Contents

Preface

It’s not often that an engineer sits down to write a book out of frustration. However, after struggling with nonstandard and sometimes cryptic documentation from Inmos, I set out to do so, hoping at first only to crystalize the information in my own head, then later realizing that others may be facing the same problem.

For other conventional microprocessors on the market, documentation is plentiful, but the transputer is a strange bird. It is probably the most successful CPU chip conceived of, designed, and manufactured outside of the United States. Yet it has not attracted the following of technical writers that the latest chips from Motorola, Intel, or other American manufacturers have. Let the others follow the pack. The ability to multitask processes with assembly level instructions or to connect multiple transputers together like "electronic leggo" sets this CPU chip apart from the rest. Whether others will follow its lead remains to be seen, but I believe the transputer has had and will continue to have an important role to play in the area of computing known as "parallel processing."

Parallel processing essentially means more than one computing element working on the same problem. Hopefully, this means the problem can be solved faster (but not always). Usually, the type of problems that can be solved faster involve heavy number crunching, the kind of problems scientists sink their teeth into. In the never-ending search for more computer speed, scientists are rushing headlong into parallel processing and transputers are there waiting for them.

Acknowledgments

This book would not have been possible without the assistance of important individuals. Will Burgett provided great assistance in editing early drafts and making them human-readable. Philip Goward also reviewed many copies and was mercilessly correct in pointing out technical errors or omissions. Colin Plumb provided the inspiration with his editorializing, but excellent technical note on the bizarre nature of transputer assembly language. Wayne Davison cleared the fog on the IEEE 754 Standard by patient answers to my naive questions. Tom Merrow let me apprentice with him on some important tasks involving operating system essentials. And Charles Vollum gave me an extraordinary opportunity to learn about transputers.

Thank you everybody!

Introduction

Why the transputer? The overwhelming market share of microprocessors today are either members of the Intel 80x86 or Motorola 680x0 family. These traditional processors have the numbers behind them, but the transputer reflects an architecture we're going to see more of in the future. The on-chip floating point unit (in the T800 transputer) yields performance faster than microprocessors paired with their coprocessor cousins. Multitasking is the wave of the present as well as the future, and the transputer provides assembler-level support for multiple processes; in fact, the transputer has a process scheduler built into it.

The most important feature of the transputer, however, is its support for communication between processors. From a hardware perspective, the transputer has four bidirectional serial links that can be connected to other transputers; this hardware mechanism can be exploited for parallel processing. From a software perspective, communication to another transputer occurs in the same fashion that communication occurs between two processes on the same transputer, allowing for easier software design of parallel programs.

Parallel programming is where computers are going. Personal computers are still benefiting from the increase in single processor performance, but at the mainframe and minicomputer level, multiprocessor architectures are starting to dominate the scene. The problems encountered in single processor architectures will also eventually arise in personal computers, and parallelism will be used to increase their performance as well.

When Inmos first introduced the transputer, they decided to insulate programmers from the basic instruction set of the transputer. Instead of providing information about the native assembly language, they provided a language called Occam. Occam is a higher level than assember, but lower than most programming languages (such as Pascal or C). Occam allows a programmer to exploit the parallelism and special communication features of the transputer in a well-defined manner, yet it was a new and completely different programming language. Programmers want to program in the languages they are familar with. Inmos eventually relented and began providing information on the underlying instruction set of the transputer. Now, programming languages in C, C++, Fortran, Pascal, Modula-2, and Ada are all available for the transputer, and the list is growing.

Inmos originally designed the transputer for the embedded microcontroller market. There are some features of the transputer that reflect this influence. One example of this is that if certain flags are set in a certain order the transputer will shut down. This may be desirable for a coffee maker, but not a general purpose computer. As more users and designers saw the transputer, however, they began to realize its inherent power as a computing engine, since it was possible to cascade multiple transputers together to solve problems in significantly less time than it would take single processor systems.

So, due to the original product push and the lack of detailed information about the instruction set, transputer-based computers did not emerge at once. Instead, add-on accelerator boards consisting of multiple transputers that could be plugged into existing computers came forth. However, transputer-based computers are starting to enter the marketplace today.

There are three basic models of the transputer: the T212, the T414, and the T800. The T212 has a 16-bit wide register length and is not discussed in this book, although much of what is written about the T414 applies to it as well. By far the more popular models are the T414 and T800 (both with 32-bit wide registers), which are discussed in this book. The basic difference between the T414 and T800 is that the T800 has an on-chip floating point arithmetic engine. There are other minor differences that are noted in the pages that follow.

Most assembly language programmers have already mastered the basics of binary arithmetic and hexadecimal representation of binary numbers, so these subjects are not discussed. Hexadecimal numbers are indicated by the prefix "0x" as in the C programming language. For example, "0xA" means the hexadecimal value "A" (which is equivalent to the decimal value 10).

Inmos invented some new terminology when it produced the transputer, perhaps just to be different. For the sake of consistency, their terminology is used where it differs with conventional computerese. The most agregious terms are Inmos's use of "workspace" for "stack" and "instruction pointer" for "program counter". There is also the "workspace pointer," which everyone else in the computer industry would call the "stack pointer."

The basic organization of the book is as follows:

Chapter 1 Parallel processing
Chapter 2 Transputer architecture
Chapter 3 The instruction set
Chapter 4 Programming the transputer
Chapter 5 Newer Transputers
Instruction Set Reference Detailed descriptions about each instruction
Appendixes Miscellaneous useful information
Suggested Readings List of useful references

This book is intended to be useful for reference even after it has been read thoroughly. The information does not depend on any one operating system or assembler. Programming examples have sufficient comments to make them easily portable. It is hoped that the dissemination of this information will generate further excitement and interest in the transputer.

Introduction to Parallel Processing and the Transputer

What is parallel processing? Parallel processing is the ability to perform multiple computations simultaneously. In some multitasking operating environments, it may appear that you are computing two things at the same time, but in reality the computer is only doing one thing at a time, switching between tasks so quickly that it appears to be doing more than one thing at a time when it isn't. True parallel processing involves physically separate computing engines, each chewing on some computation.

Since microprocessors were first invented, engineers and hobbyists have been envisioning a computer system consisting of many processing elements. However, advances in the performance of single processor units has been so great in such a short time that many engineers and scientists believe that uniprocessor computers will always provide the best performance and that higher performing computers are only a generation away. Yet parallel processing is starting to emerge as a way of bringing more computing power to bear against problems that even the fastest computers have found intractable. Essentially, parallel processing is a mechanism to allow computers to calculate faster.

Although it may seem perfectly rational that two computers working together on a problem should be able to solve it faster than one, it's not always the case. Consider the following C program fragment, which multiplies each element of a 1000 element integer array (called a) by 100:

```
for (i = 0 ; i < 1000 ; i++)
a[i] = a[i] * 100;
```

Each element of this array could be multiplied in parallel, and if you had 1000 processors you could parallelize this to all 1000 of them like so:

```
processor 1 a[1] = a[1] * 100
processor 2 a[2] = a[2] * 100
processor 3 a[3] = a[3] * 100
. . .
. . .
. . .
```

However, suppose the program loop was:

```
for (i = 1 ; i < 1000 ; i++)
a[i] = a[i] * a[i - 1]
```

This is not so easy to parallelize, since the current computation depends on the result of the previous one. That is why it is important for programmers to structure their programs in a manner so it is possible to take advantage of parallelism.

There are two basic models of parallel processing: shared memory and distributed memory. Shared memory processing is where multiple processors are connected to the same system memory. This scheme is used in many minicomputers and mainframes today. An advantage of shared memory is that multiple processors can use the shared system memory as a fast way to communicate and exchange data. The main problem of shared memory is that several processors can attempt to access the same memory location at the same time. When this happens, the requests have to be serialized, that is put into a sequential ordering. The contention that arises due to serializing is referred to as "memory contention." This slows down overall performance. Thus, shared memory systems tend to require faster memory so that processors will not have to wait too long before having a memory request satisfied. For shared memory in general, the more processors, the faster the memory required.

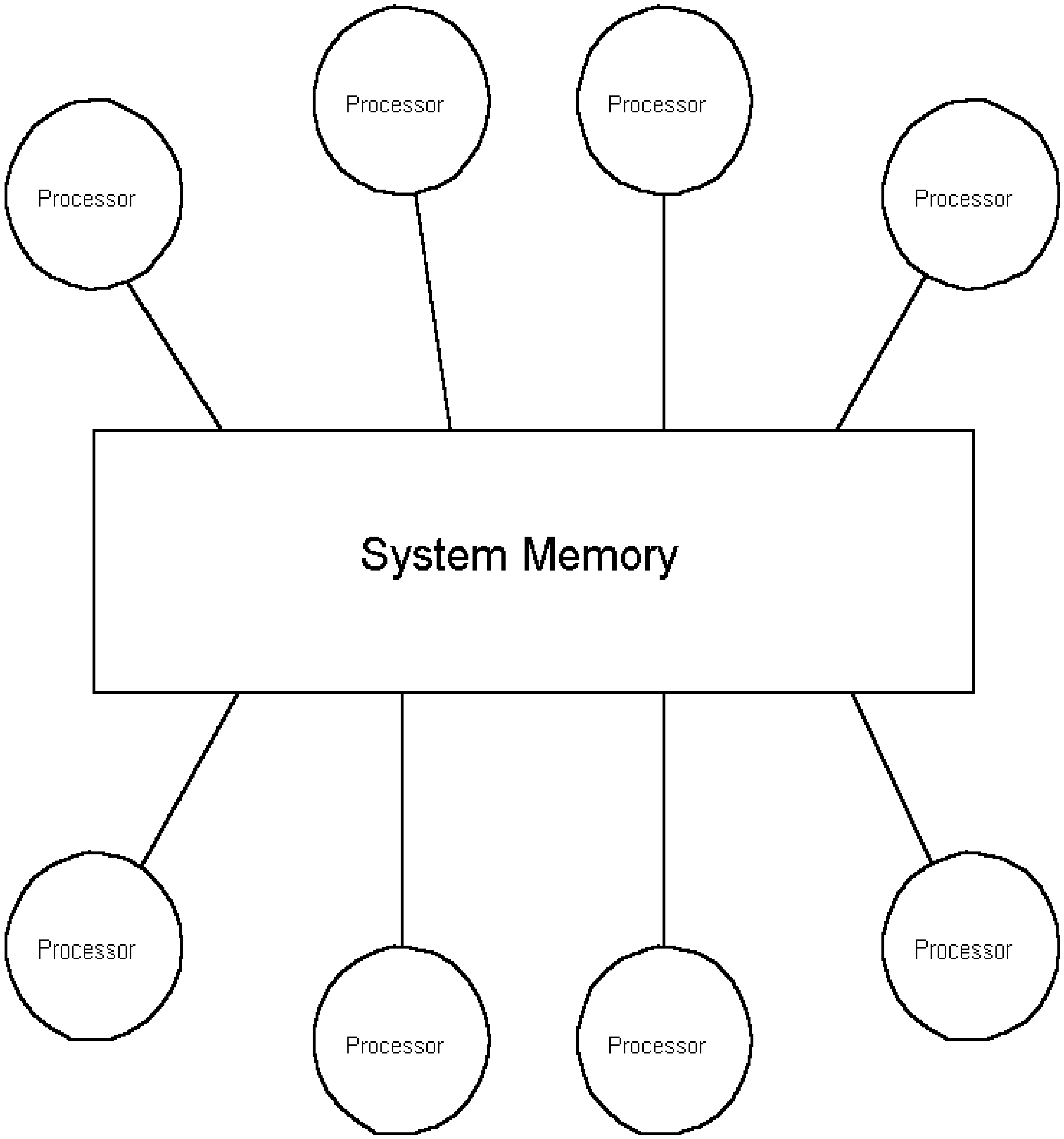

Figure 1-1 An example of a shared memory architecture.

The alternate memory model, distributed memory, is where the transputer fits in. In a distributed memory model, each processor has its own local memory. The key question in a distributed memory model is What is the nature of the communication between processors? There is no common "memory pool" for communication in a distributed memory environment, so instead the processors must have some other method of communication. On the transputer, this method is to use serial "links," which act very much like serial ports on personal computers. Each transputer has four bidirectional links that can be connected to links on other transputers. Each link provides a flow of data from one processor in the system to another. Thus, like building blocks or leggo, one can connect many transputers together into various configurations. Such a mechanism is typical of a distributed processing system.

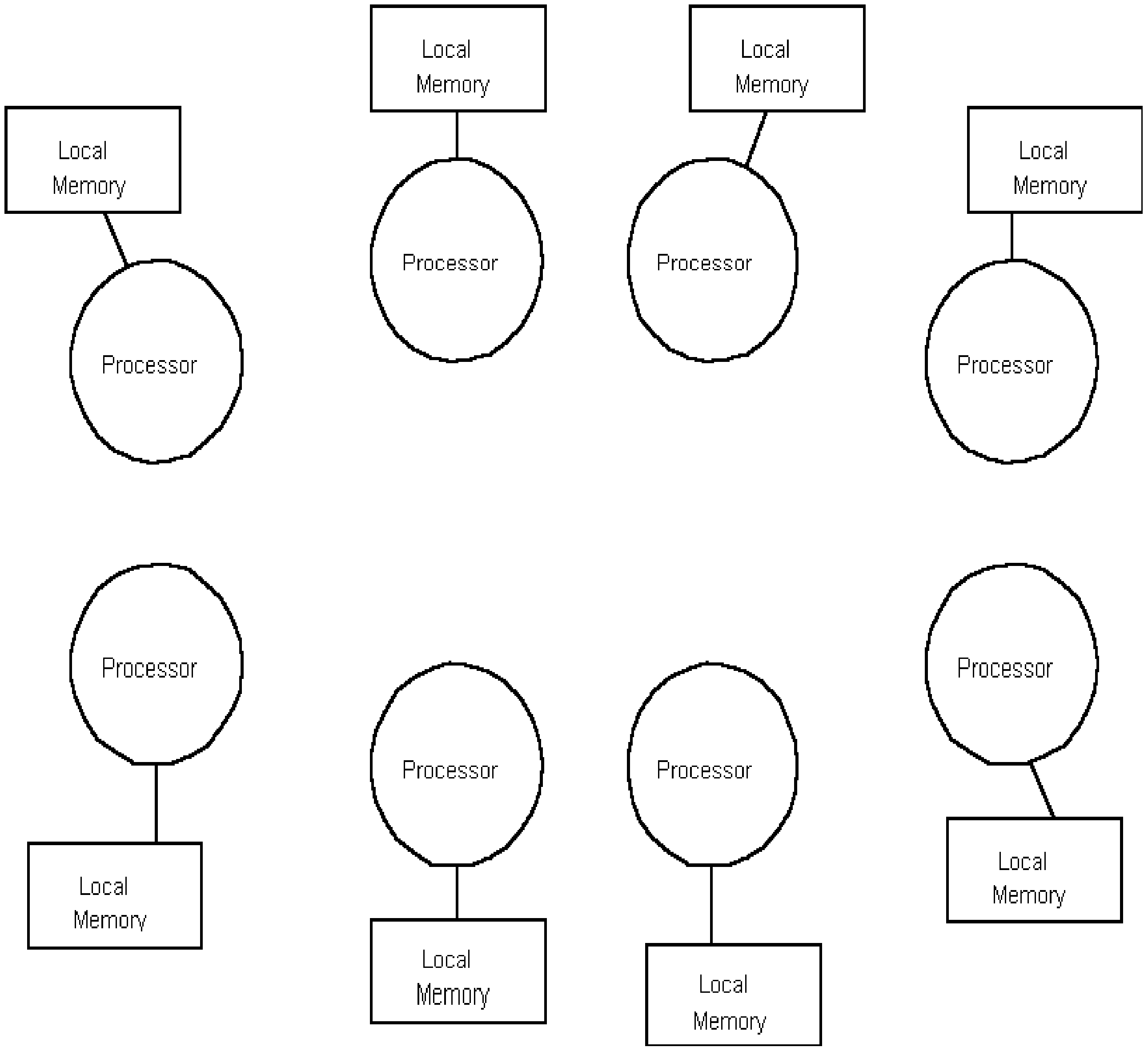

Figure 1-2 An example of distributed memory.

One popular software mechanism for using distributed memory as a kind of global shared memory is "Linda." Linda was developed by David Gelernter and Nicholas Carriero at Yale and is essentially a set of communication primitives that are added to an ordinary computer language. In a Linda program, a programmer places data into and reads or removes data from an abstract shared memory area called "tuple space." The data objects inside the tuple space are referred to as "tuples." A tuple space can exist in either shared or distributed memory. This abstraction has the advantage of portability. In particular, Linda programs can run on either shared or distributed memory machines. Underlying system libraries implement the machine-dependent communication functions upon which Linda relies. Linda programs ignore the underlying processor topology and aim at a higher level of abstraction for interprocess and interprocessor communication.

However, for most distributed memory environments, the processor topology is the main consideration. The next question to ask then is how to connect multiple transputers. With four links per transputer, there are various topologies, or network configurations, that are possible. It is possible to configure four transputers in a "ring," with each processor connected to two others, much like children holding hands to form a ring. Ring topologies are common in computer networks, since they only require an input line from one system and an output line to another.

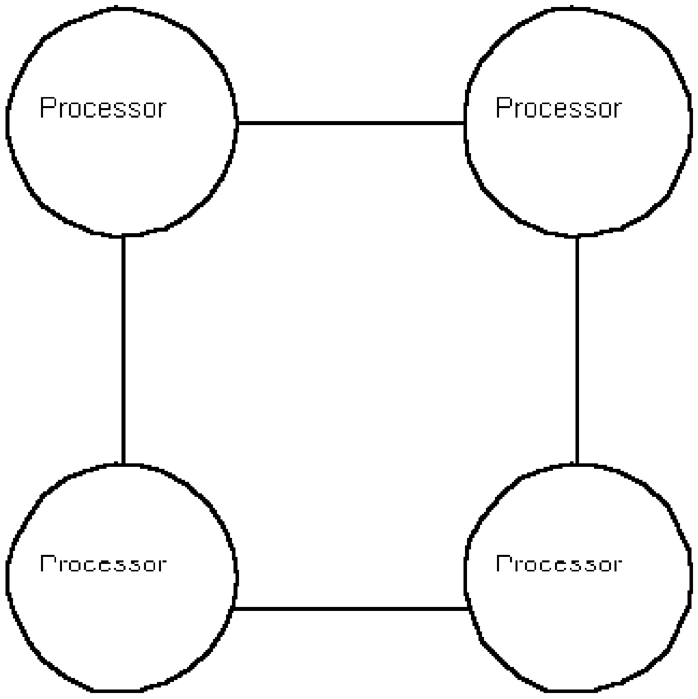

Figure 1-3 Four processors configured in a ring.

However, the transputer has four links, so it is possible to provide more interconnections between processing elements. If another link was used to connect each processor, another configuration would be possible. When each processing element in a network of processors is connected to all the other processors in the network, the network is said to be "fully-connected."

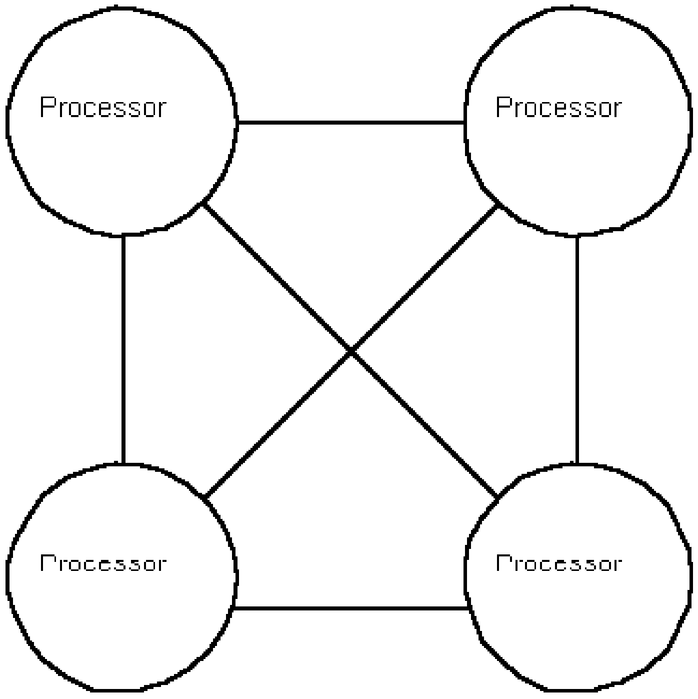

Figure 1-4 Four processors configured in a fully-connected ring.

Transputers are not limited to ring topologies. In general, the topology of a computer network is determined by the communication needs of the application program. These needs reflect the algorithmic solution for the problem it is trying to solve. For some applications, it may be desirable to configure a series of transputers into a tree structure.

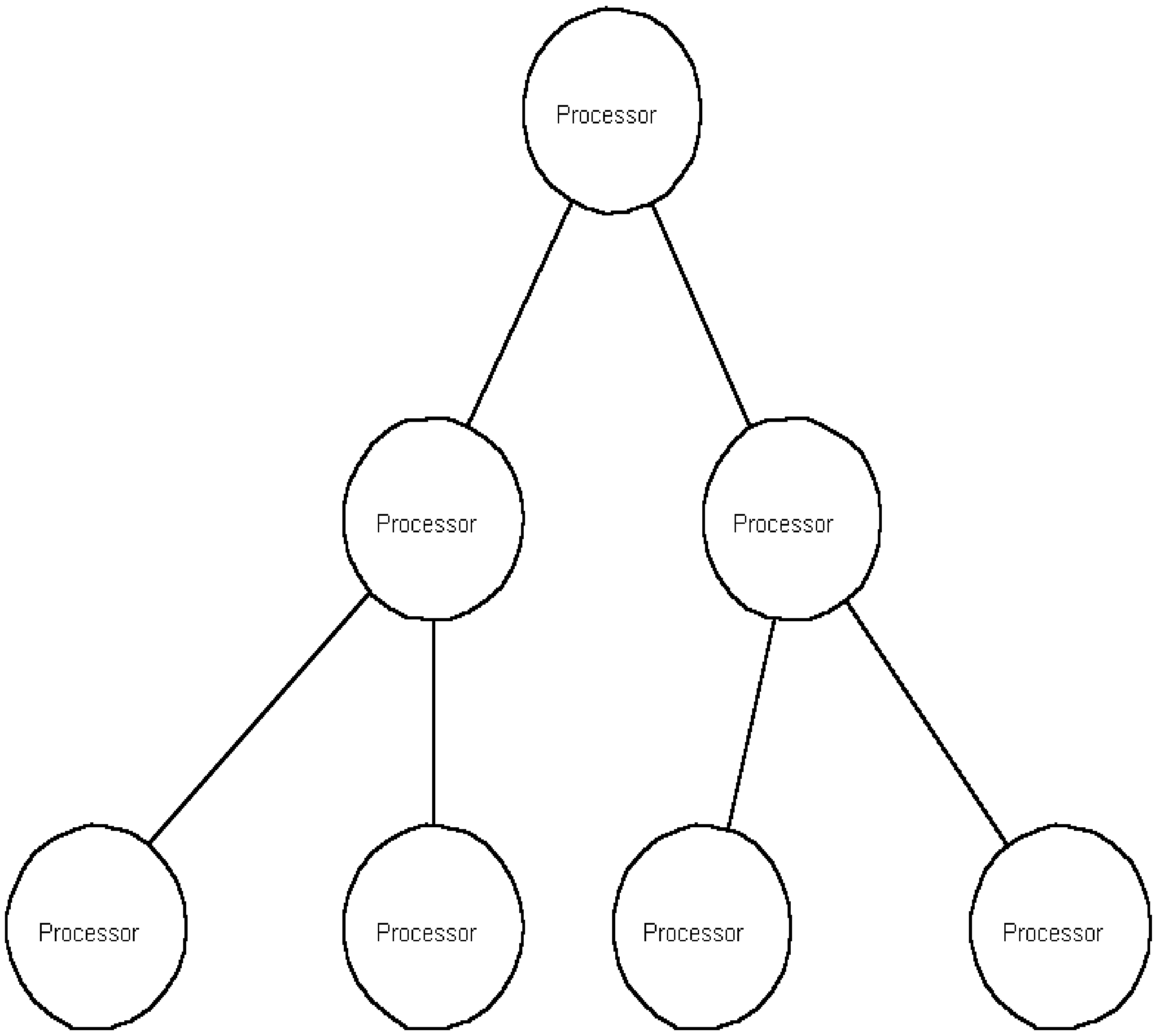

Figure 1-5 Seven processors configured into a binary tree.

Key questions surrounding processor topologies are: what is the nature of the problem to be solved, how can the problem be parallelized, and how will data migrate from processor to processor? The data migration issue is perhaps the easiest to focus on. For instance, consider a problem where each computing engine acts like a worker on an assembly line, performing some computation on the problem, then passing its result onto another worker on the line. The data drives the computation. This scenario is referred to as a "pipeline," where data is input at one end, then piped through multiple processors, each of which performs some intermediate step towards the final result. The last processor, presumably, achieves the final result and outputs it.

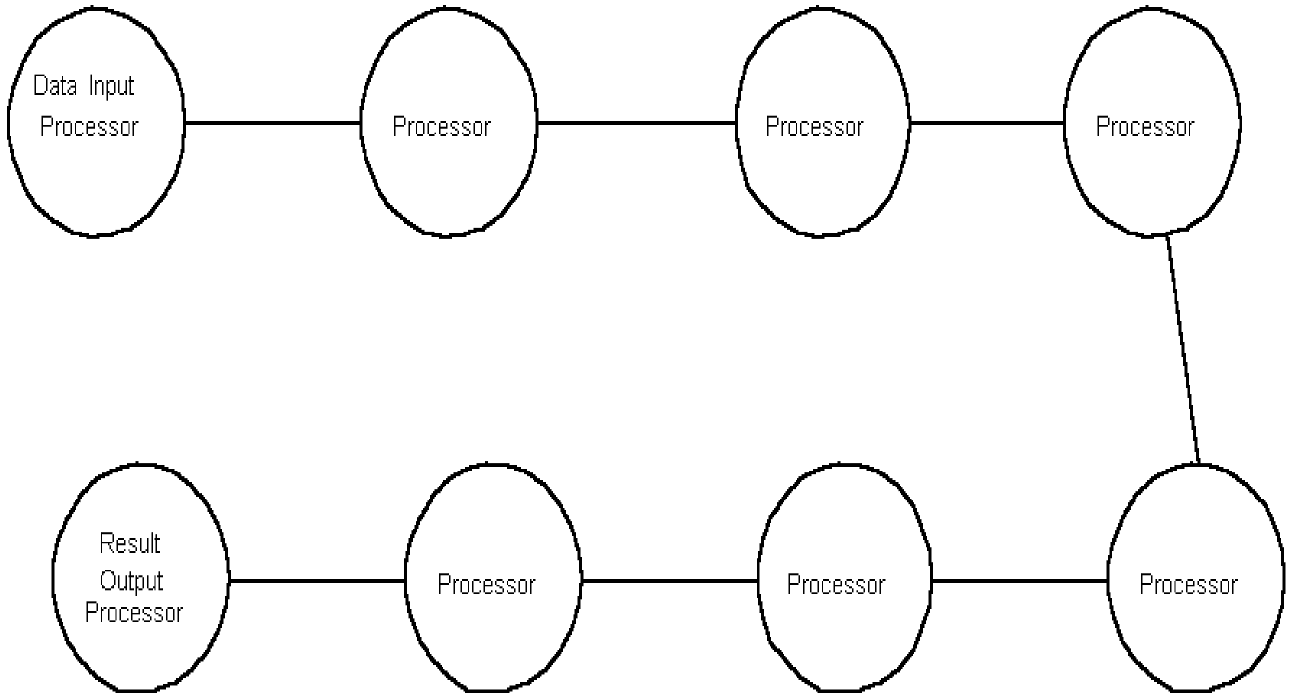

Figure 1-6 Eight processors configured in a pipeline.

In general, computers can be broken down into one of four categories:

SISD	single instruction, single data stream
MISD	multiple instruction, single data stream
SIMD	single instruction, multiple data stream
MIMD	multiple instruction, multiple data stream

A single instruction, single data stream (SISD) computer is a traditional single central processing unit (CPU) computer with only one processor working on one set of data at a time. Ordinary personal computers fall into the SISD category. A multiple instruction, single data stream (MISD) computer has multiple processors, each of which executes instructions on a single stream of data. Fault-tolerant computers, which duplicate hardware functionality, are an example of this category, as a fault-tolerant system typically has at least two processors each performing the same computation on the same piece of data, then if the primary unit fails, the backup unit takes over. Single instruction, multiple data stream (SIMD) computers are typically "arrays" of processors that each execute the same instruction on a different set of data. A grid of processors performing a matrix multiplication (hence the term array) would be an example of SIMD computing; each processor performs the same multiplication instruction, but on different elements of the matrix. Multiple instruction, multiple data stream (MIMD) computers are where each processor executes a different sequence of instructions on a different set of data. A computer network can be thought of as an MIMD system (since it is usually composed of SISD computers all executing different instructions on different data).

The power of the transputer design is that it is flexible enough to be configured into any of these categories. A computer architectural design can now be fitted to the exact problem you are trying to solve. This provides a performance boost for programs previously constrained to run on a traditional single-processor computer architecture.

Each of the four serial links on a transputer can be connected to a serial link on another transputer. This allows you to construct a network of transputers just as you could connect a network of personal computers together. In the same way that computer network topologies vary, so too can transputer network topologies vary. Much of the analysis that applies to computer networks also applies to transputer networks. The main idea is to get additional computational power by using multiple processors together on a problem. The unique aspect of the transputer that makes this possible is its built-in ability to communicate with other transputers over its serial links. Since each transputer uses its own local memory, the memory is said to be "distributed" over the processor network.

Transputers, in short, are distributed memory processing engines that can be configured into a variety of architectures.

The architecture of the processor network can be tailored to the problem to be solved. Usually the network topology is fixed; however it is possible to construct transputer networks that can be reconfigured on the fly, that is as the processor is running a program. In addition to the serial links, there are assembly language level instructions designed for communication between transputers. The transputer is unique among popular microprocessors today in providing both hardware and software level support for parallel processing.

Transputer Hardware Architecture

There are two strong influences in the transputer’s design philosophy and one minor one: parallel processing and RISC (Reduced Instruction Set Computing) concepts have strongly shaped the transputer, while embedded microcontroller notions have also had an impact. The microcontroller influence is seen most strongly where the transputer can shut itself down if a computation produces an error. This may be necessary if a coffee pot starts to overflow, but no modern computer system really runs without allowing computation errors of some kind. Another influence that stems from embedded microcontrollers is that transputers are capable of servicing external interrupts quickly. This is primarily due to the small amount of state associated with each transputer "process" or thread of execution, thus changing the current process, also called "context switching," is relatively inexpensive.

Note: The term "thread" refers to a single sequence of instructions being executed. If you follow the consectutive instructions a program executes, then you are following the "thread of execution" for the program. Some operating systems, like MS-DOS, can only support a single thread of program execution. Other operation systems, like Unix, can support multiple threads of program execution, implying that more than one program can execute at the same time.

The major features of the chip, however, are inherent support for parallel processing and RISC. The parallel processing influence gave the transputer communication mechanisms that enable multiple transputers to behave as a single computing engine with each individual component transputer passing necessary messages (about a computation) to another. The RISC influence has made the instruction set sparse, but not small. This means that due to the relatively large number of special features on the transputer (built-in chip-to-chip communication channels called "links," on-chip timers, assembler level multiprocessing, etc.), the instruction set is not small, but each instruction exercises only a narrow aspect of a particular hardware feature. In this sense the transputer comes closest to the RISC concept. For instance, unlike many new 32-bit microprocessors the transputer does not provide support for indexed addressing in a single instruction. The same functionality exists, but instead of having the index in one register, the base address in another, and a single instruction to compute the effective address and fetch its contents, the transputer requires several instructions to perform the effective address computation and fetch. In general, this is because the transputer’s instructions are less functional than conventional or CISC (Complex Instruction Set Computing) instructions. The advantage of the transputer’s RISC approach is that the instructions themselves, while less powerful, execute much faster.

In addition to these software-oriented features, the transputer has built into it a programmable external memory interface and thus requires very little support circuitry. The most costly part of a transputer-based system is typically the RAM.

Inmos introduced the transputer in 1984. The INMOS transputer family is currently composed of three different processors:

- The 16-bit T212,
- The 32-bit T414, and
- The 32-bit T800.

The T212 is not discussed in this book, but much of what is said about the T414 applies to it. (Note that the T212 has 16-bit registers, thus a word length of 16 bits or 2 bytes, while the T414 and T800 have 32-bit registers, thus a word length of 32 bits or 4 bytes.) The T800 is essentially a T414 with an on-board floating point co-processor engine and is available in 20, 25 or 30 MHz versions. The T414 is currently available in a 15 or 20 MHz version.

The 20 MHz versions of the T414 and T800 can provide for about 15 MIPS (million instructions per second) performance. However, since transputer instructions are more primitive (i.e., less functional) than conventional microprocessor instructions, this translates to a relative performance of about 3 to 5 MIPS (when compared with other conventional microprocessors).

Register Set

The transputer has three general purpose registers (forming a stack), a program counter (called the "instruction pointer" in Inmos parlance), a stack pointer (called "workspace pointer" by Inmos), and an operand register.

The general purpose registers on the transputer form a three deep stack (and operate in a similar fashion as RPN calculators, like those manufactured by Hewlett-Packard). Values are pushed onto the register stack and popped off it. These three integer registers are named A, B, and C. (Note: On the T800 there are three additional registers named FA, FB, and FC for floating point arithmetic.)

If you push a value into A, first B will be pushed into C and the contents of A will be pushed into B; then A is set to the new value. Similarly, if you pop the register stack, A receives the contents of B and then B receives the contents of C. After popping the register stack, C is undefined. (Although it has been observed that the value of C is replicated, that is, it retains its value after the register stack is popped. Some transputer compilers take advantage of this feature, but Inmos warns that it may be discontinued in the future and software should not rely on it.) The A, B, and C registers are 16 bits wide on the T212 and 32 bits wide on the T414 and T800.

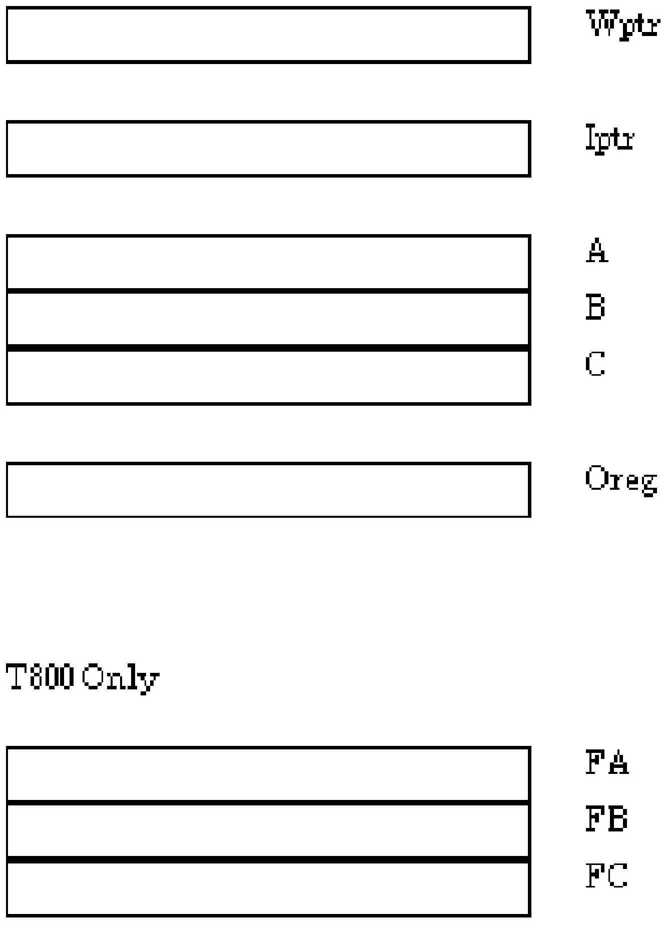

Figure 2-1 Transputer register set.

You always access the register stack through the A register. When you pop the value in register A, its contents will come off the stack (into memory), while those values of B and C move down (into A and B). Conversely, when you push a value onto the stack:

1. The C register receives the value in the B register.

2. The B register receives the value in the A register.

3. The new value is placed in register A.

In addition to the register stack, the transputer has three other major registers:

- The instruction pointer (program counter),
- The operand register, and
- The workspace pointer (stack pointer).

The operand register serves as the focal point for instruction processing. All transputer instructions are one byte long and typically execute in one or two clock cycles. The four upper bits of a transputer instruction contain the operation to perform, the four lower bits contain an operand for that operation. There are special transputer instructions to form longer operands. These are called "prefix" instructions, since typically they prefix (or come just before) other instructions that use the operand register.

A prefix instruction loads its operand field into the four least significant bits of the operand register, and then shifts the operand register left four places. Consecutive prefix instructions can be used to fill the operand register with any value, one nibble at a time. By using the contents of the entire operand register as an operand along with the operand field of the current instruction (its lower four bits), any 32-bit value can be used as an instruction operand.

Register Notation and Summary

There are six registers on the transputer that are used in virtually all programs. They are Iptr, Wptr, the A-B-C stack, and Oreg.

Iptr - Instruction pointer. Normally referred to as a program counter, Inmos chose to call it the instruction pointer. This contains the address of the next instruction to execute.

Wptr - Workspace pointer. Again, this is normally called a stack pointer, but Inmos chose to call it a "workspace pointer." The transputer instruction set is strongly geared for using memory locations that are at small positive offsets from Wptr for the storage of variables. Note that loading from an absolute address requires two instructions, while loading from the workspace (stack) only requires one.

A, B, C - General purpose registers that form a three deep stack.

Oreg - Operand register. This register is used to assemble small operands from multiple instructions (e.g., prefix instructions) into one long operand.

Flags

There are three flags in the transputer: Error, HaltOnError, and FP_Error.

Error - The transputer has an error flag similar to an overflow flag in conventional microprocessors, except that its "sticky," that is once set it stays set until it’s explicitly cleared. The Error flag value is also sent out over an external pin of the same name. This harkens back to the microcontroller influence in the transputer, so other transputers can detect that one has made a computation error and take some appropriate action. This is probably never used in reality.

HaltOnError - This flag, when set, halts the entire transputer if Error is set (provided Error was previously clear). It is probably wise to leave this flag clear, otherwise you could shut down the entire transputer if Error is set. Again, this is the microcontroller influence that says, if you’re controlling some device and you make a mistake, shut everything down.

FP_Error - This flag is the floating point error flag and behaves like the Error flag does, except that it is used for floating point errors. It is only present on the T800.

T800 Floating Point Unit

There is a measure of support for IEEE floating point arithmetic in both the T414 and T800 transputer. The T414 has several instructions designed to assist single-length (32-bit) floating point arithmetic. These instructions are implemented in microcode on the chip.

The T800 does not have those instructions, instead it has something much more powerful, an on-chip floating point arithmetic engine. Most conventional CPUs, such as the Intel 80x86 and Motorola 680x0, have sister coprocessor chips for floating point arithmetic (the Intel 80x87 and Morotola 68881 and 68882). Inmos has done them one better and incorporated the floating point unit on the same chip in the T800. The floating point performance of the 20 MHz T800 is in the 1.5 to 2.0 Megaflop (million floating point operations per second) range. This is roughly 50 percent faster than a 16 MHz 80386/80387 combination.

The T800 floating point unit supports IEEE floating point arithmetic for both single length values (32-bit) and double length values (64-bit). The floating point registers are organized in a three deep stack, like the integer registers. The floating point engine and the integer unit can act autonomously to a degree, so while the integer unit is computing an address, the floating point unit can perform some numerical computation simultaneously. This "overlapped execution" is the one way to maximize transputer performance (and some compilers exploit this).

The floating point unit registers are called FA, FB, and FC. These registers are 64 bits long and can hold either single (32-bit) or double (64-bit) precision floating point values. There is no way to query the precision of the number that one of these registers contains (i.e., is it a 32-bit or 64-bit number?), so it is necessary for the programmer to be aware of this limitation when writing code.

Instruction Format

Transputer instructions are a single byte in length, although one instruction is merely "execute the contents of the operand register as an instruction," which is the way around a one-byte instruction limitation. Instructions that are one byte in length are called "direct instructions." Instructions that use the "execute the contents of the operand register" trick and are more than one byte in length are called "indirect instructions." Indirect instructions are composed of prefix instructions that load a value into the operand register, followed by an operate instruction that triggers the actual execution.

The idea behind single-byte instructions is to improve the effectiveness of the instruction pre-fetch, which in turn improves processor performance. Since there is an extra word in the processor’s pre-fetch buffer, the transputer rarely has to wait for an instruction fetch before proceeding to decode and execute the next instruction. Since transputer instructions are only one byte in length and one word is fetched from memory at a time, transputers are effectively equipped with a four-byte instruction cache.

The most commonly used instructions were chosen to be the direct instructions, since direct instructions are the most compact (one byte) and execute the fastest. However, it is obviously impossible to code all necessary instructions into a single byte. As a result, some operations in the transputer require more than one instruction.

As mentioned earlier, the upper four bits of a byte determine the instruction code, so this allows for 16 possible instructions. Only 13 of these are commonly used by assembly language programmers (for example, load, store, jump, and call). Each of these instructions requires an operand that is partially supplied by the four-bit operand field.

The other three instructions (not commonly used by assembler programmers) are the two prefix instructions, which are used to build operands longer than four bits, and the operate instruction, which says "execute the contents of the operand register as an instruction." Assemblers insulate programmers from having to build longer instructions out of these more primitive assembler instructions.

More often than not, four bits per operand is often insufficient, so the transputer has two special instructions called PFIX and NFIX that are used to load the operand register. When one of these instructions is executed, the operand register is shifted four bits to the left and then the instruction’s operand (from the NFIX or PFIX) is placed in the four bits that were cleared by the shift (the lowest four bits or least significant nibble). Using these instructions, you can load the operand register with any 32-bit value (the word length of the transputer).

Data Organization in Memory

The transputer architecture provides inherent support for 32-bit integer or long values, as well as byte or character values. It is more difficult to provide support for the short 16-bit values, such as in the C programming language. Most instructions are geared around word (32-bit) quantities; however, there are two instructions LB (load byte) and SB (store byte) that support character-sized (8-bit) data. Support for 16-bit quantities must be emulated by using the byte instructions and shifting; there is no instruction level support for short (16-bit) quantities. For example, to load a 16-bit quantity, first you load one byte, then shift left eight bits, and then add the next byte. (Note: These comments apply to the T414 and T800, which are 32-bit word length machines, not the T212, which is a 16-bit word length machine. On the T212, support for shorts is inherent, support for "longs" or long-types is not.)

The transputer is little endian in nature, meaning that the least significant bytes of data are stored first in the lower memory addresses. This is like the Intel 80x86 architecture and is the opposite of the Motorola 680x0 architecture.

Addresses

The addresses on the transputer are one word wide, yielding a continuous address space of four gigabytes for the 32-bit T414 and the T800. The address bus is multiplexed with the data bus. An interesting facet of the transputer is that addresses are "signed," that is, if the most significant bit of an address is set, the address is negative and actually precedes any positive address.

