designed entirely in occam, as all input and output is formalized as channel communication. A high priority process may wait for the first of several different inputs to become ready by using the `ALT` construct.

A high priority process proceeds until it terminates or has to wait for a communication. A system can thus be designed to meet real-time constrains by designing each high priority process so that the amount of processor time it requires over a given period is bounded, thus placing a bound on the total time that a high priority process may have to wait for the cpu. In many cases, it may be possible to reason that two or more high priority processes will never conflict, and that the latency reduces to the time required to switch from a low priority process to a hgh priority process. Each transputer implementation places a bound on this time.

A global synchronized sense of time is not practicable, and not representative of real-world situations. There is therefore a local concept of time, each timer being implemented as an incrementing clock.

Logically, access to a timer is treated as an input. A delayed input may be used, which waits until the value of the clock reaches an appropriate value. A timer input may be used in an alternative construct. This can be used to provide timeout on a communication.

2.3 Inter-transputer links

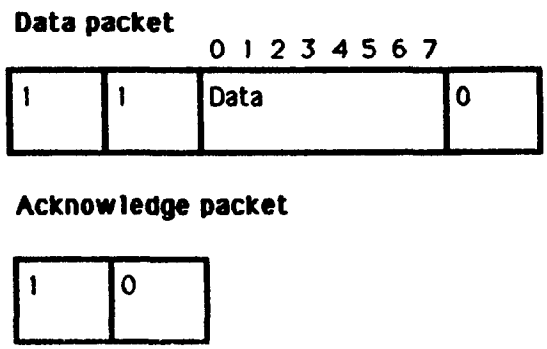
A link between two transputers provides a pair of occam channels, one in each direction. A link between two transputers is implemented by connecting a link interface on one transputer to a link interface on the other transputer by two one-directional signal lines. Each signal line carries data and control information.

Communication through a link involves a simple protocol, which provides the synchronized communication of occam. The protocol provides for the transmission of an arbitrary sequence of bytes, which allows transputers of different wordlength to be connected.

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 information is lost.

Each byte is transmitted as a start bit followed by a one bit followed by the eight data bits followed by a stop bit. After transmitting a data byte, the sender waits until an acknowledge is received; this consists of a start bit followed by a zero bit. The acknowledge signifies both that a process was able to receive the acknowledged byte, and that the receiving link is able to receive another byte. The sending process may proceed only after the acknowledge for the final byte of the message has been received.

Figure 1 Link protocol



Data bytes and acknowledges are multiplexed down each signal line. An acknowledge is transmitted as soon as reception of a data byte starts (if there is a process waiting for it, and if there is room to buffer another one). Consequently transmission may be continuous, with no delays between data bytes.

Using point to point serial communications, rather than busses has a number of advantages:

> Board layout is much simplified.
>
> Communications bandwidth is increased, as many links in a system can operate concurrently.
>
> Devices of different word lengths and performance can be easily interconnected.

Transputers with different word lengths and performance will all interwork together, as will all future products, ensuring that systems can be readily upgraded as the technology advances. It is not necessary to downgrade the performance of a connected set of components to that of the slowest.

2.3.1 Electrical properties of links

The signals are TTL compatible and their range can be extended by inserting industry standard line drivers and receivers. The standard transmission rate is 10MHz, providing a maximum performance of about 1MByte/sec in each direction on each link.

The links are designed to make the engineering of transputer systems as easy as possible. Irrespective of internal performance, all transputers use a reference clock of 5MHz, and this is required only for approximate frequency information and not for phase. All future transputers will also use this same frequency. The low frequency was chosen to simplify the distribution of the clock in a large system and it is not neccesary for all transputers to be on the same clock, enabling interworking between independently designed systems. Thus, transputers can be interconnected just as easily as TTL gates - indeed, the constraint on the designer is just the same - he must not exceed the maximum capacitance.

3 Implementation

3.1 Instruction set requirements and overview

The first transputer product is the T424, a general purpose 32 bit machine with 4K bytes of on-chip memory (which can be extended with off chip memory) and four bi-directional communications links, which provide a total of 8Mbytes per second of communications bandwidth. This will shortly be followed by the T222, a 16 bit machine providing similar facilities.

