TEMPUS
Parallel Computing

Lecture notes

on

transputers

Hardware description

Barcelona
1997

Contents

GENERAL PRINCIPS OF TRANSPUTERS **2**

System design rationale **4**

System architecture rationale **6**

Occam model **8**
Overview 8
Processes 10
Constructions 10

Physical architecture **13**
INMOS serial links 13
Link electrical specifications 13
System services 13

Basic architecture and concepts of transputer **17**
Programable device 17
VLSI technology 17
Simplified processor with microcoded scheduler 18
Transputer internal architecture 18
Sequential processing 20
Instructions 21
Direct functions 21
Prefix instructions 21
Indirect functions 22
Efficiency of encoding 23
Support for concurrency 23
Communication 24
Internal channel communication 25
Communication links 28
Timer 29
Alternative 30
Floating point instructions 30
Optimising use of the stack 31
Concurrent operation of FPU and CPU 31

ADVANCED FEATURES OF TRANSPUTERS (T9000) **33**

APPENDIX: TRANSPUTER CHARACTERISTICS **34**

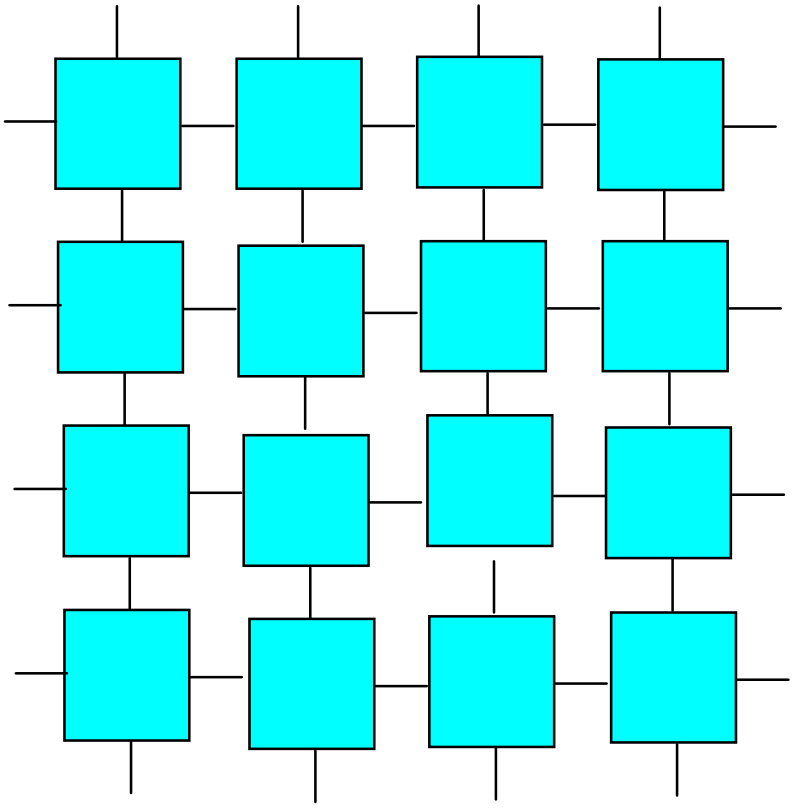
General princips of transputers

A transputer is a microcomputer with its own local memory and with links for connecting one transputer to another transputer.

The transputer architecture defines a family of programable VLSI components.

A typical member of the transputer product family is a single chip containing processor, memory and communication links which provide point to point connection between transputers. In addition, each transputer product contains special circuitry and interfaces adapting it to a particular use. For example, a peripheral control transputer, such as a graphics or disk controller, has interfaces tailored to the requirements of a specific device.

A transputer can be used in a single processor system or in networks to build high performance concurrent systems. A network of transputers and peripheral controllers is easily constructed using point-to-point communication.

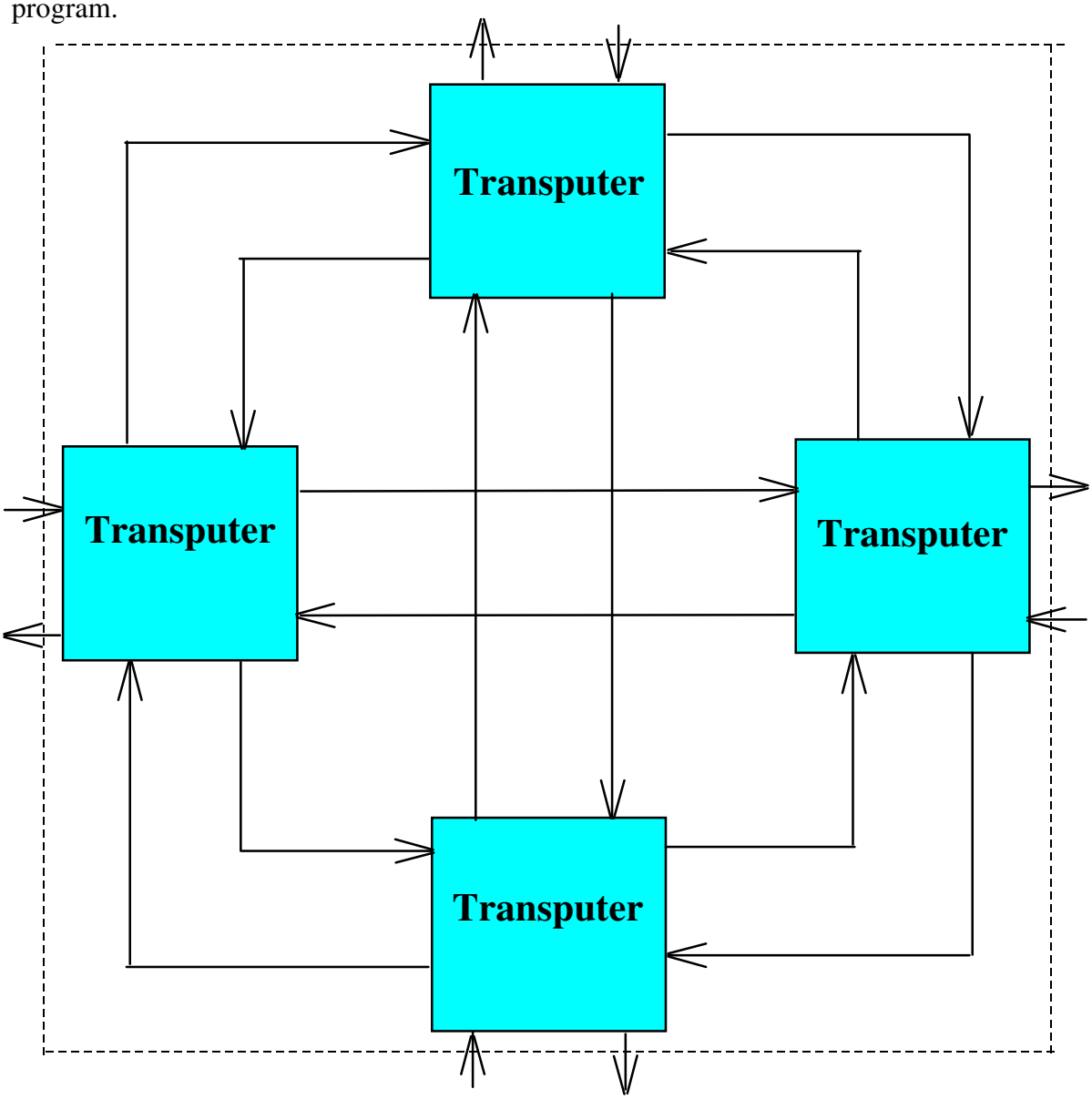


Transputer network

Transputers can be programmed in most high level languages, and are designed to ensure that compiled programs will be efficient. Where it is required to explicit concurrency, but still to use standard languages, occam can be used as a harness to link modules written in the selected languages.

To gain most benefit from the transputer architecture, the whole system can be programmed in occam. This provides all the advantages of a high level language, the maximum program efficienty and the ability to use the special features of the transputer.

Occam provides a framework for designing concurrent systems using transputers in just the same way that boolean algebra provides a framework for designing electronic systems from logic gates. The system designer's task is eased because of the architectural relationship between occam and the transputer. A program running in a transputer is formaly equivalent to an occam process, so that network of transputers can be described directly as an occam program.



A node of four transputers

System design rationale

The transputer architecture simplifies system design by the use of processes as standard software and hardware buildings blocks.

An entire system can be designed and programmed in occam, from system configuration down to low level I/O and real time interrupts.

The software building block is the process. A system is designed in terms of and interconnected set of processes. Each process can be regarded as an independent unit of design. It communicates with other processes along point-to point channels. Its internal design is hidden, and it is completely specified by the messages it sends and receives. Communication between processes is synchronized, removing the need for any separate synchronisation mechanism.

Internally, each process can be designed as a set of communicating processes. The system design is therefore hierarchically structured. At any level of design, the designer is concerned only with a small and managable set of processes.

Occam is based on these concepts, and provides the definition of the transputer architecture from the logical point of view.

Processes can be implemented in hardware. A transputer, executing an occam program, is a hardware process. The process can be independently designed and compiled. Its internal structure is hidden and it communicates and synchronizes with other transputers via its links, which implement occam channels.

Other hardware implementations of the process are possible. For example, a transputer with a different instruction set may be used to provide a different cost/performance trade-off. Alternatively, an implementation of the process may be designed in terms of hard-wired logic for enhances performance.

The ability to specify a hard-wired function as an occam process provides the architectural framework for transputers with specialized capabilities (e. g., graphics). The required function (e. g., a graphic drawing and display engine) is defined as an occam process, and implemented in hardware with standard occam channel interface. It can be simulated by an occam implementation, which in turn can be used to test the application on a development system.

A transputer can be programmed to perform a specialized function, and be regarded as a “black box” thereafter. Some processes, preprogrammed transputers and hardware processes. Such a system can, itself, be regarded as a component of larger system.

