





John Roberts



# Transputer Assembly Language Programming

# Transputer Assembly Language Programming

John Roberts



VAN NOSTRAND REINHOLD  
New York

Copyright © 1992 by Van Nostrand Reinhold

Library of Congress Catalog Card Number 91-32704  
ISBN 0-442-00872-4

All rights reserved. No part of this work covered by  
the copyright hereon may be reproduced or used in any  
form or by any means—graphic, electronic, or  
mechanical, including photocopying, recording, taping,  
or information storage and retrieval systems—without  
written permission of the publisher.

Printed in the United States of America

Van Nostrand Reinhold  
115 Fifth Avenue  
New York, New York 10003

Chapman and Hall  
2-6 Boundary Row  
London, SE 1 8HN, England

Thomas Nelson Australia  
102 Dodds Street  
South Melbourne 3205  
Victoria, Australia

Nelson Canada  
1120 Birchmount Road  
Scarborough, Ontario M1K 5G4, Canada

16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1

# **Library of Congress Cataloging-in-Publication Data**

Roberts, John, 1959—

Transputer assembly language programming / John Roberts.

p. cm.

Includes bibliographical references and index.

ISBN 0-442-00872-4

1. Transputers—Programming. 2. Parallel programming (Computer  
science) I. Title.

QA76.6.R625 1992

005.26—dc20

91-32704

CIP

# Transputer Assembly Language Programming

## 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 assembler, 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 familiar 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 egregious 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
Chapter 6	Instruction Set Reference



Appendixes  
Suggested Readings

Miscellaneous useful information  
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.

# Contents

**Preface ix**

ACKNOWLEDGMENTS ix

**Introduction xi**

**Chapter 1 Introduction to Parallel Processing and the Transputer 1**

**Chapter 2 Transputer Hardware Architecture 9**

REGISTER SET 10

Register Notation and Summary 12

FLAGS 13

T800 FLOATING POINT UNIT 13

INSTRUCTION FORMAT 14

DATA ORGANIZATION IN MEMORY 15

Addresses 15

On-Chip Memory 15

MICROCODE SCHEDULER 16

COMMUNICATION 18

EVENT CHANNEL 20

ON-CHIP TIMERS 20

**Chapter 3 Transputer Instruction Set 22**

DIRECT INSTRUCTIONS 23

Stack Operations 23

Workspaces 25

Space on the Workspace 33

Prefix Functions 34

Flow Control Instructions 36

Summary of Direct Instructions 41

INDIRECT INSTRUCTIONS 42

General Operation Codes 43

Arithmetic and Logical Operation Codes 44

Long Arithmetic Operation Codes 47

Indexing and Array Operation Codes 48

Control Operation Codes 50

Scheduling Operation Codes 52

Timer Handling Operation Codes 56

Input/Output Operation Codes 57

Alternation 60

Error Handling Operation Codes 65

Processor Initialization Operation Codes 68

T800 SPECIFIC INSTRUCTIONS 70

Block Move Operation Codes 70

Bit Operation Codes 72

CRC Operation Codes 73

Floating Point Numbers 75

IEEE Standard for Binary Floating-Point  
Arithmetic 75

Floating Point Errors 80

T414 Floating Point Support 81

T800 Floating Point Support 83

Load and Store Operation Codes 83

General Operation Codes	85
Arithmetic Operation Codes	85
Rounding Operation Codes	88
Error Operation Codes	89
Comparison Operation Codes	90
Conversion Operation Codes	91
Optimizing Floating Point Calculations	94

## **Chapter 4      Programming the Transputer      95**

BOOTING A TRANSPUTER	95
Booting from ROM	96
Booting from a Link	96
BEHAVIOR OF THE C REGISTER	99
SOFT RESET	99
STARTING MULTIPLE PROCESSES	100
CHANGING THE PRIORITY OF A PROCESS	103
TRANSPUTER TIMER BUG	104
SUBROUTINES	107
THE STATIC LINK	112
TIMER	115
COMMUNICATION	115
ALTERNATION	116
USING THE T800 FLOATING POINT UNIT	120
OPTIMIZING T800 FLOATING POINT INSTRUCTIONS	125