The design objectives of the I1 instruction set and the processor for these first transputers were as follows:-

> To provide an efficient implementation of occam, so that the use of high level languages results in efficient use of silicon capability, and that highly concurrent programs execute with minimum overheads.
>
> To provide a simple and direct implementation of occam so that programs can be compiled simply and straightforwardly, and to ensure that there is no need to consider programming at a lower level than that defined architecturally.
>
> To provide word length independence, so that a program can be executed using processors of different word lengths without recompilation.
>
> To provide position independence, so that program and workspaces may be allocated anywhere in memory after compilation.
>
> To provide low latency response to communications with external devices.

The lowest level of programming transputers is to use occam (occam is equivalent in effectiveness to a conventional microprocessor's assembler). The instruction set, and the use of occam as its programming language, is therefore illustrated by describing the main usage of the various registers in the machine, and by giving typical instruction sequences for simple occam constructs. Note that it is not common practice to abbreviate the names of the instructions, or to use mnemonics. Transputer system designers have no general need to write down instruction sequences, and using full names aids readability of the examples.

3.2 The I1 instruction set

3.2.1 Performance note

Two important performance measures are the number of bytes to hold the program, and the speed of execution provided by an implementation. It should be realized that the speed of execution of individual instructions is less important than the speed with which key system functions are performed, bearing in mind the intended uses of the machine.

The I1 instruction set is designed specifically with a view to efficient and fast VLSI implementation, although various trade-offs of performance versus silicon area are still possible. On the first transputers, each instruction is executed in one or more processor cycles using one level microcode. The figures given in this paper assume that program and data are stored on chip. Extra cycles may be required if program and/or data are stored off chip, though the significance of this can be reduced to a low level with careful organisation of the application. Full details are given in [14].

It should be noted that although all transputers have an external clock cycling at 5 MHz, the internal speed is set as part of the manufacturing process. It is expected that the range of speeds of the first transputers will provide internal processor cycle rates of up to 20MHz.

The design of the first transputers carefully balances the costs of memory access and alu operation, and contains sufficient overlap to ensure a high degree of efficiency. Many of the instructions execute in a single cycle, and typical sequences of commonly used instructions can deliver a 15 MIPS execution rate.

3.2.2 Memory organization

The memory address space comprises a signed linear address space. The instruction architecture does not differentiate between on-chip and off-chip memory. This allows the application designer to have complete control over the placement of code and data to take advantage of the performance benefits of on-chip memory.

A byte in memory is identified by a single word value called a pointer. A pointer consists of two parts: a word address and a byte selector. The byte selector contains as many bits as are needed to identify a single byte within a word and occupies the least significant bits of the pointer. For example, in a 24 bit transputer the word address would occupy the 22 most significant bits and the byte selector the 2 least significant bits.

Special instructions, such as *load local pointer* and *word subscript*, are provided to construct and manipulate pointers. Pointer values are treated as signed integers, starting from the most negative integer and continuing, through zero, to the most positive integer. This enables the standard comparison functions to be used on pointer values in the same way that they are used on numerical values.

The addressing instructions provide access to items in data structures, using short sequences of single byte instructions, allowing the representation of data structure access to be independent of the word length of the processor.

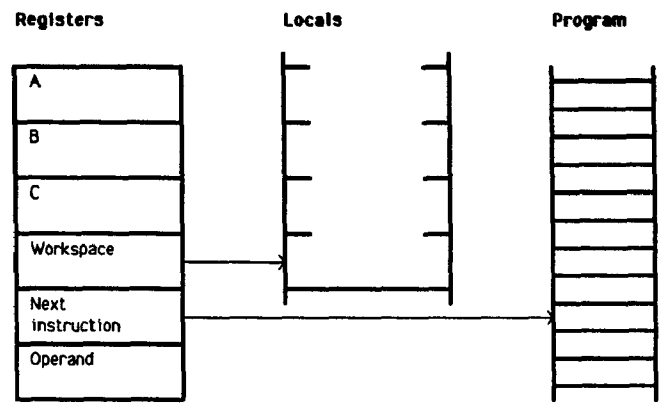
3.2.3 Registers

The design of the transputer processor exploits the availability of fast on-chip memory by having only a small number of registers; six registers are used in the execution of a sequential process. In the internal organization of the processor, all internal registers and data paths are the wordlength number of bits wide. The small number of registers, together with the simplicity of the instruction set, enables the processor to have relatively simple (and fast) data paths and control logic.

