February 1991

The Huges Array Co-Processor and Its Application to Robotics

Craig Sayers
University of Pennsylvania

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-91-17.

The Huges Array Co-Processor and Its Application to Robotics

Abstract

This report describes the results of twelve months research involving the Hughes array co-processor. This work began with the testing and debugging of the existing system, continued with the development of software to interface the co-processor to a host machine and concluded with the implementation of a trajectory planning algorithm for redundant manipulators.

A loader program has been developed which allows simple programs to be executed. A library of C-callable routines has also been created and this enables the fabrication of more complex systems which require a high level of interaction between the host and co-processor.

Routines to perform square root, sine and cosine functions have been designed and these have been used successfully in the development of a trajectory planning algorithm. This algorithm uses the co-processor to compute in parallel a large number of forward kinematics solutions and by doing so is able to convert a cartesian space trajectory into a joint space path for a redundant manipulator.

The performance of the processor has been analyzed and a number of recommendations have been made concerning future implementations.

Comments

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-91-17.

The Huges Array Co-Processor and Its Application To Robotics

MS-CIS-91-17
GRASP LAB 255

Craig Sayers

Department of Computer and Information Science
School of Engineering and Applied Science
University of Pennsylvania
Philadelphia, PA 19104-6389

February 1991

The Hughes Array Co-Processor and its Application to Robotics.

Craig Sayers

This work was supported by the National Science Foundation under Grant Number MIP87-14689. Any opinions, findings, conclusions or recommendations expressed in this report are those of the author and do not necessarily reflect the views of the National Science Foundation.

Summary

This report describes the results of twelve months research involving the Hughes array co-processor. This work began with the testing and debugging of the existing system, continued with the development of software to interface the co-processor to a host machine and concluded with the implementation of a trajectory planning algorithm for redundant manipulators.

A loader program has been developed which allows simple programs to be executed. A library of C-callable routines has also been created and this enables the fabrication of more complex systems which require a high level of interaction between the host and co-processor.

Routines to perform square root, sine and cosine functions have been designed and these have been used successfully in the development of a trajectory planning algorithm. This algorithm uses the co-processor to compute in parallel a large number of forward kinematics solutions and by doing so is able to convert a cartesian space trajectory into a joint space path for a redundant manipulator.

The performance of the processor has been analysed and a number of recommendations have been made concerning future implementations.

Acknowledgements

The author wishes to thank Janez Funda for his assistance with regard to the use of the simulator and Charles Martin and K.Wojtek Przytula for their invaluable help in correcting hardware related errors.

Contents

1. Introduction

The Hughes array co-processor is the result of a joint effort between Hughes Research Laboratories and the University of Pennsylvania. The hardware was developed at Hughes by Ted Carmely, Lap Wai Chow, Charles Martin, J.Greg Nash, K. Wojtek Pryztula and Dale Simpa. The software development system was created at the University of Pennsylvania GRASP Lab by Miriam Hartholtz (assembler) and Janez Funda (simulator).

This report describes the results of twelve months of research involving the Hughes array co-processor. In its initial stages this work involved the testing and debugging of the existing system (see Appendix A). Subsequent research was aimed at the development of software to allow for a high level of interaction between the co-processor and the host machine. A number of routines were developed to run on the co-processor and this work culminated in the implementation of a trajectory planning algorithm for redundant manipulators.

Section two of the report describes the machine at the hardware level and includes a detailed analysis of the major functional units. This will prove important in the investigation of errors produced by the software described later in the report.

The third section describes the addition of both a loader and a library of C-callable routines to the software development environment. The loader allows small stand-alone programs to be executed on the processor while the C-callable routines provide the means by which more complex programs may be constructed. By way of example a simple addition program is presented.

The fourth section of this report discusses some of the less obvious factors involved in developing software for the co-processor. In particular the selection of appropriate algorithms and the programming methodology involved in turning such algorithms into efficient working programs are described.

Section five describes the implementation of a square-root routine on the processor. Such a routine is required for a number of applications yet its implementation is non-trivial due to the unique hardware constraints inherent in the array co-processor.

The sixth section focuses on the development of a single routine to calculate sine and cosine values. This will form the basis for the forward kinematics solution used in the trajectory planning application.

Section seven describes the development of trajectory planning scheme for redundant manipulators. This application makes good use of the parallelism inherent in the array and also demonstrates the practicality of splitting the work-load between the array co-processor and the host machine.

Section eight provides a discussion of possible improvements to both the hardware and software of the current system. It is intended that this discussion should provide some direction for future work.

2. Hardware

The hardware component of the Hughes co-processor has already been described in some detail [1] and this section of the report is intended to complement this existing documentation. A brief description of the hardware will be presented followed by an in depth look at the current performance of the functional units within each processing element. Such an analysis is fundamental to the understanding of errors in the mathematical routines described later in the report.

2.1 An Overview of the Processor

The Hughes co-processor is composed of 256 processing elements configured in a square 16x16 array. Systolic architectures of the type originally proposed by Kung[2] are typically made up of a number of simple processing elements - each of which is dedicated to a simple task. Data is manipulated by feeding it into the array at the array boundaries. The data is then continuously processed as it moves through the array with the results appearing at the output of boundary processing elements on the far side of the array. The Hughes processor differs a little from this definition in that each processing element is a programmable arithmetic logic unit. In effect this makes the processor into what Kung[3] terms a "programmable systolic array". While this will clearly not be as efficient as a dedicated system it does have the significant advantage of allowing the same hardware to be used for a number of applications.

---

1 Przytula, K.W., Systolic Cellular System, Hughes Research Laboratories, March 1988.

2 Kung, H.T., "Why Systolic Architectures", IEEE Computer, Jan, 1982, pp37-46.

3 Kung, H.T., "On the Implementation and use of Systolic Array Processors", Proc. IEEE Int. Conference on Computer Design, Port Chester, N.Y., 1983, pp370-373.

2.2 The Connection to a Host Machine

The host machine (a Sun workstation) communicates with the co-processor through a VME connection by means of a memory-mapped interface. The processor data and program memories appear as the Sun's own memory and may be read from, or written to, directly (see Figure 2.2).

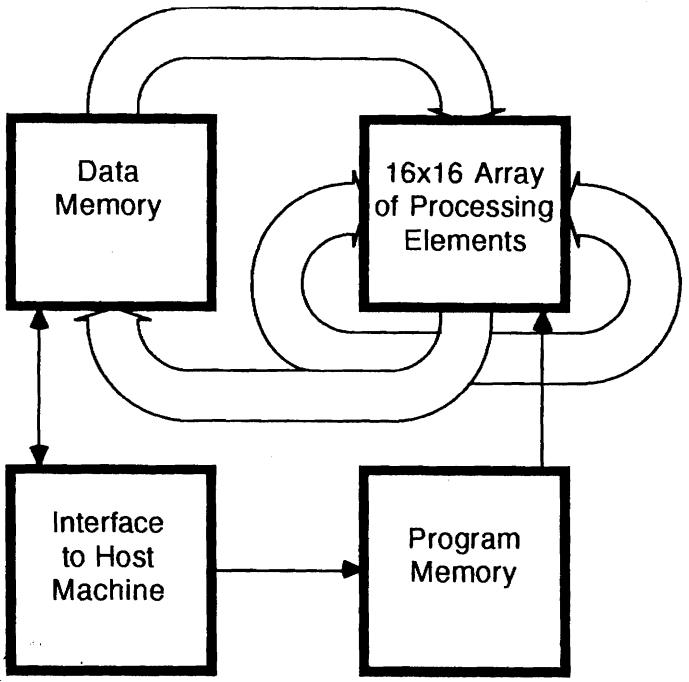


Figure 2.2 **An overview of the array co-processor and its connection to the host machine (the fifo queues are not shown in this simplified view).**

The host controls the co-processor by reading or writing specific memory addresses. These accesses are detected by the processor and interpreted as command signals. As a result the controlling program will often contain several reads/writes to the same memory location and care must be taken to ensure that an over-zealous optimising compiler does not mistakenly consider these commands to be superfluous.

2.3 The Processing Elements

Each processing element is composed of four input/output ports, 24 static registers and six functional units. These components are connected by two buses (see Figure 2.3.1).