The architecture has been designed to permit a network of programmable components to have any desired topology, limited only by the number of links on each transputer. The architecture has been designed to permit a network of programmable components to have any desired topology, limited only by the number of links on each computer. The architecture minimizes the constrainst on the size of such systems, and the hierarchical structuring provided by occam simplifies the task of system design and programming.

The result is to provide new orders of magnitude of performance for any given application, which can now exploit the concurrency provided by a large number of programmable components.

System architecture rationale

The transputer architecture simplifies system design by using *point to point communication* links. Every member of the transputer family has one or more standart links, each of which can be connected to a link of some other component. This allows transputer networks of arbitary size and topologyto be constructed.

Point to point communication links have many advantages over multiprocessor buses:
- there is no contention for the communication mechanism, regardless of the number of transputers in the system;
- there is no capacitive load penalty as transputers are added to a system;
- the communication bandwith does not saturate as the size of the system increases. Rather, the larger the number of transputers in the system, the higher the total communication bandwith of the system. However large the system, all the connections between transputers can be short and local.

Each transputer in the system uses its own *local memory*. Overall memory bandwith is proportional to the number of transputers in the system. In contrast to a large global memory, where the additional processors must share the memory bandwith.

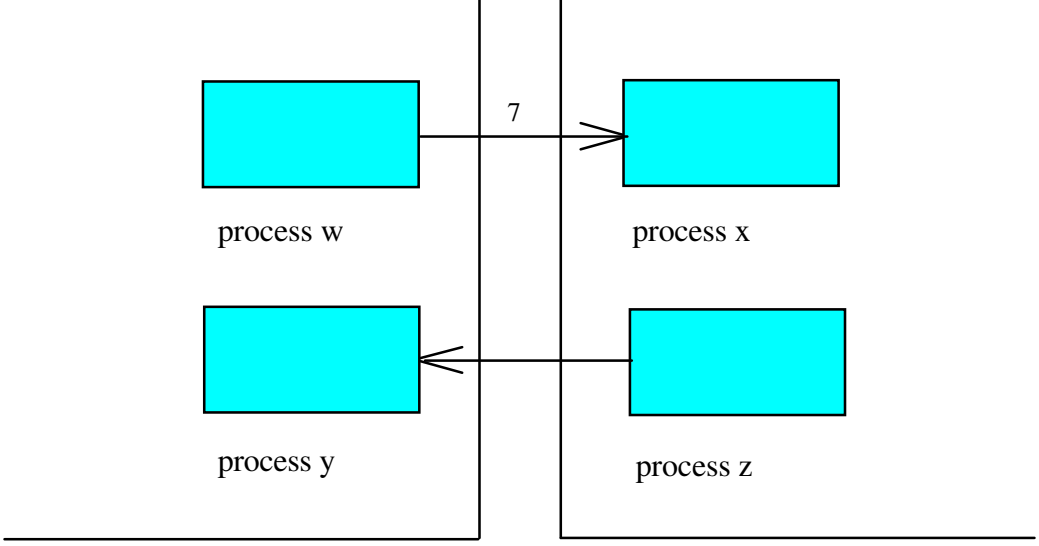
Because memory interfaces are not shared, and are separate from the communications interfaces, they can be individually optimised on different transputer products to provide high bandwith with the minimum of external components.

To provide synchronised communication, each message must be acknowledged. Consequently, a link requires at least one signal wire 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, along which data is transmitted serially.

The two signal wires on the link can be used to provide two occam channels, one in each direction. This requires a simple protocol. Each signal line carries data and control information.

The link protocol provides the sinchronized communication of occam. The use of a protocol providing for the transmission of an arbitrary sequence of bytes allows transputers of different word lenght to be connected.

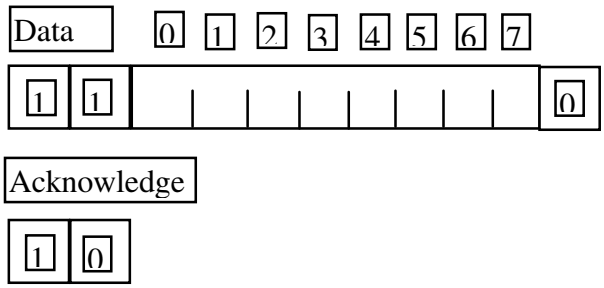


Each message is transmitted as a sequence of single byte communications, requiring only the presence of a single byte buffer in the receiving transputer to ensure that no information is lost. Each byte is transmitted as a start bit followed by a one bit followed by the eight data bits followed by a stop bit. After transmitting a data byte, the sender waits until an acknowledge is received; this consits 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 is able to receive another byte.

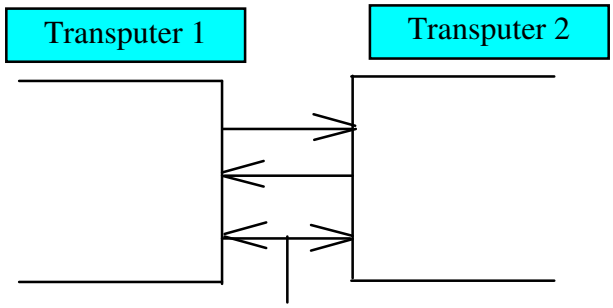
The sending link reshedules the sending process only after the acknoledge for the final byte of the message has been received.

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

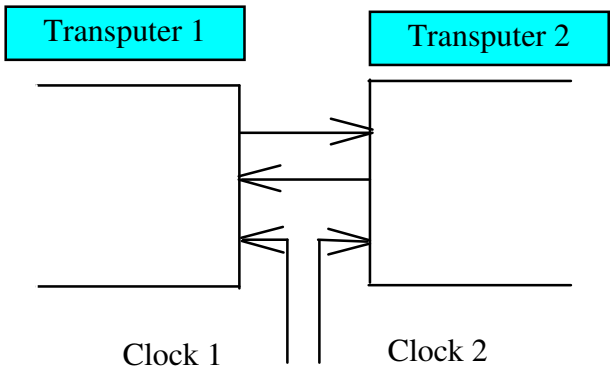


Link protocol

The links are designed to make the engineering of transputer systems straightforward. Board layout of two wire connections is easy to design and area efficient. All transputers will support a standard communications frequency of 10 Mbits/sec., regardless of processor performance. Thus transputers of different performance can be directly connected and future transputer systems will directly communicate with those today.



Common clock



Clocking transputers

Link communication is not sensitive to clock phase. Thus, communication can be achived between independently clocked systems as long as the communication frequency is the same.

The transputer family includes a number of link adapor devices which provide a means of interfacing transputer links to non-transputer devices.

Occam model

The programming model for transputers is defined by occam. The purpose of this section is to describe how to access and control the resources of transputers using occam.

The transputer development system will enable transputers to be programmed in other industry stundard languages. Where it is required to explicit concurrency, but still to use standard languages, occam can be used as a harness to link modules written in selected languages.

Overview

In occam processes are connected to form concurrent systems. Each process can be regarded as a black box with internal state, which can communicate with other processes using point to point communication channels. Processes can be used to represent the behavior of many things, for example a logic gate, a microprocessor, a machine tool or an office.

The processes themselves are finite. Each process starts, performs a number of actions and then terminates. An action may be a set of sequential processes performed one after another, as in a conventional programming language, or a set of parallel processes to be performed at the same time a one another. Since a process is itself composed of processes, some of which may be executed in parallel, a process may contain any amount of internal concurrency, and this may change whith time as processes start and terminate.

Ultimately, all processes are constructed from three primitive processes - assignment, input and output. An assignment computes the value of an expression and sets a variable to the value.

Input and output are used for communicating between processes. A pair of concurrent processes communicate using a one way channel connecting the two processes. One process outputs a message to the channel and the other process inputs the message from the channel.

The key concept is that communication is synchronised and unbuffered. If a channel is used for input in one process, and output in another, communication takes place when both processes are ready. The value to be output is copied from the outputting processes then proceed. Thus communication between processes is like the handshake method of communication used in hardware systems.

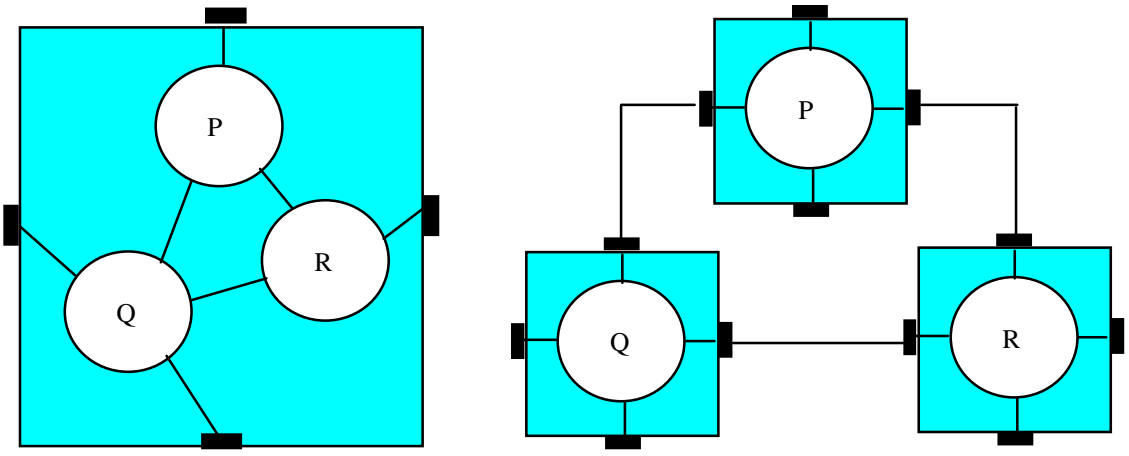
Since a process may have internal concurrency, it may input channels and output channels performing communication at the same time.

Every transputer implements the occam concepts of concurrency and communication. As a result, occam can be used to program an individual transputer or to program a network of transputers. When occam is used to program an individual transputer, the transputer shares its time between the concurrent processes and channel communication is implemented by moving data with the memory. When occam is used to program a network of transputers, each transputer executes the process allocated to it. Communication between occam processes on different transputers is implemented directly by transputer links. Thus the same occam program can be implemented on variety of transputer configurations, with one configuration optimised for cost, another for performance, or another for an appropriate balance of cost and performance.