The six registers are:

> The workspace pointer which points to an area of store where local variables are kept.
>
> The instruction pointer which points to the next instruction to be executed.
>
> The operand register which is used in the formation of instruction operands.
>
> The A, B and C registers which form an evaluation stack. The evaluation stack is used for expression evaluation, to hold the operands of scheduling and communication instructions, and to hold parameters of procedure calls.

Figure 2 Registers for sequential programming



The evaluation stack removes the need for instructions to specify registers explicitly. Consequently, most of the executed operations (typically 80%) are encoded in a single byte. The I1 instruction set saves on time and area through not having to decode secondary control fields or register fields.

3.2.4 Support for concurrency

The processor provides efficient support for the occam model of concurrency and communication. It has a 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 processor 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. There is also no need for the hardware to perform access checking on every memory reference, resulting in an overall improvement in performance.

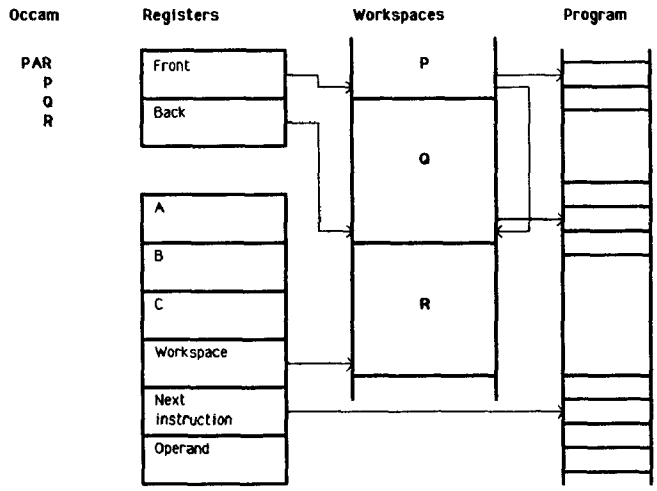
At any time, a concurrent process may be

active
- being executed
- on a list awaiting execution

inactive
- ready to input
- ready to output
- waiting until a specified 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.

Figure 3 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 together with the ones already being 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 still to terminate. When the components have all terminated, the counter reaches zero, and a specified process can then proceed.

The processor supports two priority levels, implemented using two lists as described above. A switch from a priority 1 process (low priority) to priority 0 process (high priority), or vice versa, may occur when a process stops, when a channel becomes ready, or when a communication completes and causes a priority 0 process to become ready.

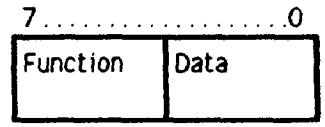
To allow a maximum latency figure to be calculated, the instructions which may take a long time to execute have been implemented to allow a switch during execution. Consequently, the maximum time taken to switch from priority 1 to priority 0 is 58 cycles (less than three microseconds with a 50ns processor cycle time). The switch from priority 0 to priority 1 only takes place when there is no priority 0 work available. The time taken for the switch is 17 cycles.

A context switch between processes, both executing at priority 1, occurs only at times when the evaluation stack has no useful contents, and therefore affects only the instruction pointer and the workspace pointer. With the need to save and restore registers at a minimum, the implementation of concurrency is very efficient.

3.2.5 Instruction format

All instructions have the same format. Each is one byte long, and is 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.

Figure 4 Instruction format



The use of a single instruction format requires only a simple decode mechanism in the processor, which reduces area and increases speed. The use of single byte instructions decouples the instruction format from the wordlength of the machine. In particular it avoids the commonly found problems concerned with aligning instructions on word boundaries.

Short instructions also improve the effectiveness of the instruction fetch mechanism, which in turn improves processor performance. The processor uses otherwise spare memory cycles to fetch instructions. As memory is word accessed, a 32 bit transputer will receive four instructions for every fetch. There are two words of instruction fetch buffer so that the processor rarely has to wait for an instruction fetch before proceeding (only on transfers of control if on-chip memory is used). Since the buffer is short, there is little time penalty when a jump instruction causes the buffer contents to be filled.

There is no instruction cache, as only rarely would such a cache reduce the number of processor cycles required. An on-chip cache incurs a significant cost in terms of chip area, as a cache requires several times the area of a simple memory to store the same amount of information. An off-chip cache complicates the external interface. Both require extra logic, even when aided by software (as in the IBM 801 [7]), which would be likely to slow down the overall speed of operation and use up even more chip area. The view is taken that the chip area is better spent on providing memory for the application.