## **Chapter 5      Newer Transputers      127**

COMPLAINTS ABOUT TRANSPUTERS	127
T425 AND T805	128
T9000	130

<b>Chapter 6</b>	<b>Instruction Set Reference</b>	<b>132</b>
	SYMBOLLOGY SUMMARY	136
<b>Appendix A</b>	<b>Instructions Sorted by Operation Code</b>	<b>266</b>
<b>Appendix B</b>	<b>Sample Floating Point Number Representations</b>	<b>270</b>
<b>Appendix C</b>	<b>Special Workspace Locations</b>	<b>273</b>
<b>Appendix D</b>	<b>Transputer Memory Map</b>	<b>275</b>
<b>Appendix E</b>	<b>Instructions Which Can Set Error</b>	<b>278</b>
<b>Appendix F</b>	<b>T414 and T800 Instruction Set Differences</b>	<b>279</b>
<b>Appendix G</b>	<b>Summary of Different Models of Transputers</b>	<b>281</b>
<b>Suggested Readings</b>	<b>283</b>	
	TRANSPUTER-RELATED ARTICLES	283
	FROM INMOS	284
	FLOATING POINT ARITHMETIC	284
<b>Index</b>	<b>285</b>	

# Chapter 1

## 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
...
...
...
```

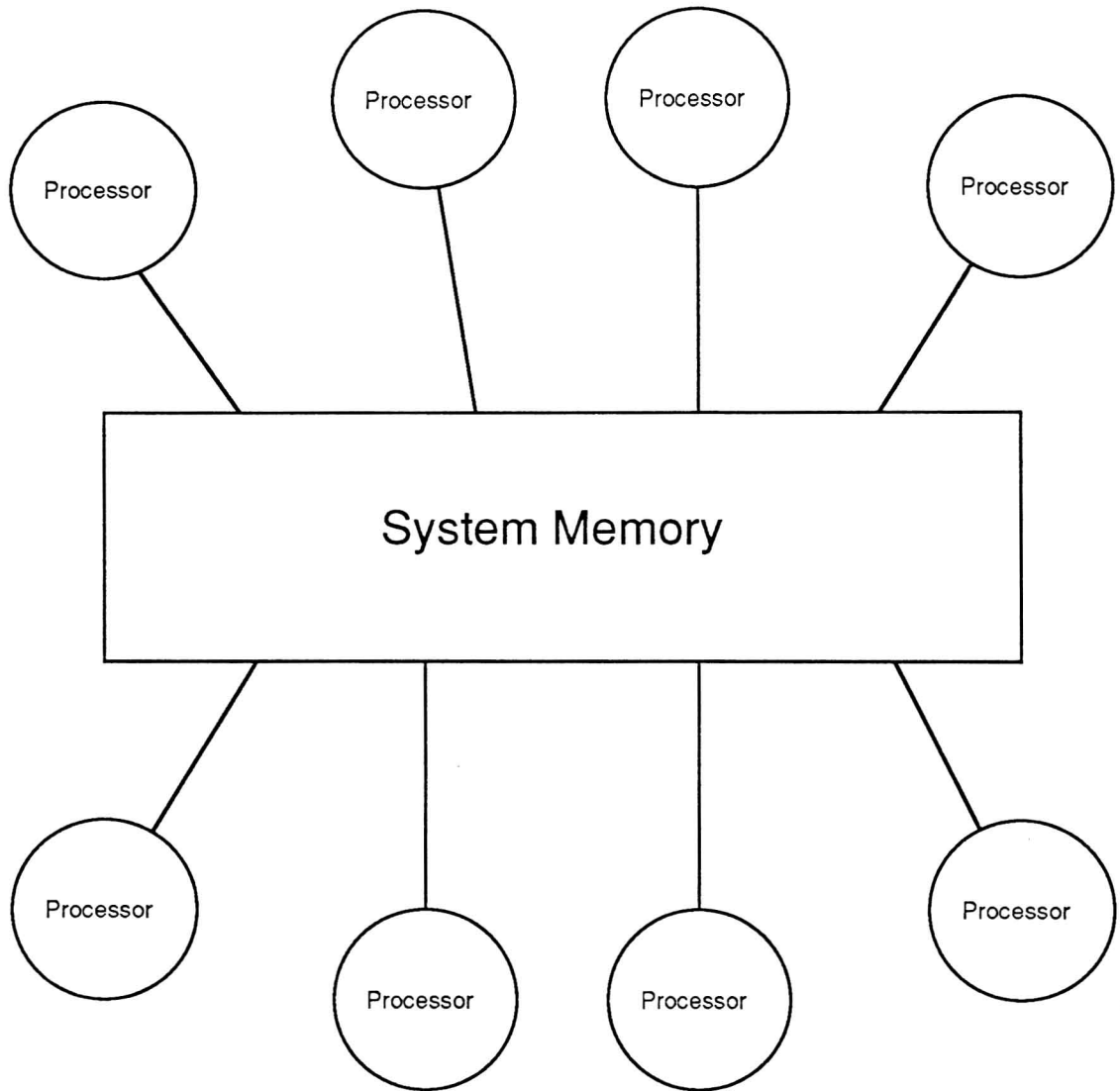
However, suppose the program loop was:

```
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 as illustrated in Figure 1-1. 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.