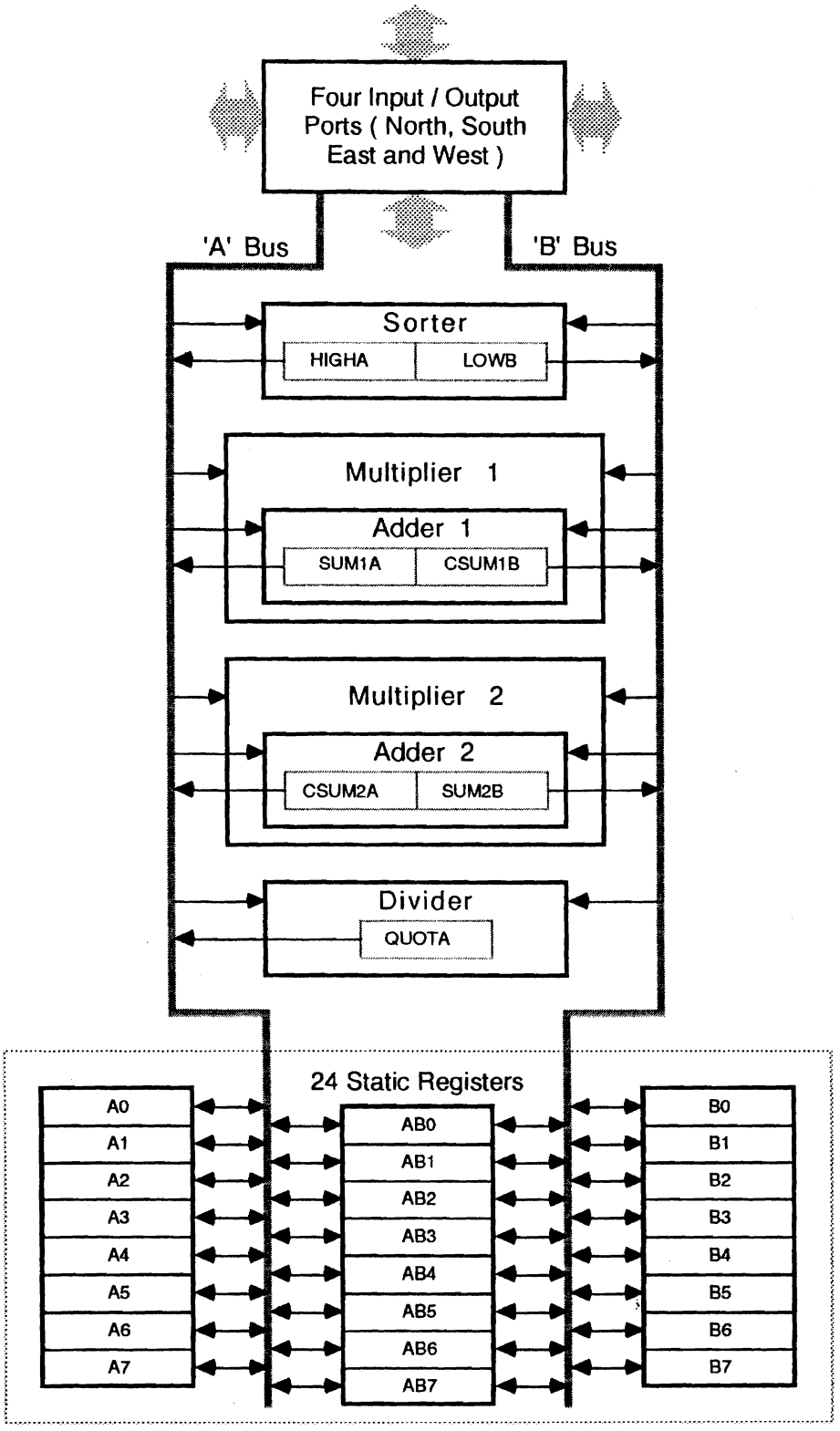


Figure 2.3.1 **Schematic representation of a single processing element showing the i/o connections, the 24 static registers and the six functional units with their internal dynamic registers.**

The input/output ports facilitate connection to each of the four nearest-neighbour processing elements in the square array and are logically termed the West, East, North and South ports. The West port of the Western-most processing elements are connected to the East port of the Eastern-most processing elements. This is logically equivalent to folding the array into a cylinder about a North-South axis. The North port of the Northern-most processing elements receives input from the data memory, while the South port of the Southern-most elements may be used to write to data memory. In most cases input data will move from the data memory into the processing array via the Northern-most elements. It will then pass through the array in a Southerly direction while being processed by the functional units within each element. The final results will eventually appear at the Southern-most processing elements from where they may be written into the data memory. More complex arrangements are also possible. For example it is possible to write results to data memory from the Southern edge of the array and then read those same results back into the Northern edge. This is logically equivalent to folding the array into a cylinder about the east-west axis.

The 24 static registers are split into three groups of eight. The first being accessible from either bus, the second only from bus A and the third only from bus B. In practice the limitation than two thirds of the registers can only be accessed from one bus is not a major inconvenience but it does mean than some care must be taken when assigning logical variables to physical registers.

The six functional units are a sorter, divider, two adders and two multipliers. The two inputs to each functional unit are provided via the two buses and the output(s) are stored in dynamic output registers within each functional unit. These dynamic registers do not retain their values indefinitely and care must be taken to ensure that results are read before they decay away. The inputs may come either from the static registers or from the dynamic output registers in each functional unit. This means that intermediate values need not be stored in static registers and may

instead be passed directly from the output of one functional unit to the input of another. For example its possible to use multiplier one to calculate the product of the previous result from multiplier one and the previous sum from the second adder.

The sorter takes one input from each bus, sorts them and stores the result in two dynamic registers. No problems were found with the sorter during testing.

The divider takes two inputs, normalizes them, performs a division and then stores the result in a dynamic register. When calculating the value of x/y it has been specified that x and y must be chosen such that the magnitude of the result is always less than 1. Tests on the divider unit were conducted using randomly chosen inputs and the results are shown in Figure 2.3.2.

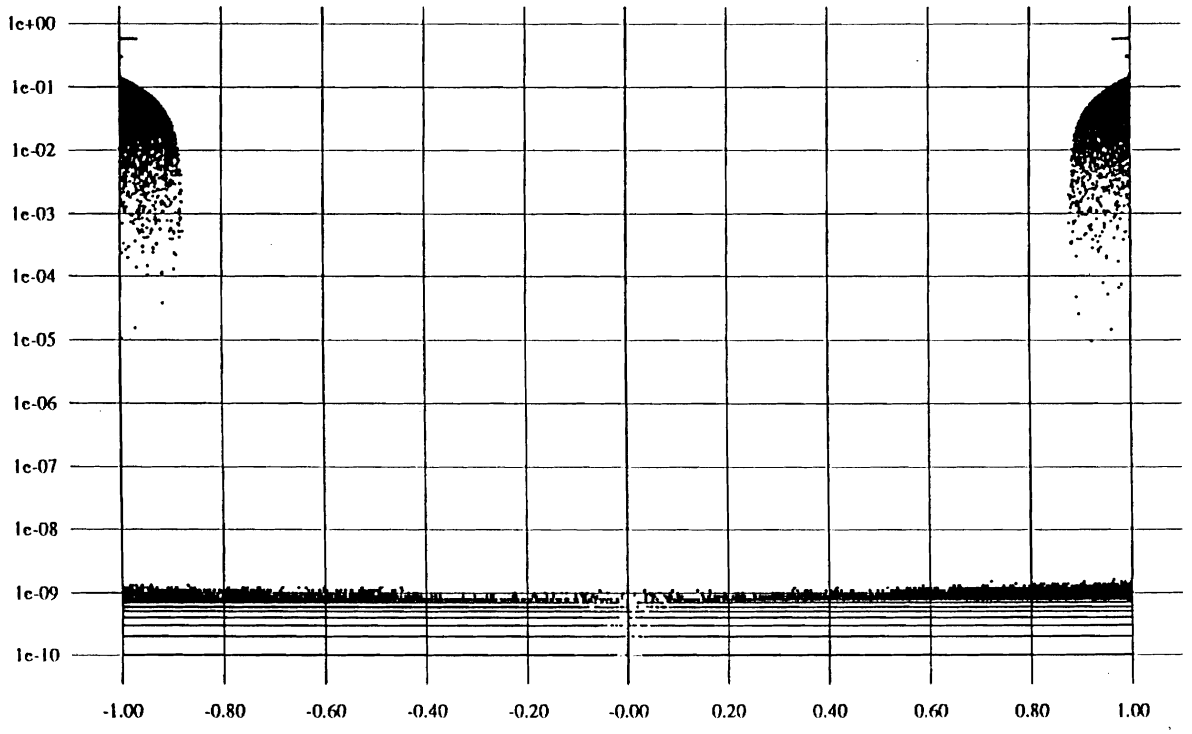


Figure 2.3.2. **A graph of expected output versus error magnitude for the divider unit in each processing element. (The bands which appear in the lower section of the graph are an artifact of the 10-digit precision to which the results were stored).**

Examination of the above graph shows that the result is unreliable whenever the magnitude of the expected result is greater than 0.875. Provided this limit is not exceeded then reasonably good results are obtained. However, even in these cases the error is still greater than might be expected.