3.2.6 Direct functions

The representation provides for sixteen functions, each encoded as a value in the range 0 to 15. Thirteen of these values are used to encode the most important functions performed by any computer. These include:

load constant	*load non local*
add constant	*store non local*
load local	*jump*
store local	*conditional jump*
load local pointer	
	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 onto the evaluation stack with a single byte instruction. The *load local* and *store local* instructions access locations in memory relative to the workspace pointer. The first 16 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. This eliminates the need for complicated and difficult-to-use addressing modes.

In the following examples, x and y are assumed to be local variables allocated to offsets *x* and *y* respectively in the first sixteen words of workspace.

occam	instruction sequence	bytes	cycles
x := 0	*load constant 0*	1	1
	store local x	1	1
x := y	*load local y*	1	2
	store local x	1	1

In this example, z is assumed to have been declared externally to the PROC which contains this assignment statement. The compiler allocates a local workspace location, at offset *staticlink*, to hold the address of the workspace that contains the variable z.

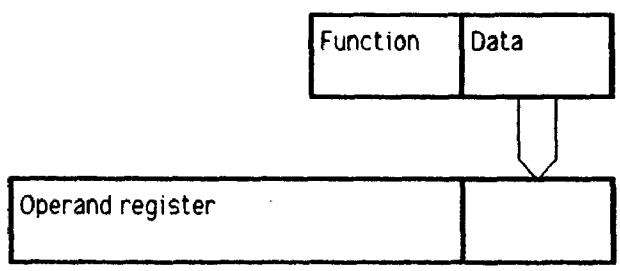
occam	instruction sequence	bytes	cycles
z := 1	*load constant 1*	1	1
	load local staticlink	1	2
	store non local z	1	2

3.2.7 Prefixing functions

Two more of the function codes, prefix and negative prefix, are used to allow the operand of any instruction to be extended in length.

All instructions are executed by loading the four data bits into the least significant four bits of the operand register, which is then used as the the instruction's operand. All instructions except the prefixing instructions end by clearing the operand register, ready for the next instruction.

Figure 5 Use of operand register



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 complements 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 prefixing instructions. In particular, operands in the range -256 to 255 can be represented using one prefixing instruction.

The following example shows the instruction sequence for loading the hexadecimal constant #754 into the A register, and gives the contents of the O register and the A register after executing each instruction

	O register	A register
prefix #7	#7	?
prefix #5	#75	?
load constant #4	0	#754

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

Each prefixing instruction occupies one byte and takes one cycle to execute.

3.2.8 Indirect functions

The remaining function code, *operate*, causes its operand to be interpreted as an operation on the values held in the evaluation stack. For example, the plus operation adds the values of the A and B registers. The result is left in the A register, and C is copied into the B register.

The *operate* instruction allows up to 16 such operations to be encoded in a single byte instruction. However, the prefixing instructions can be used to extend the operand of an operate instruction just like any other.

The encoding of the indirect functions is chosen so that the most frequently occurring operations are represented without the use of a prefixing instruction. These include arithmetic, logical and comparison operations, together with the most frequently used control functions and register manipulation functions.

Less frequently occuring operations have encodings which require a single prefixing operation (the transputer instruction set is not large enough to require more than 512 operations to be encoded!).

3.2.9 Expression evaluation

Loading a value onto the evaluation stack pushes B into C, and A into B, before loading A. Storing a value from A, pops B into A and C into B.

The A, B and C registers are the sources and destinations for arithmetic and logical operations. For example, the *add* instruction adds the A and B registers, places the result in the A register, and copies C into B.

If there is insufficient room to evaluate an expression on the stack, then the compiler introduces the necessary temporary variables in the local workspace. However, expressions of such complexity are, in practice, rarely encountered. Three registers provide a good balance between code compactness and implementation complexity.

Single length signed and single length modulo arithmetic is directly supported. In addition, a quick unchecked multiply is provided, in which the time taken is proportional to the logarithm of the second operand. The performance of these instruction sequences compares favourably, in both space and time, to that achieved by more complex instruction sets. Where a more complex instruction set cannot achieve the same effect in a single instruction, the performance gain is significant.

occam	instruction sequence	bytes	cycles
x + 2	*load local x*	1	2
	add constant 2	1	1
(v + w) * (y + z)			
	load local v	1	2
	load local w	1	2
	add	1	1
	load local y	1	2
	load local z	1	2
	add	1	1
	multiply	2	7+wordlength

3.2.10 Input and output

A channel provides a communication path between two processes. Channels between processes executing on the same transputer are implemented by single words in memory (internal channels); channels between processes executing on different transputers are implemented by point-to-point links (external channels).