Transputers have two basic ways of addressing data in memory: (1) stack operations and (2) random access methods. Both techniques are valid, but stack operations are faster, whereas random access methods are more typically convenient for the programmer. In either case it is faster to access words in memory rather than bytes. Important: Many instructions dealing with memory require addresses on word boundaries (i.e., addresses that are multiples of four), otherwise unpredictable values can result from a memory access.

The transputer also has instructions geared for moving entire blocks of memory quickly, for copying arrays or to provide a graphics "bit blit" function.

On-Chip Memory

Transputers have fast on-chip memory, which is mapped into the low end of the transputer’s address space. There are 2K bytes on the T212 and T414, and 4K bytes on the T800. This internal memory can be accessed by the transputer without wait states, whereas at least two processor cycles are added for external memory accesses. A program that fits into this on-chip memory will execute an instruction in the transputer’s cycle time. In the case of the 30 MHz version of T800, this is a 33 nanosecond cycle.

The on-chip memory is considered to start at the bottom of the address space, which corresponds to the largest negative number, namely 0x80000000. A few values in the on-chip memory are used for link communication channels and process swap information.

A typical use of on-chip memory is for register variables (since the transputer has only three registers). An on-chip register variable can be accessed in a single cycle, while variables held in external RAM require extra cycles to be latched and read. Although code executing out of on-chip memory will experience a speedup, empirical evidence seems to indicate that placing the workspace into on-chip RAM provides the biggest performance boost to most programs by speeding up data references.

Each thread of execution on the transputer is called a "process." Every process has a stack (i.e., a workspace) associated with it. The workspace associated with the currently executing process lies above a base address (i.e., in higher, more positive, memory). The transputer’s workspace pointer register points to the base address of the workspace for the process currently being executed.

Memory locations within a small offset from the workspaces are "local," and they can be accessed extremely fast by using the load local (LDL) and the store local (STL) instructions. If a memory location is within 16 words of the workspace pointer, it can be accessed with a one-byte instruction. When the workspace in a process resides in the transputer’s internal, on-chip memory, workspace data can be accessed in one processor cycle (50 nanoseconds on a 20 MHz T414 or T800). In this case, the workspace acts like a large set of registers on a conventional microprocessor. In fact, some transputer compilers implement register variables by simply moving the variable to one of the first 16 locations on the workspace.

Microcode Scheduler

Many operating systems allow you to "fork" or "spawn" or "exec" multiple processes (threads of execution) with a program;however, it is very rare to be able to do this in a native assembly language. The transputer has inherent support for multitasking via an on-chip microcoded scheduler. Certain instructions add new processes to the scheduler’s queue of processes; other instructions cause a process to be removed from that queue. The task scheduler can schedule processes all on its own. Task switching in the transputer takes less than one microsecond. The scheduler executes processes from the queue in an endless loop, that is, once it reaches the end of the queue, it proceeds to the start of the queue for its next process to schedule. There are two separate queues the scheduler manages: one for high priority processes and one for low priority processes.

Processes running on a transputer can have any one of the following status conditions:

- Running,
- Waiting in a queue to be executed,
- Waiting for input,
- Waiting for output, and
- Waiting for a certain timer value.

If the status is "running" or "waiting in a queue to be executed," the process is active**.** If the status is any one of the remaining three conditions, the process is inactive.

Transputer processes can run in either low or high priority. A high priority process can interrupt a low priority process at any time. The high priority process will continue running:

1. Until it is completely finished;
2. Until it has to wait for a communication; or
3. Until it has to wait for a certain timer value.

A low priority process will always be preempted by any high priority process, for example if a high priority process is queued up for the scheduler, the running low priority process will be interrupted and the high priority process will begin executing. All high priority processes must terminate or be waiting for input or output before any low priority process will be scheduled.

Before it has to give way to a high priority process, a low priority process finishes executing its current instruction and then its process state (register values, flag values, etc.) is saved in reserved memory locations. Depending on what instruction was being performed at the time of interruption, the task switch time from high to low priority can take anywhere from 19 to 58 processor cycles (1 to 3 microseconds on a 20 MHz T414). When a process that has high priority must wait, the low priority process that was interrupted is rescheduled, but only if there are no other active high priority processes. This switching of processes from one to another is often referred to as a "context switch."

Typically, a high priority process is tied to the Event pin to handle external interrupts. This high priority process then "wakes up" when there is an external event it needs to service. If another interrupt occurs over the Event pin, another high priority process is spawned to service that event. This is how a transputer responds to external interrupts by creating a process to handle each interrupt.

Scheduling is done differently for processes of the same priority. Descheduling one process will only occur when a certain type of instruction has been executed. These instructions are also known as "descheduling points," since it is a point in a program where the current process can be descheduled and another can be scheduled to run. Furthermore, almost no process state is saved on a switch to another process of the same priority. In particular, the values in the A, B, C register stack are not preserved when another process of the same priority level is scheduled.

So you should be careful when programming a transputer that the register stack doesn’t have any information that can be lost when you reach one of these special instructions that can cause the current process to be descheduled. This does place an additional burden on the programmer or compiler writer, but while the transputer stores much less process information during a same priority to same priority context switch, this makes process switching much faster. Remember that when a process is descheduled by another process of the same priority, the integer stack values are "lost," however, the integer stack is saved when a high priority process interrupts a low priority process.

Also remember that only one process is ever running at any given instant of time. The scheduler is in an endless loop of saving the state of one process, transferring execution to another process (to run), waiting as the process runs, then interrupting it and saving its state so it can transfer execution to another process, and so on. The rapid switching of the CPU between processes produces the illusion that multiple programs are running at the same time.

Communication

An important concept for parallel programming is the ability to connect multiple transputers together to form a network of processors. Connections between processors allow transputers to communicate information and share data. This means that more than one CPU or "computing element" can work on a problem, and hopefully the problem can be solved faster than it would be with only one computing element. There are built-in hardware connections to allow transputers to communicate with each other.

Each transputer has four bidirectional one-bit wide serial ports called "links." A link on one transputer can be connected to a link on any other transputer. Inmos developed a general technique for one process to communicate to another process. Each such communicating process writes to and reads from memory addresses called "channels." Channels can be used for communication between processes inside a single transputer (in that case, the channels can reside almost anywhere in memory), or channels can be used to communicate over links to a process on another transputer (in which case certain fixed memory addresses in on-chip RAM must be used).

Typically, when a transputer runs several processes at the same time, these processes will want to communicate. To accommodate this need, the transputer offers use of internal channels for communication. These internal channels are used exactly the same way that external channels (i.e., links) are. The only difference is that any memory location will work in place of the reserved locations used by links.

Using internal channels gives you a sophisticated way to synchronize processes without semaphores or other fancy software tricks. For instance, examine the case where two processes need to share data. One process may be ready to send data to the other process, but the receiver process isn’t ready to receive it yet. The sending process will execute an OUT instruction over a channel and be descheduled until the receiving process is ready (i.e., executes an IN instruction). When the receiving process executes the IN instruction, the sending process in this example is automatically rescheduled and the communication starts. This is a more efficient use of processor time, since the sending process hasn’t had to continually poll (query) some status flag to check if the second (receiving) process is ready. This same trick can also be used to synchronize processes on different transputers.

Currently, links operate at a maximum speed of 20 MBits/sec, yielding an effective unidirectional data rate of 800 kilobytes/sec for the T212 and the T414, and 1.8 megabytes/sec for the T800. Transputer links use a handshake in which each byte that is sent must be acknowledged before the next byte can be sent. The T800 has an improved handshake mechanism, which is why it has a faster transfer rate.

Links have assigned locations in internal, on-chip memory. To send a message to another transputer, you must use one of these reserved locations. These locations correspond directly to pins on the transputer that will be electrically connected to either other transputers or to switching chips that will eventually connect to another transputer’s link pin. Note that each input and output link channel is one word (four bytes) wide:

Link	Address (hexadecimal)
Link 3 input	0x8000001C
Link 2 input	0x80000018
Link 1 input	0x80000014
Link 0 input	0x80000010
Link 3 output	0x8000000C
Link 2 output	0x80000008
Link 1 output	0x80000004

Link 0 output 0x80000000 (base address of memory)

So, for example, a process communicating across link three would read from address 0x8000001C and write to address 0x8000000C. Note that there are special instructions used for communication.

Programs communicate across a link by executing an input (IN) instruction or output (OUT) instruction. Parameters to these instructions are the memory-start address of the data and the length of the data block. When executed, the input or output instruction invokes the link engine. Once the instruction is started, the CPU is free to go ahead and do something else.

The link hardware acts autonomously, stealing cycles from the processor to get data from memory and pass it over the link channel. After every byte sent, the receiving transputer sends an acknowledgement signal and stores the data in the buffer of the receiving process. If there is no such receiving process, the receiving transputer doesn’t acknowledge the byte. The sending transputer still waits for an acknowledgment, though. So a receiving process must be started to receive the byte, or the sender will block (i.e., wait) until there is one.

When the message has been transferred, the communicating processes are placed back on the process ready queues (maintained internally by the transputer’s microcoded scheduler). As a result of the internal link engine stealing cycles from the CPU, there is some small CPU overhead once the communication begins.

Because the link engines don’t do any buffering, communications can only take place when a sending process and a receiving process are both ready to communicate. This means that one side has executed an output instruction and the other side has executed an input instruction.

A process that wants to communicate through a hardware link will always be descheduled. The transputer will reschedule that process only after the message has been sent or received. Thus, transputer processes sleep as long as they’re waiting for communication.

When a process is descheduled, the transputer puts information about it at the end of a "ready" queue. This is a queue of processes ready to run when the current process reaches a descheduling point. It can take some time before the process works its way back to the start of the queue and is able to run again. Because of this delay, for you to use the hardware links efficiently, you should use the largest possible message. By doing this a process will be descheduled a minimum number of times. In this case, you can see that it would make sense to calculate and pass 100 or even 1,000 pixels worth of data at a time, rather than just one pixel.

Event Channel

In addition to the links, there is another hardware mechanism that acts as a channel. The external Event pin on the transputer uses location 0x80000020 as its "channel." Whenever the external event pin receives a signal, any process waiting on this channel is awakened. This is the transputer’s equivalent of an interrupt. In this sense it is a one-way channel, since a process must continually wait on the channel (for input) by executing an IN instruction, but never sends data through it. The length of the expected message does not matter as no data is actually transferred through this channel; thus it makes the most sense to input a byte from it as this is the smallest unit. Caution: This pin is "edge-triggered," which means that the smallest glitch will latch it. Also note that the pin must be pulled low before it can be triggered again.

On-chip Timers

Another unique feature of the transputer are its on-chip timers that can be set or read by a program. There are several considerations that made it essential that the transputer have an on-chip timer. First, it is necessary to be able to "timeout" (i.e., abort after a certain amount of time has elapsed) on a link channel if communication is not taking place. This requires a clock to check the elapsed time during which no communication occurs. Second, the unusual multiprocessing ability of the transputer requires the microcode scheduler (which schedules processes to be run) to swap processes in a "fair" manner. This can prevent one process from "hogging" the CPU all the time. Normally, process scheduling is implemented at the software level in an operating system, but the transputer permits multiple processes to be run on the chip at the assembly code level. A scheduler is built into the microcode that gives each process a timeslice. It tries to be fair and give each process a relatively equal slice of CPU time, although a programmer can defeat its "fairness" by not using instructions that can cause descheduling. In order to do this, the scheduler needs a clock. Programmers can take advantage of this clock to schedule processes themselves. A process can be programmed to sleep for a certain amount of time, then "wake up" and continue.

The transputer has two on-chip timers, one for high priority processes and one for low priority processes. Each timer is a 32-bit register. The high priority timer ticks once each microsecond for a period of about 71 minutes, while the low priority timer ticks once each 64 microseconds for a longer period of about 76 hours. A process can only access the timer that is associated with its own priority level.

Times between (now + one tick) and (now + 0x7FFFFFFF ticks) are "in the future." If you attempt to wait for them, the process will deschedule and wait for the specified time to appear in the timer register. Times between (now + 0x80000000) and now are in the past. If you attempt to wait for them you won’t wait at all, that is your process will not be descheduled. This mechanism can prevent a process from going to sleep for over three days (76 hours) in the case that it just wanted to pause for a brief millisecond, but a high priority process preempted it for two milliseconds before it could deschedule itself. Thus, what the process thought was the "future" became the "past" before the process could actually execute.

Transputer Instruction Set

The Basic Types of Transputer Instructions

Each transputer instruction consists of a single byte divided into two four-bit parts. The four most significant bits are a **function code,** and the four least significant bits are a **data value** (see Figure 3-1). This allows for 16 function codes or 16 instructions; however, one of the 16 instructions (operate) allows indirect access to many more instructions.

Function Data
Bits 7 to 4 3 to 0

Figure 3-1 Function Code and Data Value within a byte.

Based on this arrangement, transputer instructions can be divided into two categories: **direct** and **indirect** instructions. Inmos studied which instructions are most often executed and designed the transputer so that the most common instructions are encoded into a single byte and execute very rapidly. These are the direct instructions. The remaining instructions require multiple bytes of memory and are referred to as indirect instructions.

Direct Instructions

The most common programming operations are the direct instructions. As a result, the direct instructions have been designed to be the fastest and most compact instructions on the transputer. Inmos claims that most programs on the transputer will typically spend 70 percent of their time executing direct instructions.

Direct instructions can be broken into three general groups:

ï Stack operations;
ï Prefix functions; and
ï Flow control instructions.

The transputer's architecture consists of a three register stack that uses registers A, B, and C. Register A is at the top of the stack and register C is at the bottom. Table 3-1 is a full list of the direct instructions available for the INMOS transputer.

Table 3-1. Direct Instructions

Opcode	Mnemonic	Description
0X	j	jump
1X	ldlp	load local pointer
2X	pfix	prefix
3X	ldnl	load non-local
4X	ldc	load constant
5X	ldnlp	load non-local pointer
6X	nfix	negative prefix
7X	ldl	load local
8X	adc	add constant
9X	call	call subroutine
AX	cj	conditional jump
BX	ajw	adjust workspace
CX	eqc	equals constant
DX	stl	store local
EX	stnl	store non-local
FX	opr	operate

Stack Operations

To load a constant value into a transputer register, use the LDC (load constant) instruction.

1**.** When the transputer executes *load constant*, the low order nibble of the instruction is placed in the lowest four bits of the operand register.

2**.** The contents of B are pushed into the C register, and the contents of A are pushed into B (the registers form a stack).

3**.** Finally, the contents of the operand register are placed into the A register.

Example 3-1

Register Contents:
A 10
B 20
C 30

Instruction Executed
ldc 5 ; This loads the value 5 into the A register.

Result:
A 5
B 10
C 20

Comment: In this example, the contents of C are "lost" (i.e., pushed off the bottom of the stack).

Example 3-2

For the same initial values of A, B, and C used in Example 3-1, the instruction sequence:

Register Contents:
A 10
B 20
C 30

Instructions Executed
ldc 9
ldc 2

Result:
A 2
B 9
C 10

Comment: When the ldc 9 is executed, the contents of B are pushed into C, and the contents of A are pushed into B. Next, 9 is placed into the A register. When the ldc 2 is executed, the stack is pushed again, B is pushed into C, and A is pushed into B before 2 is placed into register A.

The add constant instruction is used to add a constant value to a register and is represented in mnemonic form by ADC. An example using ADC follows:

1**.** The lower nibble of the *add constant* instruction is placed in the lower four bits of the operand register.

2**.** The contents of the operand register are then added to register A.

Example 3-3

Register Contents:
A 10
B 20
C 30

Instructions Executed
ldc 6
adc 3

Result:
A 9
B 10
C 20

Comment: First, 6 is loaded into register A. This pushes the old contents of A and B, while the old contents of C are pushed off the stack. Then, 3 is added to A (which contains 6 after the ldc 6) and 9 remains.

One of the important points to remember is that, if the addition **overflows** (i.e., causes the A register to wrap around from 0x7FFFFFFF to 0x80000000 in either direction), the Error flag is set. This means that, if the most significant bit of A changes because of an add constant instruction, the Error flag is set.

Workspaces

Besides the register stack, a program must also access the main memory to perform operations. The transputer extends the register stack model to address memory in a stack-based fashion. Inmos, however, chose not to use the word "stack" in their documentation for memory-based stacks. Instead Inmos calls them "workspaces"; however, they are simply stacks in memory. So if the word "workspace" seems foreign to you, just mentally substitute "stack" wherever you see it.

Workspaces are organized as "falling" or "decreasing" stacks, in that the top of the workspace grows towards low memory. (In other words, towards the most negative address 0x80000000. Recall that addresses are signed.) All workspace-based operations must be word-aligned. Whenever you are addressing memory you must do so on addresses that are multiples of four.

There is a special register that the transputer uses just to contain the address at the top of the workspace. It is called the ***workspace pointer*** and is abbreviated ***Wptr.***

Wptr must always contain a word-aligned value; it must point to the beginning of a word (not a byte) in memory. Since the transputer has a 32-bit word length (or four bytes), the *Wptr* is always an even multiple of four.

To retrieve a value from a workspace, use the load local instruction.

The *load local* instruction:

1**.** Pushes B into C;
2**.** Pushes A into B; and
3**.** Loads the 32-bit value into A that is located at the address of four times the operand added to the workspace pointer.

Example 3-4

This example uses the following values for the registers and memory:

Registers
A 10
B 20
C 30
Wptr 10004000

Memory
10004000 5
10004004 7
10004008 9

If the following instruction is executed:
ldl 0

Then the value at 10004000 is loaded into A, resulting in:
A 5
B 10
C 20

If instead, you execute:
ldl 1

Then the value at 10004004 is loaded into A, resulting in:
A 7
B 10
C 20

In a similar fashion, if you execute:
ldl 2

It will then yield:
A 9
B 10
C 20

Comment: In all of the cases shown in Example 3-4, the workspace remains unchanged, while the operands in the *load local* instructions refer to word-offsets (four bytes) from the workspace pointer.

Use the *store local* instruction to place values from the A register into a workspace.

Just as in the case with the *load local* instruction, *store local* :

1**.** Calculates the address of a memory location by adding four times its operand to the workspace pointer.

2**.** It then places a copy of the contents of the A register at that location.

Example 3-5

Again, using the following values for the registers and memory:

Registers:
A 10
B 20
C 30
Wptr 10004000

Memory:
10004000 5
10004004 7
10004008 9

If you execute a:
stl 0

Then the value at 10004000 is replaced by A and thus:
10004000 10
10004004 7
10004008 9

If instead, you execute:
stl 1

Then the value at 10004004 is replaced by A and thus:

10004000 5
10004004 10
10004008 9

And likewise (if instead you execute):
stl 2

Yields:
10004000 5
10004004 7
10004008 10

Comment: In all of the above cases, the A, B, and C registers remain unchanged.

Use the *load local pointer* instruction to calculate addresses related to the workspace.

The *load local pointer* instruction:

1. Loads the workspace pointer plus four times the operand into the A register (after pushing B into C and A into B).

Example 3-6

Using the same values as Example 3-5:

Registers

A 10
B 20
C 30
Wptr 10004000

Then:
ldlp 0

Results in:
A 10004000
B 10

C 20

If instead, you execute:
ldlp 1

Then you have:
A 10004004
B 10
C 20

And likewise (if instead you execute):
ldlp 2

Yields:
A 1004008
B 10
C 20

Comment: In all of the above cases, the workspace remains unchanged.

In the previous workspace-related instructions, the word "local" is key. All of the operations are local to the workspace in use. But suppose you want to randomly access some memory location that isn't in the current workspace? The transputer provides for non-local memory accessing.

The *load non-local* instruction allows you to access any address in memory**.**

Load *non-local* works in about the same way that *load local* does, except that:

1**.** Register A is used as a base address instead of the workspace pointer *Wptr*.

2. The B and C registers are unaffected by this operation. Also note that the address in A must be word-aligned (divisible by four).

Example 3-7

Given the following:

Registers
A 10004000
B 20
C 30

Memory
10004000 5
10004004 7
10004008 9

Then:

ldnl 0

Yields:

A 5
B 20

C 30

If instead, you execute:
ldnl 1

Then you have:
A 7
B 20
C 30

If instead you execute:

ldnl 2

It yields:
A 9
B 20
C 30

Comment: A common technique is to:

1. First load register A with an address by way of the *load constant* instruction.

2. Then load the contents of the address using *load non-local* with an operand of zero:
ldc address
ldnl 0

Example 3-8

The reverse operation, *store non-local*, places the contents of the B register to the word-aligned address pointed to by register A.

So given:

Registers
A 10004000
B 20
C 30

Memory
10004000 5
10004004 7
10004008 9

Then:
stnl 0

Yields:
10004000 20
10004004 7
10004008 9

If instead, you execute:
stnl 1

Then you have:

10004000 5
10004004 20
10004008 9

If instead you execute:
stnl 2

It yields:

10004000 5
10004004 7
10004008 20

There is an analagous *load pointer* instruction to the *load local pointer* called the *load non-local pointer*.

The *load non-local pointer* instruction:

1. Loads the contents of the A register plus four times the operand into the A register.

2. Registers B and C are unaffected.

Example 3-9

So if you have in this example:

A 10004000
B 20
C 30

Then:
ldnlp 0

Yields:
A 10004000
B 20
C 30

While:
ldnlp 1

Yields:
A 10004004
B 20
C 30

And:
ldnlp 2

Yields:
A 10004008
B 20
C 30

Space on the Workspace

Space on the workspace is allocated and deallocated using an adjust instruction called *ajw* (for adjust workspace). Because the workspace grows towards the low memory addresses, you must allocate memory to the workspace by:

1. Decrementing the workspace pointer (by adjusting the workspace with a negative number); and

2. Deallocating space by incrementing the workspace pointer.

Example 3-10

In this example, all the addresses are hexadecimal:

ï This allocates two words (eight bytes) to the workspace:
; Wptr = 10004008 before the ajw
ajw -2
; Wptr = 10004000 after the ajw

ï While the following deallocates two words from the workspace:
; Wptr = 10004008 before the ajw
ajw 2
; Wptr = 10004010 after the ajw

Prefix Functions

With only four bits for data in an instruction byte, the maximum operand any instruction can have is 15. To allow values greater than 15 to be loaded into a register, other instructions are required to load the operand register. Two more instructions allow the operand of any instruction to be extended in length. These instructions load the data into the operand register that a future instruction can then use. These two instructions are *prefix* and *negative prefix*.

Prefix Instruction.

The *prefix* instruction:

1. Loads its four data bits into the operand register; and

2. Then shifts the operand register to the left by four places.

Consequently, operands of any length can be represented up to the length of the operand register (which is 32 bits).

Example 3-11

To load the hexadecimal value 0x324 into the operand register, the code would be:
pfix 3
pfix 2
ldc 4

After this sequence, register A would contain 324 (hexadecimal). The last instruction is a *load constant*, since the prefix instruction always shifts the operand register to the left by four places after placing its data bits in the lower four bits.

Negative Prefix

Negative numbers are loaded in a similar fashion using the *negative prefix* instruction. The *negative prefix* instruction is similar to the *prefix instruction* except that it complements the operand register before shifting it to the left by four places.

Example 3-12

To load the value -31 (represented as FFFFFFE1) into the operand register, the code would be:

```
nfix 1
ldc 1
```

After this sequence, A would contain -31 [hexadecimal FFFFFFE1]. The nfix 1 instruction should be examined in detail:

ï First, 1 is loaded into the operand register.

ï Then all the bits are complemented, so the operand register now contains FFFFFFFE.

ï The operand register is shifted to the left four places and contains FFFFFFE0 at the end of the *nfix 1* instruction.

Comment: Normally, you will never have to deal with the prefix instructions because most assemblers take care of the prefix bytes for you and allow you to write code like:

```
ldc 0x4321
```

Then the assembler automatically generates:

```
pfix 4
pfix 3
pfix 2
ldc 1
```

Flow Control Instructions

Most computer programs need to test a set of conditions. Then they decide what part of the program to execute, based on the results of the test. The transputer provides these necessary instructions for controlling the flow of execution.

The most obvious instruction for controlling the execution flow is the *jump* instruction. *Jump* causes the instruction pointer to be incremented by the amount of the operand. *Jump* is a *relative* instruction because the target address is computing by adding the operand to the current address (rather than loading some absolute address).

Most assemblers simply allow the name of a label to reference where to jump (see Example 3-13). The mnemonic for jump is "j."

Example 3-13

```
j THERE
ldc 2
ldc 1
THERE: ldc 0
```

The *ldc 2 and ldc 1* will be skipped in the above program fragment because the *jump* instruction causes the flow of program execution to be transferred to the *ldc 0* instruction.

It has already been mentioned that the computation is relative. An equivalent way of expressing the program in Example 3-13 would be:

```
00004000 j 2
00004001 ldc 2
00004002 ldc 1
```

00004003 ldc 0

Because all of the above instructions are direct, they are all one byte long. Suppose the numbers on the left are the instruction addresses in memory:

1. The "jump" with operand two (or the *j 2* instruction) adds two to the instruction pointer.

2. The instruction pointer is incremented prior to executing the instruction, so when it starts executing *j 2* , it contains 00004001.

3. Then the *j 2* instruction causes two to be added to the instruction pointer so that it will contain 00004003.

4. The instruction pointer continues executing instructions starting from 000040003.

Tip: Don't store information across a jump. The *jump* instruction can potentially trash the contents of the A, B and C registers.

Jump is actually a descheduling instruction: If another process is pending, the current process may be suspended and the pending process started (or resumed). The newly scheduled process may then use the registers as it will, thus invalidating any data in the registers before the "jump" in the first (and now sleeping) process.

There are times in a program when you might want to return to the point that is just after control was transferred. That way, a frequently used code fragment can be repeatedly used without having to contain multiple copies in the program. Such code fragments are known as *subroutines*, since they are part of the program but not inside the main routine. The main routine transfers executions to subroutines; eventually, the subroutine transfers executions back to the main routine. Subroutines are implemented through the traditional *call* instruction:

1. The *call* instruction pushes the C register, the B register, the A register, and the instruction pointer (in that order) onto the workspace.

2. Then it jumps to the "relative" address specifed by the instruction pointer that has been added to the operand of the call.

3. The workspace pointer is adjusted (decremented) by four to compensate for the information pushed onto it.

Comment: The corresponding *return* instruction, which returns control to the main line routine (mnemonic RET), is not a direct instruction and is two bytes long. When the *return* instruction is executed, any workspace claimed by the subroutine should be released by an *ajw* instruction) so that the *Wptr* has the same value it had prior to calling the subroutine.

The state of the workspace after the call instruction is:

Address	**Saved Values**
Wptr+4	(old Wptr pointed here)
Wptr+3	C register
Wptr+2	B register
Wptr+1	A register
Wptr+0	instruction pointer (Iptr)

You might recall that the workspaces grow towards low memory; after a call, the workspace pointer will contain four less than the old value. Like the *jump* instruction, *call* jumps to the "relative" address specified by the operand plus the instruction pointer.

Example 3-14

An example of a subroutine follows. This routine just decrements the single parameter it passes:

```
ldc 4
stl 0 ; place parameter on stack
call decrement
. . .
. . .
. . .

parameter = 5
locals = 0

decrement:
ajw -locals

ldl parameter
adc -1
stl parameter

ajw locals
ret
```

In Example 3-14, the first parameter to a routine is located at five words above the current workspace pointer because four items were pushed on the workspace as a result of the call instruction.

Also notice that to allocate local variable-space, you use *ajw*'s at the start and end of the subroutine: The first allocates stack space by decrementing the *Wptr*; the second deallocates the stack space by incrementing the*Wptr*. In Example 3-14, the *ajw*'s are not needed, since local variables were not used; they are there to help illustrate how a typical subroutine looks.

In Example 3-14, the first two lines above the decrement routine are directives to the assembler to replace the symbolic names *parameter* and *ws* with the constants on the right of the equals sign. This makes the code more readable because you're using symbolic names rather than numbers. Some assemblers use the "EQU" mnemonic for this directive, others use "#define" just like C.

In the code used in Example 3-14, the main routine:

1. First pushes the value it wants decremented on the stack, then it calls the decrement subroutine.

2. Next, the subroutine allocates stack space by adjusting the stack according to the number of local variables it needs (in this case, zero). This way, the local subroutine can use the stack for its own computations while preserving the data already put there by the main line routine.

The *conditional jump* (cj) instruction jumps if the value in the A register is false (zero), but not if its true (nonzero).

1. If the *conditional jump* is taken, the contents of A, B, and C are unaffected.

2. If the *conditional jump* is not taken, then the A register is popped and replaced by the contents of B and the contents of C are duplicated into the contents of B. To illustrate the usage of *cj*, let's modify the subroutine from Example 3-14 to decrement the parameter it has passed to zero, but not below zero. In Example 3-15, the routine will have to test if the parameter is zero, and if it is, return without decrementing the parameter.

Example 3-15

```
ldc 4
```

```
stl 0 ; parameter to pass
call decr_to_zero
. . .
. . .
. . .
parameter = 5
locals = 1

decr_to_zero:
ajw -locals

ldl parameter
stl 0 ; save in a local
cj return ; if zero, skip
ldl 0 ; reload, cj popped A
adc -1
stl parameter

return: ajw locals
ret
```

In this program, the following sequence occurs:

1. The parameter is pushed on the workspace and then the subroutine is called.
2. First, the subroutine *decr_to_zero* adjusts the workspace for the number of local variables it will use (in this case, one).
3. Next, it retrieves the parameter that was passed and saves it in this local variable.
4. Then the *cj* instruction tests the A register for zero. If the paramater was zero, A will be zero and the jump will be taken. The workspace that was allocated for local variables will be deallocated, and the subroutine will return to the main routine. If on the other hand (as in this instance where parameter is four) the parameter is nonzero, then the *cj* will fail to take the jump and the A register will be popped.
5. The subroutine continues executing through the *ldl 0* instruction. *ldl 0* retrieves the value of the parameter, which is then decremented.
6. Finally, the local workspace is deallocated and the subroutine returns. Note that the local variable in this instance was used simply to temporarily hold a result, since *cj* pops the register stack. It is often necessary to compare two values in a program. The *equals constant* (eqc) instruction compares the contents of the A register with the operand and replaces the contents of A with zero if the operand and A were not equal, and with one if they were equal. The B and C registers are unaffected.

Example 3-16

Let’s modify the subroutine in Example 3-15 so that it takes a parameter larger than one and decrements it to one, but not lower. The code could look like:

```
ldc 4
stl 0 ; parameter to pass
call decr_to_one
. . .
. . .
. . .
parameter = 5
locals = 1

decr_to_one:
ajw -locals

ldl parameter
stl 0 ; save in a local
eqc 1
cj continue ; if one, skip
```

```
j return

continue: ldl 0 ; reload the parameter
adc -1
stl parameter

return: ajw locals
ret
```

In Example 3-16, the parameter is tested to see if it is one by the *eqc 1* instruction.

1. If it is one, the A register will contain 1 (true = 1). If it is zero, the result of the *eqc 1* test was false (false = 0).

2. If the A register contains zero, it means that the test has failed. For instance, the parameter was not one, so the *cj* tests the A register for zero and jumps if A is zero. If the parameter is not one, the subroutine skips ahead to the where the continue label points (*ldl 0*).

3. If on the other hand the *cj* is not taken, then the A register contained something nonzero. Since the result of an *eqc* is only one or zero, this means that A contains one, which means the *eqc 1* yielded true. In this case, the subroutine unconditionally jumps to the return point.

Summary of Direct Instructions

J *jump* transfers the flow of execution. Jumps are instruction-pointer relative (i.e., the operand is added to the instruction pointer to calculate the location of the next instruction to be executed). *Jump* is also a descheduling breakpoint.

LDLP *load local pointer* instruction loads the workspace pointer, plus four times the operand register, into A. This gives you the address of a value on the workspace.

PFIX *prefix* loads the lower nibble of its operation code into the operand register and then shifts the operand register to the left by four places.

LDNL *load non-local* loads the value pointed at by the address calculated by the workspace pointer, plus four times the operand register, into A.

LDC *load constant* loads the lower nibble of its operation code into the lower nibble of the operand register and then loads the operand register into A.

LDNLP *load non-local pointer* loads A plus four times the operand.

NFIX *negative prefix* loads the lower nibble of its operation code into the operand register and then inverts all the bits in the operand register and shifts the register to the left by four places.

LDL *load local* loads the value at the word-offset of the operand register into A.

ADC *add constant* loads the lower nibble of its operation code into the operand register, then adds the contents of the operand register and A.

CALL *call* pushes the C, B, and A registers, as well as the instruction pointer on the workspace. It then jumps to the relative address specified by the operand register.

CJ *conditional jump* jumps to the relative offset in the operand register if the value in A is zero (false).

AJW *adjust workspace* adds four times the operand register to the workspace pointer.

EQC *equals constant* tests the value in A against the operand register.

STL *store local* stores the contents of A at the address calculated by adding the workspace pointer and four times the operand register.

STNL *store non-local* stores the contents of B at the address calculated by adding A and four times the contents of the operand register.

OPR *operate* is the gateway to all the *indirect* instructions. This instruction takes the value in the operand register and executes it.

Indirect Instructions

The direct instructions the transputer provides are compact and efficient, yet they are inadequate to provide the necessary functionality for writing most programs. Additional instructions are required and these are the indirect instructions.

The gateway to indirect instructions is the *operate* instruction. The *operate* instruction is classified as a direct instruction, but of a special kind. *Operate* executes the contents of the operand register as an instruction. In this way, indirect instructions are usually comprised of a prefix instruction that loads the operand register followed by an *operate* instruction, which causes the contents of the operand register to be executed. The exceptions to this rule are the indirect instructions that fit into a single byte, that is, the 16 instructions that fit into the lower nibble of the operate instruction and thus do not require a prefix instruction.

Note: Some floating point instructions on the T800 are indirect in yet another fashion. There is a floating point instruction called FPENTRY that executes the contents of the A register, rather than the operand register.

The indirect instructions fall into general groupings of functionality. The references to A, B, and C below refer to the three deep, integer register stack, with A being the A register, B meaning the B register, and C meaning the C register.

General Operation Codes

General operations include the *reverse* instruction, which swaps A and B as it is technically an indirect instruction, although only one byte in length (i.e., REV is equivalent to the direct instruction OPERATE, with an operand of zero). There are also four instructions that are used for conversion between signed values of different length: XWORD, CWORD, XDBLE, and CSNGL. There is also an instruction for loading the minimum integer value into A, which is useful for checking the sign (leftmost) bit or computing an address in on-chip memory (on-chip memory starts at location 0x80000000).

REV Reverse swaps the contents of A and B.

MINT Minimum integer loads A with the hexadecimal value 0x80000000 (Inmos refers to this as NotProcess.p). This is useful because the transputer’s on-chip RAM is located at address 0x80000000.

XWORD Extend to word sign extends a partial word value to a full 32-bit word.

CWORD Check word checks a 32-bit word to see if it will fit into a smaller bit-field.

XDBLE Exend to double sign extends a 32-bit word to a 64-bit word.

CSNGL Check single reduces a 64-bit value to a single 32-bit word.

Examples:

To extend the sign of a byte to the entire word, use:
ldc 0x80 ; set the most significant bit of the byte
xword ; sign extend the byte into the word

To see if a word can be represented by a signed byte, use:
ldc 0x80 ; set the most significant bit of the byte
cword ; perform the check
; test the Error flag to determine result of the check

Arithmetic and Logical Operation Codes

The transputer has the usual complement of arithmetic and logical operations one expects to find on a microprocessor.

The arithmetic instructions fall into two categories: those that perform checked arithmetic and those that don't. Checked arithmetic means that if overflow occurs, the Error flag will be set. The easiest way to visualize overflow is to think of an automobile odometer. When a certain number of miles (or kilometers) is reached, say 999999, the odometer returns to 000000 on the next mile. This is due to the finite number of digits it has available to represent the current mileage (in this example, six digits). Likewise, the 32-bit registers of the transputer can only represent numbers of a certain size. An attempt to represent a larger number due to the result of an arithmetic operation will cause overflow. If this matters in the calculation you are performing, use the instructions that set the Error flag and then check the Error flag after the operation. If overflow does not matter in your calculation, then use the instructions that do not set the Error flag on overflow.

The shift instructions are a bit unusual in that shifts of greater than 32 are actually preformed (even though the transputer's word lenth is 32, so a shift greater than 32 really doesn't make sense). Since the transputer does not have a barrel shifter (a hardware mechanism to perform any number of shifts in one processor cycle), the amount of time to perform a shift depends on the number of places shifted. For very large shifts, the transputer can actually "lock up" for minutes at a time performing the shift. It is of interest to note that the floating point engine of the T800 does have a barrel shifter and future versions of the transputer will probably have a barrel shifter on the integer engine, thus the shift quirks are likely to go away in future revisions of the silicon.

AND And performs a bitwise logical and operation between A and B, leaving the result in A.

OR Or performs a bitwise logical or operation between A and B, and leaves the result in A.

XOR Xor peforms a bitwise logical exclusive or operation between A and B, and leaves the result in A.

NOT Not performs a bitwise logical not on A.

SHL Shift left shifts the number in B to the left by the number of bits specified in A. Vacated bit positions are filled with zero bits.

Tip: The SHL instruction takes an amount of time to execute that is proportional to the value in the A register (one cycle for every bit shifted plus some overhead). In the worst case, for a very large value in A, the transputer can be locked up for 3 to 4 minutes.

SHR Shift right shifts the number in B to the right by the number of bits specified in the A. The shift is unsigned, so vacated bit positions are filled with zero bits.

Tip: The SHR instruction takes an amount of time to execute that is proportional to the value in the A register (one cycle for every bit shifted plus some overhead). In the worst case, for a very large value in A, the transputer can be locked up for 3 to 4 minutes.

ADD Add adds the contents of A and B, leaving the result in A. The Error flag is set if overflow occurs.

SUB Subtract subtracts the contents of A from B, leaving the result in A. The Error flag is set if overflow occurs.

MUL Multiply multiplies B and A, leaving the result in A. The Error flag is set if overflow occurs.

FMUL Fractional multiply is used to multiply two fractional numbers that are represented in fixed point arithmetic. A and B are considered to contain fixed point numbers lying in the range $-1 <= X < 1$. FMUL returns the rounded fixed-point product of these values to the A register.

DIV Divide divides B by A and leaves the result in A. The result is undefined if A is zero.

REM Remainder leaves the remainder of B divided by A in A. Again, the result is undefined if A is zero.

GT Greater than performs an unsigned comparison of A and B. If B is strictly greater than A, A is set to true (one).

Otherwise, A is set to false (zero).

DIFF Difference subtracts A from B, leaving the result in A without checking for overflow or carry.

SUM Sum adds B to A without checking for overflow or carry.

PROD Product multiplies B and A, leaving the result in A without checking for overflow or carry.

Examples:

To invert the sign of a number (in A):
```
ldl -1 ; load negative one
mul ; multiply to invert sign
or,
not ; perform one’s complement on bits
adc 1 ; add one to make two’s complement
or,
ldc 0 ; load zero onto stack
rev ; reverse A and B
diff ; subtract number from zero to invert sign
```

To multiply a number by eight using unchecked arithmetic:
```
ldc 8 ; load eight
prod ; perform the multiplication
or,
ldc 3 ; load three into A
shl ; shift the number left 3 places
```

To multiply a number by eight using checked arithmetic:
```
ldc 8 ; load eight
mul ; perform checked multiplication
```

Long Arithmetic Operation Codes

In addition to single length (32-bit) arithmetic support, the transputer has support for double length (64-bit) integers built into the instructions set. Such numbers are usually refered to as "long numbers" or "longs." Note that to perform sign extended right shifts you must use the long shift right instruction (LSHR), as the single shift right instruction (SHR) fills the leftmost bit with a zero (performing an unsigned shift). Some C programmers are used to using right shift (represented in C by the double greater than sign ">>") for division. If the C compiler you are using generates SHR instructions for the shift, the shift will be unsigned and could turn a small negative number into a large positive number. If the C compiler generates LSHR instructions, then the sign will be preserved on the shift and will behave as expected (i.e., perform correct division even on negative numbers). As of this writing, the draft ANSI C standard says that the right shift operation is implementation dependent, thus a C programmer cannot count on sign extension on right shifts.

LADD Long add adds A and B and the least significant bit of C (carry bit), leaving the result in A. Arithmetic overflow is checked.

LSUB Long subtract subtracts A from B, and also subtracts the least significant bit of C (carry bit) from B, leaving the result in A. Arithmetic overflow is checked.

LSUM Long sum is designed to be an intermediate instruction in multi-precision arithmetic. LSUM adds A, B, and the least significant bit of C (carry bit).

LDIFF Long diff is designed to be an intermediate instruction in multi-precision arithmetic. LDIFF subtracts A from B and also subtracts the least significant bit of C.

LMUL Long multiply multiplies A and B as two single-word, unsigned operands adding the single word carry-operand in C to form a double length unsigned result.

LDIV Long divide divides the double length value held in B and C (the most significant word in C) by the single-length unsigned value held in A. Overflow occurs if the result cannot be represented as a single word value and causes the Error flag to be set.

LSHL Long shift left shifts the double-length value held in B and C (the most significant word in C) left by the number of bit positions equal to the value held in A. Vacated bit positions are filled with zero bits.

LSHR Long shift right shifts the double-length value held in B and C (the most significant word in C) right by the number of bit positions equal to the value held in A. Vacated bit positions are filled with zero bits.

NORM Normalize normalizes the unsigned double-length value held in A and B (the most significant word in B). This value is shifted to the left until its most significant bit is one. The shifted double-length value remains in A and B, while C receives the number of left shifts performed. If the double-length value is initially zero, C is set to twice the number of bits in a word (64).

Example:

A sign extended right shift can be performed via:

```
xdble ; extend value to be shifted to double
ldl X ; load the number of places to shift (from X)
lshr ; perform the shift, result is in A
```

Indexing and Array Operation Codes

Since arrays are often used in programs, the transputer provides instructions just for computing addresses of elements with an array. Instructions for arrays of bytes, words, and double words are all present.

BSUB Byte subscript adds A and B and leaves the result in A. It is designed for computing offsets into byte arrays.

Tip: BSUB does not check the sum for overflow, so it is useful for unsigned arithmetic.

WSUB Word subscript adds A plus four times B, leaving the result in A. It computes offsets into word arrays. The sum is not checked for overflow, so it also useful for unsigned arithmetic.

WSUBDB Form double word is used for generating indexes for double word data items.

Note: The WSUBDB instruction is available on T800 transputers, but not on T414 transputers.

BCNT Byte count returns four times A into A. This is useful for computing byte offsets from word offsets.

WCNT Word count is used to find the word offset of an address from zero and return it in A; the byte offset (0-3) from the word is returned in B.

LB Load byte loads an unsigned byte from the address given in A.

Note: The transputer is little-endian: If you store 0x12345678 at address 00000000, then byte 0 will contain 0x78, byte 1 will contain 0x56, byte 2 will contain 0x34, and byte 3 will contain 0x12.

SB Store byte stores the byte in B at the address given in A.

MOVE Move message moves a block of memory by copying the number of bytes held in A from the starting source address, held in C, to the starting destination address, held in B.