Each of the two adders takes two inputs (one via each bus) and stores their sum and its complement in two dynamic registers which are each accessible via one of the buses. Tests on the adders showed no errors in the output.

Each of the two multipliers takes two inputs (again one from each bus) and produces two partial products. These are then summed in the adders to produce the final product and its complement which are stored in two dynamic registers. The two multipliers can be used in parallel to multiply two different pairs of numbers and this configuration was found to be particularly useful (see the sine and cosine routines for example).

Tests on the output from the multipliers failed to reveal any substantial errors however it was found that the least significant two bits of the resulting word were often incorrect.

3. Description of the Development Environment.

The development environment previously consisted of an assembler and a simulator. Since both of these have already been well documented [4,5] this section of the report will concentrate on describing the two new additions to the system: a loader and a library of C-callable routines. An example program which demonstrates the transfer of data to and from the co-processor is also presented.

3.1 The Loader Program

The loader program simplifies the task of writing and testing simple programs for the Hughes system. It can load the processor data memory with input data, load the program memory and fifo queues with instructions from an assembler-generated executable file, run the program and then create an output file by reading values from the processor data memory.

This process is set in motion by typing the command:

```
load [-h] filename
```

Where filename is the name of a control file. This file contains all the information which the loader program requires and it allows the process to be carried out independently of further operator action.

The format for the control file is:

```
executable_data_filename
input_data_filename
output_data_filename
first_row_of_output_data
last_row_of_output_data
```

---

4 Hartholz, M.A., The Systolic/Cellular System Assembler: User's Guide, Master's Thesis, The University of Pennsylvania, August, 1988.

5 Funda, J., Symbolic Simulator/Debugger for the Systolic/Cellular Array Processor, The University of Pennsylvania, January, 1991, Technical Report # MS-CIS-91-07

Where input_data_filename is the name of a file containing the data values with which the array is to be initialized. The format for this file is identical to the data file used by the simulator with the addition that data values may be in either floating-point or hexadecimal format. When called with the -h option the input file is assumed to be in hexadecimal format and the output file is created in hexadecimal format. Alternatively if the -h switch is not specified then both files are in floating-point format.

The executable_data_filename is the name of an output file created by the assembler.

Once the program has been executed the loader reads values from data memory in the range first_row_of_output_data through last_row_of_output_data and writes them to the file output_data_filename using the format specified by the presence or absence of the -h switch.

The loader is intended to facilitate the testing of simple stand-alone programs. However, to access the full power of the array co-processor it is necessary to have a finer level of control over the hardware and it is particularly desirable to have a greater level of interaction between the host and co-processor. It was to provide this additional power that a library of C-callable routines was created.

3.2 The Library of C-callable routines

The library of C-callable routines allows a programmer to have a high level of control over the co-processor without necessarily needing to have a detailed understanding of the underlying hardware. A description of these routines will be followed by an example of their use.

There is no elegant way to recover from most processor-related errors. As a result the following routines handle errors by printing an error text to stderr and then exiting with code 1.

3.2.1 map_processor_into_memory()

This routine opens the VME device driver, allocates memory space for the co-processor and maps that space into the Sun's address space. It should

be called only once at the start of any C program which wishes to use the co-processor.

3.2.2 initialise_processor()

This routine initialises the processor and should be used after the processor has been mapped into memory and before any program is run. It puts the processor into halt mode (terminating any program which may have been running). It then resets the co-processor and clears the three fifo queues. The program and data memory is unaffected by this operation.

3.2.3 print_status()

This routine prints the current status of the processor. This includes the full/empty condition of each fifo queue as well as the paused/running status of the program.

With long programs this routine may be used to watch the fifo queues empty during program execution. However, some care should be exercised in the interpretation of the output because of the speed differential between the co-processor and the host. For example with small programs it is possible for the co-processor to complete execution before the print_status routine has had time to print the machine's condition.

3.2.4 load_data_word(location, data)

This function loads a data word into a specified location in the processor data memory. The location number is an offset from the start of data memory (zero being the first location) measured in a row-wise followed by column-wise format. For example location 18 would be the third column of the second row of the data memory.

Access to the data memory is simplified since this function insulates the user from the need to know either where the processor is mapped into memory or how that mapping is performed.

For efficiency reasons there is no run-time checking of the validity of the specified location. If a location outside of the data memory is specified then unpredictable behaviour may result.

3.2.5 read_data_word(location)

This is the inverse of the load_data_word function. It takes a location in data memory and returns the contents of that location.

3.2.6 load_data(filename)

This routine takes as its argument the pointer to a character string containing a filename. It opens the specified file and reads the contents into increasing data memory locations starting at location zero. The format for the input file is the same as that for the simulator with the exception that input numbers may be in either hexadecimal or floating-point format. This choice being controlled by the truth or falsity of the global variable use_hex_numbers.

By permitting data values to be read from a file this routine allows the user to change the input data without the necessity of re-compiling the controlling program.

The routine makes use of the fact that data memory locations are sequential and it is thus more efficient to use this routine than to use repeated load_data_word() function calls to load data into memory.

3.2.7 load_program(filename)

This routine takes as its argument a pointer to a character string containing the name of an assembler-generated executable file. It opens the specified file and uses the contents to load the read, write and program fifos with addresses and the program memory with instructions. It also checks for overflows in either fifo or program memory and prints the statistics for the current program to stdout.

The compiled code on the host depends only on the filename of the co-processor executable file. As a result the user has the ability to modify and re-assemble the co-processor code independently of the host program.

3.2.8 create_output_file(filename, first_row, last_row)

This routine takes as its argument a pointer to a string containing the name of the output file which is to be created. It then reads data values from data memory starting with those in the first_row and ending with those in the last_row. These values are written to the output file using an appropriate format as specified by the use_hex_numbers global variable.

3.2.9 start_processor()

This function resets the pointers to the read, write and program fifo queues. It then loads the starting address into the program counter and starts the co-processor running.

The routine does not wait until the co-processor has completed execution - it merely starts it running. This gives the user the opportunity to run other code while the co-processor is executing. In practice most programs are sufficiently short that little useful work could be achieved and so a typical implementation would be:

```
start_processor();
while( RUNNING )
      ;
```

Where the while loop enforces a wait until the co-processor has finished executing the current program. (The macro RUNNING is described below)

3.2.10 Assert(control_line)

This macro causes the specified control line to be activated. The addresses of all valid control lines are defined in the header file processor.h using the same terminology used in the hardware manual [6].

For example, the write fifo queue can be reset using the command:

```
Assert( W_FIFO_RESET );
```

In practice it is not usually necessary to make use of such low-level commands.

---

6 Przytula, K.W., Systolic Cellular System, Hughes Research Laboratories, March 1988.

3.2.11 dtofx(number)

This macro takes an input of type double and converts it into an unsigned integer using the fixed-point format utilised by the co-processor.

For example to load 1.0 into the first data memory location one could use either:

```
load_data_word(0, 0x40000000 )
```

or:

```
load_data_word(0, dtofx(1.0) )
```

3.2.12 fxtod(number)

This is the inverse of dtofx. It converts a number in fixed-point format into a floating-point equivalent of type double.

3.2.13 Macros to Test the Processor Status

Eight macros have been defined which return true/false (0/1) values. Each depends on a single bit in the processor status register and their use allows the programmer to determine the empty/full state of each fifo queue as well as the paused/running state of the co-processor. The only one which is commonly required is RUNNING which returns true if the co-processor is currently executing code and false otherwise.

3.3 An Example Program

By way of example consider a program to add 256 pairs of numbers together. While this would clearly not be a very effective use of the system it does serve to illustrate the transfer of data to and from the co-processor.

The program to perform the addition on the co-processor is as follows:

```
DEFQUEUE Q1 32;
DEFQUEUE Q2 16;

      NOP;
      READQ Q1;
      WRITEQ Q2;

L0:   GETNR(A1,A1);           read in A
      LOOP 15 L0;

L1:   GETNR( B1,B1);          read in B
      LOOP 15 L1;

      ADDD( A1, B1);          add A and B

      MOV( :SUM2B,AB0); store result

L2:   GETNW( AB0,AB0);  and write it to memory
      LOOP 15 L2;

      STOP;
      END;
```

Assuming that this is stored in the file add.a then it may be assembled using the command:

```
% xscs add.a add.exe
```

This causes the assembler to generate an executable output file with the name add.exe.

At this point the user could create data and control files and then load and execute the program using the loader. However, while being adequate for some applications, the level of interaction between the host and co-processor afforded by this scheme would be minimal. A much more powerful implementation inovolves writing the following C program using the library routines.