As in the occam model, communication takes place when both the inputting and outputting processes are ready. Consequently, the process which first becomes ready must wait until the second one is also ready.

A process prepares for an input or an output by loading the evaluation stack with a pointer to a buffer, the identity of the channel, and the count of the number of bytes to be transferred. It then executes an *input message* or an *output message* instruction as appropriate.

The *input message* and *output message* instructions use the address of a channel to determine whether the channel is internal or external. This means that the same instruction sequence can be used for both internal and external channels, allowing a process to be written and compiled without knowledge of where its channels are connected. In particular, either an internal or an external channel can be used as the actual parameter for a channel parameter of a named process.

A communication primitive communicating a block of size n bytes requires only one byte of program, and on average the maximum of (24, 21+(8*n/wordlength)) cycles (including the scheduling overhead).

Instructions for enabling and disabling channels provide support for an implementation of alternative input without the use of polling.

3.3 Discussion

The requirements of the transputer indicate that a transputer processor should have a simple design. A transputer has a substantial amount of area given over to memory and communications, indeed a transputer can be thought of as a memory chip with a processor in one corner. In fact, the processor on the first transputers occupies about 25% of the available area.

It was clear that a simple processor could be constructed which would leave the majority of a chip area available for other purposes. The early RISC experiences [6, 7, 8, 9] lent further support to the evaluation that performance resulting from using a simple processor need not suffer.

Various projects, for example the IBM 801 [7] and MIPS [8], are willing to pay a price of software complexity in order to achieve implementation efficiency. However, the evidence of interpretive schemes for high level languages was that a simple instruction set could be designed which would lead to a better hardware/software relationship, and hence simplify the software as well. This would probably mean rejecting the strategy of compiling to a level best considered as microcode.

The justification for the use of multiple cycle instructions must be that the instructions well match the software requirements. In the transputer processor for the I1, repetitive operations, such as multiply, and block move, are implemented by microcode (with hardware assistance). The alternative RISC implementation [9] is to provide, for example, a single cycle multiply step, and for the software to compile the appropriate loop. The efficiency, in both code space and execution speed, resulting from the microcoded solution outweighs the cost of area and capacitance in the microcode ROM.

The I1 instruction set achieves word length independence, in that a program which manipulates bytes, words and truth values can be translated into an instruction sequence which behaves identically whatever the wordlength of the processor executing it (apart from overflow conditions resulting from word length dependencies). This results from the fact that the instruction size is independent of wordlength, the method of representing long operands as a sequence of prefixing instructions, and the memory addressing structure.

Workspaces are held in addressable memory, which the designer can choose to allocate on chip or off chip. Holding workspaces on chip forms a very effective alternative to the use of cache memory [11], the cost of which has already been discussed. A further advantage is that, unlike cache memory, rarely accessed data need not be brought on chip.

In general, a program needs much less store to hold it than an equivalent program in a conventional microprocessor. Since a program requires less store to represent it, less of the memory bandwidth is taken up with fetching instructions. As memory is word accessed, the processor will receive several instructions for every fetch (depending upon the number of bytes in a word).

The overall effect is thus that both compactness and speed have been achieved, together with economical use of silicon.

4 The transputer as a family

The T424 32 bit transputer is the first of a range of transputer products [14]. The next products will be a 16 bit transputer offering similar facilities to the T424, a high performance disk controller and a high performance graphics controller.

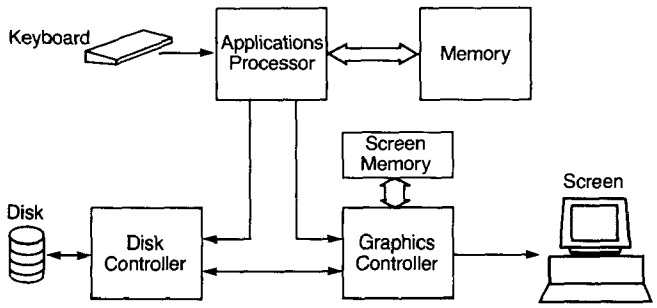
A transputer family device controller has the same organisation as a transputer, with the addition of special high speed control logic and interfaces. Device controllers are programmable, in occam, in the same way as transputers. This allows a designer to tailor the controller's function to his particular application.