The transputer and occam were designed together. All transputers include special instructions and hardware to provide maximum performance and optimal implementations of the occam model of concurrency and communications.

All transputer instructions sets are designed to enable simple, direct and efficient compillation of occam. Programming of I/O, interrupts and timing is standard on all transputers and conforms to occam model.

Different transputer variants may have different instruction sets, depending on the desired balance of costs, performance, internal concurrency and special hardware. The occam level interface will, however, remain standard across all products.



Mapping processes onto one or several transputers

Processes

After process starts execution, performs a number of actions, and then either stops and terminates. Each action may be an assignment, an input or an output. An assignment changes the value of the variable, an input receives a value from a channel and an output sends a value to a channel. At any time between its start and termination, a process may be ready to communicate on one or more of its channels. Each channel provides a one way connection between two concurrent processes; one of the processes may only output to the channel, and the other may only input from it.

Assignment

An assignment is indicated by the symbol :=. The example:

```
v := e
```

sets the value of the variable v to the value of the expression e and then terminates, for example: x: = 0 sets x to zero, and x := x+1 increases the value of x by 1.

Input

An input is indicated by the symbol ?. The example:

```
c ? x
```

inputs a value from the channel c, assigns it to the variable x and then terminates.

Output

An output indicated by the symbol !. The example:

```
c ! e
```

outputs the value of the expression e to the channel c.

Constructions

A number of processes can be combined to form a construct. A construct is itself a process and can therefore be used as a component of another construct. Each component process of a construct is written two spaces futher from the left hand margin, to indicate that it is part of the construct. There are four classes of constructs namely the sequential, parallel, conditional and the alternative constructs.

Sequence

A sequential construct is represented by

```
SEQ
  P1
  P2
  P3
```

The component process P1, P2, P3 ... are executed one after another. Each component process starts after the previous one terminates and the construct terminates after the last component process terminates.

For example:

```
SEQ
  c1 ? x
  x := x+1
  c2 ! x
```

inputs a value, adds one to it, and then outputs the result.

Sequential constructs in occam are similar to program written in conventional programming languages. Note, however, that they provide the performance and efficiency equivalent to that of an assembler for a conventional microprocessor.

Parallel

A parallel construct is represented by

```
PAR
  c1 ? x
  c2 ! y
```

allows the communication on channels c1 and c2 to take place together.

The parallel construct is unique in occam. It provides a straightforward way of writing programs which directly reflects the concurrency inherent in real systems. The implementation of parallelism on a single transputer is highly optimised so as to incur minmal process sheduling overhead.

Communication

Communication between two processes is the essence of occam programming. At its simplest it requires two processes executing in parallel and a channel joining them:

```
INT x:
CHAN OF INT comm:
PAR
  comm ! 2
  comm ! x
```

This trivial program merely outputs the value 2 from one process and inputs it into the variable x in the second. Its overall effect is exactly as if we had a single process which assigned 2 to x.

Alternation

A process may use several channels for input, or several for output, or both, in order to communicate with one other process. So we can make choices according to the state of channels. This is made possible by the ALT construction, whose name is short for alternation.

ALT joins together any number of components into a single construction, but the component part of an ALT called alternatives are some more complicated.

The simplest kind of ALT has as each alternative an input process followed by a process to be executed. The ALT watches all the input processes and executes the process associated with the first input to become ready. Thus ALT is basically a first-past-the-post race between a group of channels, with only the winner's process being executed:

```
CHAN OF INT chan1, chan2, chan3 :
INTx:
ALT
  chan1 ? x
     .... first process
  chan2 ? x
     .... second process
  chan3 ? x
     .... third process
```

If chan2 were the first to produce an input, then only the second process would be executed. Here choice is being decided in the time dimension, the inputs causing the program to wait until one of them is ready.

Physical architecture

INMOS serial links

All transputers have several links. The link protocol and electrical characteristics form are standard for all INMOS transputer and peripheral products.

All transputers support a standard communication frequency of 10 Mbits/sec. Some devices also support other data rates. Maintaining a standard communication frequency means that devices of mixed performance and type can intercommunicate easily.

Each link consists of two unidirectional signal wires carrying both data and control bits. The link signals are TTL-compatible so that their range can be easily extended by inserting buffers.

The INMOS communiciation links provide for communication between devices on the same printed circuitboard or between printed circuit boards via black plane. They are intended to be used in electrically quiet environments in the same way as logic signals between TTL-gates.

The number of links and any communication speeds in addition of to the standard speed of 10 Mbits/sec, are given in the product data for each product.

Link electrical specifications

The quiescent state of the link signals is low, for a zero. The link input signals and output signals are standard TTL-compatible signals.

For correct functioning of the links the specifications for maximum variation in clock frequency between two transputers joined by a link and maximum capacitive load must be met. Each transputer product also has specified the maximum permissible variation in delay in buffering, and minimum permissible edge gradients. Details of these specifications are provided in the product data.

System services

Powering up and down, running and stopping

At all times the specification of input voltages with respect to the GDN and VCC pins must be met. This includes the times when the VCC pins are ramping to 5 V, and also while they are ramping from 5 V down to 0 V.

The system services comprise the clocks, power and signals used for initialization.

The specificator includes minimum times that VCC must be within specification, the input clock must be oscillating, and the Reset signal must be high before Reset goes low. These specifications ensure that internal clocks and logic have settled before the transputer starts.

When the transputer is reset the memory interface is initialised (if present and configurable).

The processor and INMOS serial links starts after reset. The transputer obeys a bootstrap program which can either be in off-chip ROM can be received from one of the links. How to specify where the bootstrap program is taken from depends upon the type of the transputer is beeing used. The program will normally load up a large program either from ROM or peripheral such as a disk.

During power down, as during power up, the input and output pins must remain within specification with respect to both GND and VCC.

A software error, such as arithmetic owerflow, array bounds violation or divide by zero, causes an error flag to be set in transputer processor. The flag is directly connected to the Error pin. Both the flag and the pin can be ignored, or the transputer stopped. Stopping the transputer on an error means that the error cannot cause further corruption.

As well as containing the error in this way it is possible to determine the state of the transputer and its memory at the time the error occured.

Clock distribution

All transputers operate from a standard 5 Mhz input clock. High speed clocks are derived internally from the low frequency input to avoid the problems of distributing high frequency clocks. Within limits the mark-to-space ratio, the voltage levels and the transition times are immaterial. The limits on these are given in the differences in the product data for each product. The asynchronous data reception of the links means that differences in the clock phase between chips is unimportant.

Bootstrapping from ROM or from link

The program which is executed after reset can either reside in ROM in the transputer′s address space or it can be loaded via any one of the transputer′s INMOS serial links.

The transputer bootstraps from ROM by transfering control to the top two bytes in memory, which will invariably contain a backward jump into ROM.

If bootstrapping from a link, the transputer bootstraps from the first link to receive a message. The first byte of the message is the count of the number of bytes of program which follow. The program is loaded into memory starting at a product dependent location MemStart and then control is transferred to this address.

Messages subsequently arriving on the links are not acknowledged until the transputer processor obeys a process which inputs from them. The loading of a network of transputers is

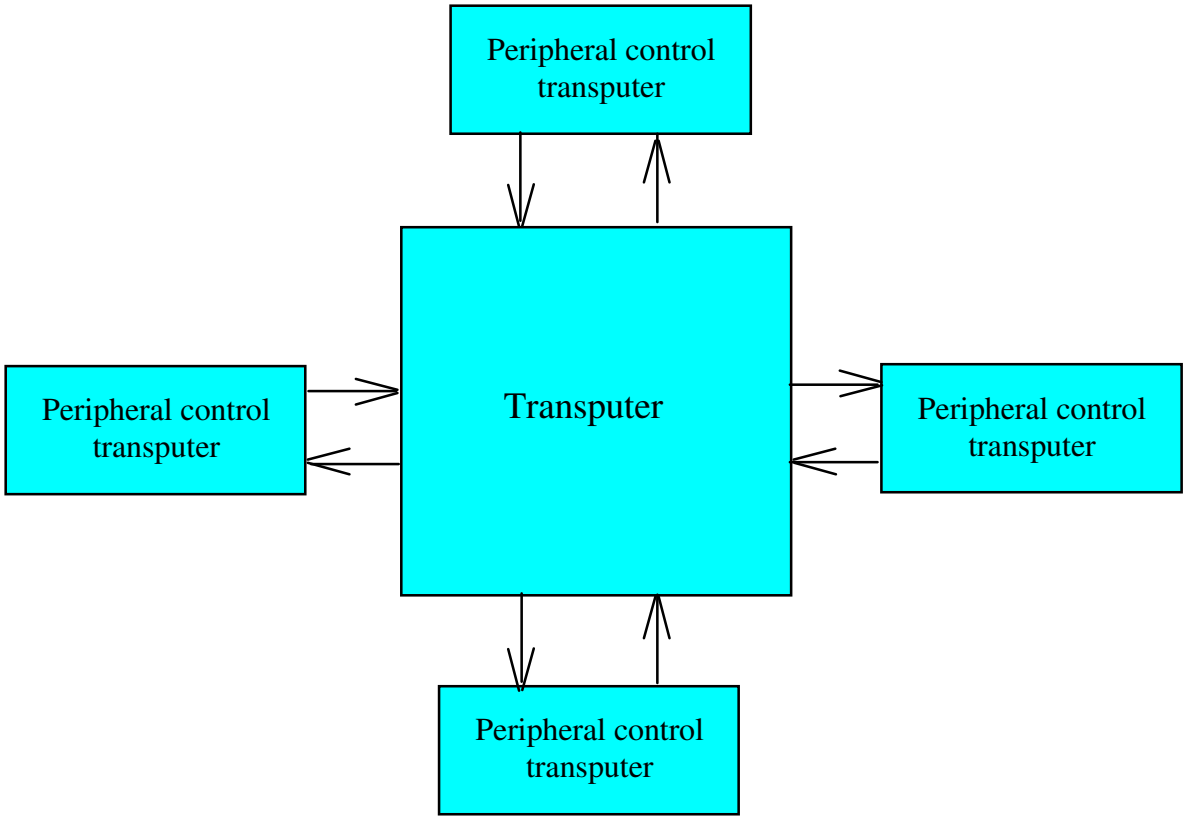
controlled by the transputer development system, which ensures that the first message each transputer receives is the bootstrap program.