The code required to map the co-processor into memory, initialise it and then load the three fifo queues and the program memory with data is as follows:

```
map_processor_into_memory();
initialise_processor();
load_program( "/usr/users/sayers/hughes/tests/add.exe" );
```

The numbers to be added may then be written to the data memory of the co-processor using the following C commands:

```
for( i=0; i<512; i++)
    load_data_word(i, dtofx( (double)i/512) );
```

It should be noted that it is the responsibility of the user to ensure that the memory locations accessed by the C program correspond to those used by the program on the co-processor. For example in this case the co-processor reads from the first 32 rows (512 locations) and writes to the following 16 rows (256 locations).

Once the program and data have been loaded the program may be executed with the code:

```
start_processor();
while( RUNNING )
    ;
```

And the results may then be read and printed out with the code:

```
for(i=0; i< 256; i++)
    printf(" Result of addition # %d is %f\n", i,
                        fxtod( read_data_word(i+512)) );
```

At this point the program could simply terminate or it could modify the input data and/or the program and then re-start the co-processor. In many applications it is expected that a single program and all constant input data would initially be loaded. The co-processor could then be started many times with only the variable input data being written to the co-processor prior to each invocation.

4. Writing Software for the Hughes Processor

4.1 Overview

Perhaps the most difficult aspect of the software development process is the determination of a suitable implementation. In the traditional systolic architecture the hardware is a direct implementation of the chosen algorithm [7]. This means that the engineer can effectively start each design with a blank sheet of paper - the constraints on the design imposed by hardware considerations being only limited by what is physically realisable. The disadvantage with this method is that each design requires specialised hardware which, once created, is difficult to modify or adapt to other problems. The Hughes processor seeks to overcome this limitation by employing a programmable system where there is software control over the implemented function. Unfortunately this additional flexibility does not come without cost and the catch is that it is a non-trivial process to map existing algorithms to a fixed hardware structure.

The array may be used in a number of different logical configurations. One possibility is to simulate a systolic architecture with each processing element performing the tasks of one component in that design. The advantage of this is that data may be processed continuously as it moves through the array. (In some cases data transfer can be performed in parallel with multiplication or division operations - thus i/o can be performed with no time penalty.) The disadvantage is that the systolic architecture being implemented may not map well on to the 16x16 processor array. There may also be some loss in parallelism if different components of the array are required to perform different tasks since in this case some masking of the array would almost certainly be required. It may also be necessary to artificially introduce delays into the system in order to have all the multiplications/ divisions occurring simultaneously.

---

7 Kung, H.T., "Why Systolic Architectures", IEEE Computer, Jan, 1982, pp37-46.

An alternative is to treat the array as a SIMD (single-instruction multiple-data) machine with 256 parallel processors. This implementation is particularly efficient since all the processors are busy all the time. The disadvantage is that some time may be wasted transferring data into, and results out of, the array - though this is mitigated to a certain extent by the extremely high bandwidth connection between the array and data memory.

Another possibility is to treat the array as sixteen parallel pipelines (or one-dimensional systolic machines). It would even be possible, though not particularly efficient, to use the array to simulate a 256-stage linear array.

4.2 Optimising the code

When composing code for the processor it is possible to take account of optimisations as the code is written however the resulting code is difficult to read and debug. An alternative method is to write the code without considering any optimisations and then wait until a working version exists before attempting any improvements. This however has the disadvantage that substantial changes to the coding may be necessary in order to realize relatively minor speed improvements. It was found in practice that a combination of the above two techniques usually produced the best results in terms of both programming and execution time.

There are two main levels at which improvements can be achieved. At the higher level there are global efficiency considerations (such as the average time each processing element spends performing useful work). Improvements at this level generally require the selection of an alternative algorithm which maps more cleanly onto the fixed array.

At the lower level it is possible to achieve some speed improvement by optimising the coding of the algorithm. There are three main areas where these gains can be realised:

Firstly, the use of static registers for temporary storage of results can be minimised by making use of the fact that the dynamic output registers within each functional unit can be used directly as inputs for successive operations.

Secondly, the use of multiplication and division operations may be optimised by rearranging equations to minimise the number of divisions and, wherever possible, to allow two multiplications to be performed in parallel.

Thirdly, it is often possible to replace the NOP instructions performed during multiplication and division operations by instructions which perform useful work. It should be noted that the dynamic output registers for these operations do not need to be read as soon as the results are available and so it is possible to replace NOPs by instructions which may take slightly longer to execute.

5. Description of the Square Root Routine

The aim of this section is to present an efficient algorithm for determining square roots using the processing element in the Hughes systolic array co-processor.

5.1 Overview

It is possible to calculate the square root of any number by hand using a method similar to long division. This manual method can be readily translated into a binary format with the resulting algorithm being particularly well suited to computer implementation [8]. An alternative method, described by Johnson [9] is to use a binary search technique to locate the square root. Yet another method is to consider finding the square root of a number to be equivalent to finding the root of the equation $f(x)=x^2-y$. This equation may then be solved using an iterative method [10].

The decision as to which of these alternative methods to use was effectively pre-determined by the unique hardware design of the processing element. Since each element lacks any conditional, bit-wise, logical or table-lookup operations the only method which may be implemented efficiently is the iterative solution.

It may be possible to distribute the calculation of a square root across the systolic array (for example Majithia and Kitai [11] describe a cellular array for computing square roots) however in the present situation it is desirable

---

8 Cowgill, D., "Logic Equations for a Built-In Square Root Method", IEEE Transactions on Electronic Computers, April 1964, pp156-157.

9 Johnson, K.C., "Algorithm 650: Efficient Square Root Implementation on the 68000", ACM Transactions on Mathematical Software, Vol.13, No.2, June, 1987, pp138-151.

10 Borse, G.J., Fortran 77 and Numerical Methods for Engineers, PWS Engineering, U.S.A, 1985, pp358-361.

11 Majithia, J.C. and Kitai, R., "A Cellular Array for the Nonrestoring Extraction of Square Roots", IEEE Transactions on Computers, December, 1971, pp1617-1618.

for each processing element to be capable of independently calculating the square root function. Such an implementation allows a large number of square roots to be computed in parallel while simultaneously reducing the amount of communication required between processing elements. This section will therefore consider only the case where a single processing element is required to calculate the square root of a single number.

5.2 Algorithmic Description

For most non-linear functions the computational solution is obtained by some form of non-iterative approximation. However, in the case of the square root function a particularly simple and very fast iterative technique exists. This method is known by various titles including Newton's method and the Newton Raphson method.

Newton's method is a general technique for finding the root of an equation and may be written:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \quad \text{.....(5.1)}$$

Where x_k is the value of x after k iterations (x_0 is the initial guess) and f and f' are the function and its derivative for which the root is required.

In the case of the square root function:

$$f(x) = x^2 - y \quad \text{.....(5.2)}$$

Clearly this will equal zero only when $x = \sqrt{y}$

Substituting into (5.1) yields:

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{y}{x_n} \right) \quad \text{.....(5.3)}$$

Intuitively it seems clear that this iterative formula should always converge (provided the initial guess is positive) and it has been shown [12] that this convergence is quadratic.

---

12 Bajpai A.C., Mustoe L.R. and Walker D., Engineering Mathematics, Wiley, USA, 1982, p352.

Perhaps the most common method for implementing this iterative technique in software is to use the following algorithm:

i) Perform range reduction

ii) Choose an apropriate initial guess.

iii) Solve iteratively until a desired degree of accuracy is obtained

iv) Transform the result to nullify the effects of step (i).

5.3 Range Reduction

If the input numbers are permitted to vary over a very wide range then it becomes difficult to make an accurate guess as to the square root of the number. This increases the number of iterations required in step (iii) and consequently slows down the routine. It is therefore common to reduce the range of the input numbers.

This range reduction is typically performed by shifting the input number by some appropriate number of bits until the input falls within some desired range (say [1/4,1]). Such a reduction is particularly easy to nullify after the square root has been performed since the result can simply be shifted in the reverse direction by half as many places.

The absence of any shift operations makes it difficult to perform range reduction in this way using the Hughes processor. Even if the shift operations were replaced by multiplication/division operations it is not clear how one could determine which number to multiply/divide by and even if this could be determined it would not be easy to nullify the effects of such an operation after the square root had been performed.

Therefore, while there is definitely an advantage to performing range reduction, it is not practical to use it in this implementation. The range of possible input numbers is therefore $[2^{-30},2)$. One side effect of this decision is that the choice of an accurate initial guess becomes even more important.

5.4 Choosing an Initial Guess

Determining a suitable method for finding an initial guess is a compromise between two alternatives. If a very simple method is used (for example just

using a constant) then a relatively large number of iterations will be required. Alternatively, if a more complex method (for example a 3rd degree polynomial approximation) is used then this will reduce the number of iterations but the savings in iterative calculations may be overshadowed by the increased calculation required for the initial guess. In this case a first-order approximation was considered to provide the best compromise.