4.1 A personal workstation

This section explores the design possibilities provided by the transputer architecture. The first step is the outline design of a personal workstation, which can be designed and built using functionally distributed transputers. One transputer, the applications processor, accepts the user's commands and carries out the appropriate processing, calling on two other transputers, which look after a disk system and a graphics display system respectively. Each of the latter two transputers

and associated hardware can be replaced by transputer based device controllers as they become available.

Figure 6 Personal computer workstation



The transputers are connected together using the standard transputer communications links. The resulting system can be engineered onto a single card.

The architecture permits a number of variations on the implementation of the workstation to be made without major redesign.

For example, the disk controller can double as the applications processor, and the applications transputer removed completely. Alternatively, more processors can be added, and the occam processes redistributed to take advantage of the additional concurrency. Vastly more than 1 Mbyte of memory could be attached.

4.2 Transputer without external memory

This second example explores the design and use of a large amount of processing power based on a transputer with only link interfaces in, say, a 28 pin chip carrier.

Figure 7 Single board transputer system

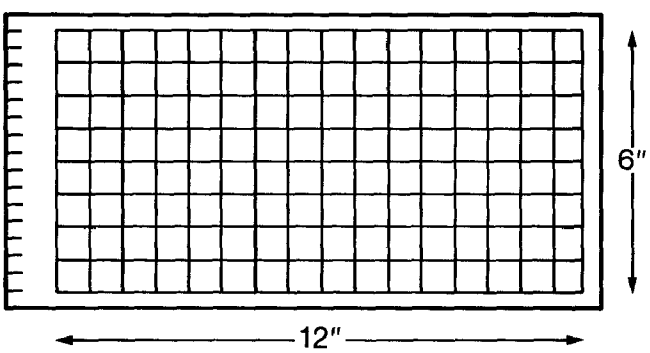
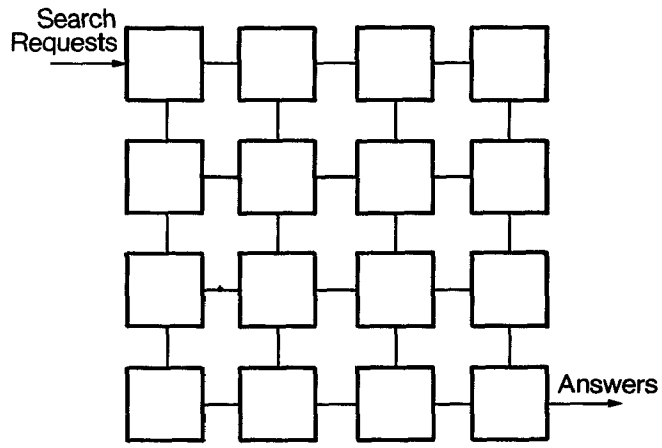


Figure 7 shows 128 transputers on a single printed circuit board. The board has 1/2Mbyte of fast static RAM and up to 1 GIPS (Giga Instruction Per Second) of processing power.

In this application, the board is used to provide high performance database searching. We assume that the database is partitioned, so that the most commonly accessed parts of a database can be placed in the transputer array.

The concept is shown in a simplified form in figure 8.

Figure 8 Concurrent database search



Here 16 transputers are connected into a square array with search requests input at one corner of the array, and answers being output from the other corner. Each transputer keeps a small part of the database in its local memory.

A small program in each transputer does the search. It can receive two sorts of input. A search request is forwarded to any connected transputer which has not yet received the request and simultaneously a search is made through the local data. The other sort of input is an answer from a transputer which has just searched its own local memory. This answer is merged with the answer generated from the local data and forwarded.

A simple performance analysis indicates the latency and throughput of this application on the 128 transputer board. Assume that each record is 16 bytes long, and that a search key is four bytes long. Each transputer can hold 200 records and the whole system can hold 25,000 records. For each transputer to search its own records against a request will take less than a millisecond.

The time taken to transmit a search request to each transputer in the array is proportional to the longest path across the system, in this case 24 links.

It takes about 6 microseconds to send a 4 byte message from one transputer to another. It will thus take about 150 microseconds to transmit a search request to the whole array, and about another 150 microseconds to transmit the answer. The whole search of 25,000 records will take less than 1.3 milliseconds.

However just as an individual transputer can be performing input, output and processing at the same time, so can the array. Requests can be pipelined through the system with a further request being input before the previous one has come out.