Tip: The two address regions must not overlap. Also, the number of bytes to move in A is a signed value and must not be negative.

Examples:

To load the element x[5] from an array with word-sized elements:

```
ldc 5 ; load the index into the array
ldlp x ; load the pointer to the array
wsub ; computes the address of x[5]
ldnl 0 ; load x[5]
```

To load a byte from memory and extend its sign into the word:

```
lb ; load the byte, the higher bits are zeroed
ldc 0x80 ; set the most significant bit of a byte
xword ; extend the sign of the byte loaded via LB
```

Control Operation Codes

Flow control instructions are necessary in the construction of computer programs, and the transputer provides the needed support. There are indirect instructions for looping and subroutine calls and returns. Since loops may execute for a long time, it is of interest to note that LEND can cause a process to be descheduled. Atomic loops may be constructed with the direct jump instruction J.

RET Return returns the execution flow from a subroutine back to the calling thread of execution.

LDPI Load pointer to instruction adds the current value of the instruction pointer to A.

Tip: The instruction pointer contains the address of the instruction after the LDPI. This can be useful for computing relative addresses.

GAJW General adjust workspace exchanges the contents of the workspace pointer and A. A should be word-aligned.

DUP Duplicate top of stack duplicates the contents of A into B.

Note: The DUP instruction is available on the T800 and is not present on the T414.

GCALL General call exchanges the contents of the instruction pointer and A; execution then continues at the new address formerly contained in A. This can be used to generate a subroutine call at run time by:

1. Build a stack (workspace) frame like the one
the CALL instruction uses.
2. Load A with the address of the subroutine.
3. Execute GCALL.

B and C are unaffected by the general call instruction.

LEND Loop end is used to implement looping constructs. Before executing LEND, B contains the address of a two (32-bit) word control block in memory. The second word of this control block (pointed at by B+4) contains the loop count, which is decremented each time LEND is executed. If this value is decremented to zero or less, the loop is considered over, and execution continues with the instruction after the LEND. However, if the value is still positive, the first word (pointed at by B) is incremented and the instruction pointer is decremented by the contents of A. This means that A must contain the offset of where to jump to if the LEND is taken. Thus, prior to LEND, A must be loaded with this offset value. In the case of relative addressing, A would be loaded with the difference of two labels.

Tip: This is a nonatomic instruction and can cause a process to be descheduled.

Examples:

To call a subroutine, use the traditional CALL/RET pairs:

```
call subroutine ; call a subroutine
. . .
. . . ; continue on
. . .
subroutine:
. . . ; perform calculations
. . .

ret ; return to instruction past CALL
```

To implement a loop from 2 to 10, using LEND:

```
ldc 2 ; load starting index for loop
stl index ; save in a loop index variable
ldc 8 ; load number of times to iterate (10-2)
stl index+1 ; save in the word just past the loop index
ldl index+1 ; load the iteration count again
cj ENDLOOP ; if counted down to zero, jump to end of loop
LOOP:
. . .
. . . ; perform desired calculations in the loop
. . .
ldlp index ; load a pointer to the loop index
ldc ENDLOOP-LOOP ; load offset from start to end of loop
lend ; decrement count, jump to LOOP if not zero
ENDLOOP:
. . .
. . .
```

Scheduling Operation Codes

The transputer has a unique ability among microprocessor chips: hardware support for concurrent processes. The transputer can run multiple processes transparently, that is without user-coded support software; the scheduling and context switching is handled in microcode. Most programmers are familiar with such multiprocessing instructions at a higher level (such as the Unix fork() or exec() calls), but it is unusual to find them implemented at a level so deep in the silicon.

The fundamental resource of execution on a transputer is a process. A process consists of:

1. A thread of execution (represented by the instruction pointer); and

2. A workspace (defined by the workspace pointer).

Processes come in two flavors: low priority and high priority.

Tip: If a high priority process is started, all currently executing low priority processes will be suspended until the completion of the high priority process. In this manner, high priority processes can be thought of as interrupts, since they interrupt low priority processes. Multiple high priority processes will timeslice among themselves, just as low priority ones will.

The technique the transputer uses for deciding when to stop one process (temporarily) and continue with another is to assign certain instructions as descheduling points. The direct instruction for jumping, J, is one such descheduling point. If there are other processes of the same priority pending, then prior to execution of the J, the current process state will be saved and the next process, in a linked list of processes, will be executed.

Since some instructions are descheduling points, the others that aren’t will be executed without causing a timeslice (unless a high priority process interrupts a low priority one). Such instructions, which do not cause descheduling, are called atomic. The direct instruction, CJ, is an example of an atomic instruction.

Note: Only one process is ever running at a given time on the transputer. All processes are kept track of by a linked list organized as a queue. The microcode scheduler is in charge of managing this queue and determining which process to run (and for how long). When a descheduling instruction is reached by the current process, the scheduler sees if there is another process in the queue and runs it. If there is no other process in the queue, the current process continues executing.

The existance of descheduling and atomic instructions affects how you write code for the transputer. If there is some time critical section of code, say responding to some external device, you probably don’t want to be timesliced in the middle of executing this routine, so you would use atomic instructions like CJ (conditional jump).

Tip: All other processes will wait until the atomically coded one finishes or reaches a descheduling instruction. In most circumstances, it is wise to avoid sequences that have only atomic instructions, as they will only "hog the processor" and detract from the unique built-in multiprocessing support in the transputer.

Processes that have been started are kept track of in two singly-linked lists, one for each priority. In each list, or queue, the head pointer contains the workspace descriptor (i.e. address) of the process at the head of the queue, or NotProcess.p (0x80000000) if the queue is empty. NotProcess.p is a special name given to the value 0x80000000. This also happens to be the representation for -1 (in two’s complement arithmetic). This value can be loaded into the register stack using the MINT instruction (minimum integer). NotProcess.p, the value 0x80000000, indicates the end of the process queue.

Finally, when a process has been descheduled (or time-sliced), five words below the workspace are used to store the internal state of the process. Workspace locations -1 through -5 are used to save information needed when the process is rescheduled. In some cases (involving alternation), workspace location zero is also used. The saved items are:

Location 0 Offset to guard routine to execute (alternation)
Location -1 Instruction Pointer
Location -2 Wdesc of next process in queue
Location -3 Address of message buffer (called State.s)
Location -4 Wdesc of next process waiting for the timer or
NotProcess.p+1 (this location is called Tlink.s as it
links the processes in a timer queue)
Location -5 Time process waiting to awaken at

Note: Wdesc stands for workspace descriptor, meaning the workspace pointer with the priority of the process (that owns the workspace) encoded into the low bit.

Different descheduling instructions can cause different items to be saved. Locations -1 and -2 are used whenever a process is descheduled, the other locations may hold saved values, depending on the type of descheduling instruction involved.

In general, the following negative workspace locations will be used:

Process with no I/O 2 words
Process with only unconditional I/O 3 words
Process with alternative input 3 words + location 0
Process with timer input 5 words
Process with alternative timer input 5 words + location 0

So it is wise programming practice to allot five additional words below the workspace pointer as values not to disturb. When using alternation constructions, be sure to avoid using workspace location zero since the offset to the routine to execute (after the alternation has finished waiting) will be contained there.

STARTP Start process starts a new concurrent process at the same priority as the current process.

ENDP End process ends the current process. ENDP is also used to synchronize the termination of multiple processes. When a process executes an ENDP, it decrements a reference count of other processes. If the result is nonzero, it terminates and gives up control of the processor; however, if the reference count (of other processes) is zero, the workspace pointer and instruction pointer are assigned new values and continue as the current process.

Tip: ENDP is useful to synchronize some action after a group of processes have terminated, such as the freeing of allocated memory blocks to the processes or the closing of file handles that the processes may have used.

RUNP Run process starts an already existing, but stopped, process.

STOPP So that it can be restarted in the future, the stop process instruction stops the current process and leaves its instruction pointer in workspace location -1. To restart the process, a RUNP instruction must be executed as the process is removed from the scheduling list. STOPP can also be used to change a process’s priority (see example below).

LDPRI Load current priority loads the current process priority into A. Here a value of one means that the process has a low priority, and zero means the process has a high priority.

Example:

The following code sequence starts a process located at NEW_PROC with the same priority as the currently executing process:

```
ldc NEW_PROC - L_START ; load offset to process to start
ldlp WORKSPACE ; load address of workspace for new ; process
L_START:
startp ; start the new process at NEW_PROC
. . .
. . .
. . .
NEW_PROC:
; the new process located here starts
```

Note above that WORKSPACE is the address of some uninitialized area of memory for the process to be started (NEW_PROC).

The following code sequence drops a high priority process to low priority:

```
ldlp 0 ; load the workspace pointer
adc 1 ; set the zero bit for low priority
runp ; add self to low priority run queue
stopp ; save the Iptr and stop
. . . ; continue
```

Timer Handling Operation Codes

The transputer has two 32-bit on-chip timers. Inmos intended the timers to be used by processes for descheduling themselves, waiting until a specified time was reached before the sleeping process would awaken. One of the timers is only accessible by low priority processes, while the other is accessible only by high priority processes.

There are other applications for timers besides descheduling and synchronization. For instance, you can use the timers to check elapsed time simply by loading a timer's contents, doing whatever else you have to, loading the timer's contents again, and then taking the difference of the first and second timer values to get the time elapsed. However, the instruction set reflects the design considerations for having processes "sleep" until a designated time.

The high priority process timer, designated Timer0, is incremented once every microsecond; it completely cycles (or flips back to zero) in 4295 seconds (about 71 minutes). The low priority process timer, designated Timer1, is incremented once every 64 microseconds (or 15,625 times every second) and completely cycles in approximately 76 hours.

LDTIMER Load timer loads A with the value of the current priority timer.

TIN Timer input interprets the contents of A as a timer value and compares it with the value of the current priority timer.

1. If A is less than the current value of the timer (i.e., in the "past"), the process continues and the instruction has no effect.

2. However, if A is greater than the current priority timer, then the time is in the "future" and the process is descheduled (blocks) until that time occurs. Upon reaching that time, the process will awaken. This is a descheduling instruction and the values of A, B, and C are undefined after its execution.

Example:

To pause a low priority process for one second, use:

```
ldc 15625 ; 64 microseconds * 15625 = 1 second
ldtimer ; load the current timer value
sum ; calculate one second past the current timer value
tin ; wait until the timer has ticked one second
```

Input/Output Operation Codes

The transputer has hardware (links) and software (microcoded instructions) that support synchronized communication between processes. The important thing to remember is that communication occurs between processes. Even if the processes are on separate transputers, the messages will still be passed between two processes.

The channel is the pipe used for communication between processes and is used to synchronize and communicate messages. A channel is simply a 32-bit word in memory. Before a memory location can be used as a channel, it must be initialized to the value NotProcess.p, which is 0x80000000. You can easily do this by using the MINT instruction (which loads the A register with 0x80000000). The channel is automatically reset to NotProcess.p after executing input or output instructions, so it only needs to be initialized once.

When two processes try to communicate over a channel, it is important that both sender and receiver know how much data is to be transferred (this means both processes must use the same message length). If too little data is sent, the receiving process will stay blocked as it awaits more data. If too much data is sent, the sender risks writing over (and therefore corrupting) the receiver's data area.

As mentioned in Chapter 2, each transputer has four high-speed serial "links" that are used to connect transputers together. This allows transputers to pass messages to each other so that computations can be spread out over multiple transputers, and results collected by a single transputer.

The links behave in a similar fashion to a Direct Memory Access (DMA) channel. When sending or receiving messages, the link hardware acts autonomously, as a though it were a seperate software process (in a very real sense, it steals cycles from the processor since high message traffic over the links degrades the overall performance of the transputer). In the case of sending a message, data is fetched from memory and sent over the link a single byte at a time. After each byte, the receiving transputer sends an acknowledgement signal and stores the byte into its memory. If there is no receiving process listening for the transmitted bytes, the receiving transputer won't acknowledge the byte transfer and the transputer process sending the byte will "hang" on sending the message. In such a case, it is important that both the sending and receiving processes of the communicating transputers know the size of the message that will be passed between them. If one side sends a 16-byte message and the other side is waiting for a 32-byte message, then the receiving transputer will wait indefinitely. Conversely, if one side sends 64 bytes to another that is expecting only 32 bytes, the receiving transputer will acknowledge the first 32 bytes, and then, having its message request satisfied, it will continue execution without acknowledging (or receiving) any of the remaining 32 bytes. Again, the sending transputer will "hang" on the link.

No special instructions are necessary for communicating across the links. Messages are passed between links the same way they are passed between processes. The only difference is the memory location used for the channel, which is the area where communication occurs. For links, these locations are defined to be at the bottom of memory (remember that addresses are signed quantities on the transputers and 0x80000000 represents the largest negative address, i.e., the bottom of memory).

To communicate over a link, use the following addresses for channels:

Link	Input	Output
0	0x80000010	0x80000000
1	0x80000014	0x80000004
2	0x80000018	0x80000008
3	0x8000001C	0x8000000C

IN Input message inputs a message with the length (in bytes) of A from the channel pointed to by the address in B, leaving the message at the memory pointed to by C.

Tip: This instruction can cause the process to be descheduled.

OUT Output message outputs a message equal in length to the number of bytes in A to the channel pointed to by the address in B from the memory pointed at by the address in C.

Tip: This instruction can cause the process to be descheduled.

OUTWORD Output word outputs the 32-bit word contained in A to the channel pointed to by B. Outword uses location Wptr+0.

Tip: This instruction can cause the process to be descheduled.

OUTBYTE Output byte outputs the byte in the lower eight bits of A to the channel pointed to by B. Outbyte uses location Wptr+0.

Tip: This instruction can cause the process to be descheduled.

RESETCH Reset channel resets the channel pointed to by the A register. If A points to a link channel, the link hardware is reset. The channel is reinitialized to NotProcess.p.

RESETCH is used if communication that was started breaks down between two processes. In internal channels, like in the case where there is communication between processes on the same transputer, communication is guaranteed to finish, as the data transfer is achieved by just copying a block of memory. However in communication between transputers over links, if one transputer fails, a process could block indefinitely as it waits to communicate to the failed transputer. RESETCH provides the ability to reset the channel and restart the communication. The A, B, C register stack is unaffected by this instruction.

Note: This instruction will allow programs to check channels for "timing out" (not achieving communication) and reset the channel for future use (or retries).

Example:

To receive a byte from a channel:

```
ldl chan_addr ; load the channel address
ldl dest_addr ; load the destination address for the message
ldc 1 ; load the number of bytes to receive (one)
in ; wait for the message to arrive
```

To send a word out over a channel:

```
ldl chan_addr ; load the channel address
ldl message ; load the value to send out over the channel
outword ; output the word over the channel
```

Alternation

The transputer also introduces the concept of alternative inputs. In the transputer, interrupts are discouraged because the transputer supports on-chip multiprocessing; the idea is to use different processes to handle external events. A process can be "put to sleep" waiting for events and then "wake up" when an event occurs and start another process to service the event. This is determined by the type of event that has occurred and is somewhat like an interrupt service routine in that it captures an exception and dispatches it to the correct routine. The type of software construct that listens for events and then resumes execution based on what it receives is called an alternative input or alternative construct. INMOS refers to such a construct as an alternation that has component alternatives (meaning different possible branches of execution).

The best way to think about alternation, or an ALT construct, is to compare it to a SWITCH statement in the programming language C. In C, a SWITCH statement consists of a keyword, called SWITCH, that indicates the start of a series of possible execution choices, followed by an expression to evaluate. Next come the individual CASE statements which correspond to possible values that the evaluated expression may assume; these CASE statements determine the possible execution choices. One of the CASE statements (choices for execution) may be a DEFAULT case, which is selected if no other CASE is determined to equal the value of the expression.

Figure 3-1: Sample SWITCH statement in C

```
switch (expression)
{
case ONE:
statements;
break;

case TWO:
statements;
break;

case THREE:
statements;
break;

default:
statements;
break;
}
```

The transputer ALT construct is similar to the C SWITCH statement. The "case" equivalents in an ALT are enabled channels that the transputer is listening to. When a message comes in over one of these channels, the process wakes up and its corresponding "case" is executed. The overall structure of an ALT construct is:

```
ALT ; flag the start of an alternation
enable all channels to listen to
ALTWT ; wait until a message arrives
disable all the channels
ALTEND ; causes jump to the routine associated
; with the channel the message arrived on
service routines (equivalent to C switch statement "cases")
```

A process that enters an ALT construct "sleeps" when the ALTWT instruction is executed. The process is awakened when input arrives on one of the channels enabled by prior ENBC instructions. The process isn't really sleeping, as much as listening for input on the specified channels. It simply doesn't continue past the ALTWT instruction until some input arrives.

As you might expect, in some cases input may never arrive on the channels specified. In this case you would want to only listen for a specified amount of time and then "timeout," that is continue on, if no input arrives. Alternative timer constructs give the transputer the ability to timeout on an event; this corresponds to a "default" action if a message does not arrive in a specified amount of time.

Alternative timer constructs are very similar to regular alternative constructs, except that the instruction TALT is used in place of ALT and TALTWT is used in place of ALTWT. The overall structure of TALT construct is:

```
TALT ; flag the start of an alternation
enable all channels to listen to
TALTWT ; wait until a message arrives or timeout
disable all the channels
ALTEND ; causes jump to the routine associated
; with the channel the message arrived on
service routines (equivalent to C switch statement "cases")
```

Note: The Workspace Pointer must not change between the execution of the ALT or TALT instruction and the ALTEND instruction

ALT Alt start stores the flag 0x80000001 in workspace location -3 (State.s) to show that the enabling of an ALT construct is occuring.

ALTWT Alt wait stores 0x80000001 (-1) in workspace location zero (disablestatus) and waits until State.s is ready (i.e, contains 0x80000003). This means that a flag is stored to show that no branch has been selected yet and the process is descheduled until one of the guards is ready.

ALTEND Alt end is executed after the process containing the ALT has been rescheduled. Workspace location zero contains the offset from the instruction pointer to the guard routine to execute. This offset is added to the instruction pointer, and execution continues at the appropiate guard's service routine.

ENBS Enable skip sets a flag to show that the guard is ready. It performs this by storing ready, 0x80000003, into State.s. All guards must be enabled prior to the alternate wait instruction.

DISS Disable skip disables a skip for a guard that was previously initialized with ENBS. A contains an offset to the guard to disable; B contains a flag that allows it to decide whether to select this guard or not.

ENBC Enable channel enables a channel that has been pointed to by the B register only if the contents of A is true (one); otherwise, the channel is not enabled. If A is true, there are three cases:

1. No process is waiting on the channel pointed to by B. In this case, ENBC stores the current process workspace descriptor into the channel to initiate communication.

2. The current process is waiting on the channel pointed to by B. In this case, nothing is done and the instruction is ignored.

3. Another process is waiting on the channel pointed to by B. Here the ready flag (0x80000003) is stored at workspace location -3 to show that the guard is ready.

DISC Disable channel disables a channel pointed to by C. If the flag in B is true (one) and no other guard has been selected, this guard is selected as the next execution branch. A contains an offset to the routine that starts the process. If the guard is taken, A is stored in workspace location zero and a true flag is returned in A. If the guard is not taken, then a false flag is returned in A. In essence, one of these two cases holds:

1. B is true, the channel pointed to by C is ready and no other branch has been selected. This branch will then be selected by saving A in workspace location zero; A will be set to true.

2. B is false or the channel pointed to by C is not ready, or another branch was already selected, in which case A gets set to false (zero).

TALT Timer alt start sets up an alternative input for the current process. It also stores 0x80000002 in workspace location -4 (Tlink.s). This instruction is used instead of ALTWT in the case of timer guards.

TALTWT Timer alt wait is similar to the ALTWT instruction; however, TALTWT must be used with timer guards. Just as ALTWT does, TALTWT places 0x80000001 into the workspace location zero. If the following conditions don't exist, it will cause the process to be descheduled:

- If State.s (workspace location -3) is not ready; or
- If Tlink.s isn't 0x80000001 with a time in location -5 that is in the past.

If the above conditions do exist, then the process continues on.

ENBT Enable timer enables a timer guard. A register contains a flag (0 for false and 1 for true) and B register contains a time. Assuming the flag is true, then either:

1. "Alt timer not seen yet" (Tlink.s is 0x80000002), and the flag to set the time will be set, and the "alt time" will be set to the time of the guard. Tlink.s is set to 0x80000001 and is stored in location -5.

2. "Alt time set and earlier than this guard." In this case you should ignore this guard.

DIST Disable timer disables a timer guard in the same way as DISC disables a channel guard. C contains the time, the B register contains a flag, and A contains the offset to the guard's associated service routine. The disable timer instruction works in the following ways: If the flag in B is true, and the time (in workspace location -5) is later than the time in C, and no other branch has been selected, then this branch is selected and A is saved in workspace location zero; A is set to true (one).

Otherwise, if B is false or the time is earlier than the time in C or another branch has been taken, then A is set to false.

Example:

To listen to two channels at once using unconditional channel guards:

```
alt ; flag the start of an alternation

ldc channel1 ; load the address of the 1st channel
ldc 1 ; load flag (always TRUE)
enbc ; enable the channel

ldc channel2 ; load the address of the 2nd channel
ldc 1 ; load flag (always TRUE)
enbc ; enable the channel
```

```
altwt ; start listening to both channels

ldc channel1 ; load the address of channel1
ldc 1 ; load flag (always TRUE)
ldc SERVICE1-ENDALT ; load offset to service routine
disc ; disable the channel

ldc channel2 ; load the address of channel2
ldc 1 ; load flag (always TRUE)
ldc SERVICE2-ENDALT ; load offset to service routine
disc ; disable the channel

altend ; end alternation, jump to service routine
ENDALT
SERVICE1
ldc buffer1 ; load the address of the message buffer
ldc channel1 ; load the address of the channel
ldc messlength ; load the message length
in ; input the message from the channel
j CONTINUE
SERVICE2
ldc buffer2 ; load the address of the message buffer
ldc channel2 ; load the address of the channel
ldc messlength ; load the message length
in ; input the message from the channel
CONTINUE
```

Error Handling Operation Codes

The transputer has an Error flag that is similar to the overflow flag used by most conventional microprocessors. A major difference in the transputer's Error flag is that it is "sticky"; once set, it stays set until it is explicitly cleared. The state of the flag is sent out over an external pin on the transputer, so it is possible to detect any overflow using the hardware.

The main thing to consider when using the Error flag is that the transputer mostly uses signed arithmetic operations. So if you desire to use unsigned arithmetic, you will probably end up ignoring the Error flag. However, if you write programs that are designed with signed arithmetic in mind, then the Error flag may be used to check for overflow and to abort those routines that do overflow. An offending process can be stopped (removed from the transputer's run queue altogether) if overflow is encountered by using the STOPERR instruction.

Another flag on the transputer is the HaltOnError flag. This flag can force the whole transputer to halt if two conditions hold:

- The HaltOnError flag is set to one; and
- The Error flag changes from false to true (0 to 1).

If the Error flag is already set and you have set the HaltOnError flag, further set operations on the Error flag do not cause the transputer to halt.

The idea behind the Error and HaltOnError flags is that the offending processes, or even the transputer itself, can be shut down if an arithmetic error occurs preventing bad results from being passed on. The STOPERR instruction can halt a process if an overflow occurs; in this way it will not report erroneous results that could corrupt an entire computation. Similarly, so it will not report a bogus answer, an entire transputer that is performing some computation fragment (presumably with multiple processes working on it) can be shut down if an overflow occurs.

Note that the CSUB0 and CCNT1 checks use an unsigned comparison, meaning that the values it compares are considered nonnegative (for the sake of the comparison), even if they are not. In an unsigned comparison, -1 turns out to be greater than zero, since -1 is interpreted as a positive number represented by 0x80000000.

CSUB0 Check subscript from 0 sets the Error flag if the unsigned value in B is greater than or equal to the unsigned value in A.

CCNT1 Check count from 1 verifies that the value in B is greater than zero and less than or equal to the value in A.

Tip: INMOS suggests using CCNT1 to make sure that the count of an output or input instruction is greater than zero, yet less than the number of bytes in the message buffer.

TESTERR Test Error false and clear pushes the inverse of the current state of the Error flag into A. The Error flag is cleared as a result of this instruction.

SETERR Set Error sets the error flag.

Tip: If HaltOnError is set prior to the set Error instruction, and the Error flag was zero (meaning false), this instruction will cause all processes on the transputer to halt.

STOPERR Stop on error stops the process that is currently executing if the Error flag is set. Note the current process will be removed entirely and not simply halted or temporarily descheduled.

Note: STOPERR performs the same action as the HaltOnError flag when it is used, but STOPERR only affects the currently executing process (while HaltOnError affects all processes on the transputer).

CLRHALTERR Clear HaltOnError clears the HaltOnError flag (sets it to zero meaning false).

SETHALTERR Set HaltOnError sets the HaltOnError flag (sets it to one meaning true).

TESTHALTERR Test HaltOnError loads A with the state of the HaltOnError flag.

Examples:

To preform checked arithmetic, check the value of the Error flag:

```
add ; perform checked addition, Error is set on overflow
testerr ; test for overflow
cj overflow_handler ; Error was TRUE, jump to error handler

. . . ; otherwise no overflow, continue processing
```

To check if an index into an array is negative:

```
ldl index ; load the index (to check) for the array
mint ; load -1 (to check against)
csub0 ; perform the index check
testerr ; the Error flag it set if the index is negative
cj negative_index ; jump to handler if negative index
. . . ; otherwise index was nonnegative, continue
```

To check that an array index falls with a valid range, use the following:

```
ldl index ; load the index to check
mint ; load -1 (for comparison)
csub0 ; perform the check (that index > 0)
testerr ; the Error flag is set if index is less than zero
cj negative_index ; jump to handler for negative index
```

```
ldl MAXINDEX ; load the largest valid array index
csub0 ; perform the check (that index < MAXINDEX)
testerr ; the Error flag is set if index > MAXINDEX
cj index_too_big ; jump to handler for indexes too large
```

The index instructions can also be used to test boolean expressions. To check if a value is TRUE (i.e., the expression is not zero), use:

```
ldl value ; load the value to test
ldc 1 ; load 1 (to check against)
csub0 ; perform the check
testerr ; the Error flag it set if the value was TRUE
cj value_true ; jump to handler if value was TRUE
. . . ; otherwise value was FALSE
```

To check to see if the result of some expression is FALSE (i.e., equal to zero), use:

```
ldl value ; load the value to test
ldc 1 ; load 1 (to check against)
ccnt1 ; perform the check
testerr ; the Error flag it set if the value was FALSE
cj value_false ; jump to handler if value FALSE
. . . ; otherwise value was TRUE
```

Processor Initialization Operation Codes

The transputer has various setup instructions that are used to configure the transputer when it boots. The transputer keeps track of two sets of pointers to two queues. These two queues are the linked list of processes that the transputer is to execute one for high priority and one for low priority processes. Both the head (first element) and tail (last element) pointers to these queues are saved in special internal registers.

Queue Control Registers

Fptr0: Pointer to front of high priority active process list.
Bptr0: Pointer to back of high priority active process list.
Fptr1: Pointer to front of low priority active process list.
Bptr1: Pointer to back of low priority active process list.

Tip: INMOS warns that explicit manipulation of the scheduling queues is dangerous, since the queues are manipulated both by scheduling instructions and external events (such as a link transfer completion). Also remember that any awakening high priority process will interrupt all low priority processes.

The idea is that the microcoded on-chip scheduler will be the only process affecting the run queues. You should not use the following instructions at any time other than bootup because they could be confused with the on-chip scheduler and cause unexpected results.

TESTPRANAL Test processor analysing returns a flag in A that is true (one) if the last time the transputer was reset it was an "analysis" instead of a reset. If a special analyze pin is asserted while the transputer is being reset, some of the state of the processor is saved to allow a post-mortem debug. This instruction is useful in debugging hardware.

SAVEH Save high priority queue registers stores the high priority process queue pointers to the address pointed to by A. Location A receives the head pointer, and location A+4 receives the tail pointer.

SAVEL Save low priority queue registers stores the low priority process queue pointers to the address pointed to by A. Location A receives the head pointer, and location A+4 receives the tail pointer.

STHF Store high priority front pointer initializes the high priority process queue-head pointer with the value in A.

STHB Store high priority back pointer initializes the high priority process queue-tail pointer with the value in A.

STLF Store low priority front pointer initializes the low priority process queue-head pointer with the value in A.

STLB Store low priority back pointer initializes the low priority process queue-tail pointer, with the value in A.

STIMER Store timer initializes both the low and high priority timers to the value contained in A. STIMER also starts the timers ticking.

Example:

To initialize the timer upon system startup, do:

```
ldc 0 ; load A with zero
sttimer ; start the timer ticking from zero
```

T800 Specific Instructions

The T800 version of the transputer has some instructions that are not available on the T414. These instructions fall into three categories:

- instructions that move two dimensional blocks of memory in a single instruction,
- instructions that check cyclic redundancy and bit operations; and
- floating point support instructions.

Block Move Operation Codes

The T800 has four instructions that efficiently perform two-dimensional block moves. These instructions move blocks of bytes, so if you have any information to transfer, it must terminate at byte boundaries. You should keep in mind that these instructions do not act as a bit copy.

You will need six parameters to use a two-dimensional block-move instruction. Since the transputer has only three registers, the two-dimensional move has to be performed in two instruction steps: the first instruction initializes three of the parameters; the second instruction passes the other three parameters and performs the desired move operation.

The six parameters necessary for performing the move are:

The address of the first element in the source block;
The address of the first element in the destination block;
The width (in bytes) of each row to be copied;
The length (number of rows) to be copied;
The stride of the source block (two-dimensional array); and
The stride of the destination block (two-dimensional array).

The first two parameters are just the start and target addresses for the copy. The width means the size in bytes of each element to copy. The length is the number of rows in the block to copy. The stride in an array is the number of bytes in a row (or number of elements in a row times the width).

The reason for differing source and target strides is so that you can perform a block copy of a two-dimensional array into another two-dimensional array where the source and target are different sizes (row and/or column lengths).

MOVE2DINIT The initialize data for two-dimensional block move instruction sets up three of the six necessary parameters for the other two-dimensional move instructions. Since some of the two-dimensional move instructions require six parameters, and the transputer has only three registers, a two-step instruction sequence is necessary to perform the two-dimensional move operation.

MOVE2DALL The two-dimensional block copy instruction copies an entire block of length-rows (each of which is width bytes) from the source to the destination. This instruction performs the actual move of one block to the destination.

MOVE2DNONZERO The two-dimensional block company nonzero bytes instruction copies only the non-zero bytes in the source block to the destination block. As a result, it leaves unchanged the bytes corresponding to zeroes in the destination block. Inmos suggests using this instruction to overlay a non-rectangular picture onto another picture (e.g., copy the picture and not the background on top of another picture). This is a byte copy and not a bit copy, so any picture edge would have to terminate on a byte boundary.

MOVE2DZERO The two-dimensional block copy zero bytes instruction copies the zero bytes in the source block to the destination block. As a result, it leaves unchanged the bytes corresponding to nonzero bytes in the source. Inmos suggests that you use this instruction to mask out a non-rectangular shape from a picture.

Example:

```
ldl stride_source ; stride of source block to copy
ldl stride_dest ; stride of destination block (target block)
ldl length ; number of rows in block to copy
move2dinit ; perform the initialization step
ldl source ; source address to copy from
ldl destination ; destination address to copy to
ldl width ; size in bytes of each element to copy
move2dall ; copy an entire block
```

Bit Operation Codes

The T800 also has some additional instructions (which the T414 does not) for bit manipulation within a 32-bit word. There are bit manipulation instructions used for counting the number of bits set (to one) in a word or for reversing the bit pattern of part or all of a word. Reversing bit patterns is useful for converting data generated on another computer system. Microprocessors such as the Motorola 68000 family are "big-endian," meaning they store the most significant bits in memory first. The transputer is "little-endian" (as are the Intel 80x86 processors) and stores the least significant bits in memory first. So, to convert data from one type of microprocessor to another, it is sometimes necessary to reverse the ordering of the bits in a word of data, and the T800 provides useful built-in instructions just for that purpose.

BITCNT Count bits set in word counts the number of bits that are set (to one) in A and adds that number to the contents of B. The idea is that a total number of bits set can be accumulated if so desired.

BITREVWORD Reverse bits in word reverses the bit pattern of the word held in A. Thus, if A were to contain the hexadecimal number 0x12345678 (which expands to 0001 0010 0011 0110 0101 0110 0111 1000 in binary), after the instruction was executed A would contain 0x1E6A2C48 (which expands to 0001 1110 0110 1010 0010 1100 0100 1000 in binary).

BITREVNBITS Reverse bottom n bits in byte reverses the bottom N bits in B, where N is the value in A. All bits in B more significant than N are cleared to zero prior to the reverse operation.

Example:

To convert a word from having the leftmost digit as most significant to the rightmost digit as most significant:

```
ldl left_sig ; load the left significant word
bitrevword ; reverse the bits
stl right_sig ; store into a right significant word
```

CRC Operation Codes

Due to noise on communication lines, part of a message may be garbled, perhaps a single bit, perhaps most of the information in the message. To preserve communication integrity, it is necessary to detect when errors in a message occur and then request rebroadcast of the garbled message. A primitive technique for such error detection is the use of a parity bit, which is an extra bit added on to a transmitted character. The parity bit indicates whether the number of set bits in the character sent is even or odd. This allows the receiver of the message to check the parity bit, with the character sent to ensure that none of the bits in the transmitted character were garbled.

However, parity checking on characters is not highly reliable and requires a large overhead (typically one bit for parity and seven for data). The next level up in checking for errors is the checksum. Performing a checksum is easy: Simply add all the characters in the message together and send the result. The receiver then adds the received characters together and verifies that its sum matches the sum sent by the sender. If it does not match, an error occurred in the transmission of the message. This technique has advantages over simple parity, notably its lower data transmission overhead (i.e., there are usually fewer checksum bits than parity bits in a long message). However, there are still drawbacks to the checksum for instance, if two characters are simply received out of sequence, the checksum will fail to detect the error (since addition is commutative).

A very reliable form of error checking with low overhead is desireable. Cyclic redundancy checking (CRC) is such a technique for verifying that data was communicated correctly. The basic technique that CRCs employ is to send the data as one large binary number (sequence of bits). Appended to the end of this number is a remainder value. The remainder value is computed by dividing the message data (encoded into a number) by a specially chosen, agreed upon value (with certain mathematical properties). The sender computes the remainder and sends it along after the message data. The receiver receives the message, divides the data part by the special agreed upon value, and then checks that the remainder it computes is the same as the one received in the message. If the remainder is the same, there is an extremely high probability that the message was received correctly.

The computation for a CRC is done via division. The mathematical abstraction is to encode a number into a polynomial and then divide it by another (generator) polynomial to compute the remainder. At the bit level, this just involves simple modulo two division. The only difference between modulo two division and ordinary binary division is that the intermediate results are obtained by using exclusive or instead of subtraction. The final answer computed from modulo two division is the same as that computed using binary division. The reason modulo two division is used (via employing the exclusive or operation) is to avoid having to compute the borrows or carries required for binary subtraction.

The T800 has two special instructions for computing CRCs. The agreed upon divisor value (or generator) is placed into the C register, the "accumulated thus far" CRC value is placed into the B register, and the data to perform the CRC on is placed into the A register. In the case of a single byte (using CRCBYTE), the data is located in the most significant (high) byte of the word.

The T800 computes the CRC value by shifting B and A left one place (as a double word), then exclusive or-ing C into B if the bit shifted out of B was set to one.

The CRC instructions are designed to be used inside of a loop construct if a message will not fit into a single byte or word.

Note: It is important to remember that the transputer computes CRCs in little-endian fashion and that this may differ with the machine the CRC values will be sent to or tested against.

CRCWORD Calculate CRC on word calculates the CRC value of the word held in A using the generator in C. B contains the CRC that has been accumulated over successive CRCWORD instructions.

CRCBYTE Calculate CRC on byte calculates the CRC value of the byte held in the most significant byte of A using the generator in the C register. B contains the CRC that has been accumulated over successive CRCBYTE instructions.

Example:

The following evaluates the CRC of a word-aligned message of length words:
```
ldc 0 ; load zero for start of loop
stl index ; store into index
```

```
ldl length ; load the message length
stl index+1 ; store into word past index (for LEND)
ldc generator ; load the CRC generator polynomial
ldc 0 ; load zero for clearing temporary vars
stl temp1 ; clear current CRC temp var
stl temp2 ; clear accumulated CRC temp var
LOOP: ldl index ; load loop index
ldl message ; load the message address
wsub ; index to the word to compute CRC on
ldnl 0 ; load a word of the message
ldl temp2 ; load temp2
rev ; reverse
ldl temp1 ; load temp1
rev ; reverse
crcword ; compute the CRC on the message word
stl temp1 ; store CRC of current word into temp1
stl temp2 ; store accumulated CRC into temp2
ldlp index ; load a pointer to index for LEND
ldc END-LOOP ; load relative offset for LEND
lend ; loop until done (length iterations)
END:
```

Floating Point Numbers

It is very common now for microprocessors to be marketed with sister coprocessors that perform additional functions. The most popular kind of coprocessor is one that performs floating point arithmetic (instead of the conventional integer arithmetic in the microprocessor). The designers of the transputer have added features to support floating point arithmetic to the T414 and in the case of the T800 have provided the floating point coprocessor on the same chip.

IEEE Standard for Binary Floating-Point Arithmetic

In 1985 the Institute of Electrical and Electronics Engineers issued a document specifiying how binary floating point arithmetic should be handled on computers. This is now known as the IEEE Standard for Binary Floating-Point Arithmetic (also referenced as IEEE Standard 754-1985) and is generally accepted in the computer industry as the way to deal with floating point arithmetic. All the major industry manufacturers support the format: Intel on the 8087, 80287, and 80387, Motorola on the 68881 and 68882, and Inmos on the T414 and T800, to name three. The standard describes how to represent floating point numbers in a computer’s memory (i.e., their binary representation) as well as the rules governing operations using floating point numbers such as how operations affect rounding, truncating, and accuracy.

Floating point numbers come in two flavors: 32-bit and 64-bit. This conforms to the usual FLOAT and DOUBLE data types in C, for instance. In general, operations on 32-bit floating point numbers are swifter than on 64-bit floating point numbers; however they carry less precision, while 64-bit floating point numbers are more precise, but are generally slower than 32-bit floating point operations.

The exact bit formats are represented in the following figure:

Single (32-bit) Floating Point Format

bits	1	2 to 9	10 to 32
meaning	sign	exponent	mantissa
size	1	8	23

Double (64-bit) Floating Point Format

bits	1	2 to 12	13 to 64
meaning	sign	exponent	mantissa
size	1	11	52

Note: The maximum numbers representable in these formats are approximately
3.4 * 10 ** 38 for single and 1.8 * 10 ** 308 for double. The smallest numbers representable in these formats are approximately 1.2 * 10 ** -38 for single and
2.2 * 10 ** -308 for double.

All floating point numbers are composed of three parts: a sign bit, a biased exponent, and a fraction or mantissa. The sign bit indicates whether the number is positive (sign bit equals zero) or negative (sign bit equals one). The mantissa represents a fractional number between zero and one. Usually it is normalized, meaning its leftmost bit is set to one, however, if you assume all valid floating point numbers are normalized, you can assume the leftmost bit of the mantissa is a one, hence you can achieve an extra bit of precision by leaving out the implied leftmost one bit for normalized numbers. The IEEE standard and the transputer implementation infer this extra bit of precision for normalized numbers.

The exponent represents a power of two to multiply the mantissa by. Since it is desirable to represent very large and very small numbers in floating point format, the exponent is biased. Biasing means that instead of using one of the exponent bits as a sign bit to indicate whether the exponent is positive or negative, a fixed positive value is added to the exponent before it is represented in a floating point number. The problem with a sign bit is that an exponent of zero has two representations (one for each sign), thus wasting an opportunity to express more numbers. Biasing the exponent eliminates this probem while allowing the exponent to be considered as either positive or negative (note that, while raising a number to a large positive exponent makes the result large, raising a number to a large negative exponent makes the result small). In the case of single (32-bit) floating point numbers, the bias is 127. This means that 127 is subtracted from the exponent before it is raised to two (and then multiplied by the mantissa). In symbolic notation this means that for a single (32-bit) floating point number:

number = (-1)s 2e-127 (1 ï f) for 0 < e < 255
where s is the sign bit
e is the biased exponent
f is the fraction (mantissa)
(1 ï f) denotes the inferred one bit for a
normalized number

There are certain special problems that arise with floating point numbers. It may be the case that multiplying two numbers produces a number too small or too large for the format to handle. In these cases, the result should be rounded to be either plus or minus infinity, depending on the sign of the result. The IEEE standard provides for an encoding to represent infinity:

number = (-1)s infinity
where s is the sign bit
infinity is a number representing infinity

It may also be the case that the conversion from a decimal to a binary number yields a quantity out of range or an operation occurs that has no meaning, like dividing by zero or adding positive infinity to negative infinity. In these cases, a value arises that's Not a Number or a NaN for short. There is also a special encoding for NaNs. The single floating point number case is:

If e = 255 and f is not equal to zero, then the number
is a NaN regardless of s.
where s is the sign bit
e is the biased exponent
f is the fractional part of the number (including the
inferred one bit in normalized numbers)

Another special case arises when the mantissa is no longer normalized. Usually, the most significant bit of the mantissa is assumed to be one (normalized), so it is not actually present in the floating point representation. This yields an extra bit of precision for the mantissa and the number. Now suppose you divide a number between zero and one by the largest possible number. The result would represent a fraction of the smallest possible number. It would be convienent to relax the rule requiring the most significant mantissa bit to be inferred as one so that these numbers could be represented, although a loss of precision has occurred since you have lost the inferred one bit. These numbers are called denormalized numbers, since they are no longer normalized. In the case of denormalized numbers, the most significant mantissa bit will be zero and the exponent will always be the same as the exponent of the smallest representable floating point number.

Note: The IEEE standard calls for an underflow exception when a number becomes denormalized. The underflow exception is not implemented on the transputer, so it is difficult to detect this loss of precision when a number becomes denormalized.

The rule for interpreting denormalized numbers (for single floating point numbers) is:

if e = 0 and f does *not* equal zero,
then number = (-1)s 2e-126 (0 ï f)
where:
s is the sign bit
e is the biased exponent
f is the fraction (mantissa)

(0 ï f) denotes that the most significant bit of
f is zero (and also implies the number is
denormalized).

Finally, there is a special case representation for the number zero:

If e = 0 and f = 0, then the number equals 0.
where e is the biased exponent and f is the fraction.

There are analagous special cases for double (64-bit) floating point numbers. The following figure indicates the five rules for interpreting or encoding floating point numbers:

Single (32-bit) Floating Point Number Interpretation

s is a one-bit sign bit
e is an eight-bit biased exponent (the bias is 127)
f is a 23-bit fraction

[1] If e = 255 and f is *not* zero, then the number is a NaN
(regardless of s)

[2] If e = 255 and f = 0, then the number = (-1)s infinity

[3] If 0 < e < 255, then number = (-1)s 2e-127 (1 ï f)

[4] If e = 0 and f is *not* zero then number = (-1)s 2e-126 (0 ï f)
(denormalized numbers)

[5] If e = 0 and f = 0, then the number equals zero.