Peripheral interfacing

All tansputers contain one or more INMOS serial links. Certain transputer products also have other application specific interfaces. The peripherial control transputers contain specialized interfaces to control specific periperal or peripheral family.

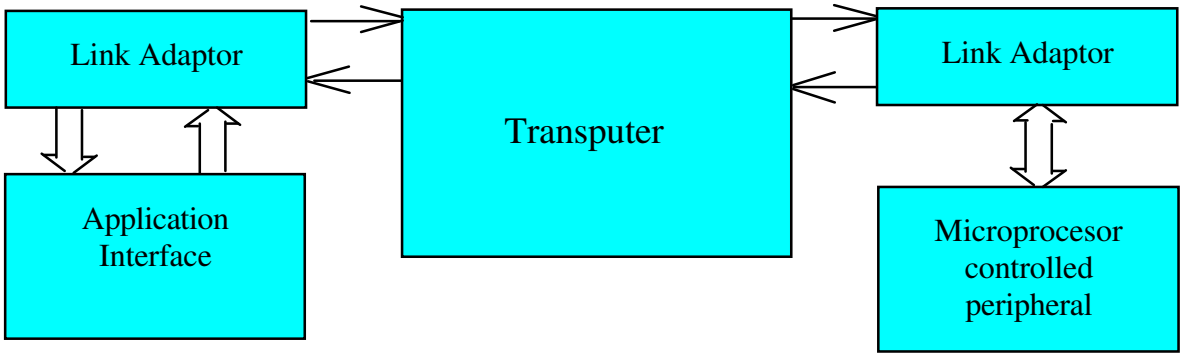
In general a transputer based application will comprise a number of transputers which communicate using INMOS links. There are three methods of communicating with peripherals:

- the first is by employing peripheral control transputers (eg. graphics or disks), in which the transputer chip connects directly to the peripheral concerned. The interface to the peripheral is implemented by special purpose hardware within the transputer. The application software in the transputer is implemented as an occam process, and contorls the interface via occam channels linking the processor to the special purpose hardware;



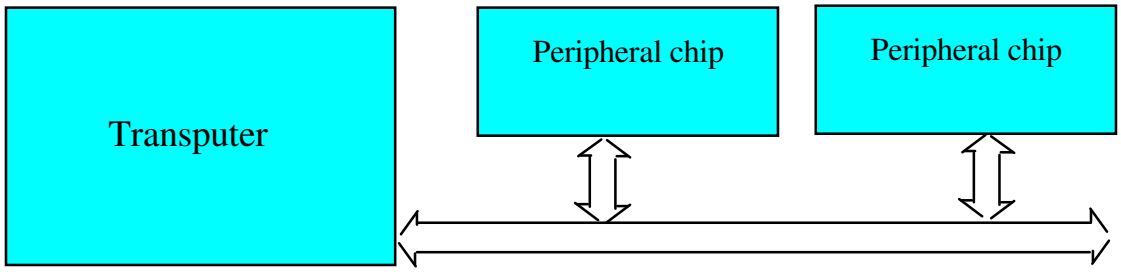
Transputer with peripheral control transputers

- the second method is by employing link adaptors. These devices convert between a link and a specialised interface. The link adaptor is connected to the link of an appropriate transputer, which contains the application designer's peripheral device handler implemented as an occam process;



Transputer with link adaptors

- the third method is by memory mapping the peripheral onto the memory bus of the transputer. The peripheral is controlled by memory accesses issued as a result of PORT inputs and outputs. The application designer's peripheral device handler provides a standard occam channel interface to the rest of the application.



Memory mapped peripherals

The first transputers implement an event pin which provides a simple means for an external peripheral to request attention from the transputer.

In all three methods, the peripheral driver interfaces to the rest of the application via occam channels. Consequently, a peripheral device can be simulated by an occam process. This enables testing of all aspects of transputer system before the construction of hardware.

Basic architecture and concepts of transputer

The INMOS transputer family is a range of system components each of which combines processing, memory and interconnect in a single VLSI chip. A concurrent system can be constructed from a colection of transputers which operate concurrently and communicate through serial communication links. Such systems can be designed and programmed in occam, a language based on communicating processes, and in other industry standard languages. Transputers have been succesfully used in application areas ranging from enbedded systems to supercomputers.

The first member of the family, the IMS T414 32-bit transputer, was introduced in September 1985, and has enabled concurrency to be applied in a wide variety of applications such as simulation, robot control, image synthesis, and digital signal processing. Many computationally intensive applications can explicit large arrays of transputers; the system performance depending on the number of transputers, the speed of inter-transputer communication and the performance of each transputer processor.

The power of transputer-based systems lies in the smoothly scaleable performance offered by adding more transputers. The transputer embodies the concepts required for effective parallel processing.

Futher transputer products are contimually being developed which increase in the memory, processing performance and communication performance. An important example in the floating point transputer, first introduced in 1987.

Programable device

The transputer is a component designed to exploit the potential of VLSI. This technology allows large numbers of identical devices to be manufactured cheaply. for this reason, it is attractive implement a concurrent system using a number of identical components, each of which is custumised by an appropriate program. The transputer is therefore, a VLSI device with a processor, and communication links for direct connection to other transputers . Transputer system can be designed and programmed using occam which allows an application to be described as a collection of processes which operate concurrently and communicate through channels. The transputer can therefore be used as a building block processing systems, with occam as the associated design formalism.

VLSI technology

One important property of VLSI technology is that communication between devices is very much slower than communication within a device. In a computer, almost every operation that the processor performs involves the use of memory. For this reason a transputer includes both processor and memory in the same integrated circuit design.

In any system constructed from integrated circuit devices, much of the physical bulk arises from connections between devices. The size of the package for an integrated circuit is determined more by the number of connection pins than by the size of the device itself. In addition, connections between devices provided by path on a circuit board consume a considerable amount of space.

The speed of communication between electronic devices is optimised by the use of one-directional signal wires, each connecting two devices. If many devices are connected by a shared bus, electrical problems of dividing the bus require that the speed is reduced. Also, additional control logic and wiring are required to control sharing of the bus.

To provide maximum speed with minimal wiring, the transputer uses point to point serial communication links for direct connection to other transputers.

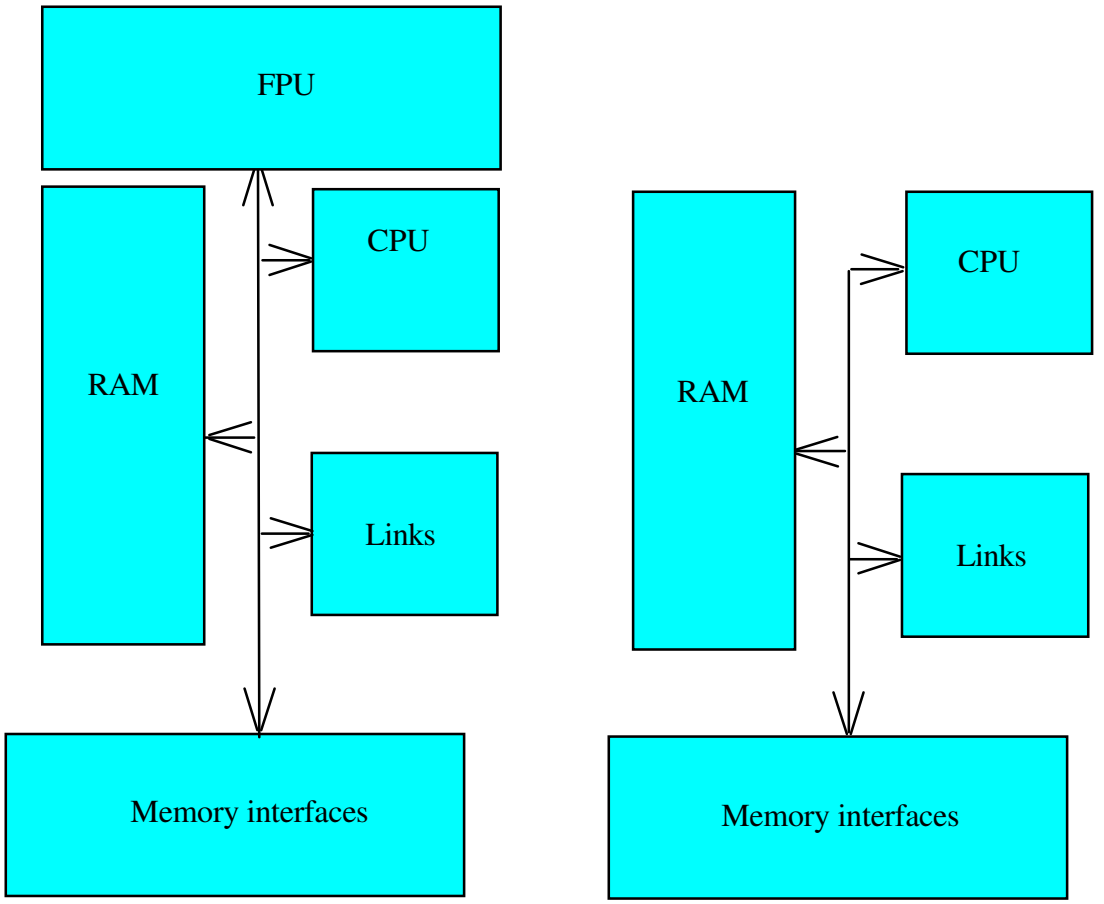
Simplified processor with microcoded scheduler

The most effective implementation of simple programs by a programmabel computer is provided by a sequential processor. Consequently, the transputer has a fairly conventional microcoded processor. There is a small core of about 32 instructions which are used to implement sequential programs. In addition there are other, more specialised groups of instructions which provide facilities such as long arithmetic and process scheduling.