There are several alternative methods for determining "optimal" values for the two constants in the first-order approximation. For purposes of comparison consider an approximation for a routine designed to cope with numbers in the range [0.25,1].

5.4.1 Approximation using Minimax Methods

One way of determining a square root approximation is to minimise the maximum relative error between the approximation and the square root function.

If the approximation is of the form:

$$P(x) = A+Bx \quad \text{.....(5.4)}$$

then the relative error is:

$$E(x) = \frac{A + Bx - \sqrt{x}}{\sqrt{x}} \quad \text{..... (5.5)}$$

Suitable values for A and B are required in order to minimise:

$$\max_{0.25 \le x \le 1} \left| \frac{A + Bx - \sqrt{x}}{\sqrt{x}} \right| \quad \text{..... (5.6)}$$

Solving this using minimax techniques [13] gives the approximation [14]:

$$P(x) = 0.3431 + 0.6863\,x \quad \text{.....(5.7)}$$

---

13 Fike, C.T., Computer Evaluation of Mathematical Functions, Prentice Hall, U.S.A., 1968, pp75-76.

14 Eve, J., "Starting Approximations for the Iterative Calculation of Square Roots", Comput. J., October, 1966, pp274-275.

5.4.2 Approximation using Moursund's Method

Moursund [15] has shown that the above-described result may be improved by considering the relative error after m iterations of Newton's method (as opposed to the relative error at the output from the approximation step).

Thus, if $N_m(y)$ is the result after m iterations of Newton's method using a starting approximation of y then values for A and B are required in order to minimise:

$$\max_{0.25 \le x \le 1} \left| \frac{N_m(A + Bx) - \sqrt{x}}{\sqrt{x}} \right| \quad \text{.....(5.8)}$$

It turns out that the values of A and B remain constant as m increases (assuming $m \geq 1$) and Moursund has tabulated values for a number of different situations. For an input range of [0.25,1] the approximation is:

$$P(x) = 0.34329 + 0.68659x \quad \text{.....(5.9)}$$

5.4.3 Approximation using Absolute Error Method

Both of the above two methods aim to minimise relative errors. However, in a computer implementation it is desirable to have a routine which produces answers accurate to the last bit and this implies that it is the absolute, rather than relative, errors which are important.

Hence we require values for A and B which minimise:

$$\max_{0.25 \le x \le 1} \left| N_m(A + Bx) - \sqrt{x} \right| \quad \text{.....(5.10)}$$

A simple, but computationally intensive, technique for determining the values for A nd B in this equation is to test a number of different values for

---

15 Moursund, D., "Optimal Starting Values for Newton Raphson Calculation of $\sqrt{x}$", Communications of the A.C.M., Vol.10, No.7, July, 1967, pp430-432.

A and B until a suitable result is obtained [16]. In essence the algorithm is as follows:

(i) Step A through a range of values

(ii) Step B through a range of values

(iii) Determine an approximate value for:
$$\max_{0.25 \le x \le 1} \left| N_m(A + Bx) - \sqrt{x} \right|$$
by testing the procedure at discrete values of x.

(iv) If the absolute error for this choice of A and B is lower than the best known then remember these values.

v) Output the values for A and B which produced the lowest absolute error.

It should be noted that in this case the "optimal" values will change as the number of iterations is altered. Approximate values for A and B using the above method are:

$P(x) = 0.3486 + 0.6766\ x$ (for m=1)

$P(x) = 0.3459 + 0.6816\ x$ (for m=2)

This method only produces approximate answers for A and B since it only tests at discrete points. Nevertheless it produces values which appear to perform better than either of the two alternative methods (in terms of absolute accuracy of the final result).

A comparison of the three methods is shown in Figures 5.4.3a and 5.4.3b.

---

16 This is similar to the technique used by Andrews [Andrews, M., "Mathematical Microprocessor Software: A $\sqrt{x}$ Comparison", IEEE Micro, May, 1982, pp63-79.] although in that case the numerical precision was much lower and so it was practical to test every possible input number.

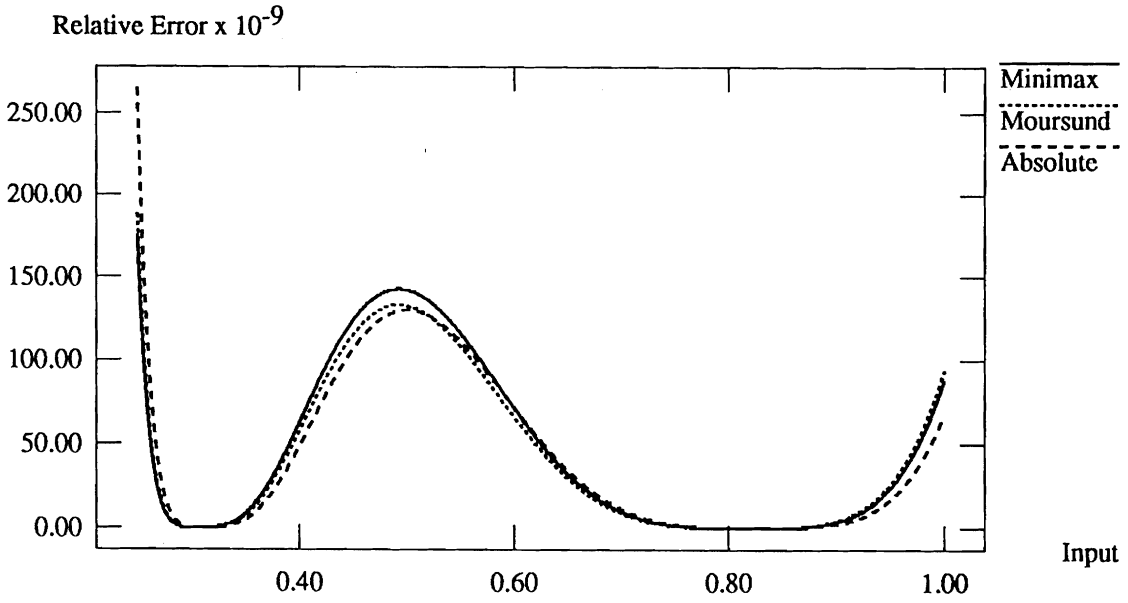


Figure 5.4.3a **Comparison of relative error after two iterations with three different starting approximations.**

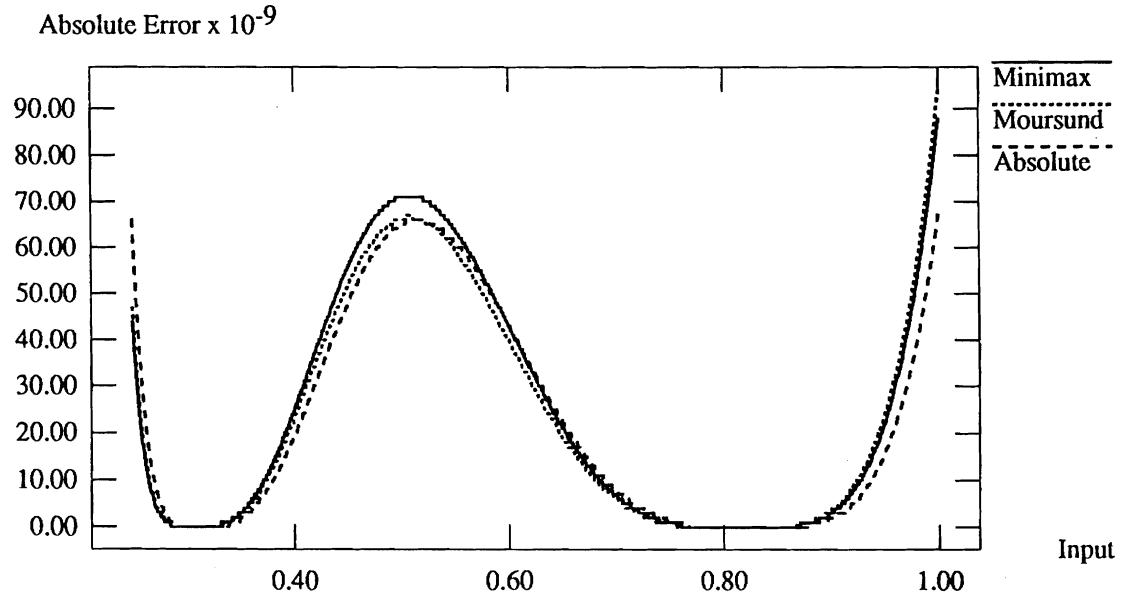


Figure 5.4.3b **Comparison of absolute error after two iterations with three different starting approximations.**

Examination of these graphs shows that, at least in terms of absolute error, there is some advantage in using the absolute error method of determining initial approximations. In this example the difference between the three methods is small, however it may be expected that these difference would be magnified if the range of input numbers were to be greatly increased.