Double (64-bit) Floating Point Number Interpretation

s is a one-bit sign bit
e is an 11-bit biased exponent (the bias is 1023)
f is a 52-bit fraction

[1] If e = 2047 and f is *not* zero, then the number is a NaN
(reguardless of s)

[2] If e = 2047 and f = 0, then the number = (-1)s infinity

[3] If 0 < e < 2047, then number = (-1)s 2e-1023 (1 ï f)

[4] If e = 0 and f is *not* zero, then number = (-1)s 2e-1022 (0 ï f)
(denormalized numbers)

[5] If e = 0 and f = 0, then the number equals zero.

Floating Point Errors

There are error conditions in integer arithmetic, notably division by zero and overflow. There are considerably more types of arithmetic errors that can occur in floating point arithmetic than in ordinary integer arithmetic. What types of errors can occur and how to interpret them are important facets of floating point arithmetic.

The IEEE 754 Standard calls for five testable flags (say encoded into a status word) to test for five kinds of floating point arithmetic errors: invalid operation, division by zero, overflow, underflow, and inexact. Inmos deviated from the standard on this point when designing the floating point unit for the T800. The transputer designers argue that one Error flag is sufficient since correct programs shouldn’t generate floating point arithmetic errors and, in any case, floating point errors are still detected. The level of discrimination on what the error was is not as fine as what the IEEE 754

Standard calls for (in some cases, underflow and inexact errors are not supported at all, while other types of errors are diagnosed completely). The advantage to only one flag is simpler and, thus faster, hardware. The 20 MHz T800 achieves 1.5 megaflops (million floating point operations per second), which is an order of magnitude above other floating point coprocessors.

Note: The two missing exceptions on the transputer are underflow and inexact. Underflow occurs when an extremely small positive or negative number is generated (i.e., a number that is very close to zero); such a number cannot be represented in a normalized form. Operations resulting in denormalized numbers should cause an underflow exception according to the IEEE standard. The inexact exception should occur when an operation yields a floating point number that is not an exact representation of the actual number (i.e., some form of rounding has taken place).

The floating point unit of the T800 does provide for certain error detection despite not conforming to the full IEEE standard. Any invalid floating point operations set the floating point Error flag called FP_Error. (Note: It is also possible to construct programs that do not check for floating point errors.) Invalid operations set the FP_Error flag and return an encoded NaN which represents the type of error that occurred. Divison by zero sets FP_Error and returns either plus or minus infinity (depending on the numerator’s sign). Overflow sets FP_Error and rounds the result appropiately.

Note: Any operation involving a NaN or an infinity sets FP_Error.

Infinity is encoded as all of the exponent bits set and a mantissa of zero. This means that, for the transputer, infinity is represented as:

Type	Single	Double
positive infinity	7F800000	7FF00000 00000000
negative infinity	FF800000	FFF00000 00000000

The value of a Not a Number signifies the type of evaluation error that occurred. The following table illustrates what the various errors NaNs represent. Note that the numbers are in hexadecimal.

Error	**Single**	**Double**
Divide zero by zero	7FC00000	7FF80000 00000000
Divide infinity by infinity	7FA00000	7FF40000 00000000
Multiply zero by infinity	7F900000	7FF20000 00000000
Addition of opposite signed infinities	7F880000	7FF10000 00000000
Subtraction of same signed infinities	7F880000	7FF10000 00000000
Negative square root	7F840000	7FF08000 00000000
Double to Single NaN conversion	7F820000	7FF04000 00000000
Remainder from infinity	7F804000	7FF00800 00000000
Remainder by zero	7F802000	7FF00400 00000000

Inmos has also added three new NaN encodings (not in the IEEE standard):

Error	**Single**	**Double**
Result not defined mathematically	7F800010	7FF00002 00000000
Result unstable	7F800008	7FF00001 00000000
Result inaccurate	7F800004	7FF00000 80000000

T414 Floating Point Support

There are five instructions unique to the T414, which provides a measure of support for single-length floating point arithmetic. Note that the term "guard word" refers to an extension of the fractional part of a floating point number into another word. This additional word is used to store extra bits of accuracy so that precision will not be lost during the floating point computation. If the guard word is zero, the floating point number is said to be exact.

CFLERR Check single length floating point infinity or NaN examines the value held in the A register and sets the Error flag if it is a floating point infinity or NaN.

UNPACKSN Unpack single length floating point number decodes the IEEE format single-length floating point number contained in the A register and returns its exponent in B and the fractional field (mantissa) in A. The implied, most significant bit in normalized numbers is not included in the return value for A.

ROUNDSN Round single-length floating point number encodes initial values of C containing an exponent, B containing the fraction or mantissa, and A containing the guard word into a single-length, appropriately rounded floating point number (returned in A).

POSTNORMSN Post-normalize single-length floating point number is intended to be used after the NORM instruction to provide normalization correction.

LDINF Load single length infinity loads the single-length floating point number for positive infinity (0x7F800000) into the A register.

An example of using the T414 floating point instructions is the packing of a number into floating point format. Suppose you have an exponent, guard word, fraction, and sign bit in integer format and you wish to construct a floating point number. The code is as follows:

```
ldl exp ; load the exponent
stl 0 ; save it on the workspace in location 0
ldl frac ; load the fractional part of the number
ldl guard ; load the guard bits to protect precision
norm ; normalize the fractional part
postnormsn ; complete the normalization
roundsn ; round the number to single length
ldl sign ; load the sign for the number
or ; OR in the sign bit
```

After completing this code sequence, A will contain a single-length, IEEE format floating point number. Note that workspace location zero is used and the sign should be present only in the most significant bit of "sign," all other bits of "sign" should be zero.

T800 Floating Point Support

The T800 has an on-chip floating point arithmetic engine that conforms to the IEEE 754 Standard for floating point arithmetic. In addition to the three deep integer register stack A, B, C, the T800 has a three deep floating point register stack FA, FB, FC. Each floating point register can hold either a 32-bit single or 64-bit double floating point number, and each register also has associated with it an internal flag signifying the length of the data it contains (double or single). The floating point stack behaves in a manner similar to the integer stack, as values are pushed onto and popped off it.

In order to produce efficient microcode, certain floating point instructions are not executed as indirect instruction sequences using the operand register and OPERATE. Instead, the desired floating point instruction is loaded into the A register, and a special floating point instruction called FPENTRY is executed. This causes the floating point instruction in A to be executed. So, while all of the floating point instructions are represented by single mnemonic names, the actual code generated is either an indirect instruction sequence that ends with OPERATE or a floating point indirect instruction sequence that is comprised of loading the A register and executing FPENTRY. For example, FPUABS is equivalent to the instruction sequence LDC 0x0B ; FPENTRY (just as AND is equivalent to PFIX 4 ; OPR 6 in the indirect instruction case).

Load and Store Operation Codes

Values are pushed onto the floating point register stack via use of special floating point nonlocal load instructions. The address of the value to load is held in the A register on the integer stack. Similarly, values are popped off the register stack into memory by special store nonlocal instructions. Again, the address is held in A.

Note: For 64-bit double length numbers, the least significant word is contained at the lower address and the most significant word at the higher address.

As an optimization, some of the load instructions also perform an operation. These instructions are equivalent to other pairs of instructions but are executed faster than the equivalent instruction pair.

FPLDNLSN Floating load nonlocal single pushes the single-length floating point number pointed to by A into FA.

FPLDNLDB Floating load nonlocal double pushes the double-length floating point number pointed to by A into FA.

FPLDNLSNI Floating load nonlocal single indexed pushes the single-length floating point number pointed to by A and indexed by B into FA. It is equivalent to the instruction sequence WSUB ; FPLDNLSN.

FPLDNLDBI Floating load nonlocal double indexed pushes the double -length floating point number pointed to by A and indexed by B into FA. It is equivalent to the instruction sequence WSUBDB ; FPLDNLDB.

FPLDNLADDSN Floating load nonlocal and add single loads the single-length floating point number pointed to by A into FA and then adds FA and FB. This instruction is equivalent to FPLDNLSN ; FPADD.

FPLDNLADDDB Floating load nonlocal and add double loads the double-length floating point number pointed to by A into FA and then adds FA and FB. This instruction is equivalent to FPLDNLDB ; FPADD.

FPLDNLMULSN Floating load nonlocal and multiply single loads the single-length floating point number pointed to by A into FA and then multiplies FA and FB. This instruction is equivalent to FPLDNLSN ; FPMUL.

FPLDNLMULDB Floating load nonlocal and multiply double loads the double-length floating point number pointed to by A into FA and then multiplies FA and FB. This instruction is equivalent to FPLDNLDB ; FPMUL.

FPSTNLSN Floating point store nonlocal single pops the single floating point number in FA and stores it at the address pointed to by A.

FPSTNLDB Floating point store nonlocal double pops the double floating point number in FA and stores it at the address pointed to by A.

FPSTNLI32 Store nonlocal int32 truncates FA into a 32-bit integer value (FA can be double or single) and then stores that integer value at the address pointed to by A.

To load the floating point number at the workspace location (Wptr + 2) as a single 32 bit floating point number, use the code sequence:

```
ldlp 2 ; load a pointer to the number’s location
fpldnlsn ; perform the fp load single
```

To load the double floating point number located at (Wptr + 1) and (Wptr + 2) as a double 64 bit floating point number use the code sequence:

```
ldlp 1 ; load a pointer to the number’s location
fpldnldb ; perform the fp load double
```

General Operation Codes

The T800 has two general operations for reversing the two top floating point stack elements as well as duplicating the top stack element. There is also an instruction similar to OPR that executes floating point instructions in an indirect manner.

FPENTRY FPENTRY causes the value in A to be executed as a floating point unit instruction. The value is actually an entry point into the microcode ROM on the floating point unit. The FPENTRY instruction is similar to the OPERATE instruction, except that it is used exclusively for floating point related instructions.

FPREV FPREV reverses the top of the floating point stack, swapping FA and FB.

FPDUP FPDUP duplicates the top of the floating point stack.

So instead of:

```
ldlp 1 ; load a pointer to a single fp number
fpldnlsn ; load the number
ldlp 1 ; load the pointer again
fpldnlsn ; load the number again
```

Use:

```
ldlp 1 ; load a pointer to a single fp number
fpldnlsn ; load the number
fpdup ; duplicate it
```

Arithmetic Operation Codes

Naturally, the T800 floating point unit has instructions to perform the usual arithmetic operations. Two of the operations it can perform require multiple instruction sequences to perform an evaluation. These are the square root and remainder operations. The square root operation requires more intermediate steps for a double-length floating point number than for a single-length number. Other instructions are provided for common operations, such as multiplying and dividing by two. For special use in conversion routines, multiplying and dividing by 2**32 is also provided.

Note: For instructions performing arithmetic operations on more than one floating point register, all registers involved in the computation must be the same format (either all single or all double).

FPADD Floating point add adds FB to FA.

FPSUB Floating point subtract subtracts FA from FB.

FPMUL Floating point multiply multiplies FA by FB.

FPDIV Floating point divide divides FB by FA.

FPUABS Floating point absolute value replaces FA with the absolute value of FA. This is done by making the sign bit positive (cleared to zero). Note that the sign bit of a NaN will be cleared even though the absolute value of a NaN is meaningless.

FPREMFIRST Floating point remainder first step initiates the first step in the code sequence necessary to compute a floating point remainder (or modulus). The evaluation performed is FB REM FA, leaving the result in FA.

Note: The code to implement the remainder operation is:

```
fpremfirst
eqc 0
cj next
loop: fpremstep
cj loop
next:
```

FPREMSTEP Floating point remainder iteration step is an intermediate step in the code sequence necessary to compute a floating point remainder.

FPUSQRTFIRST Floating point square root first step initiates the first step in the code sequence necessary to perform a square root operation. The square root operation requires multiple instructions and performs the square root on the value in FA.

The code sequence for a single-length square root is:

```
fpusqrtfirst
fpusqrtstep
fpusqrtstep
fpusqrtlast
```

The code sequence for a double-length square root is

```
fpusqrtfirst
fpusqrtstep
fpusqrtstep
fpusqrtstep
fpusqrtstep
fpusqrtstep
fpusqrtlast
```

FPUSQRTSTEP Floating point step is an intermediate step in computing a square root.

FPUSQRTLAST Floating point square root end is the last step in performing a square root operation. If rounding other than round nearest is required, it should be specified just before this instruction.

For example, the code sequence for a single-length square root rounded to zero is:

```
fpusqrtfirst
fpusqrtstep
fpusqrtstep
fpurz
fpusqrtlast
```

FPUEXPINC32 Floating multiply by 2**32 multiplies FA by 2**32 and rounds the result as specified by the current rounding mode.

FPUEXPDEC32 Floating divide by 2**32 divides FA by 2**32 and rounds the result as specified by the current rounding mode.

FPUMULBY2 Floating multiply by 2 multiplies FA by 2.0 and rounds the result as specified by the current rounding mode.

FPUDIVBY2 Floating divide by 2 divides FA by 2.0 and rounds the result as specified by the current rounding mode.

Rounding Operation Codes

There are four types of rounding specified in the IEEE 754 Standard, and the T800 supports all four. The default mode is rounding to the nearest number, and this is used if no type of rounding is selected. It is important to note that unless round to nearest is desired, the rounding mode must be set prior to each floating point operation, since the rounding mode is reset to round nearest after each floating point operation. This avoids the need to store the rounding mode of a process if it is descheduled.

Note: None of the floating point instructions cause the current process to be descheduled, they are all atomic.

FPURN Set rounding mode to round nearest sets the rounding mode to nearest rounding.

FPURZ Set rounding mode to round zero sets the rounding mode to zero or truncation rounding.

FPURP Set rounding mode to round positive sets the rounding mode to round towards plus infinity.

FPURM Set rounding mode to round minus sets the rounding mode to round towards minus infinity.

For example, to round the sum of two numbers towards minus infinity:

```
ldlp 3 ; load a pointer to a single fp number
fpldnlsn ; load the number
ldlp 4 ; load pointer to the next single fp number
fpldnlsn ; load the next number
fpurm ; set the rounding mode to minus infinity
fpadd ; perform the addition
```

To round the sum of three numbers towards plus infinity:

```
ldlp 1 ; load a pointer to a double fp number
fpldnldb ; load the number
ldlp 3 ; load pointer to the next double fp number
fpldnldb ; load the next number
ldlp 5 ; load the pointer to the last double fp number
fpldnldb ; load the last number
fpurp ; set the rounding mode to plus infinity
fpadd ; perform the addition
fpurp ; set the rounding mode again
fpadd ; perform the addition
```

Error Operation Codes

There are four instructions for testing, setting, and clearing the floating point Error flag FPError.

FPCHKERROR Check floating error sets Error to the logical OR of Error and FPError. This is intended for use in checked arithmetic.

Note: If the HaltOnError flag is set, FPCHKERROR causes the transputer to halt when FPError is set to TRUE.

FPTESTERROR Test floating error false and clear sets A to true (one) if FPError is clear or false (zero) if FPError is set. FPError is then cleared.

FPUSETERROR Set floating point error sets FPError.

FPUCLEARERROR Clear floating point error clears FPError.

For example, to check for overflow on multiplication:

```
ldlp 3 ; load a pointer to a single fp number
fpldnlsn ; load the number
ldlp 4 ; load pointer to the next single fp number
fpldnlsn ; load the next number
fpuclrerr ; clear the FP_Error flag
fpmul ; perform the multiplication
fpchkerr ; check the FP_Error flag
```

Comparison Operation Codes

The T800 provides for the variety of comparisons that the IEEE 754 Standard calls for. Note that, in all of the comparisons, if a NaN (Not a Number) is one of the operands, FPError will be set.

FPGT Floating point greater than tests if FA > FB.

FPEQ Floating point equality tests if FA = FB.

FPORDERED Floating point orderability tests to see if FA and FB can be ordered, that is they are not NaNs. A is set true if both FA and FB are not NaNs; otherwise, A is cleared to false.

Tip: The IEEE says that NaNs are not comparible with anything, meaning that X compare Y is always false if either X or Y is a NaN. The IEEE comparisons for floating point numbers can be implemented via:
Unordered fpordered; eqc 0
IEEE greater than fpordered; fpgt; and
IEEE equality fpordered; fpeq; and

FPNAN Floating point Not a Number tests whether FA is a NaN. If so, A is set true; otherwise A is cleared to false.

FPNOTFINITE Floating point finite tests whether FA is a NaN or an infinity. If so, A is set true; otherwise, A is cleared to false.

FPUCHKI32 Check in the range of type int 32 checks to see if the value in FA lies in the range of a 32-bit integer. If so, it will be possible to convert it to a 32-bit integer value.

FPUCHKI64 Check in the range of type int 64 checks to see if the value in FA lies in the range of a 64-bit integer. If so, it will be possible to convert it to a 64-bit integer value.

For example, to compare two floating point numbers with a check that the comparison was not meaningless (i.e., one of the numbers was not a NaN):

```
ldlp 3 ; load a pointer to a single fp number
fpldnlsn ; load the number
ldlp 4 ; load pointer to the next single fp number
fpldnlsn ; load the next number
fpordered ; check for NaNs in FA and FB
fpgt ; perform the comparison
and ; combine the orderability and gt results
eqc 0 ; invert the flag for cj following
cj FALSE_COMPARE ; the greater than was false
. . . ; continue here, the greater than was true
```

Conversion Operation Codes

The T800 provides instructions to convert numbers from floating point format to integer format and visa versa. To perform some of the conversions, more than one instruction may be necessary.

FPUR32TOR64 Real 32 to Real 64 converts a single-length floating point value in FA to a double-length value. This is an exact conversion that does not involve rounding.

FPUR64TOR32 Real 64 to Real 32 converts a double-length floating point value in FA to a single length floating value using the current rounding mode.

FPRTOI32 Real to Int 32 is a single instruction that is equivalent to the sequence: fpint ; fpuchki32. This is used to convert a floating point number to an integer value in the floating point format and check that the conversion was valid.

FPI32TOR32 Int 32 to Real 32 takes a 32-bit integer value pointed to by A and rounds it to a single-length floating point number that is pushed into FA.

FPI32TOR64 Int32 to Real 64 takes a 32-bit integer value pointed to by A and converts it to a double-length floating point number that is pushed into FA.

FPB32TOR64 Bit32 to Real 64 or load unsigned word as Real 64 takes the unsigned 32-bit integer value pointed to by the address in A and converts it to a double-length floating point number that is pushed into FA.

FPUNOROUND Real 64 to Real 32 without rounding converts a double-length floating point number to a single-length floating point number without rounding the mantissa. This only works for normalized numbers and zeroes and is intended to remove the possible introduction of double rounding errors when 64-bit integers are converted into single-length floats.

FPINT Round to floating integer converts a floating point number held in FA to an integer value in the floating point format by simply rounding it (using the current rounding mode).

For the following examples, A is assumed to contain an address pointing to the number to convert.

An example of converting a 32-bit integer to a single-length floating point number rounded to minus infinity is:

```
fpurm ; set rounding mode to minus infinity
fpi32tor32 ; convert the 32-bit integer to a single FP number
```

To convert a 32-bit integer to a double-length floating point number with the default rounding (round to nearest) use: fpi32tor64.

The most common code for conversion from a floating point number to an integer is:

```
fpint ; perform the conversion to integer
fpuchki32 ; check that the conversion was valid
```

The transputer has one instruction that is equivalent to the above two instructions: fprtoi32. This is faster and should be used instead of the two-instruction sequence.

An example of converting a single (or double) floating point number to a 32-bit integer with error checking is:

```
fpint ; perform the conversion of FA to an integer
fpuclrerr ; clear the FP_Error flag
fpuchki32 ; check that the conversion was within range
fpchkerr ; check the result of the FP_Error flag
testerr ; check the Error flag (fpchkerr will set Error if
; FP_Error is set)
cj ERROR ; jump to error handling routine on error
fpstnli32 ; otherwise conversion was valid, save converted number
```

Note: The above example works on both single and double floating point numbers since FPINT is used for converting both types.

An example of converting a 64-bit integer to a double floating point number rounded to nearest (i.e., the default rounding) is:

```
dup ; duplicate the address of the 32-bit integer in A
fpb32tor64 ; convert the lower portion of the integer as unsigned
ldnlp 1 ; load a pointer to the second word of the 64-bit integer
fpi32tor64 ; convert the upper portion of the integer (signed)
fpuexpinc32 ; multiply the upper portion by 2**32
fpadd ; add the lower and upper portions (using the default
; rounding mode)
```

An example of converting a 64-bit integer to a single floating point number rounded to plus infinity is:

```
dup ; duplicate the address of the 64-bit integer in A
fpb32tor64 ; convert the lower portion of the integer as unsigned
fpunoround ; convert the lower portion double to a single
ldnlp 1 ; load a pointer to the second word of the 64-bit integer
fpi32tor64 ; convert the upper portion of the integer (signed)
```

```
fpunoround ; convert the upper portion double to a single
fpuexpinc32 ; multiply the upper portion by 2**32
fpurp ; set rounding mode to plus infinity
fpadd ; add the lower and upper portions
```

Optimizing Floating Point Calculations

The main processor and floaing point arithmetic engines on the T800 transputer are sufficiently disconnected that to a degree they can execute instructions concurrently. This means that, while an instruction on the floating point engine is still executing, the next instruction in the program can be fetched and run on the integer unit. The primary optimization to take advantage of is the ability to continue to execute integer instructions while a floating point instruction (which requires a relatively long time) is executing. It is therefore desirable to write code that "overlaps" between the floating point unit and the main (integer) processor.

Floating point instructions fall into two classes. The first consists of instructions that can return values to the integer unit, such as FPEQ, which returns a boolean value into A (the result of testing FA and FB for equality). The second class consists of instructions that affect solely the floating point processor and make no changes to the state of the main (integer) processor. Instructions such as FPADD or FPMUL fall into this category, as they act only on the floating point registers. Only the second class of floating point instructions is suitable for exploiting concurrency between the two computing engines. Instructions of the first class cause the integer unit to wait until the floating point engine is finished executing and returns a value. However, instructions of the second class are merely communicated to the floating point engine (requiring about a two-cycle overhead) then the next instruction is executed.

When searching for techniques to optimize a piece of code, floating point instructions that take a long time to execute are a good place to start. Instructions such as FPMUL or FPDIV take 18 to 31 processor cycles to execute for double floating point numbers. In this time, the integer unit could calculate another address to be used for the next floating point operation. In the case of an FPMUL, the integer unit will communicate the FPMUL to the floating point unit and then proceed to execute the next instruction. If the next instruction is for the integer processor, the integer processor may execute it before the floating point unit is finished executing the FPMUL. In fact, the integer processor could execute several instructions before the FPMUL is finished. In this way, floating point computations can be overlapped with integer computations.

One of the main factors that introduces unnecessary delays into floating point code is calculating addresses where floating point numbers reside without overlapping the address calculation with the desired floating point computation. The main consideration is that the integer processor will sit idle waiting for the floating point unit to synchronize with it unless it is kept busy (calculating addresses, for example).

Programming the Transputer

Introduction

There is no learning substitute for reading and understanding actual programs. With this in mind, and with the caveat that all programs are subject to the peculiar hardware configurations of any arbitrary system (especially for I/O), source-level transputer assembler program fragments are presented here. Each addresses a specific topic related to the transputer. The source code should be relatively portable.

Booting a Transputer

The first job of any programmer dealing with fresh hardware will be to "boot" or start the transputer. When the transputer hardware wakes up, it takes certain actions in a certain order. To start the software environment correctly, it is important to understand what actions the transputer performs when it "wakes up." Note that the terms "held high" and "held low" below mean "held at logical high voltage" and "held at logical low voltage."

The transputer is usually "reset," meaning the Reset pin on the transputer is held high (which performs a system reset); however, it is also possible to reset the transputer not to run it, but to examine its internal state. This is done by holding the Analyze pin high. Then a program can execute the TESTPRANAL instruction to check if the processor is being analyzed and take appropiate actions to help debug hardware. It is beyond the scope of this book to discuss hardware analyzing the transputer.

Assuming reset (and not analysis), there are two modes that the transputer can be powered up in: boot from ROM or boot from a link. The boot mode is controlled by a pin on the transputer package. If the BootFromROM pin is held high, the transputer will boot from a ROM (Read-Only Memory); otherwise, if the BootFromROM pin is held low, the transputer will boot from a link.

Booting from ROM

If the BootFromROM pin is held high, the transputer will try to execute instructions starting at a certain location in its address space. The address where the transputer will look for the first instruction is at 0x7FFFFFFE. Notice that this is almost at the top of the transputer’s address space (since addresses on the transputer are signed). This only leaves two bytes where instructions can be placed, namely at addresses 0x7FFFFFFE and 0x7FFFFFFF.

The usual practice is to place a "backwards jump" at this location, consisting of one prefix byte and one jump instruction. The reason this is done is to allow transfer of execution to some bootstrapping program (where there is more room in memory for it to execute). Obviously, there is precious little even the transputer can do in two bytes! Since the operand to the jump can only be two nibbles (or one byte), the farthest distance that the backwards jump can take is 255 (= 0xFF) bytes. If this is not enough room to execute the user’s bootstrap program, the user can execute another, much longer backwards jump.

At power-up, the workspace pointer points to MemStart, the start of user-accessible on-chip memory. These are locations 0x80000048 on the T414 and 0x80000070 on the T800. The instruction pointer contains 0x7FFFFFFE. The "current" and only process is in low priority. Both the high and low priority clocks are stopped. Since it is possible to do a "post-mortem" analysis on the transputer, the Error and HaltOnError flags retain their previous value prior to the transputer being "reset." For the same reason, the A register contains the previous value of the instruction pointer and the B register contains the previous value of the workspace descriptor (workspace pointer with priority encoded into the low bit) prior to the reset.

Note: There is a bug in the transputer hardware associated with booting from ROM. See the "Transputer Timer Bug" section below for more on this.

Booting from a Link

If the BootFromROM pin is held low, the transputer will "listen" to its links and try to receive a message from the first link to become active. The message should consist of a small boot program. There are three actions the transputer can take, depending on the value of the first byte of the received message.

If the value of the first byte of the received message is zero, the transputer expects to receive two more words. The first word is an address and the second word is data to write to that address. The transputer writes the data to the address and then returns to its previous state of listening to its links. These messages can be used to initialize memory.

If the value of the first byte of the received message is one, the transputer expects to receive one more word that contains an address. After receiving that word (containing the address), the transputer reads the data at that address and sends it out the output channel of the same link the message came in on. The transputer then returns to its previous state of listening to its links. These messages can be used to query the state of a transputer's memory.

If the value of the first byte of the received message is two or greater, the transputer inputs that number of bytes (2 or greater, whatever the value of the first byte was) into its memory starting at MemStart (the beginning of user memory in on-chip RAM). Then, after receiving the entire message, it transfers execution to MemStart, that is begins running the program that was sent in the message. Note that since the entire message length must be represented in one byte, the maximum size of a boot program is 255 bytes (since the largest number representable in one byte is 255). Such a boot program may in fact be only the first stage of a larger boot program, since the initial boot program may simply be designed to receive a much larger program over a link.

There are several tasks that this first-stage boot program must preform. Namely, the high and low priority process queue front pointers must be initialized (to NotProcess.p, which is 0x80000000). This must be done before running any processes. The timer queue pointers must also be initialized to NotProcess.p. The clocks are stopped and should be started by executing store timer (STTIMER) instructions. Also, the Error and HaltOnError flags should be initially cleared. In addition, the bottom nine words of memory should also be set to NotProcess.p (these are the hardware link channels and the event channel). The following is a sample first-stage boot program (for booting over a link):

```
mint ; load A with NotProcess.p = mint = 0x80000000
sthf ; initialize high priority queue front pointer
mint ; load A with NotProcess.p
stlf ; initialize low priority process queue pointer
mint ; load A with NotProcess.p
ldc TPtrLoc0 ; load high priority timer queue pointer offset
stnl 0 ; initialize high priority timer queue
mint ; load A with NotProcess.p
ldc TPtrLoc1 ; load low priority timer queue pointer offset
stnl 0 ; initialize high priority timer queue
ldc start_time ; load time to initialize clocks at (usually zero)
sttimer ; start the clocks
testerr ; clears the Error flag
clrhalterr ; or use sethalterr here depending what you want
mint ; load A with NotProcess.p
ldc 0x80000020 ; load address of Event channel
stnl 0 ; store NotProcess.p to Event channel
mint ; load A with NotProcess.p
ldc 0x8000001C ; load address of Link 3 input channel
stnl 0 ; store NotProcess.p to Link 3 input channel
mint ; load A with NotProcess.p
ldc 0x80000018 ; load address of Link 2 input channel
stnl 0 ; store NotProcess.p to Link 2 input channel
mint ; load A with NotProcess.p
ldc 0x80000014 ; load address of Link 1 input channel
stnl 0 ; store NotProcess.p to Link 1 input channel
mint ; load A with NotProcess.p
ldc 0x80000010 ; load address of Link 0 input channel
stnl 0 ; store NotProcess.p to Link 0 input channel
mint ; load A with NotProcess.p
ldc 0x8000000C ; load address of Link 3 output channel
stnl 0 ; store NotProcess.p to Link 3 output channel
```

```
mint ; load A with NotProcess.p
ldc 0x80000008 ; load address of Link 2 output channel
stnl 0 ; store NotProcess.p to Link 2 output channel
mint ; load A with NotProcess.p
ldc 0x80000004 ; load address of Link 1 output channel
stnl 0 ; store NotProcess.p to Link 1 output channel
mint ; load A with NotProcess.p
ldc 0x80000000 ; load address of Link 0 output channel
stnl 0 ; store NotProcess.p to Link 0 output channel
```

Note: In the above program, it would be better to use a loop to initialize the lower nine words of memory to NotProcess.p. This is left to the reader as an exercise. It would also be more efficient to replace:

```
mint ; load NotProcess.p into A
ldc TptrLoc0 ; TptrLoc0 is at 0x80000024
stnl 0 ; store NotProcess.p into TptrLoc0
mint ; load NotProcess.p into A
ldc TptrLoc1 ; TptrLoc1 is at 0x80000028
stnl 0 ; store NotProcess.p into TptrLoc1
```

by:

```
mint ; load NotProcess.p into A
mint ; load the starting address of on-chip RAM
stnl TptrLocOffset0 ; TptrLocOffset0 = 0x24
mint ; load NotProcess.p into A
mint ; load the starting address of on-chip RAM
stnl TptrLocOffset1 ; TptrLocOffset1 = 0x28
```

Where TptrLocOffset0 is the offset, from the bottom of on-chip memory to TptrLoc0 and TptrLocOffset1 is the offset to TptrLoc1. This is more efficent, since MINT loads the starting address of on-chip memory into the A register in one instruction. Then the STNL can use the offset to the appropiate TPtrLoc, rather than having to load the entire address. The impact of this is to reduce the number of prefix bytes necessary to compute the effective addresses; thus, the code is more efficient.

Behavior of the C Register

When the register stack is popped, (i.e., when A is replaced by B and B is replaced by C), Inmos has said the contents of the C register are undefined. However, in reality, the contents of the C register remain the same. It is possible to write programs that rely on this feature. Inmos has warned developers not to depend on this feature, however, as they may revoke it in future versions, but for now it is present on the transputer.

Soft Reset

Many processors have a special instruction to restart the machine from software control, rather than having to perform a manual reset from hardware. The transputer has such an instruction that is not documented by Inmos. The operation code for the instruction is the three byte sequence 0x21 0x2F 0xFF, which means:

```
21 PFIX 1 ; load the operand register with 1
2F PFIX F ; now the operand register contains 1F
FF OPR F ; operate (execute) the operand 1FF
```

This sequence will cause the transputer to perform a soft reset. The processor is left in a state as if a "hard" reset had been performed, meaning that the transputer can then be booted from ROM or a link. Note that soft reset does not refresh or initialize external memory.

Warning: Since this is an undocumented (and unsupported) instruction, it may be revoked (meaning removed) by Inmos in the future, so future revisions of the transputer may not support this instruction. Soft reset will work on both the T414 and T800 transputers.

Starting Multiple Processes

One of the unusual features of the transputer is its inherent ability to run multiple tasks. Most processors require additional software to divide up the CPU’s time among programs. Such a program is usually called a scheduler or scheduling program. The transputer has a built-in, microcoded scheduler. This allows a programmer to write assembly language programs with multiple threads of execution, without requiring any additional operating system support such as the fork or exec system calls in Unix.

Inmos has used the term "process" to define a unit of execution. In conventional computer terminology, a process can consist of many resources (e.g., code, data, memory resources). On the transputer, a process consists chiefly of a workspace pointer and an instruction pointer. A process is run in one of two priority modes (aptly named low and high priority). The thing to remember about transputer processes is that they’re very "lightweight," as opposed to "heavyweight" processes, which own lots of resources. There aren’t many resources associated with a transputer process. In fact, the term "process" on the transputer is really closer to the general computer industry usage of "thread."

The upshot of this is that transputer processes are very lightweight, that is, it doesn’t take the transputer very long to switch between them (on the order of one microsecond). This switching between processes (to service them) is called a "context switch." On other processors (or systems) with heavyweight processes, context switching is usually very slow, since information about the state of the process needs to be saved with it. The more resources a process has, the more information that needs to be saved during a context switch.

Using multiple processes (or threads of execution) is important to parallel programming. If a program can be broken up into multiple processes, it may be possible to run each process on a different transputer, thus speeding up the entire program since the program would then execute in parallel. The built-in communication features of the transputer allow a programmer to send messages to other transputers. The message can consist of a program or subprogram to run. The result can then be communicated back.

The following example program illustrates the use of multiple processes. There are three threads of execution (three processes). First, let’s look at the following C program fragment and see how to translate it into transputer assembler code using multiple processes:

```
int x, y
{
x = 2;
y = 3;
}
```

Now suppose we want the assignments to x and y to take place as seperate processes:

```
ldc PROC_X - STARTP1 ; load distance between startp and PROC_X ; process
ldlp -6 ; load address of PROC_X Wptr = Wptr[-6]
STARTP1
startp ; start the process located at PROC_X
j CONT1 ; continue the current thread of execution at ; CONT1
PROC_X
ldc 2 ; load 2 into A
stl x ; x = 2
ldlp -5 ; load address of new process Iptr
endp ; end this process, synchronize processes via ; endp
CONT1
ldc PROC_Y - STARTP2 ; load distance between startp and PROC_Y ; process
ldlp -12 ; load address of PROC_Y Wptr = Wptr[-12]
STARTP2
startp ; start the process located at PROC_Y
j CONT2 ; continue the current thread of execution at ; CONT2
PROC_Y
ldc 3 ; load 3 into A
stl y ; y = 3
ldlp -11 ; load address of new process Iptr
```

```
endp ; end this process, synchronize processes via ; endp
CONT2
ldc END - HERE ; load offset to END
ldpi ; load address of HERE + offset to END which ; equals the address of END
HERE
stl -1 ; store address into Wptr[-1] (will be future ; Iptr)
ldc 3 ; load the number of processes to synchronize ; (three processes in this case)
stl 0 ; save into Wptr[0]
ldlp -1 ; load address of new process Iptr
endp ; end this process, start new process when all ; three processes end
END
```

General observations about the above code follow. There are three threads of execution, or processes. The first is the one on entry to this code fragment. This process loads the offset from the STARTP instruction to the address of the process to start and then loads a workspace pointer for the new process located at six words beneath the current workspace. (Remember it is good practice to reserve five words beneath the current workspace for internal machine use.) At this point the new process is added to the run queue and started. The first process continues on to CONT1 via jump instruction and then performs essentially the same task on a third process (starting at PROC_Y).

After starting that process, the first process continues on to CONT2, loads the address of END, stores it into Wptr[-1] as the instruction pointer of a process to be started after all three running processes end. Next, the number of processes, three, is pushed into Wptr[0] and the address of the workspace minus one is loaded into A. Then an ENDP is executed, to indicate that there are three processes waiting to execute an ENDP. The count is decremented at each ENDP (including the one that initialized the count to three). When the count reaches one and another ENDP is executed, all three processes have executed ENDP's and a new process is started. The A register contains the address of the instruction pointer for this new process. The workspace for this new process is located at the word in memory following the instruction pointer (i.e., the Iptr for the process is located at its Wptr[-1]). In this example, the new instruction pointer will be located at END, so the new process will start with whatever instruction begins there.

It is also important to note that the processes created via STARTP have the same priority of the process that executed the STARTP. Also note the usage of LDLP just prior to STARTP. By LDLPing negative multiples of six, you ensure leaving five words (as recommended) between each workspace for each of the three processes.

Summarizing this example, three threads of execution are used to execute two statements of a high-level language in parallel. One process sets up the synchronization information by loading the number of processes and executing ENDP and the other two processes actually perform the actions specified by the C assignment statements. The ENDP instruction provides for a synchronization point from which multiple processes can terminate collectively before program execution continues.

Changing the Priority of a Process

Sometimes it is desirable to change the priority of a process from low to high or visa versa. For instance, if a process is entering a critical area say, communicating with hardware that requires a response in a certain small amount of time; it would be nice to be able to raise the priority of that process from low to high during that time-critical area of the program.

It is possible to change priority by toggling the low bit of a workspace descriptor and then running the current process as a new process and stopping the old, previous process. Recall that a workspace descriptor is simply the workspace pointer for a process with the priority encoded into the low bit: zero for low priority and one for high priority.

The code to switch a process from high priority to low priority is:

```
ldlp 0 ; load A with the workspace pointer
adc 1 ; set the low bit to one to indicate low priority (A = Wdesc)
runp ; add the current process to the low priority run queue
stopp ; stop the current high priority process
```

The code to switch a process from low priority to high priority is:

```
ldlp 0 ; load A with the workspace pointer (will be word-aligned)
runp ; add the current process to the high priority run queue
stopp ; stop the current low priority process
```

Note that in the code to switch from low priority to high priority that the LDLP instruction returns a word-aligned value for the workspace pointer, that is, the low order bit will be zero already. There will be no need to explicitly clear it (unlike the high priority to low priority example, where the low order bit has to be explicitly set).

In fact, the code to switch from high priority to low priority (using an ADC 1) can also be used to switch from low priority to high priority since it also toggles the low bit, although it performs a carry to the bit next to the low bit. Still, Inmos has reserved the lower two bits for priority. On the T414 and T800, only the low bit is used to encode priority (since there are only two priority levels). In future revisions of the transputer, there may be more than two levels of priority, so the second lowest bit may be used. If so, one cannot use the code with the ADC to switch a process from low to high, as it could change the priority altogether.

Transputer Timer Bug

If you are booting from ROM (and not a link), it means there is a hardware bug in the microcode of T414 and T800 transputers that requires a software patch. This bug may be fixed by Inmos in future revisions of the transputer, but since the patch code below won’t harm anything, it’s probably best to execute it as part of the boot sequence to ensure correct operation of the transputer. Otherwise, your transputer-based machine may appear to randomly "lock-up" or freeze, even after hours of use.

The bug relates to the timer interrupt mechanism inside the transputer. When a timer interrupt occurs the scheduler doesn’t check if the timer queue is empty (i.e., the head pointer to the queue contains a null pointer). Instead, it unconditionally adds the first process in the timer queue to the run queue and then uses the next process link field (Link.s) of the newly added process to find the next process to start executing. So, if the timer queue is empty, a process built from random memory locations is effectively started when the timer interrupt occurs. The timer interrupt is enabled when the transputer is booted and the alarm time (when the next timer interrupt will occur) contains a random value.

So, at some random time, a timer interrupt will occur and start a random (most always catastrophic) process that will lock-up the transputer. This is because the timer scheduling code does not check for a null pointer in the timer queue head pointer register. However, if there is a real process in the first entry in the timer queue with a null next process pointer (i.e., Tlink.s = 0x80000000), the timer interrupt (for that priority) will be disabled after scheduling that process. Then, when future processes are added to the timer queue, the transputer will behave correctly (it always checks the first process’ link field (Tlink.s), it just does not check the pointer to the first process itself).

The fix is to build a dummy process with a null next process pointer (i.e., Tlink.s = 0x80000000), place it in the timer queue, and cause a timer interrupt. Since the next process link field is null, the timer interrupt will be disabled (until a new process is added to the timer queue). The dummy process will be scheduled, but it can consist of a simple STOPP instruction and will be immediately (and correctly) removed from the run queue.

The code for the patch is as follows:

```
; Construct a dummy workspace at a safe place in memory. Six words
; above the current workspace is safe, so set the dummy workspace there.

DUMMY_WPTR = 6
DUMMY_IPTR = DUMMY_WPTR - 1 ; Iptr
DUMMY_TLINK = DUMMY_WPTR - 4 ; Tlink.s
DUMMY_TIME = DUMMY_WPTR - 5 ; Time.s

; If this is part of a general boot routine, then reset the hardware and
; event channels first (here) and initialize the process queue pointers to
; NotProcess.p. Important: The following code must be run in high priority!

ldc DUMMY - HERE ; construct the Iptr for the dummy process
ldpi ; load the offset to the dummy process
```

```
HERE
stl DUMMY_IPTR ; place the Iptr into the workspace
mint ; load NotProcess.p (0x80000000)
stl DUMMY_TLINK ; store into timer link field of workspace
ldc 4 ; load the time for the process to awaken
stl DUMMY_TIME ; place into dummy workspace

ldlp DUMMY_WPTR ; load dummy Wptr into hi pri timer queue
mint ; load start address of on-chip memory
stnl TptrLocOffset0 ; place into queue (TptrLocOffset0 = 0x24)

ldlp DUMMY_WPTR ; load dummy Wptr into low pri timer queue
mint ; load start address of on-chip memory
stnl TptrLocOffset1 ; place into queue (TptrLocOffset1 = 0x28)

; now add self to timer queue behind dummy process

ldc 0 ; load zero for resetting the timer
sttimer ; reset the timer to zero
ldc 2 ; load a time in the future (2 is lowest)
tin ; add self to timer queue in 1st position
ldc 5 ; dummy should die at timer = 4
tin ; make sure the dummy process has died

ldlp 0 ; load current workspace pointer
adc 1 ; convert to a low priority Wdesc
runp ; drop down to low priority
stopp ; stop high priority process

; repeat the same procedure for the low priority timer

ldc 0 ; load zero for resetting the timer
sttimer ; reset the timer to zero
ldc 2 ; load a time in the future (2 is lowest)
tin ; add self to timer queue in 1st position
ldc 5 ; dummy should die at timer = 4
tin ; make sure the dummy process has died

 j CONTINUE ; continue on as a low priority process
DUMMY
stopp ; where the high priority process will die
stopp ; where the low priority process will die
CONTINUE
```

A description of the above code follows. First, a workspace for the dummy timer process is constructed at six words above the current workspace. Six was chosen because workspace locations -1 to -5 are used by the transputer for process scheduling information. Next, the dummy process Iptr is set to the dummy process, the Tlink.s field (which points to the next timer process in the timer queue) is set to NotProcess.p (meaning no next process in queue), and, finally, the time the process is to awaken is at is set to a timer value of four. Four was chosen, as it is far enough in the future to allow another process to be added to the queue in front of it.