As a process executed by transputer, therefore, has a microcoded scheduler which shares the procesor time between the conurrent processes. The scheduer provides two priority levels; any high prority pocess which can run will do so in preference to any low priority process.

Transputer internal architecture

Internally a transputer consists of a memory, processor, and communication system connected via a 32-bit bus. The bus also connects to the external memory and communications system each occupy about 25% of the total silicon area, the remainder being used for power distribution, clock generators and external connections.



Floating point transputer Transputer

The floating point transputers each have an on-chip floating point unit. The small size and high performance of this unit come from a design which takes careful note of silicon economics. This contrasts starkly with conventional co-processors, where the floating point unit occupies more area than complete microprocesor, and requires a second chip.

The CPU of the transputers contains three registers (A, B and C) used for integer and address arithmetic, which form a hardware stack. Loading a value into the 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. Similarly, the FPU includes three register floating point evaluation stack, containing AF,BF and CF registers. When values are loaded onto, or stored from the stack the AF, BF and CF registers push and pop in the same way as the A, B and C registers.

The addresses of floating point values are formed on the CPU stack, and values are transferred between the addressed memory locations the FPU stack under the control of the CPU. As the CPU stack is used only to hold the addresses of floating point values, the wordlenght of the CPU is independent of that of the FPU. Consequently, it would be possible to use the same FPU together with a 16-CPU.
The transputer scheduler provides two priority levels. FPU register stack is duplicated so that when the floating point transputer swiches for low to high priority none of the sate in the

floatin point unit is written to memory. This results in a worst-case interrupt response about 3 µs. Futhermore, the duplication of the register stack anables floating point arithmetic to be used in an interrut routine without any performance penalty.

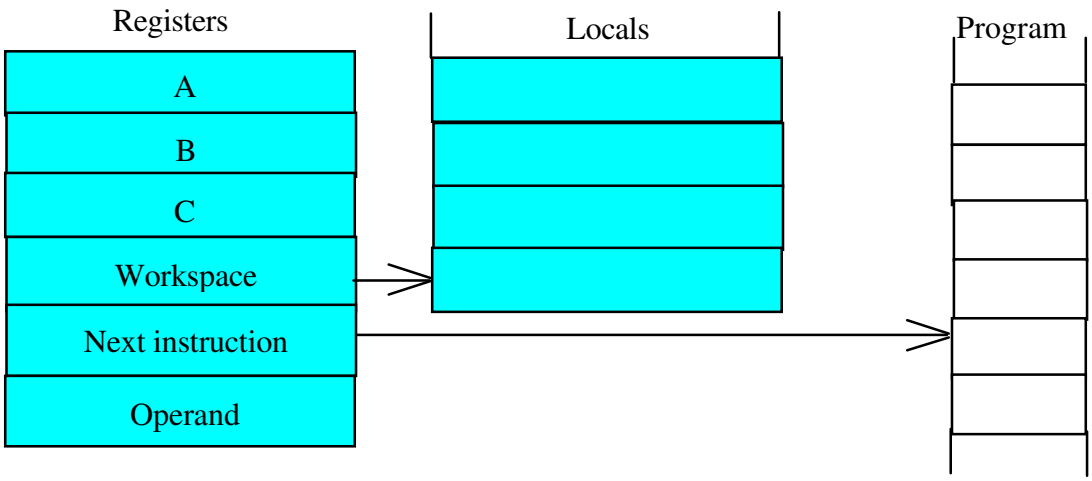
Sequential processing

The design of the transputer processor exploits the availabilitz of fast on-chip memory by having only a small number of registers; the CPU contains six registers which are used in the execution of sequential process. 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 and C registers which form an evaluation stack, and are the sources and destinations for most arithmetic and logical operations. Loading a value into the 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.



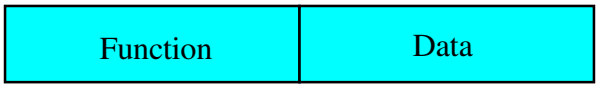
Registers

Expressions are evaluated on the evaluation stack, and instructions refer to the stack implicity. For example, the add instruction adds the top two 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. Statistics gathered from a large number of programs show that three registers provide an effective balance between code compactness and implemenation complexity.

No hardware mechanism is provided to detect that more than three values have been loaded onto the stack. It is easy for the compiler to ensure that never happens.

Instructions

It was a design decision that the transputer should be programmed in a high-level language. The instruction set has, therefore, been designed for simple and efficient compilation. It containst a relatively small number of instructions, all with the same format, chosen to give compact representation of the operations most frequently occuring in programs. The instruction set is independent of the processor wordlength, allowing the same microcode to be used for transputers with different wordlenths. Each instruction consists of a single byte divided into two bits parts. The four most significant bits of the byte are a function code, and the four least significant bits are the data value.



Instruction format

Direct functions

The representation provides for sixteen functions, each with a data value ranging from - to 15. Thirteen of these are used to encode the most important functions performed by any computer. These include:

load constant	*add constant*	
load local	*store local*	*load local pointer*
load non-local	*store non-local*	
jump	*conditional jump*	*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 instuction. The load local and store local instructions access locations in memory realtive to the workspace pointer. The first 16 locations can be accessed using single byte instruction.

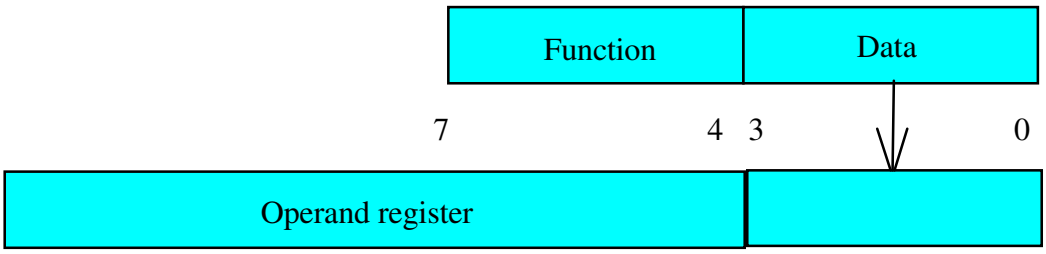
The load non-local and store non local instructions behave similarly, except that they access to data structures, and provide for simple implementations of the static links or displays used in the implementation of block structured programming language as occam.

Prefix instructions

Two more of the function codes are used to allow the operand of any instruction to be extended in length. These are:

prefix *negative prefix*

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



Instruction operand register

The prefix instruction loads ist 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 lenght 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.

The use of prefix instructions has certain beneficial consequence. Firstly they are decoded and executed in the same way as every other instruction, which simplifies and speeds instruction decoding. Secondly, they simplify the 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, indepenedent of the processor wordlength.

Indirect functions

The remaining function code, *operate*, causes its operand to be interpreted as an operation on the values held in the evaluation stack. This allows up to 16 such operations to be encoded in a single byte instruction. However, the prefix instruction can be used to extend the operand of an operate instruction just like any other. The instruction representation therefore provides for an indefinit number of operations.

The encoding of the indirect functions is chosen so that the most fequently occuring operations are represented without the use of prefix instruction. These include arithmetic, logical and comparison operations such as

add exclusive or greater than

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

The 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 colour graphics, pattern recognition and the implementation of error correcting codes. These instructions have

been added whilst retaining the existing IMS T414 instruction set. This has been possible because of the extensible instruction encoding used in transputers.

Efficiency of encoding

Measurments show that about 70% of executed instructions are encoded in a single byte (i. e. without the use of prefix instructions). Many of these instructions, such as *load constant* and *add* require just one processor cycle.

The instruction representation gives a more compact representation of high level language programs than more conventional insruction sets. Since a program requires less store to represent it, less of the memory bandwich is taken up with fetching instructions. Futhermore, as memory is word accessed the processor will receive several instructions for every fetch.

Short instructions also improve the effectiveness of instruction prefetch, which in turn improves processor performance. There is an extra word of prefetch buffer so that the processor rarely has to wait for an instruction causes the buffer contents to be discarded.

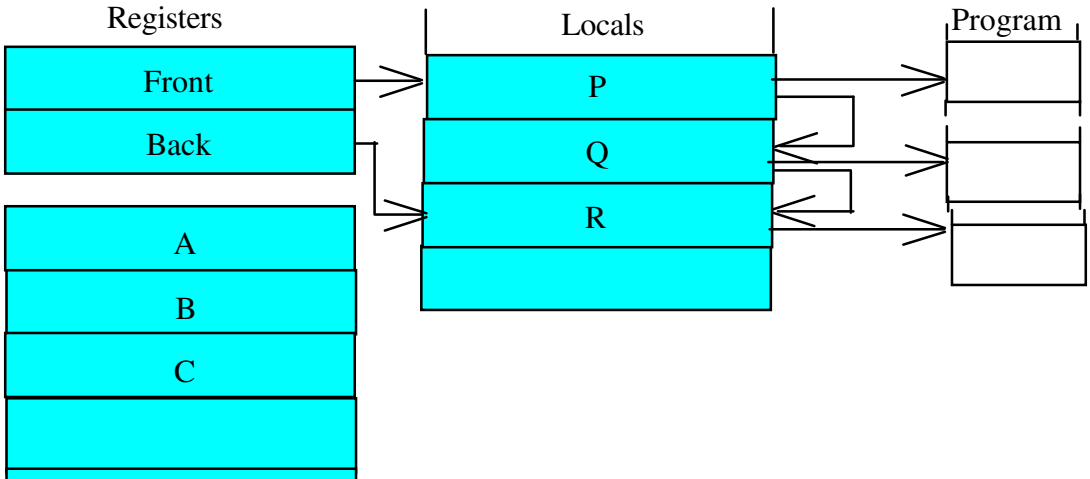
Support for concurrency

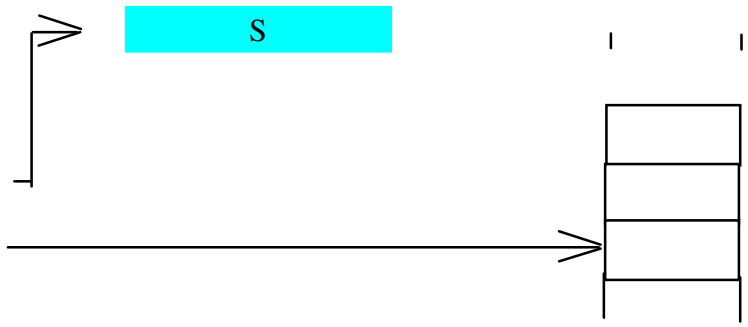
The processor provides efficient support for the occam model of concurrency and communication. It has a microcoded scheduler which enables any number of concurrent processes to be executed together, sharing the processor time. This removes the need for software kernel. Theprocessor does not need to support 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 process do not consume any processor time. The active processes waiting to be executed are held oan a list. This is a linked list, the other to the last. On the next figure S is executing, and P, Q and R are active, awaiting execution.