As a consequence of the above results the absolute error method was considered the most suitable for use in determining the optimal starting approximation for the case where input numbers fall in the range $[2^{-30},2)$. It would have been desirable to use the array processor to perform the calculations in step (iii) of the absolute error method. This would have meant that the final approximation was optimised to account for any aberrations in the arithmetic used by the processor and it would also have allowed a large number of different starting approximations to be compared quickly. However, concern over the accuracy of some of the arithmetic operations performed by the processor meant that there was some advantage in determining the starting approximation in a manor which was relatively independent of the processor hardware. It was therefore decided that the starting approximation should be determined by simulating the numerical operations using a Sun workstation. Once the best approximation had been determined the performance of the processor could be analysed using it and other values.

Applying the absolute error technique to input numbers in the range $[2^{-30},2)$ after eight iterations of Newton's method (fewer iterations did not produce an acceptable level of accuracy regardless of the constants used in the approximation) yielded the results shown in Figure 5.4.3c.

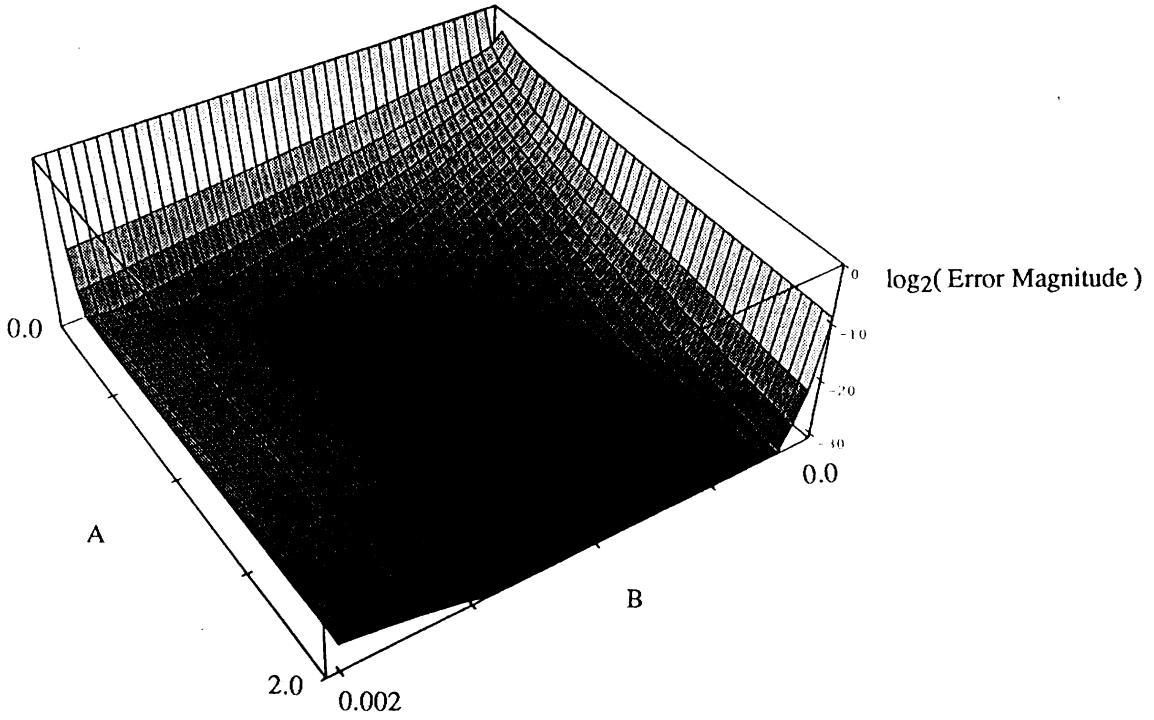


Figure 5.4.3c Graph of accuracy versus starting approximation values after eight iterations of Newton's method.

Examination of these results shows that the required degree of accuracy (an absolute error of less than 2^{-31}) may be obtained by choosing any constant values which fall within the highlighted region. The chosen approximation was 0.999x+0.00075. This falls within the desired accuracy range and also has the advantage that the initial approximation will never be greater than the largest input number - an important consideration for a fixed-point machine.

5.5 The Complete Algorithm

The complete algorithm for determining the square root of a number in the range $[2^{-30},2)$ using a single processing element in the array processor is therefore:

1) Find an approximation using the formula:

$$x_0 = 0.999.y + 0.00075$$

2) Perform 8 iterations using the formula:

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{y}{x_n} \right)$$

This algorithm could be translated directly into the machine language for the processor, however the internal architecture of the machine means that some further refinements may be introduced.

Firstly, since division operations take more than twice as long as multiplications, the algorithm may be re-written:

1) $x_0 = 0.999y + 0.00075$

2) $temp_1 = y*0.5$

3) perform 8 iterations using the formula:

$$x_{n+1} = 0.5*x_n + \frac{temp_1}{x_n}$$

Secondly, the internal parallelism within the processor can be used to advantage by re-writing the equations as:

1) Perform in parallel: $temp_1 = y*0.5$ $temp_2 = 0.999y$

2) $temp_2 = temp_2 + 0.00075$

3) Do the following eight times:

   i) Perform in parallel: $temp_3 = \frac{temp_1}{x_n}$ $temp_4 = 0.5 * x_n$

   ii) $x_{n+1} = temp_3 + temp_4$

The assembly language form for this routine is given in Appendix B.

5.6 Results and Discussion

The above-described routine was tested by using it to calculate the square root of a number of different inputs and the results of this are shown in Figure 5.6.

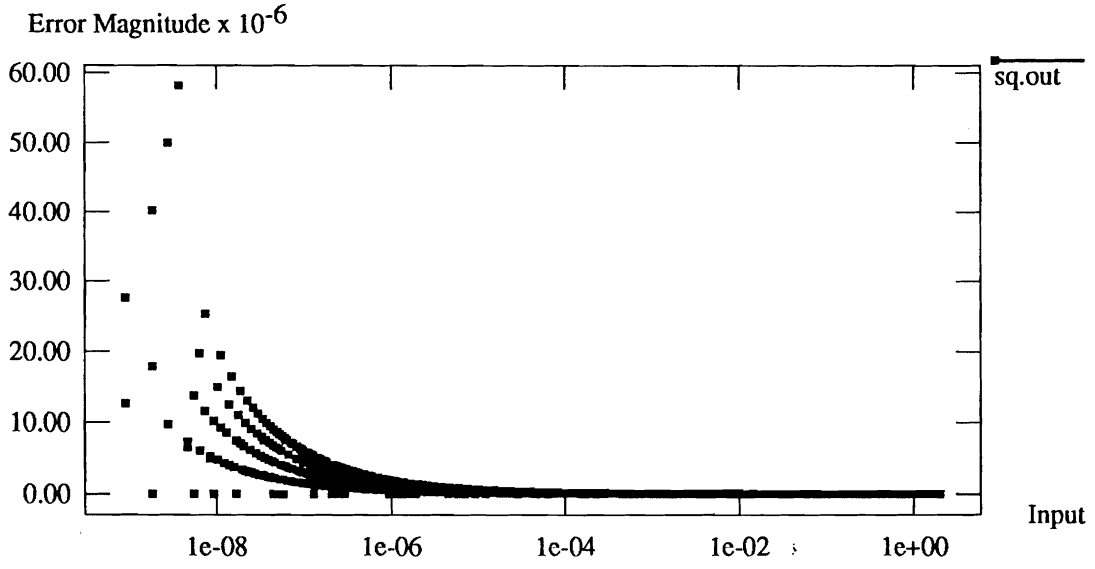


Figure 5.6 **The magnitude of the error in the output of the square root routine.**

Analysis of the results shows that for relatively large input numbers the accuracy is very good and this correlates well with that expected based on simulations performed using the Sun. As the input numbers decrease the errors increase and also there appear to be definite bands into which the errors fall. These errors were traced to errors in the arithmetic used by the processor (see Section 2.3). In many cases these inaccuracies have negligible effect since the iterative method is inherently self-correcting; however for very small input numbers the error (particularly that associated with the multiplication of x by 0.5 in the first step of the algorithm) can be larger than the number itself. This error is magnified by the square root algorithm and this leads to the errors in the final result.

Attempts to improve the result by using alternative values for the starting approximation were unsuccessful - while it was possible to produce poorer results by choosing worse starting values it was not possible to find a starting approximation which produced a better result. Attempts to improve the accuracy by changing the way in which each iteration was calculated or by increasing the number of iterations were also unsuccessful.

6. Description of the Sine and Cosine Routine

The aim of this section is to describe the development of a single routine to calculate both the sine and cosine of a given input number. This routine will form the basis for a more complex program described later in the report.

6.1 Overview

The standard implementation of sine and cosine routines is to perform some range reduction on the data first [17]. The required sine/cosine values can then be computed using either a polynomial approximation or table-lookup.