Next, the timer is reset to zero and the current process is added to the timer queue. The currently executing process will be the first process in the queue, since the microcode that adds processes to the timer queue orders the processes according to the time they are supposed to awaken at. Since the time the current process is to awaken equals two, it will be placed in front of the dummy process, which is set to awaken at timer equals four. This will cause the alarm time to be set to two (the time of the first process in the queue).

Note: It is important that the time the process is to awaken at be in the future; two was the safest, smallest value that can be chosen. If the time was not in the future, the alarm time would not be reset and would still contain a random (uninitialized) value.

With the alarm now reset, the current process will reawaken at timer equals two and continue. It then immediately adds itself to the timer queue again, waiting for the dummy process to awaken and die. The dummy process will awaken at timer equals four. Since its instruction pointer points to a STOPP instruction, it will immediately die and be removed from the process run queue.

The reason there are two STOPP's in a row in the dummy process code is that, after the first dummy process is executed (the high priority one), the instruction pointer is incremented and placed into DummyWptr[-1]. So when the second dummy process is executed (the low priority one), the instruction pointer is retrieved from DummyWptr[-1] and points to the instruction after the first STOPP, which by design is another STOPP. This is a efficient (and cute) trick.

Subroutines

The transputer supports conventional subroutine calls. When CALL is executed, the transputer pushes the return address, then A, B, and C on the stack, and then increments the workspace pointer by four words. The stack frame a subroutine will see is then:

Address	Contents
Wptr[3]	C register
Wptr[2]	B register
Wptr[1]	A register
Wptr[0]	Return Address (Iptr)

One example of using the stack frame in a subroutine is to implement a subroutine function to find the maximum of two integers. This is equivalent to the MAX(X, Y) function in some languages, which returns the larger of the two numbers.

```
MAX_INT
ldl 1 ; load the caller's A register
ldl 2 ; load the caller's B register
gt ; signed comparison caller's A > caller's B
cj B_IS_MAX ; if gt is false, jump to return caller's B
ldl 1 ; load return value, caller's A = MAX(A, B)
ret ; return to caller
B_IS_MAX
ldl 2 ; load return value, caller's B = MAX(A, B)
ret ; return to caller
```

Note: Wptr[1] = caller's A register and Wptr[2] = caller's B register. So to use MAX_INT, you first load A and B with the values to compare (for a maximum) and then make the call. An example of using this routine is:

```
ldc Value1 ; load first value to compare
ldc Value2 ; load second value to compare
call MAX_INT ; find the maximum of Value1 and Value2
eqc Value1 ; compare maximum (in A) versus Value1
cj VAL2MAX ; jump if not Value1, i.e. Value2 is maximum
; otherwise if here, then Value1 is maximum
. . .
. . .
VAL2MAX
. . . ; code if Value2 is maximum
```

A similar subroutine can be used to return the minimum of two integers, that is, implement the MIN(X, Y) function, which returns the minimum of X and Y:

```
MIN_INT
ldl 1 ; load the caller's A register
ldl 2 ; load the caller's B register
gt ; signed comparison caller's A > caller's B
```

```
cj A_IS_MIN ; if gt is false, jump to return caller's A
ldl 2 ; load return value, caller's B = MIN(A, B)
ret ; return to caller
A_IS_MIN
ldl 1 ; load return value, caller's A = MIN(A, B)
ret ; return to caller
```

Another illustration of how this stack frame can be used is for an implementation of an unsigned greater than comparison subroutine. The transputer greater than comparison instruction GT performs a signed comparison. Let's implement a subroutine to perform an unsigned greater than comparison.

First consider the following C code:

```
main()
{
unsigned int i, j;

i = 2; j = 4;
if (i > j) this();
else that();
}
```

In the transputer, this can be coded as:

```
MAIN
ajw -2 ; allocate two words of workspace for i and j
ldc 2
stl 0 ; i = 2
ldc 4
stl 1 ; j = 4
ldl 0 ; i
mint
xor
ldl 1 ; j
mint
xor
gt
cj J_IS_GREATER
call THIS
j SKIP
J_IS_GREATER
call THAT
SKIP
ajw 2 ; deallocate the two previously allocated words
ret
```

The above code illustrates an important point. In order to perform an unsigned comparison using a signed comparison instruction, you first have to toggle the high bit of the quantities you are comparing (a good way to do this is with a MINT followed by an XOR). This has the net effect of remapping the negative numbers to positive numbers, as well as remapping positive numbers to negative numbers. In the remapping scheme, the larger the positive number, the less negative the resultant number, and the less negative the number (i.e., closer to zero), the larger the positive resultant number. Thus, signed numbers are remapped to unsigned numbers. Sample values for the mapping scheme follow:

Number	Value	New Number	New Value
0x7FFFFFFF	MaxINT	0xFFFFFFFF	-1
0x7FFFFFFE	MaxINT - 1	0xFFFFFFFE	-2
0x7FFFFFFD	MaxINT - 2	0xFFFFFFFD	-3

. . . .
. . . .

```
. . . . . . . . . . . .
0x00000002 +2 0x80000002 MinINT + 2
0x00000001 +1 0x80000001 MinINT + 1
0x00000000 0 0x80000000 MinINT
0xFFFFFFFF -1 0x7FFFFFFF MaxINT
0xFFFFFFFE -2 0x7FFFFFFE MaxINT - 1
. . . . . . . . . . . .
. . . . . . . . . . . .
. . . . . . . . . . . .
0x80000002 MinINT + 2 0x00000002 2
0x80000001 MinINT + 1 0x00000001 1
0x80000000 MinINT 0x00000000 0
```

For example, in a signed comparison, -1 < 1. However, in an unsigned comparision, -1 maps to MaxINT and 1 maps to (MinINT + 1). Since MaxINT > (MinINT + 1), then -1 > 1 in an unsigned comparison. This makes sense, since 0xFFFFFFFF > 1.

Therefore the general strategy for our subroutine will be to toggle the sign bit on the two values to compare and then perform a signed greater than comparison. This is equivalent to performing an unsigned greater than comparison.

The code is as follows:

```
; This code will return the result of an unsigned greater than comparison
; test for A > B (caller's perspective).

UN_GT
ldl 2 ; load Wptr[2] = caller's B register value
mint ; load 0x80000000 (word with only sign bit set)
xor ; toggle the sign bit
ldl 1 ; load Wptr[1] = caller's A register value
mint ; load 0x80000000 (word with only sign bit set)
xor ; toggle the sign bit
gt ; perform a signed greater than (B > A)
ret ; return to the caller with the result in A
```

The above code returns zero if B is less than or equal to A, otherwise a nonzero value is returned. In other words if A is strictly greater than B (in an unsigned comparison), a nonzero value is returned, otherwise zero is returned. The A register contains the return value.

Then we could substitute the call in our original program as follows:

```
MAIN
ajw -2 ; allocate two words of workspace for i and j
ldc 2
stl 0 ; i = 2
ldc 4
stl 1 ; j = 4
call UN_GT ; compare i > j unsigned, result is left in A
cj J_IS_GREATER
call THIS
j SKIP
J_IS_GREATER
call THAT
SKIP
ajw 2 ; deallocate the two previously allocated words
ret
```

The Static Link

It is possible to write programs that depend on being loaded into fixed locations in memory. It is also possible to write programs that can be loaded into any available block in memory. The latter type of program is said to be "relocatable," meaning that it can be relocated anywhere in memory and will still run. This requires a certain coding convention or style on the transputer that keeps track of where global data is in memory.

Most issues involving relocatable code center around global data. Take the following C program, for example:

```
main()
{
int i;

i = 5;
}
```

A C compiler might generate code for the above program as follows:

```
main
ajw -1 ; allocate one word for i
ldc 5
stl 0 ; i = 5
ajw 1 ; deallocate the space for i
ret
```

However, consider the case where i is a global variable, namely:

```
int i;

main()
{
i = 5;
}
```

In the above program, i is no longer allocated from the local workspace; instead, i refers to some address, usually in a global data memory block. In this instance, a C compiler might generate the following code:

```
i WORD 1

main
ldc 5
ldl 1 ; load global data pointer
stnl i ; i = 5
ret
```

An assembler psuedo-operation, or psuedo-op, would be generated to reserve a word for i in the global data memory area. Then, there is an implicit assumption about the state of the workspace, namely that Wptr[1] contains a pointer to the start of the global data area in memory. This is refered to as the "static link," since it links an address to static, or global, data. It is also implicitly understood that "i" does not refer to an actual address, but rather is a word offset from a base address. The base address is the starting address of the global data area. Thus, references to global data are made via a base address (the static link) and a word offset (represented in this case by the variable i). This is very similar to accessing a workspace element via an offset from the workspace pointer base address.

It is assumed that the loader will initially place the address of the global data memory block into a place where the program can find it. A usual place is to put it on the first available place in the workspace. In general, there are three situations where the static link is used:

Code Description

ldl static_link ; ldnl X load global variable X

```
ldl static_link ; ldnlp X load the address of global variable X
ldl static_link ; stnl X store into global variable X
```

Now consider subroutines:

```
main()
{
int i;

sub(&i); /* pass the address of i to subroutine sub */
}
```

A C compiler might generate the following code for the above program:

```
main
ajw -1 ; allocate a word for i
ldlp 0 ; load a pointer to i
call sub ; call subroutine sub
ajw 1 ; deallocate the space for i
ret
```

However, if i is in the global data area, as in the below program:

```
int i;

main()
{
sub(&i); /* pass the address of i to subroutine sub */
}
```

Then the code generated could be:

```
main
ldl 1 ; Wptr[1] contains the global data pointer (static link)
ldnlp i ; load the address of i (&i)
ldl 1 ; load Wptr[1] into A
call sub ; call the subroutine
ret
```

It is assumed that the loader placed the address of where it located the global data in memory into Wptr[1] prior to the program's starting to execute. Note the differences primarily, loading the static link and then loading a nonlocal pointer to i. Recall that, in this case, i represents an offset from the base address of global data which is contained in the static link. Also notice that, prior to calling a subroutine, the compiler may load the global data address into A, so it can be passed to the subroutine (it does this in the above instance).

The upshot of the static link is that, if you are going to write relocatable code, you have to understand how to gain access to the base address of the global data memory area (static link). In addition, if you are going to write assembler code to interface to high-level languages (such as in a library subroutine), it is likely that such languages are producing relocatable code and will require global data access to be performed via a static link.

Timer

The transputer contains two internal clocks, one for each priority level. The high priority clock ticks once every microsecond, while the low priority clock ticks once every 64 microseconds. It is possible for a process to effectively pause (or sleep) by adding itself to the timer queue and waiting for a designated time. Here is a code fragment that will cause a low priority process to pause for one second:

```
ldtimer ; load the current timer value
```

```
ldc 15625 ; 15625 * 64 microseconds = one second
sum ; compute the time to wake up at
tin ; add self to timer queue
```

Communication

There is assembler-level support for interprocess communication on and between transputers. Messages can arrive in a message buffer, be copied to another area of memory, and be sent with each such operation only taking one instruction.

One familar subroutine call to C programmers is bcopy, sometimes called memcpy. bcopy stands for "block copy" and is used to copy a block of memory from one address to another; similarly, memcpy stands for "memory copy" and is used for the same purpose. The only difference between bcopy and memcpy is that their first two arguments are usually reversed, namely:

memcpy(A, B, C) = bcopy(B, A, C)

Where A is the destination for the block to copy, B is the address of the source block to copy from and C is the number of bytes to copy.

The transputer implements this function in one instruction with one important restriction the source and destination memory blocks cannot overlap. If they do overlap, the results are undefined. A sample use of such a bcopy for non-overlapping blocks follows:

```
BCOPY
ldl source ; load address of source block
ldl dest ; load address of destination block
ldl numbytes ; load number of bytes to copy
move ; copy the block
ret
```

Since the transputer allows multiple threads of execution (processes), it is possible to have one process waiting for a message from another. An example of waiting for a message would be:

```
ldl buffer ; load address of the buffer to receive the message
ldl channel ; load address of the channel to use to communicate
ldl messagelen ; load the message length to receive (in bytes)
in ; receive the message
```

Notice in the above example that the receiver must know the exact length (in bytes) of the message that it will receive. Missed synchronization in message length can result in deadlocked processes.

The sender of such a message would use the same channel address and do the following:

```
ldl buffer ; load address of buffer holding the message to send
ldl channel ; load address of the channel to use to communicate
ldl messagelen ; load the message length to receive (in bytes)
out ; send the message
```

Again, it is important that both receiver and sender use the same channel address and that both know exactly how many bytes to be sent in each message. Note that to use the links to send messages between two transputers you use exactly the same instruction sequence; however, there are special receserved locations in on-chip memory that must be used for the channels.

Alternation

The IN and OUT instructions provide a basic interprocess communication facility on the transputer. However, more complex instruction sequences are generally required for typical communication usage. The ALT structures in the transputer instruction set provide this. There are essentially three kinds of ALTs: a "regular" ALT, which listens to a set of channels and then jumps to a service routine when a message arrives; a "timer" ALT, which listens to a set of channels for a specified length of time and then exits if no message arrives in that time; and a "skip" ALT, which polls channels, sees if a message has arrived, and continues on even if none has.

In general, ALTs are used to listen to a number of channels and to activate a service routine for a channel when a message arrives. In this respect ALTs are similar to interrupt routines on other microprocessors, which awaken due to some event, check the event, and then execute a routine to handle the event.

Regular ALT constructs are used to listen to multiple channels at once and act accordingly when a message arrives on one of the channels. An example of a regular ALT construct is as follows:

```
; Listen to three channels at once using unconditional channel guards

alt ; flag the start of an alternation

ldc channel1 ; load the address of the 1st channel
ldc 1 ; load flag (always TRUE)
enbc ; enable the channel guard

ldc channel2 ; load the address of the 2nd channel
ldc 1 ; load flag (always TRUE)
enbc ; enable the channel guard

ldc channel3 ; load the address of the 3rd channel
ldc 1 ; load flag (always TRUE)
enbc ; enable the channel guard

altwt ; start listening to the three channels

ldc channel1 ; load the address of channel1
ldc 1 ; load flag (always TRUE)
ldc SERVICE1 - ENDALT ; load offset to service routine
disc ; disable the channel guard

ldc channel2 ; load the address of channel2
ldc 1 ; load flag (always TRUE)
ldc SERVICE2 - ENDALT ; load offset to service routine
disc ; disable the channel guard

ldc channel3 ; load the address of channel3
ldc 1 ; load flag (always TRUE)
ldc SERVICE3 - ENDALT ; load offset to service routine
disc ; disable the channel guard

altend ; end alternation, jump to service routine
ENDALT
SERVICE1
ldc buffer1 ; load the address of the message buffer
ldc channel1 ; load the address of the channel
ldc messlen1 ; load the message length
in ; input the message from the channel
j CONTINUE
SERVICE2:
ldc buffer2 ; load the address of the message buffer
ldc channel2 ; load the address of the channel
ldc messlen2 ; load the message length
in ; input the message from the channel
j CONTINUE
```

```
SERVICE3
ldc buffer3 ; load the address of the message buffer
ldc channel3 ; load the address of the channel
ldc messlen3 ; load the message length
in ; input the message from the channel
CONTINUE
```

Timer ALT constructs provide a means of listening to one or more channels for only a specified amount of time (i.e., without having to wait forever for a message, such as an IN will do). An example of a timer ALT construct to listen to two channels is as follows:

```
; Listen to two channels at once for a specified time

talt ; flag the start of an alternation

ldc channel1 ; load the address of the 1st channel
ldc 1 ; load flag (always TRUE)
enbc ; enable the channel guard

ldc channel2 ; load the address of the 2nd channel
ldc 1 ; load flag (always TRUE)
enbc ; enable the channel guard

ldc time ; load the time to stop listening at
ldc 1 ; load flag (always TRUE)
enbt ; enable the timer guard

taltwt ; start listening to the two channels

ldc channel1 ; load the address of channel1
ldc 1 ; load flag (always TRUE)
ldc SERVICE1 - ENDALT ; load offset to service routine
disc ; disable the channel guard

ldc channel2 ; load the address of channel2
ldc 1 ; load flag (always TRUE)
ldc SERVICE2 - ENDALT ; load offset to service routine
disc ; disable the channel guard

ldc time ; load the time-out time again
ldc 1 ; load flag (always TRUE)
ldc CONTINUE - ENDALT ; load offset past service routines
dist ; disable the timer guard

altend ; end alternation, jump to service routine
ENDALT
SERVICE1
ldc buffer1 ; load the address of the message buffer
ldc channel1 ; load the address of the channel
ldc messlen1 ; load the message length
in ; input the message from the channel
j CONTINUE
SERVICE2
ldc buffer2 ; load the address of the message buffer
ldc channel2 ; load the address of the channel
ldc messlen2 ; load the message length
in ; input the message from the channel
j CONTINUE
CONTINUE
```

Skip constructs in an ALT provide a way to poll channels to see if a message has arrived. After enabling a skip, the

ALTWT does not wait for channels; it simply checks which guard was ready first (a channel guard, or the skip guard). An example of a skip ALT construct is as follows:

```
; Check two channels to see if a message has arrived on one

alt ; flag the start of an alternation

ldc channel1 ; load the address of the 1st channel
ldc 1 ; load flag (always TRUE)
enbc ; enable the channel guard

ldc channel2 ; load the address of the 2nd channel
ldc 1 ; load flag (always TRUE)
enbc ; enable the channel guard

ldc 1 ; load flag (always TRUE)
enbs ; enable the skip, i.e., set State.s to Ready.p

altwt ; check the two channels

ldc channel1 ; load the address of channel1
ldc 1 ; load flag (always TRUE)
ldc SERVICE1 - ENDALT ; load offset to service routine
disc ; disable the channel guard

ldc channel2 ; load the address of channel2
ldc 1 ; load flag (always TRUE)
ldc SERVICE2 - ENDALT ; load offset to service routine
disc ; disable the channel guard

ldc 1 ; load flag (always TRUE)
ldc CONTINUE - ENDALT ; load offset to continue point
diss ; disable the channel guard

altend ; end alternation, jump to service routine
ENDALT
SERVICE1
ldc buffer1 ; load the address of the message buffer
ldc channel1 ; load the address of the channel
ldc messlen1 ; load the message length
in ; input the message from the channel
j CONTINUE
SERVICE2
ldc buffer2 ; load the address of the message buffer
ldc channel2 ; load the address of the channel
ldc messlen2 ; load the message length
in ; input the message from the channel
j CONTINUE
CONTINUE
```

Using the T800 Floating Point Unit

When using T800 floating point instructions, it is important to remember whether you are dealing with single (32-bit) or double (64-bit) quantities. There is no way to query the floating point stack to find the length of the values it contains. While some floating point instructions are specific to single or doubles, others perform their operation on whichever length number is in the register.

In general, programmers use two sets of library subroutines when dealing with floating point numbers. One set of subroutines act on single-length floating point numbers, and the other set acts on double-length floating point numbers.

Consider the minimum (MIN) and maximum (MAX) functions as described earlier. What are their floating point equivalents? Recall that a subroutine inherits a stack frame as follows:

Address Contents

Wptr[3] C register
Wptr[2] B register
Wptr[1] A register
Wptr[0] Return Address (Iptr)

Single-precision floating point maximum function:

```
FMAX
ldlp 1 ; load the address of Wptr[1] which contains A
fpldnlsn ; load caller’s A into the FPU stack
ldlp 2 ; load the address of Wptr[1] which contains B
fpldnlsn ; load caller’s B into the FPU stack
fpgt ; perform a signed greater than function
cj BGREATER ; if caller’s B > caller’s A, jump
ldl 1 ; caller’s A was greater, return it
ret
BGREATER
ldl 2 ; caller’s B was greater, return it
ret
```

Single-precision floating point minimum function:

```
FMIN
ldlp 1 ; load the address of Wptr[1] which contains A
fpldnlsn ; load caller’s A into the FPU stack
ldlp 2 ; load the address of Wptr[1] which contains B
fpldnlsn ; load caller’s B into the FPU stack
fpgt ; perform a signed greater than function
cj A_IS_LESS ; if caller’s B > caller’s A, jump
ldl 2 ; caller’s B was less, return it
ret
A_IS_LESS
ldl 1 ; caller’s A was less, return it
ret
```

The size of the transputer’s stack poses a problem for minimum and maximum functions for double-length floating point numbers. In the previous examples it was possible to pass both numbers to compare in the stack. Since a double-length number requires two words of storage and the transputer’s stack is only three words deep, it is impossible to load both doubles to compare on the stack. Thus, let’s assume that only one double is loaded on the stack and a pointer to the other is loaded in the remaining word. So the stack frame (to the subroutine) looks like:

Address Contents Values

Wptr[3] C register High word of argument 1
Wptr[2] B register Low word of argument 1
Wptr[1] A register Pointer to argument 2
Wptr[0] Return Address Iptr

Double-precision floating point maximum function:

```
DMAX
ldl 1 ; load the address of Wptr[1] which points to arg2
fpldnldb ; load arg2 into the FPU stack
ldlp 2 ; load the address of Wptr[2] where arg1 starts
```

```
fpldnldb ; load arg2 into the FPU stack
fpgt ; perform a signed greater than function
cj ARG1MORE ; if arg1 > arg2, jump
ldl 1 ; load pointer to arg2
ldnl 1 ; load arg2 high word
ldl 1 ; load pointer to arg2
ldnl 0 ; load arg2 low word
ret
ARG1MORE
ldl 3 ; caller's arg1 was greater, return it
ldl 2 ; high word in B, low word in A
ret
```

Double-precision floating point minimum function:

```
DMIN
ldlp 2 ; load the address of Wptr[2] where arg1 starts
fpldnldb ; load arg2 into the FPU stack
ldl 1 ; load the address of Wptr[1] which points to arg2
fpldnldb ; load arg2 into the FPU stack
fpgt ; perform a signed greater than function
cj ARG2LESS ; if arg1 > arg2, jump
ldl 3 ; caller's arg1 was less, return it
ldl 2 ; high word in B, low word in A
ret
ARG2LESS
ldl 1 ; load pointer to arg2
ldnl 1 ; load arg2 high word
ldl 1 ; load pointer to arg2
ldnl 0 ; load arg2 low word
ret
```

Note that, in the above double precision subroutines, two words are returned representing a double-length floating point number with its high order word in B and its low order word in A. Also note that FPORDERED was not used prior to the FPGT, so these subroutines are not valid if passed a NaN or infinity.

Another common function used in dealing with floating point numbers is the floor function. This function truncates positive floating point numbers and rounds negative floating point numbers (towards minus infinity) to the nearest integer. So the floor of 3.54 is 3.0 and the floor of -3.54 is -4.0. A C language program for the floor function looks like:

```
double floor( x )
double x;
{
int n;

n = (int)x;
if (x < 0) --n;
return (double)n;
}
```

For calling convention sake, let's assume the double parameter to floor is passed in B (low word) and C (high word) again. A possible way of coding this in T800 transputer code is:

```
FLOOR
ajw -2 ; allocate two words of workspace to use
ldlp 4 ; load pointer to low word of x (caller's B register)
fpldnldb ; load x (value to compute floor of)
fpurz ; set rounding mode to zero
fpint ; round x to a floating point integer
ldlp 0 ; use Wptr[0] as a temporary variable, load its address
```

```
fpstnli32 ; save x as an integer value in Wptr[0]
ldl 0 ; load integer value for x into A register
stl 1 ; n = (int)x (or Wptr[1] = n)
fpldzerodb ; load (double)0.0 onto FPU stack
ldlp 4 ; load pointer to low word of x (caller’s B register)
fpldnldb ; load x (for if (x < 0) test)
fpgt ; compute 0.0 > x
cj X_POSITIVE ; if (0.0 > x) is false, then jump to X_POSITIVE
ldl 1 ; load n (which has the integer value of floor(x))
adc -1 ; decrement n by one, since x is negative
stl 1 ; store n
X_POSITIVE
ldl 1 ; load n (which has the integer value of floor(x))
stl 0 ; save n in memory at Wptr[0]
ldlp 0 ; load address of Wptr[0]
fpi32tor64 ; load (double)n onto FPU stack
ajw 2 ; deallocate the two words of workspace used
ret ; return the result of the floor(x) in FA
```

In the above implementation of floor, a double-length floating point number is assumed to be in the caller’s B and C registers prior to the call to the FLOOR subroutine. This is pushed onto the stack at Wptr[2] and Wptr[3] by CALL. Next, two words of workspace are needed locally for the FLOOR function. Note that the AJW adjusts the workspace pointer so now x is at Wptr[4] and Wptr[5]. Then the value to compute is loaded, the rounding mode is set to zero (remember that round to nearest is the default mode), and x is rounded to a floating point integer (ergo 4.56 would be rounded to 4.0, which is still a floating point number). Next, the floating point integer is stored as an actual integer into a temporary variable (Wptr[0]), which is then loaded into n. Then zero is loaded onto the floating point stack and the comparison is made for the if statement. If x is negative, the integer value of x must be decremented (in keeping with the semantics of the floor function). In any case, use Wptr[0] again as an intermediate temporary variable to load the floating point result of floor(x) into the floating point unit’s FA register. Then the allocated workspace is deallocated and the result is returned in FA.

Optimizing T800 Floating Point Instructions

The integer and floating point engines on the transputer can operate (to a certain degree) independently. Therefore, it is desireable to write programs in such a manner that both the floating point engine and the integer engine remain "busy." If both units are not busy, then it is usually the case that one is waiting for the other, and then usually the case that the integer engine is waiting for floating point calculations to finish (when instead of waiting, it could be calculating an address or doing something else of value). Taking advantage of this "overlap" in integer and floating point computation primarily affects the way in which a programmer may code a certain calculation. Calculations should be ordered in such a manner as to maximize both integer and floating point unit performance.

To understand how to order calculations in this manner, it is necessary to know when the floating point and integer units synchronize. Essentially, they synchronize on operations involving data transfer between them. For instance, any time the floating point unit loads a value from memory, the address to load from is in the integer unit, making it a point of synchronization. Instructions like FPLDNLDB, FPLDNLSN, and so on is another example.

However, this means that for many instructions, like FPADD, FPDIV, FPMUL, and others, the integer and floating point units can operate independently, since there is no exchange of data. Thus it is desireable to continue performing integer operations like address calculations while these floating point instructions are being executed. Good candidates to check for possible reordering are FPMUL and FPDIV instructions, since they are common and consume more FPU time than most others.

For example, it is possible to code the following expression:

A = B * C + D

As:

```
generate address of B
```

```
FPLDNLDB
generate address of C
FPLDNLDB
FPMUL
generate address of D
FPLDNLDB
FPADD
generate address of A
FPSTNLDB
```

However, a more efficient way to order this would be:

<FONT

Chapter 5 Newer Transputers

Introduction

The preceding chapters apply to the T414 and T800 transputer chips, however Inmos did not stop there. Other versions of transputers have been released that are upwardly compatible with the T414 and T800. These newer transputers have additional features and they are discussed here.

Complaints about Transputers

No one would call the transputer chip architecture "conventional." One of the things that strikes a transputer programmer who has used other modern microprocessors is how many different ideas have been applied to the chip. Most modern microprocessors are register-based, the transputer is stack-oriented. No other modern chip has assembler level instructions to start multiple threads of execution in a program. And while other chips have built-in chip-to-chip communication ports, it is still quite a rare feature.

So at first the transputer chip seems quite exotic to most programmers. However when you do dive in and start to program the beast you find the transputer behaves much the same as other microprocessor chips and you have the same problems you do with those other chips. Problems such as "How do I debug my program?" or "Can I prevent my program from accessing memory outside of the limits I assign to it?"

The T414 and T800 do not have a convenient way to suspend the execution of a program, examine the registers, stack and memory, then resume program execution. This is very helpful in debugging a program, as you can stop it, look at what its done, then resume it. Most microprocessors have an instruction called a "breakpoint" instruction just for this purpose. When a program hits a "breakpoint", some special things happen which save the program state, then switch to another program that allows you to examine that state.

In response to the plea from programmers for a breakpoint instruction (to aid in debugging programs), Inmos brought out the T425 and T805. These processors are essentially enhanced versions of the T414 and T800 with some additional instructions, including a breakpoint instruction. Information on the T425 and T805 is provided below.

There was also the issue of memory protection. On modern microprocessors like the 80386 and 68030, it is possible to place the processor into "protected mode." This means that if a program attempts to access a memory location outside the limits assigned to it, an interrupt or exception occurs where a supervisor process can track down and stop the guilty program. Again, this is especially useful in debugging programs. It is not uncommon for a program under development to attempt to access illegal memory. If you can’t protect memory outside of that assigned to the program, then the memory of other programs, including the operating system, could be corrupted.

In response to this need, Inmos has announced the T9000 (not yet available at the time of this writing). The T9000 has a memory protection scheme that prevents a program from accessing memory.outside of its assigned areas.

In addition, the T9000 has other important enhancements and features, one of which is the support for short (16-bit) data types. On T400 and T800 series transputers (prior to the T9000), there are no single instructions for loading or storing 16-bit (two byte) data types. Many computer programs have been written using short data types, so on the T400 and T800 transputers you have implement the load and stores by loading a byte, shifting it left 8 bits, then loading a second byte, etc... Since this is a sequence of several instructions it imposes a severe performance degredation on programs that use short data types.

In the T9000, Inmos has added single instruction loads and stores for short data types (16-bits). This should speed up execution of programs that use short data types immensely. The T9000 is also discussed below.

It is worth restating that the T425, T805 and T9000 are all upwardly compatible from the T414 and T800. That means that a program written for the T414 will run on the T425 unmodified and a program written for the T800 will run on the T805 and T9000 unmodified. The same binary image will execute in the same fashion on the new transputers.

T425 and T805

Two new instructions added to the T425 and T805 are worth looking at to start. The first is the rotate (ROT) instruction. This instruction rotates the register stack so that C receives the old value of A, B receives the old value of C and A receives the old value of B. It effectively pops the stack but places the A register value into C.

The other new instruction to examine is LDMEMSTARTVAL. This instruction loads the starting address of the user section of on-chip memory, called MemStart, into the A register. On-chip memory below MemStart is reserved for interal use. On the T425 and T805 which have 4 Kbytes of on-chip memory, MemStart is 0x80000070 (while on-chip memory starts at 0x80000000) and that is the value that would be loaded into A if LDMEMSTARTVAL is executed.

The main new feature on these chips though, is the support for debugging programs via a breakpoint instruction. One of the things you'd like to be able to do with a breakpoint instruction is insert it anywhere into your code, perhaps over the tops of an existing instruction. This way a debugger could overwrite one of your program's instructions in memory with the breakpoint, then restart your program at the point it was stopped by replacing the old instruction back over the breakpoint.

In order to perform this substitution it is desirable to have the breakpoint instruction the same size as the smallest instruction. In the case of the transputer this is one byte as there are many one byte instructions (all the direct instructions). However all the one byte instructions are already spoken for (again, all the direct instructions). Inmos chose a clever solution to this dilemma by selecting the "J 0" (jump zero) opcode as the one byte breakpoint instruction. On other transputers, this instruction is a NOP or "No Operation" since it has a null effect on the machine, essentially telling the transputer to jump to the next instruction (which it would do anyway).

However on the T425 and T805 if a special debugging flag is set, the J0_BREAK_ENABLE flag, then a 'J 0' instruction causes a context switch with another process. The other process that is started can be selected by the programmer. Here's how it works. After the 'J 0' breakpoint, the workspace pointer and instruction pointer are exchanged with values located just above MemStart. Those values point to another program, presumably a debugger-type of program. When the debugger program is finished, it executes another 'J 0' to switch the Wptr and Iptr back to the original program's values.

The values for the debugger's Wptr and Iptr are stored just above MemStart. There are two different contexts present there, one for each priority as shown in figure 5-1.

Byte Address	Purpose
8000007C	Instruction Pointer (Low Priority)
80000078	Workspace Pointer (Low Priority)
80000074	Instruction Pointer (High Priority)
80000070	Workspace Pointer (High Priority)

Figure 5-1 BreakPoint Process Save Area

To cause exection of 'J 0' to be treated as a context switch instead of a NOP, the J0_BREAK_ENABLE flag must be set. This enables the breakpointing. There are three instructions present to use regarding this flag: CLRJ0BREAK clears the jump zero break enable flag thus disabling breakpointing, SETJ0BREAK sets the jump zero break enable flag thus enabling breakpointing and TESTJ0BREAK tests if the jump zero break enable flag is set.

There is also a two byte break instruction called BREAK which has the same behavior as jump zero (when breakpointing is enabled) except that BREAK always breakpoints regardless of the state of the J0_BREAK_ENABLE flag.

T9000

The T9000 represents a significant jump in technology for transputers. Inmos wanted an order of magnitude level of performance increase and is saying a 50 MHz T9000 will have 10 times the performance of a 25 MHz T800 while still maintaining binary compatibility. Other numbers quoted were 200 MIPS (transputer MIPS, probably 40 to 50 conventional processor MIPS) and 25 MegaFLOPS. This compares with 15 MIPS and 1.5 MegaFLOPS for a 25 MHz T800. Note: A MegaFLOP is a million floating point operations per second and a MIP is a million machine instructions per second.

In addition to raw performance increases the T9000 adds many new instructions and features to the transputer. There is now 16 Kbytes of on-chip memory (instead of four) and it can be configured as a memory cache. There are instructions which implement semaphores that can be shared by different processes. Perhaps most significantly to programmers, there are now instructions that support short data types directly and of course, memory protection has been implemented.

Instead of having to load a byte, shift it left eight bits, then load another byte to load a short data type, on a T9000 there are single instructions to do the same operation for loading and storing short (16 bit) data types. The addition of these instructions will make programs simpler and faster when dealing with short data types. The new partial word support instructions are listed in figure 5-2.

Mnemonic	Name	Opcode
LS	load 16 bit word	2C FA
LSX	load 16 bit word extended	2F F9
SS	store 16 bit word	2C F8
XSWORD	sign extend 16 bit word	2F F8
CS	check 16 bit word signed	2F FA
CSU	check 16 bit word unsigned	2F FB
SSUB	16 bit word subscript	2C F1
LBX	load byte extended	2B F9
XBWORD	extend byte to word	2B F8
CB	check byte signed	2B FA
CBU	check byte unsigned	2B FB

Figure 5-2 Part Word Support Operation Codes

The most important enhancement in the T9000 to many programmers is the addition of a memory protection scheme. There are two levels of trap-handling in this scheme: G-processes and L-processes. The difference is with respect to how error handling and debugging is handled. For G-processes, the error or debugging is handled exactly as on the T425 or T805 where a process descriptor stored above MemStart is called if a trap occurs (say due to a ’J 0’ instruction). While an L-process can have associated with it its own unique trap-handler. Part of the workspace of the L-process points to a trap handler data structure which contains information about the trap handler and where to find it.

The G (global) processes behave identically to processes on previous transputers. The L (local) processes trap handler can be triggered on floating point arithmetic errors, non-word aligned address memory accesses, attempts to execute illegal instructions, integer overflow or writes to a watchpointed region of memory. The L-process trap-handler data structure contains fields which describes the region of memory to "watch" for illegal writes to these areas of memory. When an error of this kind occurs (say an illegal write to memory), the trap handler is invoked by the transputer.

Instruction Set Reference

Certain specific terminology and symbology is used in the reference section that follows. The terms and symbols are defined in this introduction. The reference section itself is meant to be used by a transputer assembly language programmer as a means to gain detailed knowledge about the instructions he or she needs to use.

The layout of an individual instruction consists of its mnemonic name followed by its meaning at the top of the page. If the instruction is specific to the T414 transputer, "T414" follows; if the instruction is specific to the T800 transputer, then "T800" follows. Otherwise, the instruction is available on both the T414 and T800 transputers.

The next section is the operation code (abbreviated opcode) and the time the instruction will take to execute (clock cycles). The operation code is the representation of the instruction in memory and consists of one or more bytes (represented by hexadecimal numbers). For direct instructions, the lower four bits (nibble) of the instruction contain an encoded operand and can assume any value, so the lower nibble of the hexadecimal number representing the operation code is denoted by an "X" to indicate that the nibble can be any number. For example, ADC has an opcode of 8X, meaning that 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 8A, 8B, 8C, 8D, and 8F are all valid representations of ADC.

The execution speed of the instruction is measured in clock cycles. For most instructions this is a constant value, but for some instructions certain symbols are necessary to represent the time required for execution. These symbols are:

b	the bit number of the highest bit set in the A register (bit 0 is the lowest, i.e., least significant bit)
n	the number of bits of a shift (i.e., the amount to shift)
w	the number of words, plus non word-aligned part words in a message (for communication instructions)
p	the number of words per row (for move instructions)
r	the number of rows (for move instructions)

In some instances, typical times and maximum times for an instruction are given. These represent the usual time required to execute the instruction and the maximum time the instruction can require. For some floating point instructions, the execution time required using double-length numbers is different from single-length numbers and is so noted.

The flags section gives information on how the instruction affects the state of the available flags on the transputer. Flags have binary value and are either clear (meaning equal to zero) or set (meaning equal to one). The flags on the transputer are represented by mnemonic names:

Mnemonic Name	Meaning
Error	Error flag
FP_Error	Floating point unit Error flag
HaltOnError	Halt on Error flag

Note that the use of the word "Set" means assign the flag a value of one and the use of the word "Clear" means assign the flag a value of zero. An example of this would be "Set Error" or "Clear FP_Error". However, if a flag is being tested, then the phrase "is Set" or "is Clear" is used to check the value of the flag, without altering it. An example of this would be "Error is Clear" or "FP_Error is Set"; such statements are used for comparision and are either true or false.

The scheduling section refers to the transputer's on-chip multitasking scheduler. If an instruction is "atomic," it will execute without being preempted by any other instruction and program execution will continue normally. If the instruction is not atomic, then the transputer's scheduler may deschedule the currently executing program (process) and restart it at some undetermined time in the future. The usual implications of this are that the registers and flag bits are in an undetermined state when the program (process) resumes. Whether the instruction is "atomic" or can be descheduled is noted in the scheduling section.

A program language-like description of how the instruction works follows under the pseudocode section and then a more general English-like description follows under the description section. Finally, if there are any important points to remember about the individual instruction, they are noted under the user notes section.

The pseudocode section is intended to be an abbreviated form of how the instruction behaves and is intended to be read from the top down, like a computer program. As such, it uses symbols by convention. The symbols are defined here:

Registers

Name	Meaning
A	The A register in the integer unit
B	The B register in the integer unit
C	The C register in the integer unit
FA	The A register in the floating point unit
FB	The B register in the floating point unit
FC	The C register in the floating point unit
Oreg	The operand register
Iptr	The instruction pointer register
Wptr	The workspace pointer register
ClockReg0	The clock register for processes of priority 0 (high)
ClockReg1	The clock register for processes of priority 1 (low)
BPtrReg0	Back (tail) pointer to run queue register (high priority)
BPtrReg1	Back (tail) pointer to run queue register (low priority)
FptrReg0	Front (head) pointer to run queue register (high priority)
FptrReg1	Front (head) pointer to run queue register (low priority)
TptrLoc0	Timer queue head pointer (high priority)
TptrLoc1	Timer queue head pointer (low priority)
RoundMode	The current floating point arithmetic round mode

Recall that the A, B, and C registers form a three deep stack with A on the top and C on the bottom (so that values must be pushed onto and popped off of A). Similarly, FA, FB, and FC also form a three deep register stack. In some cases (such as the value of C after popping the integer register stack) the values that these registers contain are unknown. In these instances the values they hold are said to be undefined. For example: "C = Undefined" means that C could contain any value (i.e., the value of C is unknown).

The operand register, Oreg, is assigned values by the use of prefix instructions and operands encoded in the lower nibble of direct instructions. In pseudocode, however, the notation for assigning it a value is "Oreg = value" (ergo "Oreg = 14" or "Oreg = 0").

The instruction pointer, Iptr, can be assigned any address; however, the convention "Next Instruction" is used to mean the address immediately after the current instruction’s address, that is, the address one byte after the last byte of the current instruction.

The workspace pointer, Wptr, is always word-aligned. Note that this is different from a workspace descriptor, Wdesc, which has the priority of the process encoded into the workspace pointer’s lower bit.

Note that TptrLoc0 is located in on-chip memory at 0x80000024 and TptrLoc1 is also located in on-chip memory at 0x80000028. Each contains a head pointer to a queue of processes waiting on their respective priority’s timer to reach some value (time). When that value is reached, the suspended processes (waiting for that time) resume.

For floating point arithmetic, RoundMode is used to hold the current rounding mode. RoundMode has one of the following values:

Name	Meaning
ToNearest	round to the nearest number
ToZero	round towards zero
ToPlusInfinity	round towards plus infinity
ToMinusInfinity	round towards minus infinity

The IEEE floating point number representation that the transputer uses provides for quantities that are not numbers or are values representing positive or negative infinity. Floating point representations that are not numbers are referred to as NaNs (Not a Number). This symbol is used widely in the floating point instruction descriptions. There are many floating point representations that are NaNs; however, the use of the symbol "NaN" means any one of those representations. Similarly, there are only four representations for infinity (one each for single-length positive and negative infinity and one each for double-length positive and negative infinity). The symbol "Inf" means any one of those representations of infinity. Note that if the floating point registers are performing operations on double-length numbers, Inf refers to double-length infinities only. Similarly, for single-length operations, Inf refers to the two single-length infinities.

The symbols "NaN" and "Inf" arise when performing floating pont arithmetic. Such arithmetic involving NaNs or Infs sets the floating point Error flag (to indicate that the result of the operation is invalid). The use of the "==" comparision operator, to test for equality (discussed below), really tests for membership in a set when referring to NaNs and Infs. That is, "FA == NaN" means,compare FA to all possible NaNs (of the appropiate length, single or double, using the same length as FA). If FA is one of those NaNs, the "FA == NaN" evaluates true; otherwise, if FA is not a member of the set of NaNs, the "FA == NaN" evaluates false. In similar fashion, "FA == Inf" really means "is FA equal to positive or negative infinity? (of the same length as FA)." In this case the double equals tests for set membership rather than strict equality (since there are two infinities of the same length as FA).

Overflow can occur if the result of an arithmetic operation exceeds the ability of the register to express that result, that is the number is too big to fit in the 32-bit (or 64-bit) register. This is very much like an odometer in a car exceeding 999999 miles; it then wraps around to 000000. However, overflow has occurred. Similarly, underflow can happen when a number is too small to be represented by the register designated to hold the result. The use of the words "Overflow" and "Underflow" indicate such conditions.

The pseudocode adopts a C-like symbology for comparison and assignment operations and a more Pascal-like symbology for conditional and looping operations. Comments lines are denoted by two slashes "//" and they are in C++. (Note that the double slashes behave the same as a semi-colon in most assembler languages.) Parentheses are used to delimit expressions and arrange their order of evaluation. Indentation is used to denote blocks of loops or conditionals, that is everything on the same level of indentation is nested at the same level.