Linked process list

A process is executed until it is unable to proceed because it is waiting to input or output, or waiting for the timer. Whenever a process is unable to proceed, its instruction pointer is saved in its workspace and the next process is taken from the list. Actual process switch times are very small as little state needs to be saved; it is not necessary to save the evaluation stack or rescheduling.

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 together with the ones already being executed.

The correct termination of a parallel construct is assured by use of the *end process* construction. This uses a workspace location as a counter of the components of the parallel construct which have still to terminate. The counter is initialised to the number of components before the process are ´started´. Each component, the counter is non zero and the component is descheduled. For the last component, the counter is zero and the component continues.

Communication

Communication between processes is achived by means of channels. Occam communication is point-to-point synchronized and unbuffered. As a result, a channel needs no process queue, no message queue and no message buffer.

A channel between two processes executing on the same transputer is implemented by a single word in memory: a channel between processes executing on different transputers is implemented by point-to-point links. The processor provides a number of operations to support message passing, the most important being

input message *output message*

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

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.

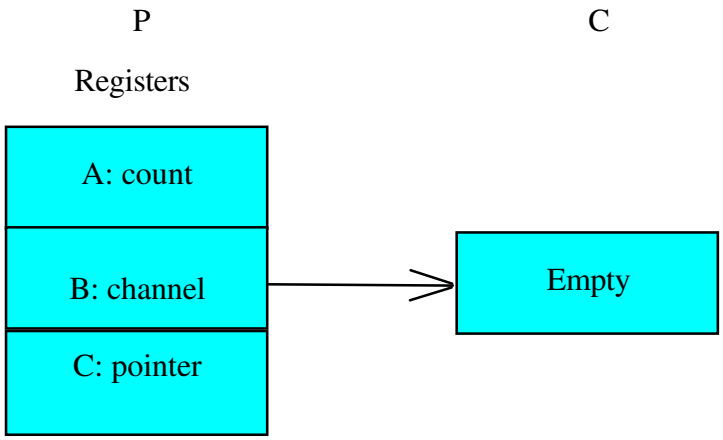
A process performs an input or output by loading the evaluation stack with a pointer to a message, the address of a channel, and a count of the number of bytes to be transferred, and then executing an *input message* or an *output message* instruction.

Internal channel communication

At any time, an internal channel (a single word in memory) either holds the identity of a process, or holds the special value empty. The channel is initialised to empty before it is used.

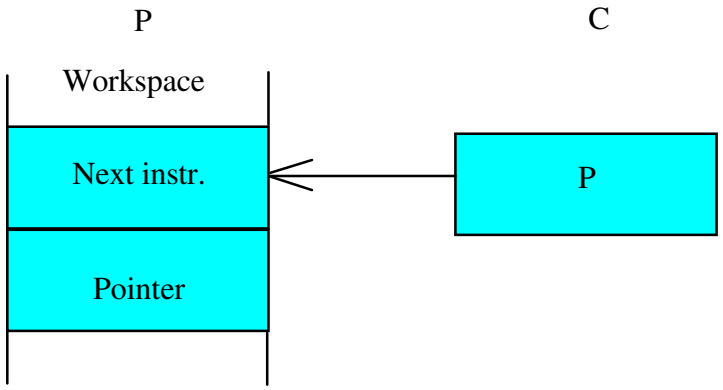
When a message is passed using the channel, and the processor starts to execute the next process from the scheduling list. When the second process to use the channel becomes ready, the message is copied, the waiting process is added to the scheduling list, and the channel reset to its initial state.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

In the next process P is about to execute an output instruction on an ´empty´ channel C. The evaluation stack holds a pointer to message, the address of channel C, and a count of the number of bytes in the message.

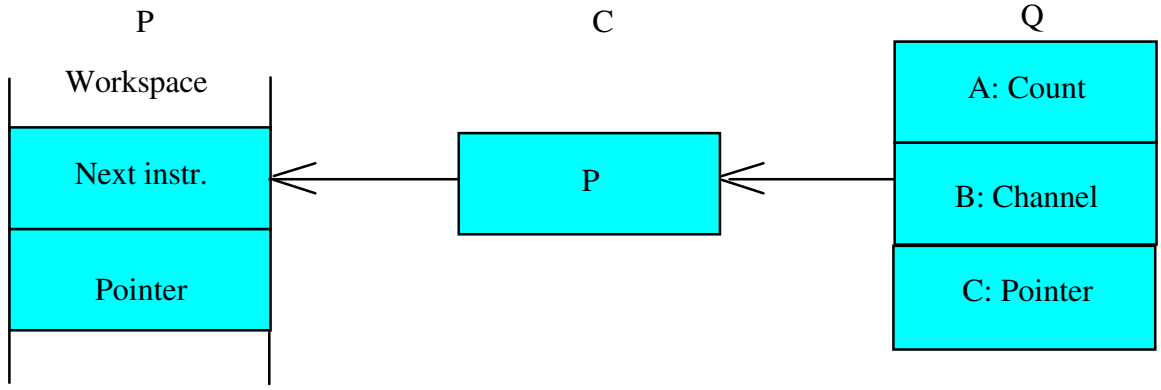


Output to an empty channel

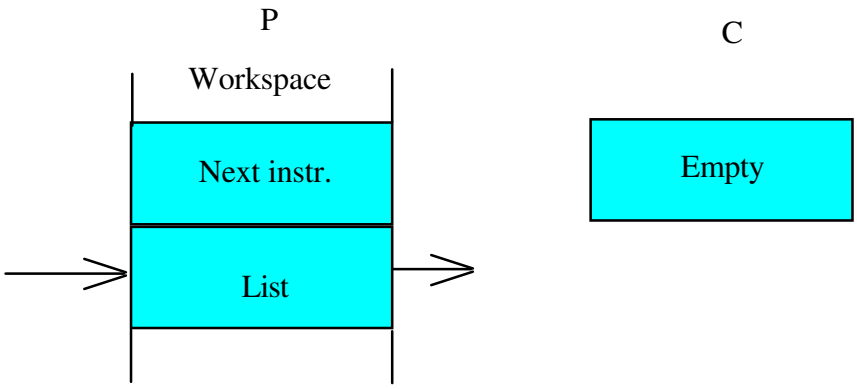
After executing the output instruction, the channel C holds the address of the workspace of P, and the address of the message to be transferred is stored in the workspace of P. P is descheduled, and the process starts to execute the next process from the scheduling list.



The channel C and the process P remain in this state until a second process, Q executes an input instruction on the channel.



The message is copied, the waiting process P is added to the scheduling list, and the channel C is reset to its initial ´empty´state.

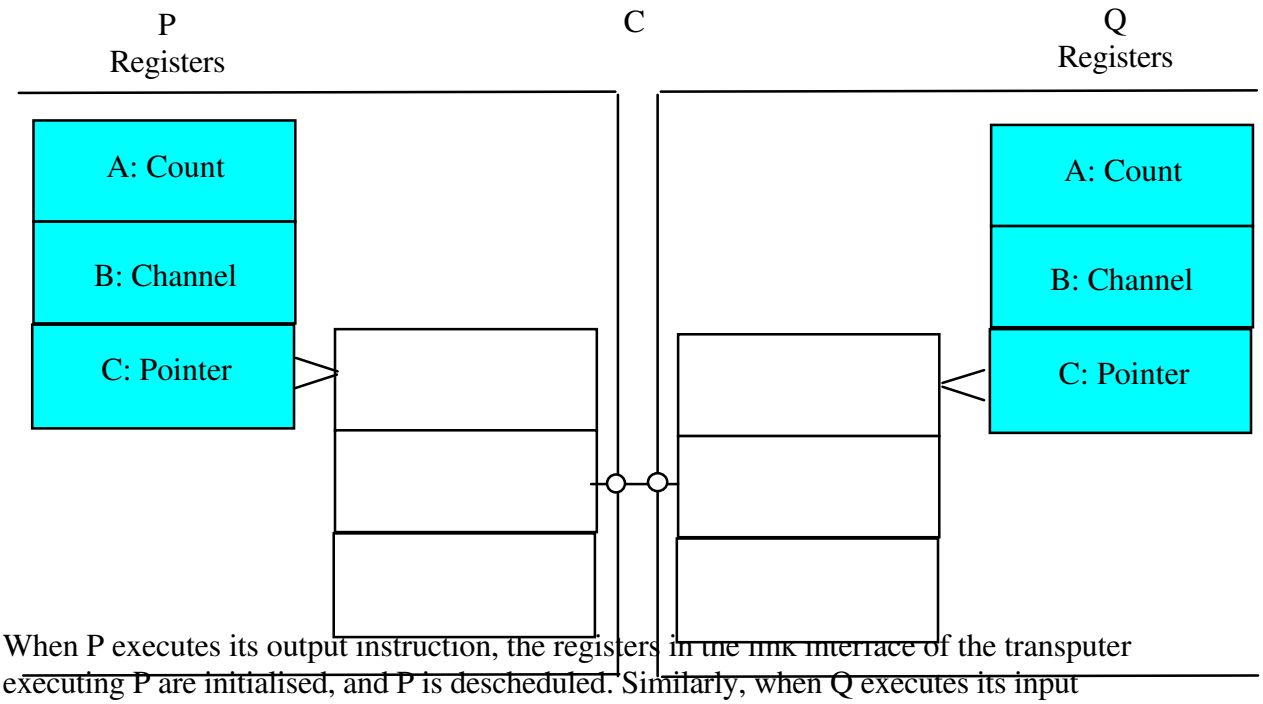


When a message is passed via an extrnal channel the processor delegates to an autonomous link interface the job of transferring the message and deschedule the process. When the message has been transferred the link interface causes the processor to reschedule the waiting process. This allows the processor to continue the execution of other processes whilst the external message transfer is taking place.