The size of the database partition can be increased by adding more boards. The search throughput is not adversely affected by this.

5 Conclusions

By taking an integrated approach to the design of a VLSI computer and a concurrent programming language it is possible to produce a new level of system building block which provides a very efficient implementation of the corresponding design formalism.

In particular, it is possible to support the use of the same concurrent programming techniques both within a single transputer and for a network of transputers. The concurrent processing features of a general purpose programming language can be efficiently implemented by a small, simple and fast processor.

The resulting transputer provides the unique concept of a programmable component enabling highly concurrent systems to be implemented within a formal design framework.

The architecture also provides a straight forward technology upgrade path. Future transputers can integrate more memory and more processors. The system architecture means that current and future products will be fully compatible and capable of interworking.

6 Acknowledgements

A large number of people have made invaluable contributions to the development of the transputer architecture and family of products, and these contributions are hereby collectively acknowledged. In particular, the original concept, and the drive to give it commercial reality, comes from Iann Barron, one of the founders of INMOS. David May designed occam, and led the team which developed the instruction set of the first products. Prof Tony Hoare, of Oxford University, has advised INMOS both generally on architecture and particularly on the basis for providing occam with a formal semantics.

7 References

[1] INMOS Limited, *Occam Programming Manual*, Prentice-Hall International, London, 1984.

[2] Barron, I.M. et al., *The Transputer*, Electronics, 17th Nov 1983, p 109.

[3] May, M.D., *OCCAM*, ACM SIGPLAN Notices vol 18-4 (Apr 1983) pp69-79.

[4] May, M.D. and Taylor, R.J.B., *OCCAM*, Microprocessors and Microsystems vol 8-2 (Mar/Apr 1984)

[5] May, M.D. and Shepherd, R, *Occam and the transputer*, IFIP WG10.3 workshop on Hardware Implementation of Concurrent Languages and Distributed Systems, North Holland (1984)

[6] Patterson, D.A and Sequin, C.H., *RISC I: A Reduced Instruction Set VLSI Computer*, Proc 8th International Symposium on Computer Architecture.

[7] Radin, G, *The 801 Minicomputer*, IBM Journal of Research and Development, Vol 27, No 3, pp237-246 (May 1983)

[8] Hennessy, J et al, *The MIPS machine*, Proceedings CompCon Spring 1982, IEEE, (February 1982)

[9] Colwell, R.P. et al, *Peering Through the RISC/CISC Fog: An Outline of Research*, Computer Architecture News, Vol 11, No 1 (March 1983)

[10] Patterson, D.A., *RISC Watch*, Computer Architecture News, Vol 12, No 1 (March 1984)

[11] Patterson, D.A. et al, *Architecture of a VLSI Instruction Cache for a RISC*, Proc 10th International Symposium on Computer Architecture, pp108-116, ACM (1983)

[12] Hoare, C.A.R. and Roscoe, A.W., *Programs as Executable Predicates*, Proc 1st Intl Conf on Fifth Generation Computer Systems, ICOT, 1984

[13] Roscoe, A.W., *Denotational Semantics for occam*, Proc NSF/SERC Workshop on Concurrency, Springer LNCS, 1984

[14] -, *IMS T424 transputer* data sheet, INMOS Limited, Bristol, England

[15] Schindler, M. *Real-time languages speak to control applications*, Electronic design, July 21, 1983, pp105-120.

[16] Fay, D. *Working with occam: a program for generating display images*, Microprocessors and Microsystems, Vol 8. No 1, Jan/Feb 1984

[17] Curry, B. Jane, *Language based architecture eases system design*, Computer Design, jan 1984, pp127-136

[18] Taylor, R., *Graphics with the transputer*, Computer Graphics 84, 1984

[19] Pountain, R., *The transputer and its special language, occam*, Byte, Vol 9, No 8, Aug 1984

[20] Kerridge, J.M. and Simpson, D., *Three solutions for a robot arm controller using Pascal-Plus, occam, and Edison*, Software Practice and Experience, Vol 14, No 1, Jan 1984

[21] Harp, J.G. et al, *Signal processing with transputers (traps)*, Computer Physics Communications (in press)

[22] Broomhead, D.S. et al, *A practical comparison of the systolic and wavefront array processing architectures*, 2nd Proc IEEE Conf on Acoustics, Speech and Signal Processing (March 1985).