Symbology Summary

Comment Lines

// double slashes denote that the remainder of the line is a comment

Arithmetic Operations (in order of precedence)

** exponentiation
*, / multiplication, division
+, - addition, subtraction

Comparison Operators

== equality (equals)
!= inequality (not equals)
>= greater than or equals
<= less than or equals
> strictly greater than
< strictly less than

Unsigned Comparison Operators

>=unsigned unsigned greater than or equals
<=unsigned unsigned less than or equals
>unsigned unsigned strictly greater than
<unsigned unsigned strictly less than

Bitwise Operators

& bitwise arithmetic AND
| bitwise inclusive OR
^ bitwise exclusive OR
>> bitwise arithmetic shift right
<< bitwise arithmetic shift left

Logical Operators

AND logical AND
OR logical OR
XOR logical XOR
NOT logical NOT (negation)

Logical Values

TRUE boolean TRUE (meaning not zero)
FALSE boolean FALSE (meaning zero)

Conditional Constructs

If expression is true, then perform statements:

IF (expression)
THEN (statements)

If expression is true, then perform "X statements"; else if
expression is false, perform "Y statements."

IF (expression)
THEN (X statements)
ELSE (Y statements)

Loop Constructs

Unconditional jump (transfer of execution):

Loop:
(statements)
GOTO Loop

Conditional Looping

Perform statements while expression is true.

WHILE (expression) (statements)

FOR (index ; expression ; index increment)
(statements)

Exiting a Loop

Unconditional exit from a loop

BREAK

Computational Functions

ABS(X) returns the absolute value of X
X REM Y returns the remainder of X divided by Y
SQRT(X) returns the value the square root of X

Exchange Contents Function

SWAP(X, Y) swaps the values of X and Y

Numeric Data Types

INT 32-bit integer value (default integer)
INT32 32-bit integer value
INT64 64-bit integer value
REAL32 32-bit single-length floating point value
REAL64 64-bit double-length floating point value

Type Casting

Variable = (TYPE)Variable

Where TYPE can be any of the numeric data types.

Addressing Operators

*(Address) byte pointed to by address
Address[index] word located at (address + 4*index)

Scheduling Related Descriptions

DESCHEDULE deschedule the specified process
START start the specified process
STOP stop the specified process

Comparisions are signed unless otherwise indicated. That is, the values involved in the comparisons will be treated as signed data types. For example, 0x80000000 = -1, not 2**31. So 0x80000000 is less than 0x00000001 in a (normal) signed comparison, but is greater than 0x00000001 in an unsigned comparison.

Note that, in the representation of 64-bit types (either real or integer) the low order word of the 64-bit quantity (64 bits equals two 32-bit words) is stored in memory first, that is, 64-bit quantities are stored in low word first and then high word next order. Double words can be represented by concatenating two registers such as BA (which denotes B as the high order word and A as the low order word of a double).

Indentation in conditionals and loops indicate the scope of the conditional or loop. Example:

```
IF (conditional)
THEN statement one
statement two
statement three
ELSE
statement four
statement five
```

statement six

If the conditional is true in the above example, statements one, two, and three will be executed; if the conditional is not true (i.e., is false), then statements four and five will be executed. In any case, after statement three (if conditional is true) or statement five (if conditional is false) executes, then statement six will be executed.

Type casting (as in the C programming language) is used to denote a conversion from one numeric representation to another. All of the integer and floating point registers have a type associated with the value they contain. For the integer unit, the values are normally INTs (INT32s) or pointers to addresses in memory, while in the floating point unit, the values are normally REAL32s or REAL64s. (Note that sometimes values in the integer registers will be half double-word values of a 64-bit word, that is, A will contain the lower word of an INT64 and B will contain the upper word.)

Addressing memory can be represented in one of two fashions. Individual bytes in memory are accessed via the C "*" operator, which finds the contents of an address. In the pseudocode following "*" is used to access the byte an address points to. To locate an entire word in memory, square brackets are used in a fashion similar to arrays in C. The index inside of the square brackets is an index to the word, that is, an offset (measured in words) from the base address of the array. This notation is used to indicate word references. For example, Wptr[1] means the word in memory located at one word past the address of the workspace pointer. The address of Wptr[1] is the same as that for *(Wptr + 4). The difference is that *(Wptr +4) references one byte in memory, while Wptr[1] references an entire word (four bytes).

The scheduling descriptions refer to the transputer’s on-chip microcoded scheduler, which switches between processes. When one process is suspended (and another started), the suspended process is said to be "descheduled." This is indicated by the pseudocode word "DESCHEDULE." In similar fashion, "START" indicates that a process is started by the scheduler. This means that the process is added to its internal queue of currently active processes. "STOP" indicates that a process is removed from that queue. Note the distinction between DESCHEDULE and STOP. DESCHEDULE simply changes which process is running and does not affect the status of the active process queue, while STOP removes a process from the scheduler’s queue of active processes.

ADC Add Constant

Opcode Clock Cycles
8X 1

Flags
Error is set on overflow.

Scheduling
Atomic.

Pseudocode
```
A = A + Oreg
IF Overflow THEN Set Error
Oreg = 0
Iptr = Next Instruction
```

Description
Add constant loads the lower nibble of its operation code into the operand register and then adds the contents of the operand register and A. The result is left in the A register. B and C are unaffected. If the addition overflows the value in A from 0x7FFFFFFF to 0x80000000 (in either direction), the Error flag is set. This reflects the transputer architecture's strong bias towards signed arithmetic.

ADD Add

Opcode Clock Cycles
F5 1

Flags
Error is set on overflow.

Scheduling
Atomic.

Pseudocode
```
A = A + B
IF Overflow THEN Set Error
B = C
C = Undefined
```

Description
Add adds the contents of A and B and leaves the result in A. C is popped into B. The Error flag is set if overflow occurs.

AJW Adjust Workspace

Opcode Clock Cycles
BX 1

Flags
Unaffected.

Scheduling
Atomic.

Pseudocode

```
Wptr = Wptr + (4 * Oreg)
Oreg = 0
Iptr = Next Instruction
```

Description

Adjust workspace adds four times the operand register to the workspace pointer. The workspace pointer must always be aligned to a 32-bit word. Since the workspace grows towards low memory, the negative operands allocate workspace, while positive operands deallocate workspace. The AJW instruction is used to generate stack frames (i.e., areas of the stack used for passing arguments to the subroutines and return values to the calling procedure). The A, B, C register stack is unaffected.

ALT Alt Start

Opcode Clock Cycles

24 F3 2

Flags

Unaffected.

Scheduling

Atomic.

Pseudocode

```
// A flag is stored in State.s to show enabling is occurring.
// Possible values State.s can have:
// Enabling.p = 0x80000001
// Waiting.p = 0x80000002
// Ready.p = 0x80000003
// Note: State.s = Wptr[-3]

State.s = Enabling.p
```

Description

Alt start stores the flag 0x80000001 in workspace location -3 *(State.s)* to show that the enabling of an ALT construct is occurring.

User Notes

The workspace pointer must not change between the execution of the ALT instruction and the ALTEND instruction.

ALTEND Alt End

Opcode Clock Cycles

24 F5 4

Flags

Unaffected.

Scheduling

Atomic.

Pseudocode

```
// Set Iptr to first instruction of branch selected by the ALT.

Iptr = Next Instruction + Wptr[0]
```

Description
Alt end is executed after the process containing the ALT has been rescheduled. Workspace location zero contains the offset from the instruction pointer to the guard routine to execute. This offset is added to the instruction pointer, and execution continues at the appropiate guard's service routine.

User Notes
The Workspace Pointer must not change between the execution of the ALT or TALT instruction and the ALTEND instruction.

ALTWT Alt Wait

Opcode Clock Cycles
24 F4 5 (channel ready)
17 (channel NOT ready)

Flags
Unaffected.

Scheduling
This instruction can cause the current process to be descheduled.

Pseudocode
```
// Set a flag (DisableStatus) to show no branch has yet been
// selected and wait until one of the guards is ready (indicated
// by the value in State.s).
//
// DisableStatus is defined to be Wptr[0].
//
// The process will be descheduled until State.s = Ready.p.
// Possible values State.s can have:
// Enabling.p = 0x80000001
// Waiting.p = 0x80000002
// Ready.p = 0x80000003
// Note: State.s = Wptr[-3]

DisableStatus = -1 // indicates no branch yet selected
A = Undefined
B = Undefined
C = Undefined
LOOP:
WHILE (State.s != Ready.p) GOTO LOOP
Iptr = Next Instruction
```

Description
Alt wait stores 0x80000001 (-1) in workspace location zero *(DisableStatus)* and waits until *State.s* is ready (i.e., contains 0x80000003). This means that a flag is stored to show that no branch has been selected yet and the process is descheduled until one of the guards is ready.

AND Logical And

Opcode Clock Cycles
24 FB 1

Flags
Unaffected.

Scheduling

Atomic.

Pseudocode

```
A = A & B
B = C
C = Undefined
Iptr = Next Instruction
```

Description

And performs a bitwise logical and operation between A and B and then leaves the result in A. C is popped into B.

BCNT Byte Count

Opcode Clock Cycles

23 F4 2

Flags

Unaffected.

Scheduling

Atomic.

Pseudocode

```
A = A * 4 // four bytes in a word
Iptr = Next Instruction
```

Description

Byte count returns four times A into A. This is useful for computing byte offsets from word offsets. B and C are unaffected.

BITCNT Count Bits Set in Word T800

Opcode Clock Cycles

27 F6 b + 2

Flags

Unaffected.

Scheduling

Atomic.

Pseudocode

```
A = B + (number of bits set in A)
B = C
C = Undefined
Iptr = Next Instruction
```

Description

Count bits set in word counts the number of bits that are set (to one) in A and adds that number to the contents of B. The idea is that a total number of bits set can be accumulated if so desired. After execution, A contains the number of bits set plus the former contents of B. C is popped into B and is undefined.

BITREVNBITS Reverse Bottom N Bits in Word T800

Opcode Clock Cycles

27 F8 n + 4

Flags
Unaffected.

Scheduling
Atomic.

Pseudocode
```
A = reversed bit pattern of (the A least significant bits of B)
B = C
C = Undefined
Iptr = Next Instruction
```

Description
Reverse bottom n bits in word reverses the bottom N bits in B, where N is the value in A. All bits in B more significant than N are cleared to zero prior to the reverse operation. After execution, A contains the reversed bit pattern and C is popped into B. C is undefined.

BITREVWORD Reverse Bits in Word T800

Opcode Clock Cycles
27 F7 36

Flags
Unaffected.

Scheduling
Atomic.

Pseudocode
```
A = reversed bit pattern of A
Iptr = Next Instruction
```

Description
Reverse bits in word reverses the bit pattern of the word held in A. Thus, if A were to contain the hexadecimal number 0x12345678 (which expands to 0001 0010 0011 0110 0101 0110 0111 1000 in binary) after the instruction was executed, A would contain 0x1E6A2C48 (which expands to 0001 1110 0110 1010 0010 1100 0100 1000 in binary). B and C are unaffected.

BSUB Byte Subscript

Opcode Clock Cycles
F2 1

Flags
Unaffected.

Scheduling
Atomic.

Pseudocode
```
A = A + B
B = C
C = Undefined
```

```
Iptr = Next Instruction
```

Description
Byte subscript assumes that A is the base address of an array and B is a byte index into the array. It adds A and B and leaves the result in A and is designed for computing offsets into byte arrays. C is popped into B.

User Notes
BSUB does not check the sum for overflow, so it is useful for unsigned arithmetic.

CALL Call Subroutine

Opcode Clock Cycles
9X 7

Flags
Unaffected.

Scheduling
Atomic.

Pseudocode
```
Wptr[3] = C
Wptr[2] = B
Wptr[1] = A
Wptr[0] = Iptr
Wptr = Wptr - 4
A = Iptr
Iptr = Iptr + Oreg
Oreg = 0
```

Description
Call pushes the C, B, and A registers, as well as the instruction pointer on the workspace. It then jumps to the relative address specified by the operand register (i.e., the operand register is added to the instruction pointer, which determines the next instruction to execute). Prior to adding the operand register, A receives the old instruction pointer (i.e., the address of the instruction to execute after returning from the subroutine). The B and C registers are unaffected.

CCNT1 Check Count From One

Opcode Clock Cycles
24 FD 3

Flags
Error is set if count is not from one (i.e., B is zero) or out of bounds (i.e., B is greater than A).

Scheduling
Atomic.

Pseudocode
```
IF (B == 0) THEN Set Error
IF (B >unsigned A) THEN Set Error
A = B
B = C
C = Undefined
Iptr = Next Instruction
```

Description

Check count from 1 verifies that the value in B is greater than zero and less than or equal to the value in A. INMOS suggests using CCNT1 to make sure that the count of an output or input instruction is greater than zero, yet less than the number of bytes in the message buffer.

CFLERR Check Single FP Inf or NaN T414

Opcode Clock Cycles
27 F3 3

Flags
Error is set if A is an infinity or a NaN; otherwise, error is unchanged.

Scheduling
Atomic.

Pseudocode
```
// SINGLE_FP_INFINITY = 0x7F800000
// Infinity has the maximum exponent and a zero fraction.
//
// A single-length FP number is a NaN when:
// NaN AND SINGLE_FP_INFINITY = SINGLE_FP_INFINITY
// and
// NaN XOR SINGLE_FP_INFINITY != 0
// NaNs have the maximum exponent and a nonzero fraction.

IF (A == SINGLE_FP_INFINITY) THEN Set Error
IF (A == NaN) THEN Set Error
Iptr = Next Instruction
```

Description
Check single length floating point infinity or NaN examines the value held in the A register and sets the Error flag if it is a floating point infinity or NaN. A, B, and C are unaffected.

CJ Conditional Jump

Opcode Clock Cycles
AX 2 (if not taken)
4 (if taken)

Flags
Unaffected.

Scheduling
Atomic.

Pseudocode
```
IF (A == 0)
THEN Iptr = Iptr + Oreg
ELSE IF (A != 0)
THEN
A = B
B = C
C = Undefined
Iptr = Next Instruction
```

Description
Conditional jump jumps to the relative offset in the operand register if the value in A is zero (false). To "invert" the

sense of a *CJ* instruction, you can use the code sequence *eqc 0 ; cj destination*, which causes the jump to occur only when the value in the A register is nonzero (true). If the jump is not taken, the A, B, and C registers are left unchanged. However, if the jump is not taken, B is inserted into A and C is inserted into B.

CLRHALTERR Clear HaltOnError Flag

Opcode Clock Cycles
25 F7 1

Flags
The HaltOnError flag is cleared (i.e., set to zero).

Scheduling
Atomic.

Pseudocode
```
Clear HaltOnError
Iptr = Next Instruction
```

Description
Clear HaltOnError clears the *HaltOnError* flag (sets it to zero, meaning false).

CRCBYTE Calculate CRC on Byte T800

Opcode Clock Cycles
27 F5 11

Flags
Unaffected.

Scheduling
Atomic.

Pseudocode
```
A = CRC of top byte of A with generator C and accumulated CRC B
B = C
C = Undefined
Iptr = Next Instruction
```

Description
Calculate CRC on byte calculates the CRC value of the byte held in the most significant byte of A using the generator in the C register. B contains the CRC that has been accumulated over successive CRCBYTE instructions. After execution, A contains the CRC and C has been popped into B.

User Notes
It is important to remember that the transputer computes CRCs in little-endian fashion and that this may differ with the machine the CRC values will be sent to or tested against.

CRCWORD Calculate CRC on Word T800

Opcode Clock Cycles
27 F4 35

Flags
Unaffected.

Scheduling
Atomic.

Pseudocode
```
A = CRC of A with generator C and accumulated CRC B
```

Description
Calculate CRC on word calculates the CRC value of the word held in A using the generator in C. B contains the CRC that has been accumulated over successive CRCWORD instructions. After execution, A contains the CRC and C has been popped into B.

User Notes
It is important to remember that the transputer computes CRCs in little-endian fashion and that this may differ with the machine the CRC values will be sent to or tested against.

CSNGL Check Single

Opcode Clock Cycles
24 FC 3

Flags
Error is set if the value will not fit into a single word.

Scheduling
Atomic.

Pseudocode
```
IF ( (A < 0 AND B != -1) OR (A >= 0 AND B != 0) )
THEN Set Error
A = Single-Length Signed Value of AB
B = C
C = Undefined
Iptr = Next Instruction
```

Description
Check single reduces a 64-bit value held in A and B to a single 32-bit word left in A. Note that C is popped into B.

CSUB0 Check Subscript from Zero

Opcode Clock Cycles
21 F3 2

Flags
Error is set if B is greater or equal to A; otherwise it remains unchanged.

Scheduling
Atomic.

Pseudocode
```
IF (B >=unsigned A) THEN Set Error
A = B
B = C
C = Undefined
Iptr = Next Instruction
```

Description
Check subscript from 0 sets the Error flag if the unsigned value in B is greater than or equal to the unsigned value in A. The register stack is popped, A receives B, B receives C, and C is undefined.

CWORD Check Word

Opcode Clock Cycles
25 F6 5

Flags
Error is set if the value will not fit into a specified partword size.

Scheduling
Atomic.

Pseudocode
```
IF ( (B >= A) OR (B < -A) ) THEN Set Error
A = B
B = C
C = Undefined
Iptr = Next Instruction
```

Description
Check word checks a 32-bit word to see if it will fit into a smaller bit-field.

DIFF Difference

Opcode Clock Cycles
F4 1

Flags
Unaffected.

Scheduling
Atomic.

Pseudocode
```
A = B - A
B = C
C = Undefined
Iptr = Next Instruction
```

Description
Difference subtracts A from B leaving the result in A without checking for overflow or carry. C is popped into B.

DISC Disable Channel

Opcode Clock Cycles
22 FF 8

Flags
Unaffected.

Scheduling
Atomic.

Pseudocode

```
// B contains flag, C contains channel address
// DisableStatus is defined to be Wptr[0]
IF ( (B == TRUE) AND (channel C ready) AND (no branch selected) )
THEN // then select this branch
DisableStatus = A
A = TRUE
ELSE A = FALSE
B = Undefined
C = Undefined
Iptr = Next Instruction
```

Description
Disable channel disables a channel pointed to by C. If the flag in B is true (one) and no other guard has been selected, this guard is selected as the next execution branch. A contains an offset to the routine that starts the process. If the guard is taken, A is stored in workspace location zero and a true flag is returned in A. If the guard is not taken, then a false flag is returned in A. In essence, one of these two cases holds:

1. B is true, the channel pointed to by C is ready, and no other branch has been selected. This branch will then be selected by saving A in workspace location zero; A will be set to true.

2. B is false, or the channel pointed to by C is not ready, or another branch was already selected, in which case A gets set to false (zero).

DISS Disable Skip

Opcode Clock Cycles
23 F0 4

Flags
Unaffected.

Scheduling
Atomic.

Pseudocode

```
// DisableStatus is defined to be Wptr[0]
IF ( (B == TRUE) AND (no branch selected) )
THEN // then select this branch
A = TRUE
DisableStatus = A
ELSE A = FALSE
B = C
C = Undefined
Iptr = Next Instruction
```

Description
Disable skip disables a *skip* for a guard that was previously initialized with ENBS. A contains an offset to the guard to disable and B contains a flag that allows it to decide whether to select this guard or not. C is popped into B.

DIST Disable Timer

Opcode Clock Cycles
22 FE 23

Flags

Unaffected.

Scheduling
Atomic.

Pseudocode
```
// DisableStatus is defined to be Wptr[0]
IF ( (B == TRUE)
AND (no branch selected)
AND (time later than guard’s time) )
THEN // select this branch
DisableStatus = A
A = TRUE
ELSE A = FALSE
B = Undefined
C = Undefined
Iptr = Next Instruction
```

Description
Disable timer disables a timer guard the same way DISC disables a channel guard. C contains the time, the B register contains a flag, and A contains the offset to the guard’s associated service routine. The *disable timer* instruction works in the following ways:

If the flag in B is true, and the time (in workspace location -5,) is later than the time in C, and

No other branch has been selected,

Then this branch is selected and A is saved in workspace location zero; A is set to true (one).

Otherwise, if B is false or the time is earlier than the time in C or another branch has been taken, then A is set to false.

DIV Divide

Opcode Clock Cycles
22 FC 39

Flags
Error is set on division by zero.

Scheduling
Atomic.

Pseudocode
```
IF ( (A == 0) OR ( (A == -1) AND (B = -2**31) )
THEN
Set Error
A = Undefined
ELSE
A = B / A
B = C
C = Undefined
Iptr = Next Instruction
```

Description
Divide divides B by A and leaves the result in A. The result is undefined if A is zero. C is popped into B.

DUP Duplicate Top of Stack T800

Opcode Clock Cycles
25 FA 1

Flags
Unaffected.

Scheduling
Atomic.

Pseudocode
```
C = B
B = A
Iptr = Next Instruction
```

Description
Duplicate top of stack duplicates the contents of A into B. First, B is pushed into C, while the previous contents of C are lost.

ENBC Enable Channel

Opcode Clock Cycles
24 F8 7 (if ready)
5 (if not ready)

Flags
Unaffected.

Scheduling
Atomic.

Pseudocode
```
// Possible values State.s can have:
// Enabling.p = 0x80000001
// Waiting.p = 0x80000002
// Ready.p = 0x80000003
// Note: State.s = Wptr[-3]

IF ( (A == TRUE) AND (no process waiting on channel B) )
THEN (initiate communication on channel B)
ELSE IF ( (A == TRUE) AND (current process waiting on channel B) )
THEN (already waiting on this channel so ignore)
ELSE IF ( (A == TRUE) AND (another process waiting on channel B) )
THEN // set flag to show a guard is ready
State.s = Ready.p
B = C
C = Undefined
Iptr = Next Instruction
```

Description
Enable channel enables a channel that has been pointed to by the B register only if the contents of A is true (one); otherwise, the channel is not enabled. If A is true, there are three cases:

1. No process is waiting on the channel pointed to by B. In this case, ENBC stores the current process workspace descriptor into the channel to initiate communication.

2. The current process is waiting on the channel pointed to by B. In this case, nothing is done and the instruction is ignored.

3. Another process is waiting on the channel pointed to by B. Here the ready flag (0x80000003) is stored at workspace location -3 to show that the guard is ready. C is popped into B.

ENBS Enable Skip

Opcode Clock Cycles
24 F9 3

Flags
Unaffected.

Scheduling
Atomic.

Pseudocode
```
// Possible values State.s can have:
// Enabling.p = 0x80000001
// Waiting.p = 0x80000002
// Ready.p = 0x80000003
// Note: State.s = Wptr[-3]

IF (A == TRUE)
THEN // set flag to show a guard is ready
State.s = Ready.p
Iptr = Next Instruction
```

Description
Enable skip sets a flag to show that the guard is ready. It performs this by storing *ready*, 0x80000003, into *State.s.* All guards must be enabled prior to the *alternate wait* instruction.

ENBT Enable Timer

Opcode Clock Cycles
24 F7 8

Flags
Unaffected.

Scheduling
Atomic.

Pseudocode
```
// Tlink.s = Wptr[-4] indicates if the guard is ready or not.
// Tlink.s can either be: TimeNotSet.p = -1 or TimeSet.p = -2.
// TimeNotSet.p means the time has not yet been reached
// or TimeSet.p, meaning the time was reached (i.e., timed out).
// Time.s = Wptr[-5] and contains the time a process is waiting
// until before it times out.

// if A is true and alt time not yet set
IF ( (A == TRUE) AND (Tlink.s == TimeNotSet.p) )
THEN // set time set flag and set alt time to time of guard
Tlink.s = TimeSet.p
Time.s = B
```

```
// else if A is true and alt time set and earlier than this guard
ELSE IF ( (A == TRUE) AND (Tlink.s == TimeSet.p) AND (Time.s < B) )
THEN // ignore this guard
BREAK // exit if-else if conditional with no action
// else if A is true and alt time set and later than this guard
ELSE IF ( (A == TRUE) AND (Tlink.s == TimeSet.p) AND (Time.s > B) )
THEN // set alt time to time of this guard
Time.s = B
B = C
C = Undefined
Iptr = Next Instruction
```

Description
Enable timer enables a timer guard. A register contains a flag (0 for false and 1 for true) and B register contains a time. Assuming the flag is true, then either:

1. "Alt timer not seen yet" (*Tlink.s* is 0x80000002) and the flag to set the time will be set, and the "alt time" will be set to the time of the guard. *Tlink.s* is set to 0x80000001 and is stored in location -5.

2. "Alt time set and earlier than this guard." In this case you should ignore this guard.

ENDP End Process

Opcode Clock Cycles
F3 13

Flags
Unaffected.

Scheduling
This instruction will cause the current process to be descheduled.

Pseudocode

```
// A = Wptr of process to be started up when count becomes zero
// Wptr[0] = Iptr value to start process at
// Wptr[1] = Count of processes yet to stop, i.e. Wptr[1] = *(A + 1)
IF ( *(A + 1) == 1 )
THEN (continue as process with waiting workspace A)
ELSE IF ( *(A + 1) != 1)
THEN (start next waiting process)
*(A + 1) = *(A + 1) - 1 // decrement *(A + 1)
```

Description
End process ends the current process. ENDP is also used to synchronize the termination of multiple processes. The A register holds the workspace pointer of a succeding process that will start when all process sets have been terminated. Location zero in that workspace (i.e., the address pointed to by the workspace pointer) contains the instruction pointer to use, and location one contains the count of other processes yet to be halted. When a process executes an ENDP, it decrements that count. If the result is nonzero, it terminates and gives up control of the processor; however, if the result is zero, the workspace pointer and instruction pointer are assigned the new values and continue as the current process.

EQC Equals Constant

Opcode Clock Cycles
CX 2

Flags
Unaffected.

Scheduling
Atomic.

Pseudocode
```
IF (A == Oreg)
THEN A = TRUE
ELSE
A = FALSE
Oreg = 0
Iptr = Next Instruction
```

Description
Equals constant tests the value in A against the operand register. The instruction leaves a "1" (true) in the A register if they are equal and a "0" (false) if they are not equal. B and C are unaffected.

FMUL Fractional Multiply

Opcode Clock Cycles
27 F2 35 (no rounding)
40 (rounding)

Flags
Error is set on overflow.

Scheduling
Atomic.

Pseudocode
```
IF ( (A == -1) AND (B == -1) ) THEN Set Error
A = (2**31) * (B * A)
B = C
C = Undefined
Iptr = Next Instruction
```

Description
Fractional multiply is used to multiply two fractional numbers that are represented in fixed point arithmetic. FMUL interprets A and B as though they contained fixed point numbers lying in the range -1 <= X < 1. The value associated with the register is 2-31 times its signed integer value. FMUL returns the rounded fixed-point product of these values to the A register. C is popped into B.

Note: This instruction is *not* on the T212 and is only on the T414 and T800.

FPADD Floating Point Add T800

Opcode Clock Cycles
28 F7 6 Typical / 9 Maximum

Flags
The FloatingPointError flag is set if overflow occurs or if FA or FB was a NaN or infinity.

Scheduling
Atomic.

Pseudocode
```
FA = FB + FA
FB = FC
FC = Undefined
IF (Overflow OR (FA == NaN) OR (FA == Inf) OR (FB == NaN) OR (FB == Inf))
THEN Set FP_Error
RoundMode = ToNearest
Iptr = Next Instruction
```

Description
Floating point add adds FA to FB, leaving the result in FA while popping FC up into FB. The current rounding mode is used. Both FA and FB should be of the same format (either both single or both double).

FPB32TOR64 Floating Point Bit 32 to Real 64 T800

Opcode Clock Cycles
29 FA 8 Typical / 8 Maximum

Flags
Unaffected.

Scheduling
Atomic.

Pseudocode
```
// A points to an unsigned integer in memory (one word) that
// will be converted to a double-length floating point number then
// pushed onto the floating point register stack. A should be
// word-aligned.
FC = FB
FB = FA
FA = (REAL64) (*A) // loads unsigned integer, converts to real 64
A = B
B = C
C = Undefined
RoundMode = ToNearest
Iptr = Next Instruction
```

Description
Bit32 to Real 64 or *load unsigned word as Real 64* takes the unsigned 32-bit integer value pointed to by the address in A and converts it to a double length floating point number that is pushed into FA. This is an exact conversion. FPB32TOR64 is used to load the bottom word of a 64 bit integer during its conversion to a floating point number. A is popped so that B is popped into A and C is popped into B. FB is pushed into FC and FA is pushed into FB.

User Notes
A should be word-aligned.

FPCHKERR Check Floating Point Error T800

Opcode Clock Cycles
28 F3 1

Flags
The Error flag is set to one if the Floating Point Error Flag FP_ERROR is set.

Scheduling
Atomic.

Pseudocode
```
IF (FP_Error == Set) THEN Set Error
RoundMode = ToNearest
Iptr = Next Instruction
```

Description
Check floating error sets Error to the logical OR of Error and FPError. This is intended for use in checked arithmetic.

User Notes
If the HaltOnError flag is set, FPCHKERROR causes the transputer to halt when FPError is set to TRUE.

FPDIV Floating Point Divide T800

Opcode Clock Cycles
28 FC 16 Typical / 28 Maximum (Single)
31 Typical / 43 Maximum (Double)

Flags
The FloatingPointError flag is set if division by zero occurs or if any number in the operation is a NaN or infinity.

Scheduling
Atomic.

Pseudocode
```
// Note: Errors in division can occur by dividing by zero, or if
// the numerator or denominator is a NaN or an infinity.
IF ((FA == 0) OR (FA == NaN) OR (FA == Inf) OR (FB == NaN) OR (FB == Inf))
THEN Set FP_Error
FA = FB / FA
FB = FC
FC = Undefined
RoundMode = ToNearest
Iptr = Next Instruction
```

Description
Floating point divide divides FB by FA, leaving the result in FA while popping FC up into FB. The current rounding mode is used. Both FA and FB should be of the same format (either both single or both double).

FPDUP Floating Point Duplicate T800

Opcode Clock Cycles
2A F3 1

Flags
Unaffected.

Scheduling
Atomic.

Pseudocode
```
FC = FB
FB = FA
FA = FA
RoundMode = ToNearest
Iptr = Next Instruction
```

Description
Fpdup duplicates the top of the floating point stack. FB is pushed into FC and FA is duplicated into FB.

FPENTRY Floating Point Unit Entry T800

Opcode Clock Cycles
2A FB 1

Flags
Unaffected.

Scheduling
Atomic.

Pseudocode
```
// This instruction starts the floating point engine executing
// using the contents of the A register as the instruction to
// execute.
A = B
B = C
C = Undefined
// Execute operation A on the floating point unit
Iptr = Next Instruction
```

Description
Fpentry causes the value in A to be executed as a floating point unit instruction. The value is actually an entry point into the microcode ROM on the floating point unit. The FPENTRY instruction is similar to the OPERATE instruction, except that it is used exclusively for floating point related instructions.

FPEQ Floating Point Equals T800

Opcode Clock Cycles
29 F5 3 Minimum / 5 Maximum

Flags
The FloatingPointError flag will be set if A or B is a NaN or infinity.

Scheduling
Atomic.

Pseudocode
```
IF ((FA == NaN) OR (FA == Inf) OR (FB == NaN) OR (FB == Inf))
THEN Set FP_Error
```

```
FA = FC
FB = Undefined
FC = Undefined
C = B
B = A
A = (FB == FA) // A receives either TRUE or FALSE
RoundMode = ToNearest
Iptr = Next Instruction
```

Description
Floating point equality tests if FA = FB and places the result of that comparison in A. Both FA and FB are popped, so FA receives FC. B is pushed into C and A is pushed into B prior to A receiving true or false. If FA or FB is a NaN or an infinity, FPError is set.

FPGT Floating Point Greater Than T800

Opcode Clock Cycles
28 F7 3 Minimum / 6 Maximum

Flags
The FloatingPointError flag will be set if A or B is a NaN or infinity.

Scheduling
Atomic.

Pseudocode
```
IF ((FA == NaN) OR (FA == Inf) OR (FB == NaN) OR (FB == Inf))
THEN Set FP_Error
FA = FC
FB = Undefined
FC = Undefined
C = B
B = A
A = (FB > FA) // A receives either TRUE or FALSE
RoundMode = ToNearest
Iptr = Next Instruction
```

Description
Floating point greater than tests if FA > FB and places the result of that comparison in A. Both FA and FB are popped, so FA receives FC. B is pushed into C and A is pushed into B prior to A receiving true or false. If FA or FB is a NaN or an infinity, FPError is set.

FPI32TOR32 Floating Point Int 32 to Real 32 T800

Opcode Clock Cycles
29 F6 8 Typical / 10 Maximum

Flags
Unaffected.

Scheduling
Atomic.

Pseudocode
```
// A points to a 32-bit integer that will be converted to a
// single-length floating point number and then pushed onto
// the floating point register stack. A should be word-aligned.
```

```
FC = FB
FB = FA
FA = (REAL32) (*A) // loads 32-bit integer, converts to real 32
A = B
B = C
C = Undefined
RoundMode = ToNearest
Iptr = Next Instruction
```

Description
Int 32 to Real 32 takes a 32-bit integer value pointed to by A and rounds it to a single-length floating point number that is pushed into FA. B is popped into A and C is popped into B. FB is pushed into FC and the previous value of FA is pushed into FB.

User Notes
In revision A of the silicon (for the transputer), FB may remain unchanged (rather than receive the value of FA). A should be word-aligned.

FPI32TOR64 Floating Point Int 32 to Real 64 T800

Opcode Clock Cycles
29 F8 8 Typical / 10 Maximum

Flags
Unaffected.

Scheduling
Atomic.

Pseudocode
```
// A points to a 32-bit integer that will be converted to
// a double-length floating point number and pushed onto the
// floating point register stack. A should be word-aligned.
FC = FB
FB = FA
FA = (REAL64) (*A) // loads 32-bit integer, converts to real 64
A = B
B = C
C = Undefined
RoundMode = ToNearest
Iptr = Next Instruction
```

Description
Int32 to Real 64 takes a 32-bit integer value pointed to by A and converts it to a double-length floating point number that is pushed into FA. This is an exact conversion. B is popped into A and C is popped into B. FB is pushed into FC and the previous value of FA is pushed into FB.

User Notes
A should be word-aligned.

FPINT Floating Point Round to Floating Integer T800

Opcode Clock Cycles
2A F1 5 Typical / 6 Maximum

Flags
Unaffected.

Scheduling
Atomic.

Pseudocode
```
FA = (REAL)((INT)FA) // FA is rounded to the nearest integer
RoundMode = ToNearest
Iptr = Next Instruction
```

Description
Round to floating integer converts a floating point number held in FA to an integer value in the floating point format by simply rounding it (using the current rounding mode). No other registers are affected.

User Notes
FPINT is used for both double and single-length floating point numbers. The result of FPINT is not defined if FA is a NaN or an infinity.

FPLDNLADDDB Floating Point Load Non-Local and Add Double T800

Opcode Clock Cycles
2A F6 3 + FPADD cycles

Flags
The FloatingPointError flag is set if overflow occurs in the addition, or if FA is a NaN or infinity, or if the number A points to is a NaN or infinity.

Scheduling
Atomic.

Pseudocode
```
// A points to two words in memory that contain a double-
// length floating point value. *A contains the lower word
// of the floating point number and*(A + 4) contains the high
// order word. A should contain a word-aligned address.
FA = FA + (REAL64)(*A) // A points to two words
IF (OVERFLOW OR (FA == NaN) OR (FA == Inf))
THEN Set FP_Error
FB = FB // FB is unchanged
FC = Undefined
A = B
B = C
C = Undefined
RoundMode = ToNearest
Iptr = Next Instruction
```

Description
Floating load non-local and add double loads the double-length floating point number pointed to by A into FA and then adds FA and FB. This instruction is equivalent to FPLDNLDB ; FPADD. B is popped into A and C into B.

User Notes
A points to two words in memory. For example, the double floating point representation for 1.0 is 0x3FF00000:00000000. *A would contain 0x00000000 and *(A + 4) would contain 0x3FF00000. If A is not word-aligned, the results of FPLDNLADDDB are undefined.

FPLDNLADDSN Floating Point Load Non-Local and Add Single T800

Opcode Clock Cycles

2A FA 2 + FPADD cycles

Flags
The FloatingPointError flag is set if overflow occurs.

Scheduling
Atomic.

Pseudocode
```
// A points to a single-length floating point number in memory.
FA = FA + (REAL32)(*A) // A points to a word in memory
IF (OVERFLOW OR (FA == NaN) OR (FA == Inf))
THEN Set FP_Error
FB = FB // FB is unchanged
FC = Undefined
A = B
B = C
C = Undefined
RoundMode = ToNearest
Iptr = Next Instruction
```

Description
Floating load non-local and add single loads the single-length floating point number pointed to by A into FA and then adds FA and FB. This instruction is equivalent to FPLDNLSN ; FPADD. B is popped into A and C into B.

User Notes
If A is not word-aligned, the results of FPLDNLADDSN are undefined.

FPLDNLDB Floating Point Load Non-Local Double T800

Opcode Clock Cycles
28 FA 3

Flags
Unaffected.

Scheduling
Atomic.

Pseudocode
```
// A points to two words in memory that contain a double-
// length floating point value. *A contains the lower word
// of the floating point number and *(A + 4) contains the high
// order word. A should contain a word-aligned address.
FC = FB
FB = FA
FA =(REAL64)(*A) // A points to two words
A = B
B = C
C = Undefined
RoundMode = ToNearest
Iptr = Next Instruction
```

Description
Floating load non-local double pushes the double-length floating point number pointed to by A into FA. FB is pushed into FC. B is popped into A and C into B.

User Notes
A points to two words in memory. For example, the double floating point representation for 1.0 is 0x3FF00000:00000000. *A would contain 0x00000000 and *(A + 4) would contain 0x3FF00000. If A is not word-aligned, the results of FPLDNLDB are undefined.

FPLDNLDBI Floating Point Load Non-Local Indexed Double T800

Opcode Clock Cycles
28 F2 6

Flags
Unaffected.

Scheduling
Atomic.

Pseudocode
```
// A contains an array base address, B is the index into
// that array of the element to load. Each element in the
// array is two words long and contains a double-length
// floating point number. So FA = A[B]
FC = FB
FB = FA
FA =(REAL64)(*(A + 4*(B*2))) // FA = A[B]
A = B
B = C
C = Undefined
RoundMode = ToNearest
Iptr = Next Instruction
```

Description
Floating load non-local double indexed pushes the double-length floating point number pointed to by A and indexed by B into FA. It is equivalent to the instruction sequence WSUBDB ; FPLDNLDB. FB is pushed into FC. B is popped into A and C into B.

User Notes
Each element of the array that A points to is two words long and contains a double-length floating point number with the low order word stored first. For example, the double floating point representation for 1.0 is 0x3FF00000:00000000. Suppose that A[B] = 1.0. Then *(A + 4*(B*2)) would contain 0x00000000 and *(A + 4*(B*2) + 4) would contain 0x3FF00000. If A is not word-aligned, the results of FPLDNLDBI are undefined.

FPLDNLMULDB Floating Point Load Non-Local and Multiply Double T800

Opcode Clock Cycles
2A F8 3 + FPMUL cycles

Flags
The FloatingPointError flag is set if FA contains a NaN or infinity, or if A points to a NaN or infinity, or if the product of FA and *A overflows or produces a NaN or infinity.

Scheduling
Atomic.

Pseudocode
```
// A points to two words in memory that contain a double-
// length floating point value. *A contains the lower word
// of the floating point number and *(A + 4) contains the high
// order word. A should contain a word-aligned address.
FA = FA * (REAL64)(*A) // A points to two words
IF (OVERFLOW OR (FA == NaN) OR (FA == Inf))
THEN Set FP_Error
FB = FB // FB is unchanged
FC = Undefined
A = B
B = C
C = Undefined
RoundMode = ToNearest
Iptr = Next Instruction
```

Description
Floating load non-local and multiply double loads the double-length floating point number pointed to by A into FA and then multiplies FA and FB. This instruction is equivalent to FPLDNLDB ; FPMUL. FC is popped into FB. B is popped into A and C into B.

User Notes
A points to two words in memory. For example, the double floating point representation for 1.0 is 0x3FF00000:00000000. *A would contain 0x00000000 and *(A + 4) would contain 0x3FF00000. If A is not word-aligned, the results of FPLDNLMULDB are undefined.

FPLDNLMULSN Floating Point Load Non-Local and Multiply Single T800

Opcode Clock Cycles
2A FC 2 + FPMUL cycles

Flags
The FloatingPointError flag is set if FA contains a NaN or infinity, or if A points to a NaN or infinity, or if the product of FA and *A overflows or produces a NaN or infinity.

Scheduling
Atomic.

Pseudocode
```
// A points to a single-length floating point number in memory.
FA = FA * (REAL32)(*A) // A points to a word in memory
IF (OVERFLOW OR (FA == NaN) OR (FA == Inf))
THEN Set FP_Error
FB = FB // FB is unchanged
FC = Undefined
A = B
B = C
C = Undefined
RoundMode = ToNearest
Iptr = Next Instruction
```

Description
Floating load non-local and multiply single loads the single-length floating point number pointed to by A into FA and then multiplies FA and FB. This instruction is equivalent to FPLDNLSN ; FPMUL. FC is popped into FB. B is popped into A and C into B.

User Notes
If A is not word-aligned, the results of FPLDNLMULSN are undefined.

FPLDNLSN Floating Point Load Non-Local Single T800

Opcode Clock Cycles
28 FE 2

Flags
Unaffected.

Scheduling
Atomic.

Pseudocode
```
// A points to a single-length floating point number in memory.
FC = FB
FB = FA
FA = (REAL32)*A // A points to a single-length floating point number
A = B
B = C
C = Undefined
RoundMode = ToNearest
Iptr = Next Instruction
```

Description
Floating load non-local single pushes the single-length floating point number pointed to by A into FA. FB is pushed into FC. B is popped into A and C into B.

User Notes
If A is not word-aligned, the results of FPLDNLSN are undefined.

FPLDNLSNI Floating Point Load Non-Local Indexed Single T800

Opcode Clock Cycles
28 F6 4

Flags
Unaffected.

Scheduling
Atomic.

Pseudocode
```
// A points to the base of an array and B indexes an element in the array.
// Each element in the array is a single-length floating point number.
FC = FB
FB = FA
FA = (REAL32) *(A + 4*B) // FA = A[B]
A = C
B = Undefined
C = Undefined
RoundMode = ToNearest
Iptr = Next Instruction
```

Description
Floating load non-local single indexed pushes the single-length floating point number pointed to by A and indexed by B into FA. It is equivalent to the instruction sequence WSUB ; FPLDNLSN. FB is pushed into FC. B is popped into A and C into B.

User Notes

If A is not word-aligned, the results of FPLDNLSNI are undefined.

FPLDZERODB Floating Point Load Zero Double T800

Opcode Clock Cycles
2A F0 2

Flags
Unaffected.

Scheduling
Atomic.

Pseudocode
```
FC = FB
FB = FA
FA = (REAL 64) 0.0 // load double-length 0.0
RoundMode = ToNearest
Iptr = Next Instruction
```

Description
The double-length floating point value for zero is pushed onto the floating point register stack. FC receives the prior value of FB and FB receives the prior value of FA. FA receives double-length 0.0 which is represented as 0x00000000:00000000.

FPLDZEROSN Floating Point Load Zero Single T800

Opcode Clock Cycles
29 FF 2

Flags
Unaffected.

Scheduling
Atomic.