Unfortunately, it is not possible to perform range reduction using the processing element within the Hughes processor. As a result a considerably larger polynomial approximation is required.

6.2 Algorithmic Description

Using Mclaurin's expansion [18]:

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots + (-1)^{(n-1)} \frac{x^{(2n-1)}}{(2n-1)!} \quad \text{.....(6.1)}$$

It was estimated that truncating this series after the eighth term would provide a reasonable compromise between execution time, register usage and numerical accuracy.

Now, since the machine can't store numbers outside the range [-2,+2) it is necessary to perform some scaling of the input values. A scaling factor of π provides the elegant result that the input numbers fall within the range

---

17 Fike, C.T., Computer Evaluation of Mathematical Functions, Prentice Hall, U.S.A., 1968, p41.

18 Bajpai A.C., Mustoe L.R. and Walker D., Engineering Mathematics, Wiley, USA, 1982, pp362-363.

[-1,+1]. Thus, using 'a' as the scaled input value, the series may be formulated as:

$$\sin(x) = \pi a - \frac{(\pi a)^3}{3!} + \frac{(\pi a)^5}{5!} - \frac{(\pi a)^7}{7!} + \cdots - \frac{(\pi a)^{15}}{15!} \quad \text{.....(6.2)}$$

$$= \pi\left(a - \frac{\pi^2 a^3}{3!} + \frac{\pi^4 a^5}{5!} - \frac{\pi^6 a^7}{7!} + \cdots - \frac{\pi^{14} a^{15}}{15!} \right) \quad \text{.....(6.3)}$$

Re-arranging this so that none of the constants fall outside the acceptable range results in the formulation:

$$\sin(x) = \sqrt{\pi}\ \sqrt{\pi}\left(a - \frac{\pi^2}{3!} a^3 + \frac{\pi^4}{5!} a^5 - \frac{\pi^6}{7!} a^7 + \cdots - \frac{\pi^{14}}{15!} a^{15} \right) \quad \text{.....(6.4)}$$

Which may be re-written as:

$$\sin(x) = B\Big(B\big(a + C_3 a^3 + C_5 a^5 + C_7 a^7 + \cdots + C_{15} a^{15} \big) \Big) \quad \text{.....(6.5)}$$

Where $B = \sqrt{\pi}$, $C_3 = -\frac{\pi^2}{3!}$, $C_5 = \frac{\pi^4}{5!}$, $\cdots$, $C_{15} = -\frac{\pi^{14}}{15!}$

In this case all constants have magnitude less than 2 and there is the added advantage that all divisions have been eliminated from the run-time routine. A further improvement may be realised by re-arranging the polynomial into a more computationally efficient form:

$$\sin(x) = B\Big(B\big((\cdots(((C_{15} a^2 + C_{13})a^2 + C_{11})\cdots)a^2 + C_3)a^2 + a \big) \Big) \quad \text{....(6.6)}$$

Similarly the cosine algorithm may be formulated as:

$$\cos(x) = B\Big(B\big((\cdots(((C_{14} a^2 + C_{12})a^2 + C_{10})\cdots)a^2 + C_2)a^2 \big) \Big) + 1 \quad \text{....(6.7)}$$

Where $B = \sqrt{\pi}$, $C_2 = -\frac{\pi}{2!}$, $C_4 = \frac{\pi^3}{4!}$, $\cdots$, $C_{14} = -\frac{\pi^{13}}{14!}$

In the cosine routine there is the possibility that, for the case where the input is of magnitude very close to 1, the value of an intermediate result may have a magnitude greater than 2. To prevent such a situation from occurring the magnitude of C_{14} was reduced slightly. This causes a minor reduction in the accuracy of the output for large input numbers but it ensures that no overflow will occur. A value for C_{14} of -0.000031 was found empirically to provide an acceptable result.

It would be possible to calculate the sine and cosine values independently, however, since each processing element has the ability to perform two multiplications in parallel there is a considerable advantage to be gained from interleaving the calculations.

Using eight terms for each series proved to be advantageous in terms of coding in that it was possible to split the 16 constants between the A and B registers. This left all of the AB registers free for holding input and output values as well as partial results. Such a mapping allows a number of sine/cosine pairs to be evaluated without any need to reload the constants into each processing element.

The resulting code is given in Appendix C.

6.3 Results and Discussion

The sine and cosine routines were tested by comparing their output with that produced by a Sun 3/240. Figure 6.3 graphs the input against the magnitude of the output error for both sine and cosine calculations.

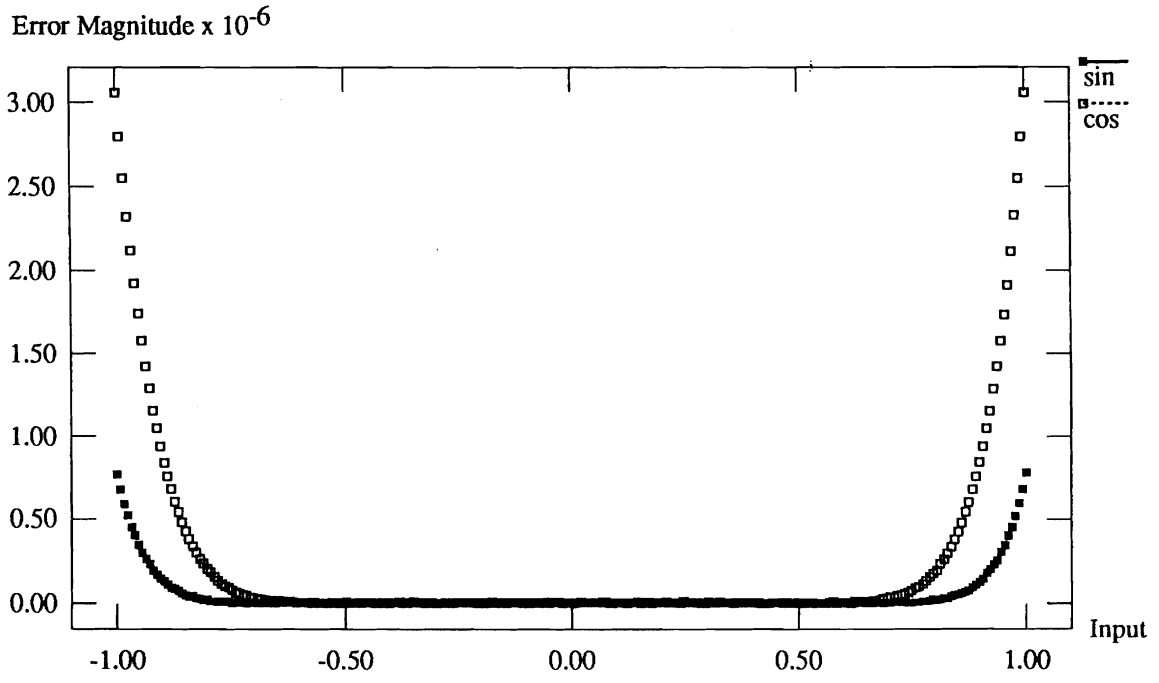


Figure 6.3. **A graph showing the magnitude of the output error for both the sine and cosine routines.**

These results indicate a maximum error of magnitude less than 0.000004 which, while being somewhat less than the maximum accuracy possible on the machine, is nevertheless still adequate for the majority of applications.

The difference in accuracy between the sine and cosine results is an artifact of the differing formulae used for each. In the case of the sine routine the eight terms translate to odd powers of x between x^1 and x^{15}; while in the cosine routine eight terms translates to even powers of x between x^0 and x^{14}. If an increased accuracy were required then a greater number of terms could be used or alternatively a better polynomial approximation could be developed.

7. Application to Trajectory Planning for Redundant Manipulators.

At present there are a number of areas in Robotics where the computationally-intensive nature of the calculations makes it difficult to perform real-time evaluation. These include "optimal" trajectory planning, obstacle avoidance and the control of redundant manipulators [19].

The application of the array co-processor will, in some cases, allow existing algorithms to be performed more quickly. However, a more significant advantage is that it will enable the application of simpler, but more computationally demanding, solutions.

The aim of this section is to describe the application of the co-processor to one such area - trajectory planning for redundant manipulators.

7.1 Overview

With a two-link planar manipulator it is possible to determine an inverse kinematics solution directly from a given cartesian end-effector position. However, if a third planar link is added then there are now an infinite number of solutions for many cartesian points within the workspace.

For such a manipulator it is very easy to find a solution but very difficult to find a "good" solution. Where "good" in this case refers to some desirable feature such as avoiding singularities, dodging objects or minimising travel time.