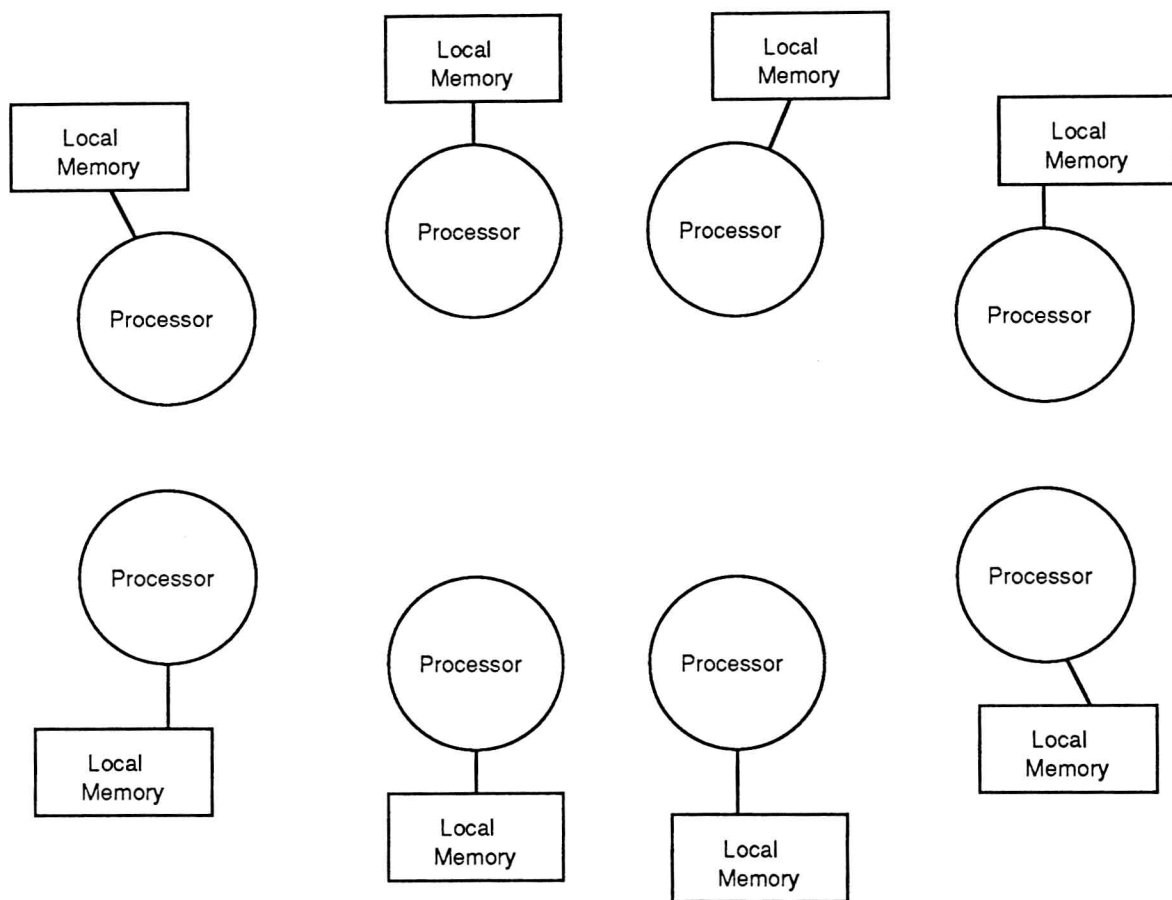
The alternate memory model, distributed memory, is where the transputer fits in. A distributed memory architecture is depicted in Figure 1-2. In a distributed memory model, each processor has its own local memory. The key question in a distributed memory model is "What is the nature of the communication 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 1-1.** An example of a shared memory architecture.

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





**Figure 1-2.** An example of distributed memory.

“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. Figure 1-3 illustrates four processors in