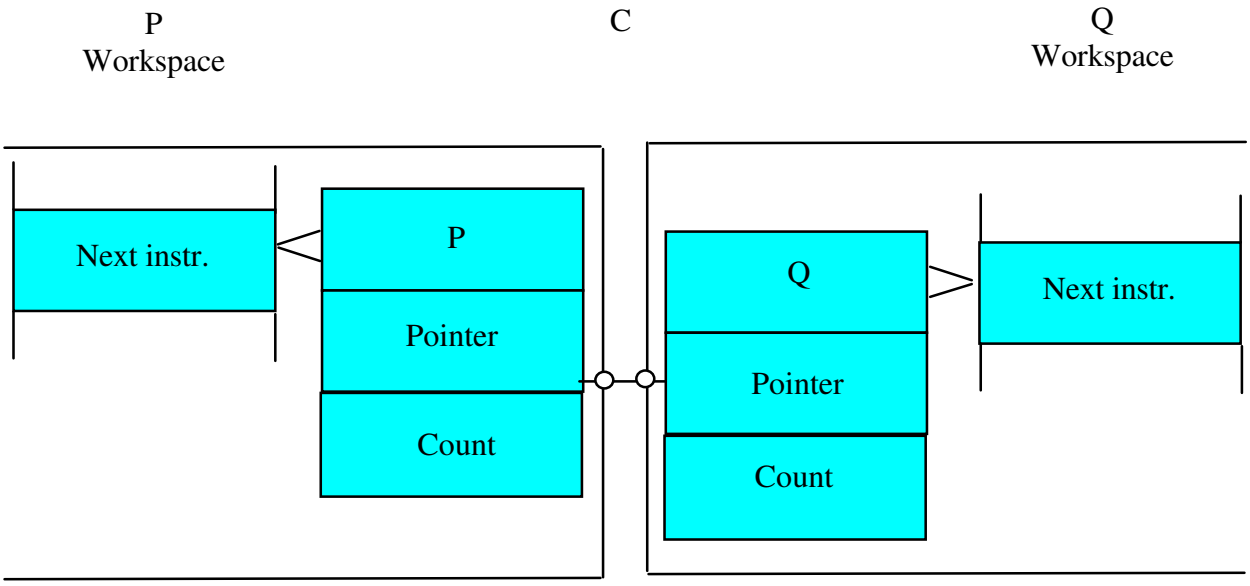
Each link interface uses three registers:

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

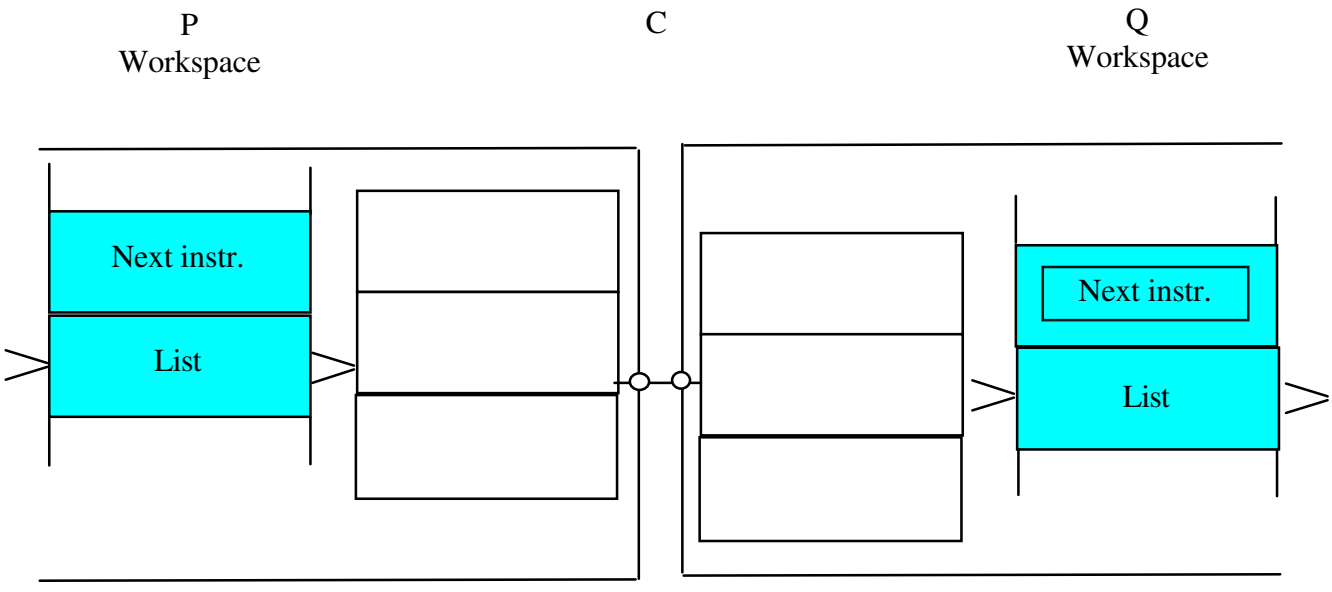
In the next figure processes P and Q executed by different transputers communicate using a channel C implemented by a link connecting two transputers. P outputs, and Q inputs:



When P executes its output instruction, the registers in the link interface of the transputer executing P are initialised, and P is descheduled. Similarly, when Q executes its input instruction, the registers in the link interface of the process executing Q are initialised, and Q is descheduled.



This message is now copied through the link, after which the workspaces of P and Q are returned to the corresponding scheduling list. The protocol used on P and Q ensures that it does not matter which of P and Q first becomes ready.

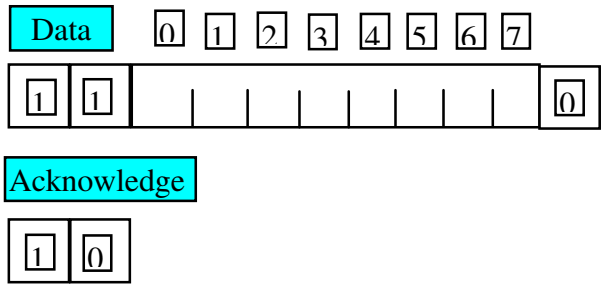


Communication links

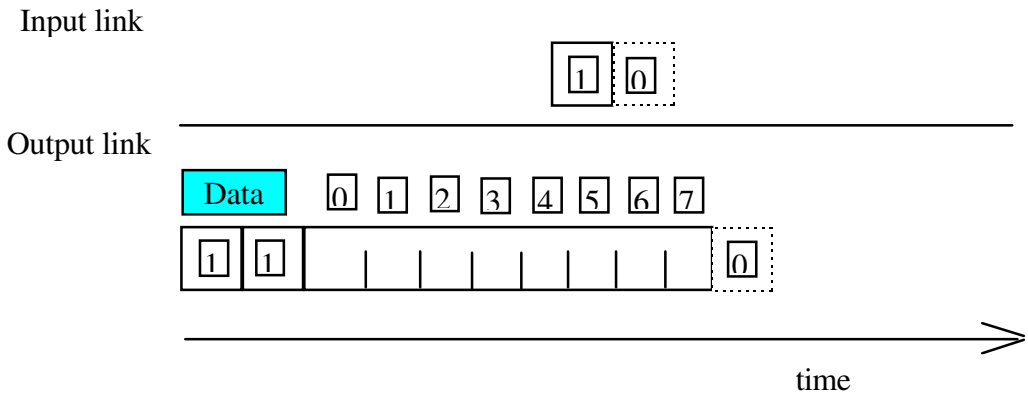
A link between two transputers is implemented by connecting a link interface on one transputer to a link interface on the other transputer by two one-directional wires, along which data is transmitted serially. The two wires provide two occam channels, one in each direction. This

requires a simple protocol to multiplex data and control information. Messages are transmitted as a sequence of bytes, each of which must be acknowledged before the next is transmitted. A byte of data is transmitted as a start bit followed by a one bit followed by eights bits of data followed by a stop bit. An acknowledgment is transmitted as a start bit followed by a stop bit. An acknowledgement indicates both tha a process was able to receive the data byte and that it is able to buffer another byte.

The protocol permits an acknoledgment to be generated as soon as the receiver has identified a data packet. In this way the acknoledgement can be received by the transmitter before all of the data packet has been transmitted and the transputer can transmit the next data packet immediately. Some transputers do not implement this overlapping and achive a data rate on 0,8 Mbytes/sec using a link to transfer data in one direction. However by implementing the overlapping and including sufficient buffering in the link hardware, the rate can be more than doubled to achive 1,8 Mbytes/sec in one directions and 2,4 Mbytes/sec when the link carries data in both directions. The diagram below shows the signals that would be observed on the two link wires when a data packet is overlapped with an acknowledgement.



Link data and acknowledgement format



Owerlapped link acknowledge

Timer

The transputer has clock which ticks every microsecond. The current value of the processor clock can be read by executing a *read timer* instruction. A processor can arrange to perform a timer input, in which case it will become ready after a specfied tiem has been reached.