Pseudocode
```
FC = FB
FB = FA
FA = (REAL 32) 0.0 // load single-length 0.0
RoundMode = ToNearest
Iptr = Next Instruction
```

Description
The single-length floating point value for zero is pushed onto the floating point register stack. FC receives the prior value of FB, and FB receives the prior value of FA. FA receives single-length 0.0, which is represented as 0x00000000.

FPMUL Floating Point Multiply T800

Opcode Clock Cycles
28 FB 11 Typical / 18 Maximum (Single)
18 Typical / 27 Maximum (Double)

Flags
The FloatingPointError flag is set if FA or FB is a NaN or an infinity, or if the product of FA and FB overflows, or is a

NaN or infinity.

Scheduling
Atomic.

Pseudocode
```
FA = FB * FA
FB = FC
FC = Undefined
IF (Overflow OR (FA == NaN) OR (FA == Inf) OR (FB == NaN) OR (FB == Inf))
THEN Set FP_Error
RoundMode = ToNearest
Iptr = Next Instruction
```

Description
Floating point multiply multiplies FA by FB, leaving the result in FA while popping FC up into FB. The current rounding mode is used. Both FA and FB should be of the same format (either both single or both double).

FPNAN Floating Point Not A Number Test T800

Opcode Clock Cycles
29 F1 2 Minimum / 3 Maximum

Flags
Unaffected.

Scheduling
Atomic.

Pseudocode
```
C = B
B = A
IF (FA == NaN) THEN A = TRUE
ELSE A = FALSE // else FA is not a NaN
RoundMode = ToNearest
Iptr = Next Instruction
```

Description
Floating point Not a Number test tests whether FA is a NaN. If so, A is set true; otherwise, A is cleared to false. B is pushed into C and the previous value of A is pushed into B. The floating point register stack is unaffected.

FPNOTFINITE Floating Point Not Finite Test T800

Opcode Clock Cycles
29 F3 2

Flags
Unaffected.

Scheduling
Atomic.

Pseudocode
```
C = B
B = A
IF ((FA == Inf) OR (FA == NaN)) THEN A = TRUE
```

```
ELSE A = FALSE // else FA is not a NaN or infinity
RoundMode = ToNearest
Iptr = Next Instruction
```

Description
Floating point finite test tests whether FA is a NaN or an infinity. If so, A is set true; otherwise, A is cleared to false. B is pushed into C and the previous value of A is pushed into B. The floating point register stack is unaffected.

User Notes
This is the opposite of the FPORDERED test.

FPORDERED Floating Point Orderability T800

Opcode Clock Cycles
29 F2 3 Minimum / 9 Maximum

Flags
Unaffected.

Scheduling
Atomic.

Pseudocode
```
C = B
B = A
IF ((FA != Inf) AND (FA != NaN)) THEN A = TRUE
ELSE A = FALSE // else FA is an infinity or NaN
RoundMode = ToNearest
Iptr = Next Instruction
```

Description
Floating point orderability tests to see if FA and FB can be ordered, that is, they are not NaNs. A is set true if both FA and FB are not NaNs; otherwise, A is cleared to false. B is pushed into C and the previous value of A is pushed into B. The floating point register stack is unaffected.

User Notes
The IEEE says that NaNs are not comparable with anything, meaning that X compare Y is always false if either X or Y is a NaN. The IEEE comparisons for floating point numbers can be implemented via:
Unordered: fpordered; eqc 0
IEEE greater than fpordered; fpgt; and
IEEE equality fpordered; fpeq; and

FPORDERED is the opposite of FPNOTFINITE.

FPREMFIRST Floating Point Remainder First Step T800

Opcode Clock Cycles
28 FF 36 Typical / 46 Maximum

Flags
The FloatingPointError flag is in an intermediate state.

Scheduling
Atomic.

Pseudocode

```
FA = intermediate value in REM computation
FB = intermediate value in REM computation
FC = intermediate value in REM computation
C = B
B = A
A = intermediate value (used internally for looping)
FP_Error = intermediate value
RoundMode = ToNearest
Iptr = Next Instruction
```

Description
Floating point remainder first step initiates the first step in the code sequence necessary to compute a floating point remainder (or modulus). The integer register stack is used to compute intermediate boolean values, so it will be undefined (trashed) after the remainder code sequence. FC will also be used and undefined after computing the remainder. The evaluation performed is FB REM FA, leaving the result in FA. The code to implement the operation is:

```
fpremfirst
eqc 0
cj next
loop: fpremstep
cj loop
next:
```

User Notes
Computing a remainder produces an exact result so that neither FPREMFIRST nor FPREMSTEP require a rounding mode. FPREMSTEP also makes use of all the registers so that only the result will be left in FA; all other registers on the floating point unit and the integer unit (i.e., FB, FC, A, B, C) will be undefined after computing a remainder.

FPREMSTEP Floating Point Remainder Iteration T800

Opcode Clock Cycles
29 F0 32 Typical / 36 Maximum

Flags
The floating point error flag will be set if FA or FB is a NaN or infinity.

Scheduling
Atomic.

Pseudocode
```
IF (LAST STEP IN COMPUTING REMAINDER) THEN
IF ((FA == Inf) OR (FA == NaN) OR (FB == Inf) OR (FB == NaN))
THEN Set FP_Error
FA = FB REM FA // FB modulo FA
FB = undefined
FC = undefined
A = undefined
B = undefined
C = undefined
ELSE
FP_Error = intermediate value
FA = intermediate value in REM computation
FB = intermediate value in REM computation
FC = intermediate value in REM computation
C = B
B = A
A = intermediate value (used internally for looping)
RoundMode = ToNearest
```

```
Iptr = Next Instruction
```

Description
Floating point remainder iteration step is an intermediate step in the code sequence necessary to compute a floating point remainder. At the end FA will contain the remainder for FB REM FA. FC and A, B, C will be undefined.

User Notes
Computing a remainder produces an exact result so that neither FPREMFIRST nor FPREMSTEP require a rounding mode. FPREMSTEP also makes use of all the registers so that only the result will be left in FA; all other registers on the floating point unit and the integer unit (i.e., FB, FC, A, B, C) will be undefined after computing a remainder.

FPREV Floating Point Reverse T800

Opcode Clock Cycles
2A F4 1

Flags
The FloatingPointError flag is set if overflow occurs.

Scheduling
Atomic.

Pseudocode
```
SWAP(FA, FB) // FA = FB and FB = FA
FC = FC // FC is unchanged
RoundMode = ToNearest
Iptr = Next Instruction
```

Description
Fprev reverses the top of the floating point stack swapping FA and FP.

FPRTOI32 Floating Point Real to Int 32 T800

Opcode Clock Cycles
29 FD 7 Typical / 9 Maximum

Flags
The FloatingPointError flag is set if FA is a NaN or infinity or if the floating point value in FA cannot be expressed as a 32-bit integer (e.g., FA being either too big or too small).

Scheduling
Atomic.

Pseudocode
```
FA = (REAL 32)((INT)FA) // Round FA to nearest integer value
IF ((FA == Inf) OR (FA == NaN) OR (FA < MinINT32) OR (FA > MaxINT32))
THEN Set FP_Error // can’t express FA as an integer
RoundMode = ToNearest
Iptr = Next Instruction
```

Description
Real to Int 32 is a single instruction that is equivalent to the sequence: fpint ; fpuchki32. This is used to convert a floating point number to an integer value in the floating point format and check that the conversion was valid by setting FPError if it was not.

User Notes
The current rounding mode is used to round FA to the nearest integer value. FA will be of single-length size after this instruction.

FPSTNLDB Floating Point Store Non-Local Double T800

Opcode Clock Cycles
28 F4 3

Flags
Unaffected.

Scheduling
Atomic.

Pseudocode
```
// A points to two words in memory to receive the double-length
// floating point value held in FA. FA should contain a double-length
// value.
FA = FB
FB = FC
FC = Undefined
*A = (REAL 64)FA // FA should contain a double-length number
A = B
B = C
C = Undefined
RoundMode = ToNearest
Iptr = Next Instruction
```

Description
Floating point store non-local double pops the double floating point number in FA and stores it at the address pointed to by A. FB is popped into FA and FC into FB. B is popped into A and C into B.

User Notes
A points to two words in memory that will receive a double-length floating point number (from FA). *A receives the low order word, *(A + 4) receives the high order word. If FA does not contain a double-length (floating point) number, the results of this instruction are undefined.

FPSTNLI32 Floating Point Store Non-Local Integer 32 T800

Opcode Clock Cycles
29 FE 4

Flags
Unaffected.

Scheduling
Atomic.

Pseudocode
```
FA = FB
FB = FC
FC = Undefined
IF ((FA < MinINT32) OR (FA > MaxINT32))
THEN *A = Undefined
ELSE
*A = (INT)FA
```

```
A = B
B = C
C = Undefined
RoundMode = ToNearest
Iptr = Next Instruction
```

Description
Store non-local int32 truncates FA into a 32-bit integer value (FA can be double or single) and then stores that integer value at the address pointed to by A. B is popped into A and C into B.

User Notes
FA is truncated (not rounded) to an integer value.

FPSTNLSN Floating Point Store Non-Local Single T800

Opcode Clock Cycles
28 F8 2

Flags
Unaffected.

Scheduling
Atomic.

Pseudocode
```
// FA should contain a single-length value to store.
FA = FB
FB = FC
FC = Undefined
*A = (REAL32)FA
A = B
B = C
C = Undefined

RoundMode = ToNearest
Iptr = Next Instruction
```

Description
Floating point store non-local single pops the single floating point number in FA and stores it at the address pointed to by A. FB is popped into FA and FC into FB. B is popped into A and C into B.

FPSUB Floating Point Subtract T800

Opcode Clock Cycles
28 F9 6 Typical / 9 Maximum

Flags
The FloatingPointError flag is set if the subtraction underflows or either FA or FB is a NaN or infinity.

Scheduling
Atomic.

Pseudocode
```
FA = FB - FA
IF (Underflow OR (FA == Inf) OR (FA == NaN) OR (FB == Inf) OR (FB == NaN))
THEN Set FP_Error
```

```
FB = FC
FC = Undefined
RoundMode = ToNearest
Iptr = Next Instruction
```

Description
Floating point subtract subtracts FA from FB, leaving the result in FA while popping FC up into FB. The current rounding mode is used. Both FA and FB should be of the same format (either both single or both double).

FPTESTERR Test Floating Point Flag False T800

Opcode Clock Cycles
29 FC 2

Flags
The FloatingPointError flag is cleared by this instruction.

Scheduling
Atomic.

Pseudocode
```
C = B
B = A
IF (FP_Error is Clear) THEN A = TRUE
ELSE A = FALSE // ELSE FP_Error is set
Clear FP_Error
RoundMode = ToNearest
Iptr = Next Instruction
```

Description
Test floating error false and clear sets A to true (one) if FPError is clear or false (zero) when FPError is set. FPError is then cleared.

FPUABS Floating Point Absolute Value T800

Opcode Clock Cycles
28 F7 6 Typical / 9 Maximum

Flags
The FloatingPointError flag is set if FA is a NaN or infinity.

Scheduling
Atomic.

Pseudocode
```
// ABS(x) = absolute value of x
IF ((FA == NaN) OR (FA == Inf)) THEN Set FP_Error
FA = ABS(FA)
RoundMode = ToNearest
Iptr = Next Instruction
```

Description
Floating point absolute value replaces FA with the absolute value of FA. This is done by making the sign bit positive (cleared to zero). Note that the sign bit of a NaN will be cleared even though the absolute value of a NaN is meaningless. FPUABS will set FPError if FA is an infinity or a NaN.

FPUCHKI32 Floating Point Check in Range of Type Integer 32 T800

Opcode Clock Cycles
4E 2A FB 3 Minimum / 4 Maximum

Flags
The FloatingPointError flag is set if FA is outside the range of a 32-bit integer value.

Scheduling
Atomic.

Pseudocode
```
IF ((FA < MinINT32) OR (FA > MaxINT32)) THEN Set FP_Error
RoundMode = ToNearest
Iptr = Next Instruction
```

Description
Check in the range of type int 32 checks to see if the value in FA lies in the range of a 32-bit integer. If so, it will be possible to convert it to a 32-bit integer value. Both register stacks are unaffected by this instruction. FPError indicates whether the check was successful or not. If the value can be represented by a 32-bit integer, FPError is clear; otherwise, FPError is set.

FPUCHKI64 Floating Point Check in Range of Type Integer 64 T800

Opcode Clock Cycles
4F 2A FB 3 Minimum / 4 Maximum

Flags
The FloatingPointError flag is set if FA is outside the range of a 64-bit integer value.

Scheduling
Atomic.

Pseudocode
```
IF ((FA < MinINT64) OR (FA > MaxINT64)) THEN Set FP_Error
RoundMode = ToNearest
Iptr = Next Instruction
```

Description
Check in the range of type int 64 checks to see if the value in FA lies in the range of a 64-bit integer. If so, it will be possible to convert it to a 64-bit integer value. Both register stacks are unaffected by this instruction. FPError indicates whether the check was successful or not. If the value can be represented by a 64-bit integer, FPError is clear; otherwise, FPError is set.

FPUCLRERR Clear Floating Point Error T800

Opcode Clock Cycles
29 4C 2A FB 1

Flags
The FloatingPointError flag is cleared (i.e., set to zero).

Scheduling
Atomic.

Pseudocode

```
Clear FP_Error
RoundMode = ToNearest
Iptr = Next Instruction
```

Description

Clear floating point error clears FP_Error.

FPUDIVBY2 Floating Point Divide by Two T800

Opcode Clock Cycles

21 41 2A FB 6 Typical / 9 Maximum

Flags

The FloatingPointError flag is set if FA is a NaN or infinity.

Scheduling

Atomic.

Pseudocode

```
IF ((FA == NaN) or (FA == Inf)) THEN Set FP_Error
FA = FA / 2
RoundMode = ToNearest
Iptr = Next Instruction
```

Description

Floating divide by 2 divides FA by 2.0 and rounds the result as specified by the current rounding mode. The result is returned in FA; no other registers are affected.

FPUEXPDEC32 Floating Point Divide by 232 T800

Opcode Clock Cycles

49 2A FB 6 Typical / 9 Maximum

Flags

The FloatingPointError flag is set if FA is a NaN or infinity or if the result of the division is a NaN or infinity.

Scheduling

Atomic.

Pseudocode

```
IF ((FA == NaN) or (FA == Inf)) THEN Set FP_Error
FA = FA / (2**32)
IF ((FA == NaN) or (FA == Inf)) THEN Set FP_Error
RoundMode = ToNearest
Iptr = Next Instruction
```

Description

Floating divide by 232 divides FA by 232 and rounds the result as specified by the current rounding mode. The result is returned in FA; no other registers are affected.

FPUEXPINC32 Floating Point Multiply by 232 T800

Opcode Clock Cycles

4A 2A FB 6 Typical / 9 Maximum

Flags
The FloatingPointError flag is set if FA is a NaN or infinity or if the result of the multiplication is a NaN or infinity.

Scheduling
Atomic.

Pseudocode
```
IF ((FA == NaN) or (FA == Inf)) THEN Set FP_Error
FA = FA * (2**32)
IF ((FA == NaN) or (FA == Inf)) THEN Set FP_Error
RoundMode = ToNearest
Iptr = Next Instruction
```

Description
Floating multiply by 232 multiplies FA by 232 and rounds the result as specified by the current rounding mode. The result is returned in FA; no other registers are affected.

FPUMULBY2 Floating Point Multiply by Two T800

Opcode Clock Cycles
21 42 2A FB 6 Typical / 9 Maximum

Flags
The FloatingPointError flag is set if FA is a NaN or infinity.

Scheduling
Atomic.

Pseudocode
```
IF ((FA == NaN) or (FA == Inf)) THEN Set FP_Error
FA = FA * 2
RoundMode = ToNearest
Iptr = Next Instruction
```

Description
Floating multiply by two multiplies FA by 2.0 and rounds the result as specified by the current rounding mode. The result is returned in FA; no other registers are affected.

FPUNOROUND Floating Point Real 64 to Real 32 No Round T800

Opcode Clock Cycles
4D 2A FB 2

Flags
Unaffected.

Scheduling
Atomic.

Pseudocode
```
// FA should contain a double-length floating point number
// that is either normalized or zero.
FA = (REAL32) FA
RoundMode = ToNearest
Iptr = Next Instruction
```

Description
Real 64 to Real 32 without rounding converts a double-length floating point number to a single-length floating point number without rounding the mantissa. This only works for normalized numbers and zeroes and is intended to remove the possible introduction of double rounding errors when 64-bit integers are converted into single-length floats. The value to convert is in FA and no other registers are affected. The exponent bias of FA is changed, as well as the length (from single to double).

User Notes
FA should contain a double-length floating point number that is either normalized or zero.

FPUR32TOR64 Floating Point Real 32 to Real 64 T800

Opcode Clock Cycles
47 2A FB 3 Typical / 4 Maximum

Flags
The FloatingPointError flag is set if FA is a NaN or infinity.

Scheduling
Atomic.

Pseudocode
```
// FA should contain a single-length floating point number (to
// be converted to a double-length floating point number).
FA = (REAL64) FA
IF ((FA == NaN) OR (FA == Inf)) THEN Set FP_Error
RoundMode = ToNearest
Iptr = Next Instruction
```

Description
Real 32 to Real 64 <FONT SIZE

GAJW General Adjust Workspace

Opcode Clock Cycles
23 FC 2

Flags
Unaffected.

Scheduling
Atomic.

Pseudocode
```
// Note: If A is not word-aligned, its lower two bits are masked
// off so that it is word-aligned.
SWAP(A, Wptr)
Iptr = Next Instruction
```

Description
General adjust workspace exchanges the contents of the workspace pointer and A. A should be word-aligned; B and C are unaffected.

GCALL General Call

Opcode Clock Cycles
F6 3

Flags
Unaffected.

Scheduling
Atomic.

Pseudocode
```
SWAP(A, Iptr)
```

Description
General call exchanges the contents of the instruction pointer and A; execution then continues at the new address formerly contained in A. This can be used to generate a subroutine call at run time by:

1. Build a stack (workspace) frame like the one the CALL instruction uses.
2. Load A with the address of the subroutine.
3. Execute GCALL.

B and C are unaffected by the *general call* instruction.

GT Greater Than

Opcode Clock Cycles
F9 2

Flags
Unaffected.

Scheduling

Atomic.

Pseudocode

```
IF (B > A)
THEN A = TRUE
ELSE // B <= A
A = FALSE
B = C
C = Undefined
Iptr = Next Instruction
```

Description

Greater than performs an unsigned comparison of A and B. If B is strictly greater than A, A is set to true (one). Otherwise, A is set to false (zero).

IN Input Message

Opcode Clock Cycles

F7 2W + 19

Flags

Unaffected.

Scheduling

The current process is descheduled until a message is received.

Pseudocode

```
FOR (i = 0; i < A; i++)
// wait for next byte to arrive over channel B
*(C+i) = B // store newly arrived byte at memory
// block starting at C
A = Undefined
B = Undefined
C = Undefined
Iptr = Next Instruction
```

Description

Input message inputs a message with the length (in bytes) of A from the channel pointed to by the address in B, leaving the message at the memory pointed to by C. A, B, and C are undefined after this instruction's execution.

J Jump

Opcode Clock Cycles

0X 3

Flags

Unaffected

Scheduling

This instruction can cause the current process to be descheduled.

Pseudocode

```
Iptr = Next Instruction + Oreg
Oreg = 0
```

Description

Jump transfers the flow of execution. Jumps are instruction pointer relative (i.e., the operand is added to the instruction pointer to calculate the location of the next instruction to be executed). This implies that *j 0* is a "no-op" (or no operation) instruction. However, as of this writing, Inmos plans to use *j 0* as a debugging-breakpoint instruction in future chip revisions. So, depending on the version of transputer you are using, *j 0* could generate an interrupt.

LADD Long Add

Opcode Clock Cycles
21 F6 2

Flags
Error is set on overflow.

Scheduling
Atomic.

Pseudocode
```
A = B + A + (C AND 1) // add lowest bit of C as carry bit
B = Undefined
C = Undefined
IF Overflow THEN Set Error
Iptr = Next Instruction
```

Description
Long add adds A and B and the least significant bit of C (carry bit), leaving the result in A. B and C are popped off the stack. Arithmetic overflow is checked.

User Notes
This instruction performs checked arithmetic and can set the Error flag.

LB Load Byte

Opcode Clock Cycles
F1 5

Flags
Unaffected.

Scheduling
Atomic.

Pseudocode
```
A = *A // load the byte pointed to by A into A
Iptr = Next Instruction
```

Description
Load byte loads an unsigned byte from the address given in A. B and C are not affected.

LDC Load Constant

Opcode Clock Cycles
4X 1

Flags
Unaffected.

Scheduling
Atomic.

Pseudocode
```
C = B
B = A
A = Oreg // push operand onto integer register stack
Oreg = 0
Iptr = Next Instruction
```

Description
Load constant loads the lower nibble of its operation code into the lower nibble of the operand register. Then the instruction loads the operand register into A, pushing the register stack first (C receives B, B receives A).

Most assemblers automatically take care of the prefix bytes so that you can just use operands greater than one nibble. For example, *ldc 0x43* would generate *pfix 4* (*ldc 3* in most assemblers).

LDIFF Long Difference

Opcode Clock Cycles
24 FF 2

Flags
Unaffected.

Scheduling
Atomic.

Pseudocode
```
A = B - A - (C & 1)
B = borrow from B - A - (C & 1)
C = Undefined
Iptr = Next Instruction
```

Description
Long diff is designed to be an intermediate instruction in multi-precision arithmetic. LDIFF subtracts A from B and also subtracts the least significant bit of C (carry bit) from B, leaving the least significant word of the result in A and the borrow bit in B. C is popped.

LDINF Load Single FP Infinity T414

Opcode Clock Cycles
27 F1 1

Flags
Unaffected.

Scheduling
Atomic.

Pseudocode
```
C = B
B = A
A = 0x7F800000 // push single-length infinity onto register stack
Iptr = Next Instruction
```

Description
Load single-length infinity loads the single-length floating point number for positive infinity into the A register. This is the hexadecimal number 7F800000. The previous value of B is pushed into C and the previous value of A is pushed into B.

LDIV Long Divide

Opcode Clock Cycles
21 FA 35

Flags
Error is set on division by zero.

Scheduling
Atomic.

Pseudocode
```
IF (C >= A) THEN Set Error
A = (C * 2**32 + B) / A
B = (C * 2**32 + B) REM A
C = Undefined
Iptr = Next Instruction
```

Description
Long divide divides the double length value held in B and C (the most significant word in C) by the single-length unsigned value held in A. The result is left in A, the remainder in B. Overflow occurs if the result cannot be represented as a single word value and causes the error flag to be set.

LDL Load Local

Opcode Clock Cycles
7X 2

Flags
Unaffected.

Scheduling
Atomic.

Pseudocode
```
C = B
B = A
A = Wptr[Oreg] // push word at Wptr + 4*Oreg onto register stack
Oreg = 0
Iptr = Next Instruction
```

Description
Load local loads the value at the word-offset of the operand register into A (B is pushed into C and A is pushed into B). This means that A receives the contents of the 32-bit value located at the workspace-pointer plus four times the operand register.

LDLP Load Local Pointer

Opcode Clock Cycles
1X 1

Flags
Unaffected.

Scheduling
Atomic.

Pseudocode
```
C = B
B = A
A = Wptr + 4*Oreg // push the address of the Oreg-th element
Oreg = 0 // of the workspace on the register stack
Iptr = Next Instruction
```

Description
Load local pointer instruction loads the workspace pointer, plus four times the operand register, into A. This gives you the address of a value on the workspace. In this case, the register stack is pushed first (i.e., C receives B, B receives A).

LDNL Load Non-Local

Opcode Clock Cycles
3X 1

Flags
Unaffected.

Scheduling
Atomic.

Pseudocode
```
// Note: A should be word-aligned when using this instruction.
A = A[Oreg] // A is replaced by the word at A + Oreg*4
Oreg = 0
Iptr = Next Instruction
```

Description
Load non-local loads the value pointed at by the address calculated by the workspace pointer, plus four times the operand register, into A. A common technique to load a value from an absolute address is:

1. Use the *ldc* address to load the address into the A register.
2. Use *ldnl 0* to load the contents at that address. This affects only A; since the stack is *not* pushed, B and C remain unchanged.

LDNLP Load Non-Local Pointer

Opcode Clock Cycles
5X 1

Flags
Unaffected.

Scheduling
Atomic.

Pseudocode

```
A = A + 4*Oreg // A is replaced by the address
Oreg = 0 // of the word at A + 4*Oreg
Iptr = Next Instruction
```

Description
Load non-local pointer loads A plus four times the operand. B and C are unaffected. This is useful if you first load A with a base address and then use LDNLP with an offset from the base address, to load the effective (or desired) address.

LDPI Load Pointer to Instruction

Opcode Clock Cycles
21 FB 2

Flags
Unaffected.

Scheduling
Atomic.

Pseudocode
```
A = A + (Address of Next Instruction)
Iptr = Next Instruction
```

Description
Load pointer to instruction adds the current value of the instruction pointer to A.

User Notes
The instruction pointer contains the address of the instruction after the LDPI. This is useful for computing relative addresses.

LDPRI Load Current Priority

Opcode Clock Cycles
21 FE 1

Flags
Unaffected.

Scheduling
Atomic.

Pseudocode
```
C = B
B = A
A = Current Priority Level // priority is either 0 for high or 1 for low
Iptr = Next Instruction
```

Description
Load current priority loads the current process priority into A. Here a value of one means that the process has a low priority, and zero means the process has a high priority.

LDTIMER Load Timer

Opcode Clock Cycles
22 F2 2

Flags
Unaffected.

Scheduling
Atomic.

Pseudocode
```
// Set A to the value of the current priority timer
C = B
B = A
IF (priority == HIGH)
THEN A = ClockReg0
ELSE // priority == LOW
A = ClockReg1
Iptr = Next Instruction
```

Description
Load timer loads A with the value of the current priority timer.

LEND Loop End

Opcode Clock Cycles
22 F1 10 (loop)
5 (exit)

Flags
Unaffected.

Scheduling
This instruction can cause the current process to be descheduled.

Pseudocode
```
// B points to a two-word control block in memory.
// The first word, B[0], contains a control variable that will be
// incremented each iteration. The iteration count is held in B[1]
// and is decremented each iteration.
IF (B[1] > 1)
THEN
B[0] = B[0] + 1 // increment control variable
B[1] = B[1] - 1 // decrement iteration count
Iptr = Next Instruction - A // loop backwards
ELSE
Iptr = Next Instruction // end of loop, continue on
```

Description
Loop end is used to implement looping constructs. Before executing LEND, B contains the address of a two (32-bit) word control-block in memory. The second word of this control block (pointed at by B+4) contains the loop count, which is decremented each time LEND is executed. If this value is decremented to zero or less, the loop is considered over and execution continues with the instruction after the LEND. However, if the value is still positive, the first word (pointed at by B) is incremented and the instruction pointer is decremented by the contents of A. This means that A must contain the offset of where to jump to if the LEND is taken. Thus, prior to LEND, A must be loaded with this offset value. In the case of relative addressing, A would be loaded with the difference of two labels. Note that C is undefined after this instruction is executed.

LMUL Long Multiply

Opcode Clock Cycles
23 F1 33

Flags
Unaffected.

Scheduling
Atomic.

Pseudocode
```
A = low word of (B * A) + C
B = high word of (B * A) + C
C = Undefined
Iptr = Next Instruction
```

Description
Long multiply multiplies A and B as two single-word, unsigned operands. It then adds the single word carry-operand in C to form a double-length unsigned result with the most significant word left in B and the least significant word left in A. C is popped.

User Notes
This instruction performs unchecked arithmetic and does not set the Error flag.

LSHL Long Shift Left

Opcode Clock Cycles
23 F6 (n + 3) for n < 32
(n - 28) n >= 32

Flags
Unaffected.

Scheduling
Atomic.

Pseudocode
```
// Shift Left operates on registers B and C as though both were
// one continuous register with the most significant word in C.
// The result of the shift is placed in registers A and B with the
// most significant word in B. The number of bit positions to
// to shift is in A.
AB = BC << A // shift BC left A places, put result in AB
C = Undefined
Iptr = Next Instruction
```

Description
Long shift left shifts the double-length value held in B and C (the most significant word in C) left by the number of bit positions equal to the value held in A. Vacated bit positions are filled with zero bits. The result is left in A and B (the most significant word in B). C is popped.

User Notes
Warning: The transputer shift instruction does not truncate the number of bits to shift to 64 or less (although more shifts than that are meaningless). Instead, the transputer performs the extra shifts. This means that for a large value of shifts, the transputer can "hang" for up to several minutes while it performs the (unnecessary) shifts.

LSHR Long Shift Right

Opcode Clock Cycles
23 F5 (n + 3) n < 32
(n - 28) n >= 32

Flags
Unaffected.

Scheduling
Atomic.

Pseudocode
```
// Shift Right operates on registers B and C as if both were
// one continuous register with the most significant word in C.
// The result of the shift if placed in registers A and B with the
// most significant word in B. The number of bit positions to
// to shift if in A.
AB = BC >> A // shift BC right A places, put result in AB
C = Undefined
Iptr = Next Instruction
```

Description
Long shift right shifts the double-length value held in B and C (the most significant word in C) right by the number of bit positions equal to the value held in A. Vacated bit positions are filled with zero bits. The result is left in A and B (the most significant word in B). C is popped.

User Notes
Warning: The transputer shift instruction does not truncate the number of bits to shift to 64 or less (although more shifts than that are meaningless). Instead, the transputer performs the extra shifts. This means that for a large value of shifts, the transputer can "hang" for up to several minutes while it performs the (unnecessary) shifts.

LSUB Long Subtract

Opcode Clock Cycles
23 F8 2

Flags
Error is set on overflow.

Scheduling
Atomic.

Pseudocode
```
A = B - A - (C AND 1) // subtract with lower bit of C as carry
B = Undefined
C = Undefined
IF Overflow THEN Set Error
Iptr = Next Instruction
```

Description
Long subtract subtracts A from B, and also subtracts the least significant bit of C (carry bit) from B, leaving the result in A. B and C are popped off the stack. Arithmetic overflow is checked.

User Notes
This instruction performs checked arithmetic and can set the Error flag.

LSUM Long Sum

Opcode Clock Cycles
23 F7 2

Flags
Unaffected.

Scheduling
Atomic.

Pseudocode
```
A = B + A + (C & 1) // sum using lower bit of C as carry
B = carry from B + A + (C & 1)
C = Undefined
Iptr = Next Instruction
```

Description
Long sum is designed to be an intermediate instruction in multi-precision arithmetic. LSUM adds A, B, and the least significant bit of C (carry bit), leaving the least significant word of the result in A and the most significant carry bit in B. C is popped.

User Notes
This instruction performs unchecked arithmetic and does not set the Error flag.

MINT Minimum Integer

Opcode Clock Cycles
24 F2 1

Flags
Unaffected.

Scheduling
Atomic.

Pseudocode
```
C = B
B = A
A = 0x80000000 // push minimum integer onto the register stack
Iptr = Next Instruction
```

Description
Minimum integer loads A with the hexadecimal value 0x80000000 (Inmos also refers to this as NotProcess.p). This is useful because the transputer's on-chip RAM starts at address 0x80000000. The value corresponds to -(2**31).

MOVE Move Message

Opcode Clock Cycles
24 FA 2W + 8

Flags
Unaffected.

Scheduling

Atomic.

Pseudocode

```
// Provided the two blocks do not overlap meaning:
// IF ((block B does not start inside block C) AND
// (block C does not start inside block B))
IF ( NOT ((B < (C + A)) AND (B >= C)) AND
NOT ((C < (B + A)) AND (C >= B))
// THEN copy A bytes starting at C to block at B
THEN
FOR (i = 1 ; i <= A ; i++)
*(B + i) = *(C + i)
ELSE undefined results

A = Undefined
B = Undefined
C = Undefined
Iptr = Next Instruction
```

Description

Move message moves a block of memory by copying the number of bytes held in A from the starting source-address held in C to the starting destination address held in B.

User Notes

The two address regions must not overlap. Also, the number of bytes to move in A is a signed value and must not be negative.

MOVE2DALL 2D Block Copy T800

Opcode Clock Cycles

25 FC (2p + 23) r

Flags

Unaffected.

Scheduling

Atomic.

Pseudocode

```
// Provided the two blocks do NOT overlap.
destination block at B = source block at C
A = Undefined
B = Undefined
C = Undefined
Iptr = Next Instruction
```

Description

The *2D block copy* instruction copies an entire block of length-rows (each of which is width bytes) from the source to the destination. This instruction performs the actual move of one block to the destination:

ï C should contain the address of the source block.

ï B should contain the address for the destination block.

ï A should contain the width (in bytes) of each row to be copied.

The contents of A, B, and C are undefined after this operation.

User Notes
This instruction requires the use of MOVE2DINIT first (to initialize the internal state of the transputer for the move).

MOVE2DINIT Initialize Data for 2D Block Move T800

Opcode Clock Cycles
25 FB 8

Flags
Unaffected.

Scheduling
Atomic.

Pseudocode
```
// Set up the first three parameters for a two-dimensional block move.
A = Undefined
B = Undefined
C = Undefined
Iptr = Next Instruction
```

Description
The *initialize data for 2D block move* instruction sets up three of the six necessary parameters for the other 2D move instructions. Since some of the 2D move instructions require six parameters, and the transputer has only three registers, a two-step instruction sequence is necessary to perform the 2D move operation:

ï C should contain the stride of the source array.

ï B should contain the stride of the destination array.

ï A should contain the length or number of rows to be copied.

The contents of A, B, and C are undefined after this operation.

MOVE2DNONZERO 2D Block Copy Non-Zero Bytes T800

Opcode Clock Cycles
25 FD (2p+23)r

Flags
Unaffected.

Scheduling
Atomic.

Pseudocode
```
// Provided the two blocks do NOT overlap.
destination block at B = nonzero bytes of source block at C
A = Undefined
B = Undefined
C = Undefined
Iptr = Next Instruction
```

Description
The *2D block company non-zero bytes* instruction copies only the non-zero bytes in the source block to the destination block. As a result, it leaves unchanged the bytes corresponding to zeroes in the destination block. Inmos suggests using this instruction to overlay a non-rectangular picture onto another picture (e.g., copy the picture and not the background on top of another picture). This is a *byte* copy and not a *bit* copy, so any picture edge would have to terminate on a byte boundary:

ï C should contain the address of the source block.

ï B should contain the address for the destination block.

ï A should contain the width (in bytes) of each row to be copied.

The contents of A, B, and C are undefined after this operation.

User Notes
This instruction requires the use of MOVE2DINIT first (to initialize the internal state of the transputer for the move).

MOVE2DZERO 2D Block Copy Zero Bytes T800

Opcode Clock Cycles
25 FE (2 p + 23) r

Flags
Unaffected.

Scheduling
Atomic.

Pseudocode
```
// Provided the two blocks do NOT overlap.
destination block at B = zero bytes of source block at C
A = Undefined
B = Undefined
C = Undefined
Iptr = Next Instruction
```

Description
The *2D block copy zero bytes* instruction copies the zero bytes in the source block to the destination block. As a result, it leaves unchanged the bytes corresponding to nonzero bytes in the source. Inmos suggests that you use this instruction to mask out a non-rectangular shape from a picture:

ï C should contain the address of the source block.

ï B should contain the address for the destination block.

ï A should contain the width (in bytes) of each row to be copied.

The contents of A, B, and C are undefined after this operation.

User Notes
This instruction requires the use of MOVE2DINIT first (to initialize the internal state of the transputer for the move).

MUL Multiply

Opcode Clock Cycles
25 F3 38

Flags
Error is set on overflow.

Scheduling
Atomic.

Pseudocode
```
A = B * A
IF (Overflow) THEN Set Error
B = C
C = Undefined
```

Description
Multiply multiplies B and A, leaving the result in A. C is popped into B. The Error flag is set if overflow occurs.

NFIX Negative Prefix

Opcode Clock Cycles
6X 1

Flags
Unaffected.

Scheduling
Atomic.

Pseudocode
```
Oreg = (bitwise NOT of Oreg) << 4
Iptr = Next Instruction
```

Description
Negative prefix loads the lower nibble of its operation code into the operand register. It then inverts all the bits in the operand register and shifts the register to the left by four places. You can use this to generate negative operands for other instructions. NFIX does not affect the A, B, C register stack. You should note that, except for PFIX and NFIX, all other instructions clear the operand register to zero after using its value.

NORM Normalize

Opcode Clock Cycles
21 F9 (n + 5) norm < 32
(n-26) norm >= 32
3 if norm = 64

Flags
Unaffected.

Scheduling
Atomic.

Pseudocode
```
C = 0
WHILE (NOT(B AND 0x80000000)
BA << 1 // shift left double word BA one
C = C + 1 // increment index
IF (C == 64) // in case BA is initially zero
THEN BREAK; // then exit from while loop
Iptr = Next Instruction
```

Description
Normalize normalizes the unsigned double-length value held in A and B (the most significant word in B). This value is shifted to the left until its most significant bit is one. The shifted double-length value remains in A and B, while C receives the number of left shifts performed. If the double-length value is initially zero, C is set to twice the number of bits in a word (64).

NOT Bitwise Not

Opcode Clock Cycles
23 F2 1

Flags
Unaffected.

Scheduling
Atomic.

Pseudocode
```
A = bitwise NOT A
Iptr = Next Instruction
```

Description
Not performs a bitwise logical *not* on A. B and C are unaffected. Example: If A = 0x80000000 prior to execution, then after executing NOT A would equal 0x7FFFFFFF.

OPR Operate

Opcode Clock Cycles
FX varies

Flags
Unaffected.

Scheduling
Atomic.

Pseudocode
```
Execute Oreg // interpret the contents of Oreg as an instruction
Oreg = 0
```

Description
Operate is the gateway to all the *indirect* instructions. This instruction takes the value in the operand register and executes it. Since the operand register is 32 bits long, this gives the transputer enough room for a very large instruction set. In reality, there are 16 one-byte and over 90 two-byte, *indirect* instructions (on the T800, there are 49 additional floating point instructions).

OR Logical Or

Opcode Clock Cycles

24 FB 1

Flags
Unaffected.

Scheduling
Atomic.

Pseudocode
```
A = B | A // A = bitwise OR of B and A
B = C
C = Undefined
Iptr = Next Instruction
```

Description
Or performs a bitwise logical *or* operation between A and B and then leaves the result in A. C is popped into B.

OUT Output Message

Opcode Clock Cycles
FB 2W + 19

Flags
Unaffected.

Scheduling
This instruction can cause the current process to be descheduled.

Pseudocode
```
// output a message of A bytes to channel at B from memory at C
A = Undefined
B = Undefined
C = Undefined
Iptr = Next Instruction
```

Description
Output message outputs a message equal in length to the number of bytes in A to the channel pointed to by the address in B from the memory pointed at by the address in C. A, B, and C are undefined after this instruction's execution.

OUTBYTE Output Byte

Opcode Clock Cycles
FE 23

Flags
Unaffected.

Scheduling
Atomic.

Pseudocode
```
// output byte in A out channel pointed to by B
A = Undefined
B = Undefined
C = Undefined
```

```
Wptr[0] = Undefined // top of workspace is lost using OUTBYTE
Iptr = Next Instruction
```

Description
Output byte outputs the byte in the lower eight bits of A to the channel pointed to by B.

ï This instruction, like OUTWORD, uses workspace location zero to save the byte before sending it out.

ï A, B, and C are undefined after this instruction’s execution.

ï The value at workspace location zero is also undefined after this instruction.

OUTWORD Output Word

Opcode Clock Cycles
FF 23

Flags
Unaffected.

Scheduling
Atomic.

Pseudocode
```
// output word in A out channel pointed to by B
A = Undefined
B = Undefined
C = Undefined
Wptr[0] = Undefined // top of workspace is lost using OUTWORD
Iptr = Next Instruction
```

Description
Output word outputs the 32-bit word contained in A to the channel pointed to by B.

ï This instruction uses workspace-location zero to save the word before sending it out the channel pointed at by B.

ï A, B, C are undefined after this instruction’s execution.

ï The value at workspace location zero is also undefined after this instruction.

PFIX Prefix

Opcode Clock Cycles
2X 1

Flags
Unaffected.

Scheduling
Atomic.

Pseudocode
```
Oreg = Oreg << 4
Iptr = Next Instruction
```

Description
prefix loads the lower nibble of its operation code into the operand register and then shifts it to the left four places. Multiple *prefix* instructions are used to load values in the operand register; then a *non-prefix* instruction can use the contents of the operand register. This is how the transputer loads values greater than 15 into registers (the maximum a nibble can hold is 15). PFIX does not affect the A, B, C register stack. You might note that, except for PFIX and NFIX, all other instructions clear the operand register to zero after using its value.

POSTNORMSN Post-Normalize Correction T414

Opcode Clock Cycles
26 FC 5 Typical / 30 Maximum

Flags
Unaffected.

Scheduling
Atomic.

Pseudocode
```
// Perform post-normalization correction on floating point number
// where normalized fraction is in B (high word) and A (guard word);
// normalized shift length is in C and exponent is in Wptr[0].
A = post-normalized guard word
B = post-normalized fraction word
C = post-normalized exponent
Iptr = Next Instruction
```

Description
Post-normalize single-length floating point number is intended to be used after the NORM instruction to provide normalization correction. This instructions takes the normalized fraction (mantissa) in B (high word) and A (guard word) with the normalizing shift length in C and exponent in Workspace location zero and returns the post-normalized or corrected guard word in A, the post-normalized fraction word in B, and the post-normalized exponent in C.

PROD Product

Opcode Clock Cycles
F8 b + 4

Flags
Unaffected.

Scheduling
Atomic.

Pseudocode
```
A = B * A
B = C
C = Undefined
Iptr = Next Instruction
```

Description
Product multiplies B and A, leaving the result in A without checking for overflow or carry. C is popped into B.

REM Remainder

Opcode Clock Cycles
21 FF 37

Flags
Unaffected.

Scheduling
Atomic.

Pseudocode
```
IF ((A == 0) OR ((A == -1) AND (B == -2**31)))
THEN
Set Error
A = Undefined
ELSE
A = B REM A // remainder - B modulo A
B = C
C = Undefined
Iptr = Next Instruction
```

Description
Remainder leaves the remainder of B divided by A in A. Again, the result is undefined if A is zero. C is popped into B.

RESETCH Reset Channel

Opcode Clock Cycles
21 F2 3

Flags
Unaffected.

Scheduling
Atomic.

Pseudocode
```
// If A points to a link channel then the link hardware is reset.
A = A[0] // A is replaced by the word pointed to by A
A[0] = NotProcess.p // NotProcess.p = -1 (constant)
Iptr = Next Instruction
```

Description
Reset channel resets the channel pointed to by the A register. If A points to a link channel, the link hardware is reset. The channel is reinitialized to *NotProcess.p.*

RESETCH is used if communication that was started breaks down between two processes. In internal channels, like in the case where there is communication between processes on the same transputer, communication is guaranteed to finish as the data transfer is achieved by just copying a block of memory. However, in communication between transputers over links, if one transputer fails, a process could block indefinitely as it waits to communicate to the failed transputer. RESETCH provides the ability to reset the channel and restart the communication. The A, B, C register stack is unaffected by this instruction.