The timer input instruction requires a time to be specified. If this time is in the ´past´(i.e. ClockReg **AFTER** SpecifiedTime) then the instruction has no effect. If the time is in the ´future´ (SpecifiedTime **AFTER** Clockreg or SpecifiedTime = Clockreg) then the process is descheduled. When the specified time is reached the process is scheduled again.

Alternative

The occam alternative construct enables a process to wait for input from any one of a number of channels, or until a specific time occurs. This requires special instructions, as the normal input instruction deschedules a process until a specific channel becomes ready, or until a specific 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 is ready; the process will be re-scheduled 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.

Floating point instructions

The core of the floating point instruction set was established fairly early in the design of the floating point transputer. This core includes simple load, store and arithmetic instructions. Examination of statistics derived from FORTRAN programs suggested that the addition of some more complex instructions would improve performance and code density. Proposed changes to the instruction set were assesed by examining their effect on a number of numerical programs. For each proposed instruction set, a compiler was constructed, the programs compiled with it, and the resulting code then run on a simulator. The resulting instruction set is noe described.

In the floating point transputer operands are transferred between the transputer´s memory and the floating point evaluation stack by means of floating point load and store instructions. There are two groups of such instructions, one for single lengthnumbers, one for double length. In the description of the load and store instructions which follow only the double length instructions are described. However, there are sinle length instructions which correspond with each of double length instructions.

The address of a floating point operand computed on the CPU´s stack and the operand is then loaded, from the addressed memory location, onto the FPU´s stack. Operands in the floating point stack are tagged with their length. The operand is loaded or is computed. The tags allow the number of instructions needed for floating point operations to be reduced; there is no need, for example, to have both *floating point add single* and *floating point add double* instructions, a single *floating point add* will sufice.

Optimising use of the stack

The depth of the register stacks in the CPU and FPU is carefully chosen. Floating point expressions commonly have embedded address calculations, as the operands of floating point operators are often elements of one dimensional or two dimensional arrays. The CPU stack is deep enough to allow most integer calculations and address calculations to be performed within it. Similarly, the depth of the FPU stack allows most floating point expressions to be evaluated within it, employing the CPU stack to form addresses for the operands.

No hardware is used to deal with stack overflow. A computer can easily eaxamin expressions and introduce temporary variables in memory to avoid stack overflow. The number of such temporary variables can be minimised by careful choise of evaluation order. The algorithm is used to optimise the use of the integer stack of the transputer CPU.

Concurrent operation of FPU and CPU

In the floating point transputers the FPU operates concurrently with the CPU. This means that it is possible to perform an address calculation in the CPU whilst the FPU performs a floating point calculation.This can lead to significant performance improvements in real application which access arrays heavily.

In designing of concurrent systems component such as transputer, it is important to maximise the performance obtained from a given area of silicon; many components can be used together to deliver more performanc. This contrasts with the design of conventionsl co-processor where the aim is to maximise the performance of single processor by use of large area of silicon. As a result, in designing the floating point transputer, the performance benefits of silicon hungry devices such as barrel shifters and flash multipliers were carefully examined.

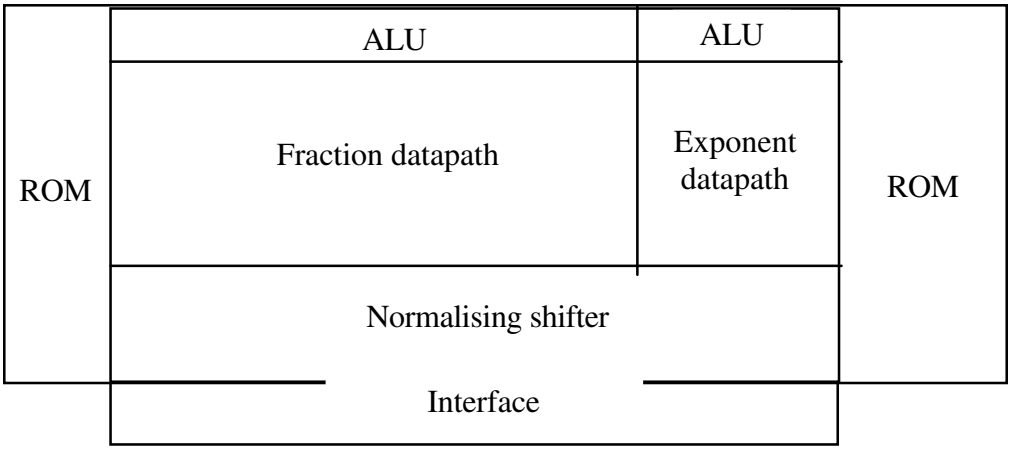
A flash multiplier is too large to fit on chip with the processor, and would therefore necessitate the use of a separate co-processor chip. The introduction of a co-processor interface to a separate chip slows down the rate at which operands can be transferred to and from the floating point unit. Higher performance can therefore, be optained from a slower multiplier on the same chip as the processor than from a fast one on a separate chip. This leads to an important conclusion: a separate co-processor chip is not appropriate for scalar floating point arithmetic. A separate co-processor would be effective where a large amount of work can be handed to the co-processor by transferring a small amount of information; for example a vector co-processor would require only the addresses of its vector operands to be trransferred via the co-processor interface.

It turns out that a flash multiplier also operates much more quickly than is necessary. Only a pipelined vector processor can deliver operands at a rate consistent with the use of such devices. In fact any useful floating point calculations involves more operand accesses than operations. As an example consider the assignment **`y[i] := y[i] + (t*x[i])`** which constitudes the core of the LINPACK floating point benchmark. To perform this it is necessary to load three operands, perform two operations and to store a result. If we assume that it takes twice as loong to perform a floating point operation as to load or store a floating point number then the execution time of this example would be evenly split between operand access time and

operation time. This means that there would be at most a factor two available in performance improvement from the use of an infinitely fast floating point unit!

Unlike a flash multiplier, a fast normalising shifter is important for fast floating point operation. When implementing IEEE arithmetic it may be necessary to perform a long shift on every floating point operation and unless a fast shifter is incorporated into the floating point unit the maximum operation time can become very long. Fortunately, unlike a fast multiplier, it is possible to design a fast shifter in a reasonable area of silicon. The shifter used is designed to perform a shift in a single cycle and to normalise in two cycles.

The next figure illustrates the physical layout of the floating point unit:



Floating point unit block diagram

Consequently, the floating point unit containts a fast normalising shifter but not a flash multiplier. However there is certain amount of logic devoted to multiplication and division. Multiplication is performed three-bits per cycle, and division is performed two-bits per cycle.

The datapath contain registers and shift paths. The fraction datapath is 59 bits wide, and the exponent data path is 13 bits wide. The normalising 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 datapath whilst the magnitude of the shift is associated with the exponent datapath. One further interesting aspect of the design is the microcode ROM. Although the diagram shows two ROMs, they are both part of the same logical ROM. This has been split in two so that control signals do not need to be bussed through datapaths.

Advanced features of transputers (T9000)

The IMS T9000 is the latest member of the transputer family and is designed for embedded systems and high performance computing apllications.

INMOS has used advanced CMOS technology to integrate a 32-bit integer processor, a 64-bit floating point processor, 16 Kbytes of cache memory, a communication processor and four high bandwith serial communication links on a single IMS T9000 chip.

Appendix: Transputer characteristics

Type of trans-puter	Word size (bits)	Internal memory (Kbytes)	Break inst-ruction	Easy link acknow-ledge	On-chip floating point unit	Number of links	Event waiting pin	Data/ /address bus
1	2	3	4	5	6	7	8	9
T212	16							
T222	16							
T225	16							
T400	32	2	Yes	Yes	No	2	Yes	Multipl.
T414	32	2	No	No	No	4	No	Multipl.
T425	32	4	Yes	Yes	No	4	Yes	Multipl.
T800	32	4	No	Yes	Yes	4	No	Multipl.
T801	32			Yes	Yes	4		
T805	32	4	Yes	Yes	Yes	4	Yes	Multipl.
T9000	32-CPU /64-FPU	16					Yes	

1. The T800 was replaced by the T805, the T414 by the T425, the T222 by the T212 and after by T225. It should be noted, that link compatibility is, hoever, maintained across the entire family of transputers.

2. Transputers are avaible in two word sizes. The T2xx models use 16 bits, while the T8xx and T4xx modells all have 32-bit words.

3. Each transputer has a quantity of high-speed SRAM. This accessed within the normal address space of the transputer. The addressing of external DRAM begins at very next location following on-chip memory.

4. Microcoded “break” instruction has been implemented on the last processor models (T805, T425, T400, T225) for use in implementing debuggers.

5. Links on early transputers, the T414 and T212, returned a byte acknowledgement only after the stop bit was received. All later transputers are capable of returning byte acknowledgements midway during receipt of the data bits.

6. On-chip floating point coprocessor units allow the T8xx family of transputers to perform floating point calculations concurrently with other operations. The T4xx processors do have special instructions to help speed up programmed execution of floating point calculations, but they lack the coprocessor.

7. All transputers, except T400, have four bidirectional serial communication links. The T400 has only two. However, since the most common transputer interconnect configuration isa linear chain, where only 2 links are required, the T400 can frequently be employed with no disadvantage.

8. With a transputer, an external interrupt is called Event. All transputer models implement the Event by way of Event Request input signal and an Event Acknoledge output signal. Following assertion of an Event Request, and subsequent recognition of that request by asserted until Event Request is removed. The transputers T805, T425, T400 and T225 all provide an additional output signal called Event Waiting. This signal is asserted by notification to external circuitry that an Event handler process is ready. The assertion of Event Waiting is removed coincident with the assertion of Event Acknowledge.

9. All transputers except the T801 and the T2xx models have multiplexed address and data lines (i. e. they use the same pins for both address and data, as do most microprocessors). The T801 and the T2xx models provide separate address and data pins, however. Separate external memory can be accessed more rapidly. This is the really only useful when external SRAM is utilized in place of denser and more economical DRAM. Due to pin-out incompatibilities, these processor models cannot be interchanged within the same circuit.