User Notes
This instruction will allow programs to check channels for "timing out" (not achieving communication) and reset the channel for future use (or retries).

RET Return

Opcode Clock Cycles
22 F0 5

Flags
Unaffected.

Scheduling
Atomic.

Pseudocode
```
Iptr = Wptr[0] // Iptr is taken off the workspace stack
Wptr = Wptr + 16 // 4 words = 16 bytes
```

Description
Return returns the execution flow from a subroutine back to the calling thread of execution. Prior to a RET instruction, any workspace claimed by the subroutine should be deallocated. The RET instruction increments the workspace pointer by four words (recall that CALL pushed four items on the workspace and decremented the workspace pointer by four).
The A, B, and C registers are unaffected by RET.

REV Reverse

Opcode Clock Cycles
F0 1

Flags
Unaffected.

Scheduling
Atomic.

Pseudocode
```
SWAP(A, B)
Iptr = Next Instruction
```

Description
Reverse swaps the contents of A and B.

ROUNDSN Round Single FP Number T414

Opcode Clock Cycles
26 FD 12 Typical / 15 Maximum

Flags
Unaffected.

Scheduling
Atomic.

Pseudocode
```
// Initially, A is guard word, B is fraction, C is exponent
A = rounded and packed floating point number
B = Undefined
C = Undefined
Iptr = Next Instruction
```

Description
Round single-length floating point number encodes initial values of C containing an exponent, B containing the fraction or mantissa, and A containing the guard word (a word extending the precision of the fraction to ensure) into a single-length, appropiately rounded floating point number (returned in A). B and C are undefined after this instruction.

User Notes
The guard word is a word extending the precision of the fraction to ensure an appropriately rounded floating point number. Essentially, the guard word is an extension of the normal word containing the fraction. The extra bits of precision makes rounding more accurate.

RUNP Run Process

Opcode Clock Cycles
23 F9 10

Flags
Unaffected.

Scheduling
Atomic.

Pseudocode
```
// Add process with descriptor A to appropiate process queue
// The current process is descheduled and the new process starts.
DESCHEDULE CURRENT PROCESS
Wptr = A & 0xFFFFFFFC // mask off two lower bits to word-align
Iptr = A[-1]
START NEW PROCESS
```

Description
Run process starts an already existing, but stopped, process. A should contain the workspace descriptor (a workspace location at which location -1 contains the instruction pointer for the process, that is, the word prior to the address of the workspace pointer contains the address to resume execution at). Note that the low bit of A denotes the priority for running the process 1 for low and 0 for high.

SAVEH Save High Priority Queue Registers

Opcode Clock Cycles
23 FE 4

Flags
Unaffected.

Scheduling
Atomic.

Pseudocode
```
A[0] = FPtrReg0
A[1] = BPtrReg0
A = B
B = C
C = Undefined
Iptr = Next Instruction
```

Description
Save high priority queue registers stores the high priority process queue pointers to the address pointed to by A.

Location A receives the head pointer, and location A+4 receives the tail pointer. B is popped into A and C is popped into B.

SAVEL Save Low Priority Queue Registers

Opcode Clock Cycles
23 FD 4

Flags
Unaffected.

Scheduling
Atomic.

Pseudocode
```
A[0] = FPtrReg1
A[1] = BPtrReg1
A = B
B = C
C = Undefined
Iptr = Next Instruction
```

Description
Save low priority queue registers stores the low priority process queue pointers to the address pointed to by A. Location A receives the head pointer and location A+4 receives the tail pointer. B is popped into A and C is popped into B.

SB Store Byte

Opcode Clock Cycles
23 FB 5

Flags
Unaffected.

Scheduling
Atomic.

Pseudocode
```
*A = (B & 0xFF) // byte in B stored at A
A = C
B = Undefined
C = Undefined
Iptr = Next Instruction
```

Description
Store byte stores the byte in B at the address given in A. The higher (than one byte) order of B are ignored. B and C are popped, with A containing the prior value of C.

SETERR Set Error Flag

Opcode Clock Cycles
21 F0 1

Flags
Error is set to one.

Scheduling
Atomic.

Pseudocode
```
Set Error
Iptr = Next Instruction
```

Description
Set Error sets the Error flag. The register stack is not affected.

User Notes
If *HaltOnError* is set prior to the *set Error* instruction, and the Error flag was zero (meaning false), this instruction will cause all processes on the transputer to halt.

SETHALTERR Set HaltOnError Flag

Opcode Clock Cycles
25 F8 1

Flags
HaltOnError is set to one.

Scheduling
Atomic.

Pseudocode
```
Set HaltOnError
```

Description
Set HaltOnError sets the *HaltOnError* flag (sets it to one, meaning true).

SHL Shift Left

Opcode Clock Cycles
24 F1 n + 2

Flags
Unaffected.

Scheduling
Atomic.

Pseudocode
```
A = B << A
B = C
C = Undefined
Iptr = Next Instruction
```

Description
Shift left shifts the number in B to the left by the number of bits specified in A. Vacated bit positions are filled with zero bits. The result is left in A and C is popped into B. In the case where A is zero, no shift is performed and B is simply popped into A.

User Notes

Warning: The SHL instruction takes an amount of time to execute that is proportional to the value in the A register (one cycle for every bit shifted plus some overhead). The transputer shift instruction does not truncate the number of bits to shift to 32 or less (although more shifts than that are meaningless). Instead, the transputer performs the extra shifts. This means that for a large value of shifts the transputer can "hang" for up to several minutes while it performs the (unnecessary) shifts.

SHR Shift Right

Opcode Clock Cycles
24 F0 n + 2

Flags
Unaffected.

Scheduling
Atomic.

Pseudocode
```
A = B >> A
B = C
C = Undefined
Iptr = Next Instruction
```

Description
Shift right shifts the number in B to the right by the number of bits specified in the A. The shift is unsigned, so vacated bit positions are filled with zero bits. The result is left in A, and B is popped into C.

User Notes
Warning: The SHR instruction takes an amount of time to execute that is proportional to the value in the A register (one cycle for every bit shifted plus some overhead). The transputer shift instruction does not truncate the number of bits to shift to 32 or less (although more shifts than that are meaningless). Instead, the transputer performs the extra shifts. This means that for a large value of shifts the transputer can "hang" for up to several minutes while it performs the (unnecessary) shifts.

STARTP Start Process

Opcode Clock Cycles
FD 12

Flags
Unaffected.

Scheduling
This instruction will cause the current process to be descheduled.

Pseudocode
```
// Add process with workspace A and Iptr at offset B bytes
// from the next instruction to the current process priority queue.
// The current process is descheduled and the new process starts.
DESCHEDULE CURRENT PROCESS
Wptr = A // A must be word-aligned
Iptr = Next Instruction + B
START NEW PROCESS
```

Description
Start process starts a new concurrent process at the same priority as the current process. B should contain the relative

address (offset) from the end of the current instruction to the first instruction of the new process. A contains the workspace pointer for the new process.

STHB Store High Priority Back Pointer

Opcode Clock Cycles
25 F0 1

Flags
Unaffected.

Scheduling
Atomic.

Pseudocode
```
BPtrReg0 = A
A = B
B = C
C = Undefined
Iptr = Next Instruction
```

Description
Store high priority back pointer initializes the high priority process queue-tail pointer with the value in A. B is popped into A and C is popped into B.

STHF Store High Priority Front Pointer

Opcode Clock Cycles
21 F8 1

Flags
Unaffected.

Scheduling
Atomic.

Pseudocode
```
FptrReg0 = A
A = B
B = C
C = Undefined
Iptr = Next Instruction
```

Description
Store high priority front pointer initializes the high priority process queue-head pointer with the value in A. B is popped into A and C is popped into B.

STL Store Local

Opcode Clock Cycles
DX 1

Flags
Unaffected.

Scheduling
Atomic.

Pseudocode
```
Wptr[Oreg] = A
A = B
B = C
C = Undefined
Oreg = 0
Iptr = Next Instruction
```

Description
Store local stores the contents of A at the address calculated by adding the workspace pointer and four times the operand register. In other words, the operand contains the 32-bit word offset from the base address of the workspace. B is inserted into A and C is inserted into B. This is the companion to the *load local* instruction.

STLB Store Low Priority Back Pointer

Opcode Clock Cycles
21 F7 1

Flags
Unaffected.

Scheduling
Atomic.

Pseudocode
```
BPtrReg1 = A
A = B
B = C
C = Undefined
Iptr = Next Instruction
```

Description
Store low priority back pointer initializes the low priority process queue-tail pointer with the value in A. B is popped into A and C is popped into B.

STLF Store Low Priority Front Pointer

Opcode Clock Cycles
21 FC 1

Flags
Unaffected.

Scheduling
Atomic.

Pseudocode
```
FPtrReg1 = A
A = B
B = C
C = Undefined
Iptr = Next Instruction
```

Description
Store low priority front pointer initializes the low priority process queue-head pointer with the value in A. B is popped into A and C is popped into B.

STNL Store Non-Local

Opcode Clock Cycles
EX 2

Flags
Unaffected.

Scheduling
Atomic.

Pseudocode
```
A[Oreg] = B
A = C
B = Undefined
C = Undefined
Oreg = 0
Iptr = Next Instruction
```

Description
Store nonlocal stores the contents of B at the address calculated by adding A and four times the contents of the operand register. Both A and B are popped and C falls into A. This is the companion instruction to the *load nonlocal* instruction, and you can access any memory location in a similar fashion. To store a value to an absolute memory address:

1. Use the code sequence *ldc* address, *stnl 0,* to load the address desired.

2. Then the *store nonlocal* (with offset zero) stores the value in B at the address loaded by way of the *load constant* instruction.

STOPERR Stop On Error

Opcode Clock Cycles
25 F5 2

Flags
Unaffected.

Scheduling
This instruction can cause the current process to be descheduled.

Pseudocode
```
IF (Error is set)
THEN
STOP // stop this process, schedule next waiting process
ELSE
Iptr = Next Instruction
```

Description
Stop on error stops the process that is currently executing if the Error flag is set. Note the current process will be removed entirely and not simply halted or temporarily descheduled. This instruction performs the same action as the *HaltOnError* flag when it is used, but STOPERR only affects the currently executing process.

STOPP Stop Process

Opcode Clock Cycles
21 F5 11

Flags
Unaffected.

Scheduling
This instruction will cause the current process to be descheduled.

Pseudocode
```
// Stop current process leaving Iptr in the workspace so it can be
// restarted (via a RUNP) and start the next waiting process
STOP CURRENT PROCESS
START NEXT WAITING PROCESS // in the active process queue
```

Description
So that it can be restarted in the future, the *stop process* instruction stops the current process and leaves its instruction pointer in workspace location -1. To restart the process, a RUNP instruction must be executed as the process is removed from the scheduling list. STOPP can also be used to change a process's priority.

STTIMER Store Timer

Opcode Clock Cycles
25 F4 1

Flags
Unaffected.

Scheduling
Atomic.

Pseudocode
```
ClockReg0 = A // this also starts ClockReg0 "ticking"
ClockReg1 = A // this also starts ClockReg1 "ticking"
A = B
B = C
C = Undefined
Iptr = Next Instruction
```

Description
Store timer initializes both the low and high priority timers to the value contained in A. STIMER also starts the timers ticking. B is popped into A and C is popped into B.

SUB Subtract

Opcode Clock Cycles
FC 1

Flags
Error is set on overflow.

Scheduling
Atomic.

Pseudocode

```
A = B = A
IF (Overflow) THEN Set Error
B = C
C = Undefined
Iptr = Next Instruction
```

Description
Subtract subtracts the contents of A from B, leaving the result in A. C is popped into B. The Error flag is set if overflow occurs.

SUM Sum

Opcode Clock Cycles
25 F2 1

Flags
Unaffected.

Scheduling
Atomic.

Pseudocode

```
A = B + A
B = C
C = Undefined
Iptr = Next Instruction
```

Description
Sum adds B to A without checking for overflow or carry. C is popped into B.

TALT Timer Alt

Opcode Clock Cycles
24 FE 4

Flags
Unaffected.

Scheduling
Atomic.

Pseudocode

```
// Store flags to show enabling is occurring and alt time not yet set

Tlink.s = TimeNotSet.p // i.e., Wptr[-4] = 0x80000002 = -2
State.s = Enabling.p // i.e., Wptr[-3] = 0x80000001 = -1
Iptr = Next Instruction
```

Description
Timer alt start sets up an *alternative input* for the current process. It also stores 0x80000002 in workspace location -4 *(Tlink.s).* This instruction is used instead of ALTWT in the case of timer guards.

User Notes
The Workspace Pointer must not change between the execution of the TALT instruction and the ALTEND instruction.

TALTWT Timer Alt Wait

Opcode Clock Cycles
25 F1 15 (time past)
48 (time future)

Flags
Unaffected.

Scheduling
This instruction can cause the current process to be descheduled.

Pseudocode
```
// Set a flag (DisableStatus) to show no branch has yet been
// selected and put alt time into the timer queue and wait until
// one of the guards is ready (indicated by the value in State.s)
// or until an already specified (by ENBT) time is reached.
// DisableStatus is defined to be Wptr[0].
//
// The process will be descheduled until State.s = Ready.p or
// until the alternation "times out," that is, until the timer reaches
// a time specified by an earlier ENBT instruction. Timing out
// is indicated by Tlink.s = Wptr[-4]. Tlink.s can either be:
// TimeNotSet.p = 0x80000001, meaning time not yet reached
// or TimeSet.p, meaning the time was reached (i.e., timed out).
//
// Possible values State.s can have:
// Enabling.p = 0x80000001
// Waiting.p = 0x80000002
// Ready.p = 0x80000003
// Note: State.s = Wptr[-3]

DisableStatus = -1
A = Undefined
B = Undefined
C = Undefined
LOOP:
WHILE ((State.s != Ready.p) OR (Tlink.s != TimeSet.p))
GOTO LOOP
Iptr = Next Instruction
```

Description
Timer alt wait is similar to the ALTWT instruction; however, TALTWT must be used with timer guards. Just as ALTWT does, TALTWT places 0x80000001 into the workspace location zero. If the following conditions don't exist, it will cause the process to be descheduled:

ï If *State.s* (workspace location -3) is not ready.

ï If *Tlink.s* isn't 0x80000001 with a time in location -5 that is in the past.

If the above conditions do exist, then the process continues on.

TESTERR Test Error False and Clear

Opcode Clock Cycles
22 F9 2 (no error)
3 (error)

Flags
The Error flag is cleared by this instruction.

Scheduling
Atomic.

Pseudocode
```
IF (Error is set) A = FALSE
ELSE A = TRUE // else Error is clear
Clear Error
B = A
C = B
Iptr = Next Instruction
```

Description
Test Error false and clear pushes the inverse of the current state of the Error flag into A. For example, A receives 0 for false if the Error flag is set and A receives 1 for true if the Error flag is not set. The Error flag is cleared as a result of this instruction. Prior to A receiving the inverse of the Error flag, B is pushed into C and A is pushed into B.

TESTHALTERR Test HaltOnError Flag

Opcode Clock Cycles
25 F9 2

Flags
Unaffected.

Scheduling
Atomic.

Pseudocode
```
IF (HaltOnError is set) A = TRUE
ELSE A = FALSE // HaltOnError is clear
B = A
C = B
Iptr = Next Instruction
```

Description
Test HaltOnError loads A with the state of the *HaltOnError* flag. If the *HaltOnError* flag is set, A is set to 1, meaning true. If the *HaltOnError* flag is clear, then A is set to 0, meaning false. Prior to A receiving this value, B is pushed into C.

TESTPRANAL Test Processor Analysing

Opcode Clock Cycles
22 FA 2

Flags
Unaffected.

Scheduling
Atomic.

Pseudocode
```
IF (PROCESSOR WAS ANALYZED) THEN A = TRUE
```

```
ELSE A = FALSE // processor was reset
B = A
C = B
Iptr = Next Instruction
```

Description
Test processor analysing returns a flag in A that is true (one) if the last time the transputer was reset it was an "analysis" instead of a reset. If a special analyze pin is asserted while the transputer is being reset, some of the state of the processor is saved to allow a post-mortem debug. This instruction is useful in debugging hardware.

User Notes
This instruction is only useful for debugging hardware.

TIN Timer Input

Opcode Clock Cycles
22 FB 30 (time future)
3 (time past)

Flags
Unaffected.

Scheduling
This instruction can cause the current process to be descheduled.

Pseudocode
```
WAIT UNTIL TIME IS AFTER VALUE IN A // skip if time is before A
A = Undefined
B = Undefined
C = Undefined
Iptr = Next Instruction
```

Description
Timer input interprets the contents of A as a timer value and compares it with the value of the current priority timer.

1. If A is less than the current value of the timer (i.e., in the "past"), the process continues and the instruction has no effect.

2. However, if A is greater than the current priority timer, then the time is in the "future" and the process is descheduled (blocks) until that time occurs. Upon reaching that time, the process will awaken. This is a descheduling instruction, and the values of A, B, and C are undefined after its execution.

UNPACKSN Unpack Single FP Number T414

Opcode Clock Cycles
26 F3 15

Flags
Unaffected.

Scheduling
Atomic.

Pseudocode
```
// A contains a single-length floating point number.
A = FRACTION FIELD CONTENTS OF A
```

```
B = EXPONENT FIELD CONTENTS OF A
IF (A == 0)
THEN C = 4 * B + 0
ELSE IF ((A == NORMALIZED) OR (A = DENORMALIZED))
THEN C = 4 * B + 1
ELSE IF (A == Inf)
THEN C = 4 * B + 2
ELSE IF (A == NaN)
THEN C = 4 * B + 3
Iptr = Next Instruction
```

Description
Unpack single-length floating point number decodes the IEEE format single-length floating point number contained in the A register and returns its exponent in B and the fractional field (mantissa) in A. The implied, most significant bit in normalized numbers is not included in the return value for A. The C register receives a value indicating the type of number (in its lower two bits). The exact quantity C receives is four times the intial value of the B register (i.e., B shifted left two bits) plus: 0 if A was zero, 1 if A was a denormalized or normalized number, 2 if A was an infinity, or 3 if A was a NaN (Not a Number).

WCNT Word Count

Opcode Clock Cycles
23 FF 5

Flags
Unaffected.

Scheduling
Atomic.

Pseudocode
```
C = B
B = A & 0x03 // byte offset from word boundary
A = A >> 2 // word offset from address 0x00000000
Iptr = Next Instruction
```

Description
Word count is used to find the word offset of an address from zero and returns it in A; the byte offset (zero to three) from the word is returned in B. It returns A divided by four into A, and the remainder (A modulo 4 or A AND 3 if you prefer) into B. The prior contents of B are pushed up into C (the contents of C are lost).

WSUB Word Subscript

Opcode Clock Cycles
FA 2

Flags
Unaffected.

Scheduling
Atomic.

Pseudocode
```
A = A + (B * 4)
B = C
C = Undefined
Iptr = Next Instruction
```

Description
Word subscript assumes A is the base address of an array and B is the word index into the array. It adds A plus four times B, leaving the result in A. This computes offsets into arrays with word-sized elements. The sum is not checked for overflow, so it also useful for unsigned arithmetic. C is popped into B.

WSUBDB Form Double Word Subscript T800

Opcode Clock Cycles
28 F1 3

Flags
Unaffected.

Scheduling
Atomic.

Pseudocode
```
A = A + (B * 8) // each double contains 8 bytes
B = C
C = Undefined
Iptr = Next Instruction
```

Description
Word subscript is used for generating indexes for double word data items. Form Double Word Subscript adds A plus eight times B, leaving the result in A. It computes offsets into word arrays. The sum is not checked for overflow, so it also useful for unsigned arithmetic. C is popped into B.

XDBLE Extend to Double

Opcode Clock Cycles
21 FD 2

Flags
Unaffected.

Scheduling
Atomic.

Pseudocode
```
C = B
BA = (INT64) A //sign-extended double of A, most significant word in B
Iptr = Next Instruction
```

Description
Exend to double sign extends a 32-bit word held in A to a 64-bit word, leaving the most significant half in B and the least significant half in A.

XOR Exclusive Or

Opcode Clock Cycles
23 F3 1

Flags
Unaffected.

Scheduling
Atomic.

Pseudocode
```
A = B ^ A
B = C
C = Undefined
Iptr = Next Instruction
```

Description
Xor peforms a bitwise logical *exclusive or* operation between A and B and then leaves the result in A. C is popped into B.

XWORD Extend to Word

Opcode Clock Cycles
23 FA 4

Flags
Unaffected.

Scheduling
Atomic.

Pseudocode
```
// B contains the partial word, A contains the length of the partial word
IF (B < A) THEN A = (INT32) B // if the part word is positive
ELSE A = (INT32)(B - A * 2) // else the part word is negative
B = C
C = Undefined
```

Description
Extend to word sign extends a partial word value to a full 32-bit word. B contains the partial word to extend to a full word. A contains the length of the partial word.

User Notes
To extend the sign of a byte (already in A) to the entire word use:
```
ldc 0x80 ; set the most significant bit of the byte
xword ; sign extend the byte into the word
```

To extend the sign of a 16-bit short (already in A) to a 32-bit word, use:
```
ldc 0x8000 ; set the most significant bit of the short
xword ; sign extend the short into the word
```

Appendix A

Instructions Sorted by Operation Code

Note: Some of the following instructions are only available on certain transputers. All numbers for the byte sequences below are in hexadecimal.

Byte Sequence	**Mnemonic**	**Description**
F0	REV	reverse
F1	LB	load byte
F2	BSUB	byte subscript
F3	ENDP	end process
F4	DIFF	difference
F5	ADD	add
F6	GCALL	general call
F7	IN	input message
F8	PROD	product
F9	GT	greater than
FA	WSUB	word subscript
FB	OUT	output message
FC	SUB	subtract
FD	STARTP	start process
FE	OUTBYTE	output byte
FF	OUTWORD	output word
21 F0	SETERR	set Error flag
21 F2	RESETCH	reset channel
21 F3	CSUB0	check from subscript zero
21 F5	STOPP	stop process
21 F6	LADD	long add
21 F7	STLB	store low priority back pointer
21 F8	STHF	store high priority front pointer
21 F9	NORM	normalize
21 FA	LDIV	long divide
21 FB	LDPI	load pointer to instruction
21 FC	STLF	store low priority front pointer
21 FD	XDBLE	extend to double
21 FE	LDPRI	load current priority
21 FF	REM	remainder
22 FO	RET	return (from subroutine call)
22 F1	LEND	loop end
22 F2	LDTIMER	load timer
22 F9	TESTERR	test error flag false and clear
22 FA	TESTPRANAL	test processor analysing
22 FB	TIN	timer input
22 FC	DIV	divide
22 FE	DIST	disable timer
22 FF	DISC	disable channel
23 F0	DISS	disable skip
23 F1	LMUL	long multiply
23 F2	NOT	bitwise not
23 F3	XOR	exclusive or
23 F4	BCNT	byte count
23 F5	LSHR	long shift right
23 F6	LSHL	long shift left
23 F7	LSUM	long sum
23 F8	LSUB	long subtract
23 F9	RUNP	run process
23 FA	XWORD	extend to word
23 FB	SB	store byte
23 FC	GAJW	general adjust workspace
23 FD	SAVEL	save low priority queue registers

23 FE SAVEH save high priority queue registers
23 FF WCNT word count
24 F0 SHR shift right
24 F1 SHL shift left
24 F2 MINT minimum integer
24 F3 ALT alt start
24 F4 ALTWT alt wait
24 F5 ALTEND alt end
24 F6 AND and
24 F7 ENBT enable timer
24 F8 ENBC enable channel
24 F9 ENBS enable skip
24 FA MOVE move message
24 FB OR or
24 FC CSNGL check single
24 FD CCNT1 check count from one
24 FE TALT timer alt start
24 FF LDIFF long diff
25 F0 STHB store high priority back pointer
25 F1 TALTWT timer alt wait
25 F2 SUM sum
25 F3 MUL multiply
25 F4 STTIMER store timer
25 F5 STOPERR stop on error
25 F6 CWORD check word
25 F7 CLRHALTERR clear halt on error flag
25 F8 SETHALTERR set halt on error flag
25 F9 TESTHALTERR test halt on error flag
25 FA DUP duplicate top of stack
25 FB MOVE2DINIT initialize data for two-dimensional block move
25 FC MOVE2DALL two-dimensional block copy
25 FD MOVE2DNONZERO two-dimensional block copy nonzero bytes
25 FE MOVE2DZERO two-dimensional block copy zero bytes
26 F3 UNPACKSN unpack single-length floating point number
26 FD ROUNDSN round single-length floating point number
27 F1 LDINF load single-length infinity
27 F2 FMUL fractional multiply
27 F3 CFLERR check single-length floating point infinity or NaN
27 F4 CRCWORD calculate CRC on word
27 F5 CRCBYTE calculate CRC on byte
27 F6 BITCNT count bits set in word
27 F7 BITREVWORD reverse bits in word
27 F8 BITREVNBITS reverse bottom N bits in word
28 F1 WSUBDB form double-word subscript
28 F2 FPLDNLDBI floating point load non-local indexed double
28 F3 FPCHKERR check floating point error
28 F4 FPSTNLDB floating point store non-local double
28 F6 FPLDNLSNI floating point load non-local indexed single
28 F7 FPADD floating point add
28 F8 FPSTNLSN floating point store non-local single
28 F9 FPSUB floating point subtract
28 FA FPLDNLDB floating point load non-local double
28 FB FPMUL floating point multiply
28 FC FPDIV floating point divide
28 FE FPLDNLSN floating point load non-local single
28 FF FPREMFIRST floating point remainder first step
29 F0 FPREMSTEP floating point remainder step
29 F1 FPNAN floating point test for Not a Number
29 F2 FPORDERED floating point orderability
29 F3 FPNOTFINITE floating point test for not finite
29 F4 FPGT floating point greater than
29 F5 FPEQ floating point equality

29 F6 FPI32TOR32 INT32 to REAL32
29 F8 FPI32TOR64 INT32 to REAL64
29 FA FPB32TOR64 BIT32 to REAL64
29 FC FPTESTERR test floating point Error flag false and clear
29 FD FPRTOI32 REAL to INT32
29 FE FPSTNLI32 store non-local INT32
29 FF FPLDZEROSN load zero single
2A F0 FPLDZERODB load zero double
2A F1 FPINT round to floating point integer
2A F3 FPDUP floating point duplicate top of stack
2A F4 FPREV floating point reverse top of stack
2A F6 FPLDNLADDDB floating point load non-local and add double
2A F8 FPLDNLMULDB floating point load non-local and multiply double
2A FA FPLDNLADDSN floating point load non-local and add single
2A FB FPENTRY floating point unit entry
2A FC FPLDNLMULSN floating point load non-local and multiply single
41 2A FB FPUSQRTFIRST floating point square root first step
42 2A FB FPUSQRTSTEP floating point square root step
43 2A FB FPUSQRTLAST floating point square root last step
44 2A FB FPURP set rounding mode to plus infinity
45 2A FB FPURM set rounding mode to minus infinity
46 2A FB FPURZ set rounding mode to zero
47 2A FB FPUR32TOR64 convert REAL32 to REAL64
48 2A FB FPR64TOR32 convert REAL64 to REAL32
49 2A FB FPUEXPDEC32 divide by 2**32
4A 2A FB FPUEXPINC32 multiply by 2**32
4B 2A FB FPUABS floating point absolute value
4D 2A FB FPUNOROUND convert REAL64 to REAL32 w/o rounding
4E 2A FB FPUCHKI32 check in the range of type INT32
4F 2A FB FPUCHKI64 check in the range of type INT64
21 41 2A FB FPUDIVBY2 divide by 2.0
21 42 2A FB FPUMULBY2 multiply by 2.0
22 42 2A FB FPURN set rounding mode to nearest
22 43 2A FB FPUSETERR set floating point Error flag
29 4C 2A FB FPUCLRERR clear floating point Error flag

Appendix B

Sample Floating Point Number Representations

Single-Length Floating Point Numbers

Value	Representation (in Hexadecimal)
-2.0	C0000000
-1.0	BF800000
0.0	00000000
1.0	3F800000
2.0	40000000
3.0	40400000
4.0	40800000
5.0	40A00000
6.0	40C00000
7.0	40E00000
8.0	41000000
9.0	41100000
10.0	41200000
Positive Infinity	7F800000
Negative Infinity	FF800000

IEEE Single-Length Not a Number (NaN) Error Encodings

Error	Representation
Divide zero by zero	7FC00000
Divide infinity by infinity	7FA00000
Multiply zero by infinity	7F900000
Addition of opposite signed infinities	7F880000
Subtraction of same signed infinities	7F880000
Negative square root	7F840000
Double to Single NaN conversion	7F820000
Remainder from infinity	7F804000
Remainder by zero	7FF00400

Additional Single-Length Not a Number (NaN) Error Encodings (Inmos)

Error	Representation
Result not defined mathematically	7F800010
Result unstable	7F800008
Result inaccurate	7F800004

Double-Length Floating Point Numbers

Value	Representation (in Hexadecimal)
-2.0	C0000000 00000000
-1.0	BFF00000 00000000
0.0	00000000 00000000
1.0	3FF00000 00000000
2.0	40000000 00000000
3.0	40080000 00000000
4.0	40100000 00000000
5.0	40140000 00000000
6.0	40180000 00000000
7.0	401C0000 00000000
8.0	40200000 00000000
9.0	40220000 00000000
10.0	40240000 00000000

Positive Infinity	7FF00000 00000000
Negative Infinity	FFF00000 00000000

IEEE Double-Length Not a Number (NaN) Error Encodings

Error	Double
Divide zero by zero	7FF80000 00000000
Divide infinity by infinity	7FF40000 00000000
Multiply zero by infinity	7FF20000 00000000
Addition of opposite signed infinities	7FF10000 00000000
Subtraction of same signed infinities	7FF10000 00000000
Negative square root	7FF08000 00000000
Double to Single NaN conversion	7FF04000 00000000
Remainder from infinity	7FF00800 00000000
Remainder by zero	7FF00400 00000000

Additional Double-Length Not a Number (NaN) Error Encodings (Inmos)

Error	Double
Result not defined mathematically	7FF00002 00000000
Result unstable	7FF00001 00000000
Result inaccurate	7FF00000 80000000

Appendix C

Special Workspace Locations

The amount of space that must be allocated to a workspace (in addition to the space for local variables) is:

Process with no I/O 2 words
Process with only unconditional I/O 3 words
Process with alternative input 3 words + location 0
Process with timer input 5 words
Process with alternative timer input 5 words + location 0

This is because, when a process is descheduled, up to five words below the current workspace are used to save the internal state of the process. The saved items are:

Wptr[0] Offset to guard routine to execute (used in alternation),
(called DisableStatus)
Wptr[-1] Instruction pointer (called Iptr.s)
Wptr[-2] Workspace descriptor of next process in queue (called Link.s)
Wptr[-3] Flag indicating alternation state (called State.s)
Wptr[-4] Timer value reached flag (called Tlink.s)
Wptr[-5] Time process waiting to awaken at (called Time.s)

Workspace location zero is used by an alternation construct to hold the offset from the end of the alternation (i.e., the instruction after an ALTEND) to the guard routine to execute (i.e., the routine corresponding to the channel on which the message was received). It is initialized to -1 by an ALTWT instruction; this value (-1) indicates that no message has arrived yet, or that no corresponding guard routine has been selected yet (via a DISC). Note: When a process resumes after an ALTWT, it typically executes disable commands to find the routine to execute that corresponds to the channel the message was received on.

Iptr.s contains the instruction pointer of a descheduled process.

Link.s contains a workspace descriptor pointing to the next process. The processes form a queue in workspace linked by the Link.s field in each process' workspace. A workspace descriptor (Wdesc) is the workspace pointer with the priority of the process that owns the workspace encoded into the low bit (either 0 for high priority or 1 for low priority).

For alternation constructs, Wptr[-3] is used as a flag to indicate the state of the alternation and is names State.s. State.s is a flag that has one of three possible three values. The flag indicates that alternation is being enabled, the process is waiting for a message, or the process is ready (has received a message). The corresponding values are:

Enabling.p = 0x80000001 ; enabling alternation
Waiting.p = 0x80000002 ; waiting for a message (channel)
Ready.p = 0x80000003 ; message is ready (received)

Tlink.s is a flag used in the implementation of timer guards and can assume one of the following two values:

TimeSet.p = 0x80000001 ; timer set
TimerNotSet.p = 0x80000002 ; timer not set

Time.s contains the time a process is waiting for before it "times out" from what it is doing and continues (used in timer alternations via the TALT and TALTWT instructions to allow a process to "time out" on a communications channel).

Different descheduling instructions can cause different items to be saved. Locations -1 and -2 are used whenever a process is descheduled; the other locations may hold saved values, depending on the type of descheduling instruction involved.

Appendix D

Transputer Memory Map

All numbers below are in hexadecimal.

The address space of the transputer is signed (i.e., addresses are considered to be either positive or negative). Addresses run from the most negative address at 80000000 to FFFFFFFF, then to 00000000 through 7FFFFFFF (the most positive address).

The T414 has 2 kilobytes of on-chip RAM at addresses 80000000 to 800007FF. The T800 has 4 kilobytes of on-chip ram at locations 80000000 to 80000FFF.

Address	**Description**
7FFFFFFF	Backwards jump consisting of a prefix
7FFFFFFE	instruction and jump instruction (ResetCodePtr)
7FFFFFFB	Memory configuration area
7FFFFF6C	
00000000	Start of nonnegative address space
80001000	Start of T800 external memory
80000FFF	End of T800 on-chip memory
80000800	Start of T414 external memory
800007FF	End of T414 on-chip memory
80000070	Start of user memory (MemStart)
8000006F	Reserved for extended functions (T800 only)
80000048	such as block moves and floating point operations
80000044	Low priority process extra register (EregIntSaveLoc)
80000040	Low priority process status (STATUSIntSaveLoc)
8000003C	Low priority process C register (CregIntSaveLoc)
80000038	Low priority process B register (BregIntSaveLoc)
80000034	Low priority process A register (AregIntSaveLoc)
80000030	Low priority process instruction pointer (IptrIntSaveLoc)
8000002C	Low priority process workspace descriptor (WdescIntSaveLoc)
80000028	Low priority timer queue head pointer (TPtrLoc1)
80000024	High priority timer queue head pointer (TPtrLoc0)
80000020	Event channel
8000001C	Link 3 input channel
80000018	Link 2 input channel
80000014	Link 1 input channel
80000010	Link 0 input channel
8000000C	Link 3 output channel
80000008	Link 2 output channel
80000004	Link 1 output channel
80000000	Link 0 output channel

The backwards jump at 7FFFFFFE (ResetCodePtr) derives from the fact that, if the transputer is booted from ROM (i.e., the BootFromROM pin on the transputer is pulled high), the transputer wakes up and executes the code contained in the last two bytes of the address space (at addresses 7FFFFFFE and 7FFFFFFF). This is typically a prefix instruction followed by a jump instruction, which causes a jump "backwards" in the address. The maximum range of this two byte jump is 255 bytes. The transputer should be configured to jump to a bootstrapping routine with this backwards jump (the routine may simply be a longer backwards jump).

The memory configuration area is used to specify an external memory interface configuration. The T414 and T800 both have a selection of 13 preprogrammed configurations. It is also possible to generate different configurations from the 13 preset ones. The external memory interface supports dynamic and static RAM as well as ROM and EPROM.

Location 80000048 is the start of user memory on the T414; however, locations 80000048 through 8000006F are used in the T800 to implement some of the new instructions present in the T800 (as opposed to the T414). Thus, the start of user memory on the T800 is 80000070.

When a high priority process interrupts a low priority process, the transputer saves the state of the low priority process in on-chip memory at locations 80000024 through 80000044 (locations with the phrase "IntSaveLoc" in their names). EregIntSaveLoc contains information if a block move is interrupted. STATUSIntSaveLoc contains process station information of the interrupted process (e.g., Error flag, HaltOnError flag).

The high and low priority timer queues are queues of descheduled processes waiting for the timer to reach a certain time. The pointers to these queues are kept in on-chip memory at locations 80000028 and 80000024.

When the external event pin on the transputer receives a signal, any process waiting on the Event channel (at 80000020) will awaken. The link communication channels are at fixed addresses at the bottom of memory.

Appendix E

Instructions that can cause the Error flag to be set:

ADC	add constant
ADD	add
CCNT1	check count from one
CSNGL	check single
CSUB0	check from subscript zero
CWORD	check word
DIV	divide
FMUL	fractional multiply
FPCHKERR	check floating point error
LADD	long add
LDIV	long divide
LSUB	long subtract
MUL	multiply
REM	remainder
SETERR	set Error
SUB	subtract

Appendix F

Instructions that are unique to the T414:

CFLERR check single-length floating point infinity or NaN
LDINF load single-length infinity
POSTNORMSN post-normalize correction of single-length fp number
ROUNDSN round single-length floating point number
UNPACKSN unpack single-length floating point number

Instructions that are unique to the T800:

BITCNT count bits set in word
BITREVNBITS reverse bottom N bits in word
BITREVWORD reverse bits in word
CRCBYTE calculate CRC on byte
CRCWORD calculate CRC on word
DUP duplicate top of stack
MOVE2DALL two-dimensional block copy
MOVE2DINIT initialize data for two-dimensional block move
MOVE2DNONZERO two-dimensional block copy non-zero bytes only
MOVE2DZERO two-dimensional block copy zero bytes only
WSUBDB form double-word subscript

In addition to the above instructions, all floating point instructions (instructions that start with the letters "FP") are unique to the T800.

Appendix G

Summary of different models of transputers.

The T212 is a 16-bit processor with 2K of on-chip RAM and a 64K address range (using separate address and data buses).

The T222 is an updated T212 with 4K of on-chip RAM.

The T225 is an updated T222 with additional instructions for program debugging. This support for a "breakpoint" instruction allows a programmer to stop his program (at a breakpoint), examine registers and memory, and then continue on. The T225 also has some additional instructions found on T800 series transputers.

The T414 is a 32-bit processor with 2K of on-chip RAM and a four gigabyte address range (using multiplexed address and data lines).

The T425 is an updated version of the T414 with additional instructions for program debugging (i.e., breakpointing instructions). The T425 also has the (two-dimensional) block move instructions and some other instructions found on T800 series transputers. For instance, FPTESTERR is on the T425, but always returns TRUE (meaning the FPU flag wasn’t set, as there is no floating point processor on the T425).

The T800 is a superset of the T414 in that it is a 32-bit processor with a floating point coprocessor on-chip, additional instructions, and 4K of on-chip RAM.

The T801 is a T800 with separate address and data buses, thus speeding up memory accesses.

The T805 is an updated T800 with additional instructions for program debugging (i.e., breakpointing instructions).

The table below summarizes the differences among transputer models:

	T212	T222	T225	T414	T425	T800	T801	T805
Memory (Kbytes)	2	4	4	2	4	4	4	4
Bytes per word	2	2	2	4	4	4	4	4
T4 Floating Point				X	X			
T8 Floating Point						X	X	X
Instructions								
2D Block Move					X	X	X	X
FMUL				X	X	X	X	X
DUP			X		X	X	X	X
WSUBDB			X		X	X	X	X
CRC instructions			X		X	X	X	X
BITCOUNT			X		X	X	X	X
FPTESTERR					X	X	X	X

Suggested Readings

Transputer-related Articles

1. Pountain, Dick. Microprocessor design: the transputer and its special language, Occam. *BYTE*, August 1984, page 361. An early article that discusses the concepts behind the transputer and Occam.

2. Walker, Paul. The transputer: a building block for parallel processing. *BYTE*, May 1985, page 219. One of the first general introductions to the transputer, Occam, and parallel distributed processing.

3. Roelofs, Bernt. The transputer: a microprocessor designed for parallel processing. *Micro Cornucopia*, November/December 1987, page 6. An excellent, easy-to-read article covering the transputer's overall architecture.

4. Kurver, Rob and Wijbrans, Klaas. Developing a parallel C compiler: powerful new extensions for a familiar language. *Micro Cornucopia*, November/December 1987, page 14. A fascinating article covering proposed parallel programming extensions to C, using the transputer as its target.

5. Poplett, John and Rob Kurver. The DSI transputer development system. *BYTE*, February 1988, page 249. This article reviews Definicon System's transputer co-processor plug-in board for an IBM PC. The most interesting aspects of this article are the parallel extensions to the C programming language that enables a programmer to use the special parallel features of the transputer. The absence of any discussion of Occam is also worth noting.

6. Pountain, Dick. A personal transputer. *BYTE*, June 1988, page 303. This article discusses Atari's plans for building a transputer-powered graphics workstation (named Abaq) and the Helios Operating System (by Perihelion), which provides a Unix-like environment for the transputer.

7. Hayes, Frank. The crossbar connection. *BYTE*, November 1988, page 278. A description of Cogent Research's transputer-based parallel workstation. Cogent Research has used transputers to implement the Linda parallel programming extensions to conventional programming languages. (Linda was developed at Yale by Dr. David Gelernter and an article about Linda (by him) is in the same issue of *BYTE* on page 301).

8. Stein, Richard M. T800 and counting. *BYTE*, November 1988, page 287. An excellent overview of the transputer from a hardware perspective along with a detailed discussion of transputer to transputer communication in Occam.

From Inmos

9. Shepard, Roger. Extraordinary use of transputer links. Inmos Technical Note 1. Bristol, England: Inmos Ltd., 1986. Technical information concerning transputer-to-transputer communication in Occam.

10. IMS T800 Architecture. Inmos Technical Note 6. Bristol, England: Inmos Ltd., 1986. General information on the T800 transputer, along with Occam descriptions of some of the T800 instructions.

11. Gore, Tony and David Cormie. Designing with the IMS T414 and IMS T800 memory interface. Inmos Technical Note 9. Bristol, England: Inmos Ltd., 1986. Useful hardware information on how to connect and configure memory with the transputer.

12. *The Tranputer Instruction Set - A Compiler Writer's Guide*. Bristol, England: Inmos Ltd., May 1987. An invaluable, but frustrating, guide to the transputer's (assembler) instruction set. The only one available from Inmos.

13. *Transputer Reference Manual*. Inmos Ltd., 1988. Prentice-Hall. A useful reference for hardware-related information about the entire transputer family. There is also a very terse introduction to Occam in this manual.

Floating Point Arithmetic

14. *IEEE Standard for Binary Floating-Point Arithmetic*. Institute of Electrical and Electronic Engineers, Inc. 1985.

ANSI/IEEE Std 754-1985. The standard specification to which all implementations aspire and all implementors sweat over.

15. Plauger, P.J. Programming on purpose: properties of floating-point arithmetic. *Computer Language*, Volume 5, Number 3, March 1988, page 17. A good discussion of floating-point arithmetic for programmers.

16. Wilson, Pete. Floating-point survival kit. *BYTE*, Volume 13, Number 3, March 1988, page 217. An excellent overview of some of the problems in floating point number representation, along with a discussion of the transputer's implementation of IEEE floating point (albeit with Occam).

17. Ochs, Tom. Theory and practice. *Computer Language*, Volume 6, Number 3, March 1989, page 67. A good in-depth covering of all the topics in the IEEE 754 Standard and some that weren't.