One possibility, proposed by Yoshikawa [20], is to formulate a solution in terms of some primary constraint (for example following a given cartesian trajectory) and then further restrict that solution by specifying some

---

19 Zhang, Y., Redundancy Control of a Robot Manipulator using a Systolic Array Processor, PhD Thesis, The Department of Mechanical Engineering and Applied Mechanics, The University of Pennsylvania, 1989.

20 Yoshikawa, T., Foundations of Robotics: Analysis and Control, M.I.T.Press, England, 1990, pp244-257.

additional constraint (such as singularity avoidance). The result of this task decomposition is a direct formula for the joint velocities.

The method proposed by the author is simpler, but possibly more computationally demanding, than the task decomposition scheme. The system is based on the premise that for many robots it is considerably easier to calculate the forward kinematics solution than it is to find its inverse. It also relies on the fact that, given a joint-space trajectory, it is not too demanding to find appropriate joint velocities to track that trajectory. Taking these factors into account led to the following algorithm:

i) Given the current position of the robot in joint-space create a set of points to which it could move if each of the joints were moved some small discrete amount.

ii) For each of these points, calculate a forward kinematics solution. The result of this step is that the system knows a range of positions in cartesian space to which the robot could move.

iii) Eliminate from this set of positions any which lie outside of the desired trajectory path.

iv) Now choose from among the remaining points based on some other criteria such that only a single "best" point remains.

v) That point becomes the next point on the robot's path and the process is repeated from step (i) until the entire trajectory has been followed.

Note that this routine makes no assumption concerning the existence of a direct inverse kinematics solution - it generates a joint-space trajectory from a cartesian space trajectory while only using forward kinematics.

The above-described algorithm is well suited to a parallel implementation with steps (i)-(iv) all being amenable to parallel computation. However, taking into account the capabilities of the array processor, it was considered that a more efficient implementation would be to have the array co-processor compute the forward kinematics solutions in step (ii) while the host machine performed the remaining calculations.

The implementation thus decomposed into two separate areas. Firstly a forward kinematics solution had to be implemented using the co-processor

and then the remainder of the algorithm had to be implemented on the host machine.

7.2 Implementation on the Co-processor

The forward kinematics solution for a three-link manipulator (see Figure 7.2) can be found by first evaluating a transformation matrix for each link and then computing the dot-product of those matrices. It was initially intended that the co-processor, being well suited to matrix operations, should perform the calculation using this scheme. However, attempts to implement the algorithm using the co-processor showed up two significant problems. Firstly, the size of the problem meant that it did not map well to the 16x16 array. Secondly, in calculating the dot-products some effort was spent performing calculations which were not actually required (for example adding 0 to a number). These calculations could not be eliminated since doing so would remove the symmetry which allowed the dot-product routine to be implemented in a systolic fashion.

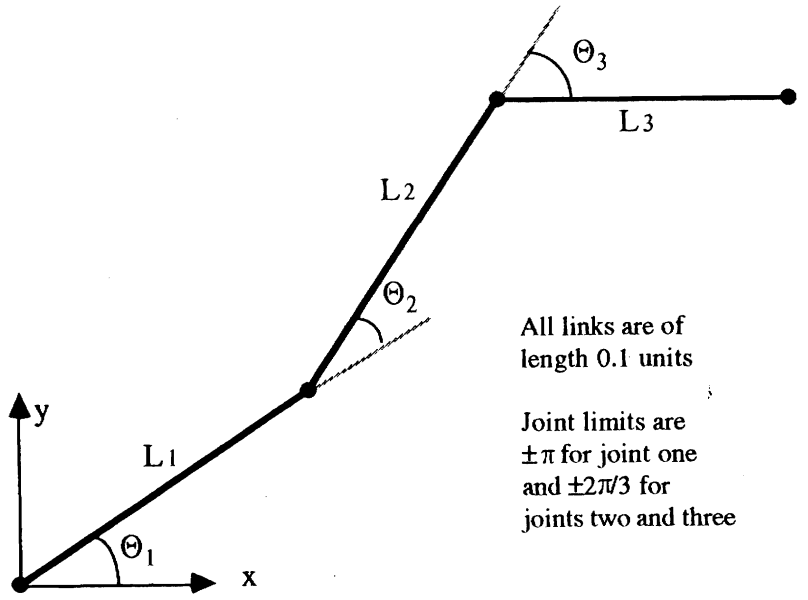


Figure 7.2. The three-link planar manipulator.

An alternative implementation was therefore considered in which each processing element within the array was used to calculate the forward kinematics solution at one point in joint-space. This allowed up to 256 points to be computed in parallel without the need for any masking of the array. It also allowed the forward kinematics equations to be simplified symbolically - thereby reducing the total amount of computation required to produce each solution point.

Using the usual notation for transformation matrices [21] the position of the end-effector with relation to the base co-ordinate frame may be represented by:

$$T_3 = A_1.A_2.A_3 \quad \text{.....(7.1)}$$

This may be simplified to give the following formulation:

$$r_{11} = r_{22} = c_1.c_2 - s_1.s_2 \quad \text{.....(7.2)}$$

$$r_{14} = L_1.c_1 + L_2.r_{11} \quad \text{.....(7.3)}$$

$$r_{21} = -r_{12} = c_2.s_1 + c_1.s_2 \quad \text{.....(7.4)}$$

$$r_{24} = L_1.s_1 + L_2.r_{21} \quad \text{.....(7.5)}$$

$$x = r_{14} + L_3.(r_{11}.c_3 + r_{12}.s_3) \quad \text{.....(7.6)}$$

$$y = r_{24} + L_3.(r_{21}.c_3 + r_{22}.s_3) \quad \text{.....(7.7)}$$

This formulation requires fourteen multiplications and nine additions but the internal parallelism within each processing element can be used to good effect by allowing the fourteen multiplications to be performed in the same time as seven.

The complete forward kinematics solution could then be obtained by combining this routine with the sine/cosine routine presented in the previous section (see Appendix D for software listing).

---

21 Paul, R.P, Robot Manipulators: Mathematics Programming and Control, MIT Press, U.S.A., 1981, pp50-55.

7.3 Implementation on the Host

The host machine is required to select points to which the manipulator could move from its previous position. This is achieved by varying θ_1 by some small discrete amount (in this case five possible values distributed evenly in the range $\theta_1 \pm 0.001$ radians were used). For each value of θ_1, θ_2 is stepped through a range of values in a similar fashion and for each of these points θ_3 is also varied. The result is a number of possible positions in joint-space. It was not practical to calculate values for the joint angles using the co-processor due to the necessity of checking for joint limits.

Once suitable values in joint-space have been located they are written to the co-processor and the host need only start the co-processor running and then await the results of the forward kinematics calculations.

After the forward kinematics values have been found it is necessary to select the "best" one. In this case some possible points were first eliminated based on the criterion that each point on the cartesian trajectory must be within a certain specified distance of a straight-line cartesian path and that it must be closer to the target than the previous point. The remaining points are then searched to find the one which minimises the cartesian distance between the end-effector and the target point.

While it would have been desirable to implement this search in a parallel manor this was impractical due to the lack of any conditional constructs in the co-processor (the co-processor does have a sort command which could be used to find the minimum cartesian distance but there is no way to relate back from that distance to the point at which it occurred).

Once the "best" point has been found it becomes the next point along the trajectory and the process is repeated until the end-effector reaches the target point.

7.4 Results and Discussion.

In the first test the system was required to create a trajectory between the cartesian points (0.3,0.0) and (0.0,0.2) with the "best" point at each step in

the algorithm being the one which minimised the cartesian distance. The results of this are shown in Figure 7.4.1.

Examination of this graph shows that initially the motion results from moving all three joints by the maximum amount at each step - this has the effect of reducing the cartesian distance by the greatest amount at each step. Toward the end of the trajectory it is no longer possible to move in this way while satisfying the constraint that the cartesian distance be minimised. As a result joint one is forced to move back to bring the end-effector to the final position.

Examination of the changing joint angles shows the interesting result that joints 2 and 3, which both had to move the maximum distance, have moved the maximum amount at each step throughout the entire motion. Thus, given the constraint that each joint can move a maximum of 0.001 radians at each step, the trajectory appears to be optimal in terms of the total number of steps required.

Figure 7.4.2 shows the result of requiring motion between the same two points but applying the constraint that the end-effector must remain within ±0.02 units of a straight-line cartesian trajectory.

In this case the arm advances from rest with all three joints moving the maximum amount at each step. This is the same as the previous case. However, once the end-effector reaches a point close to +0.02 units from the straight line trajectory further motion in this manor is impossible without violating the straight line constraint. As a result joint one is forced to move back and joint two is slowed down to achieve the desired near straight-line motion.

Examination of the joint angles shows that it is nearly always the case that one of the joints is moving at the maximum amount at each step.

Comparison of this with the previous motion shows that, as one might expect, considerably more steps were required for the case where the straight line condition was specified.

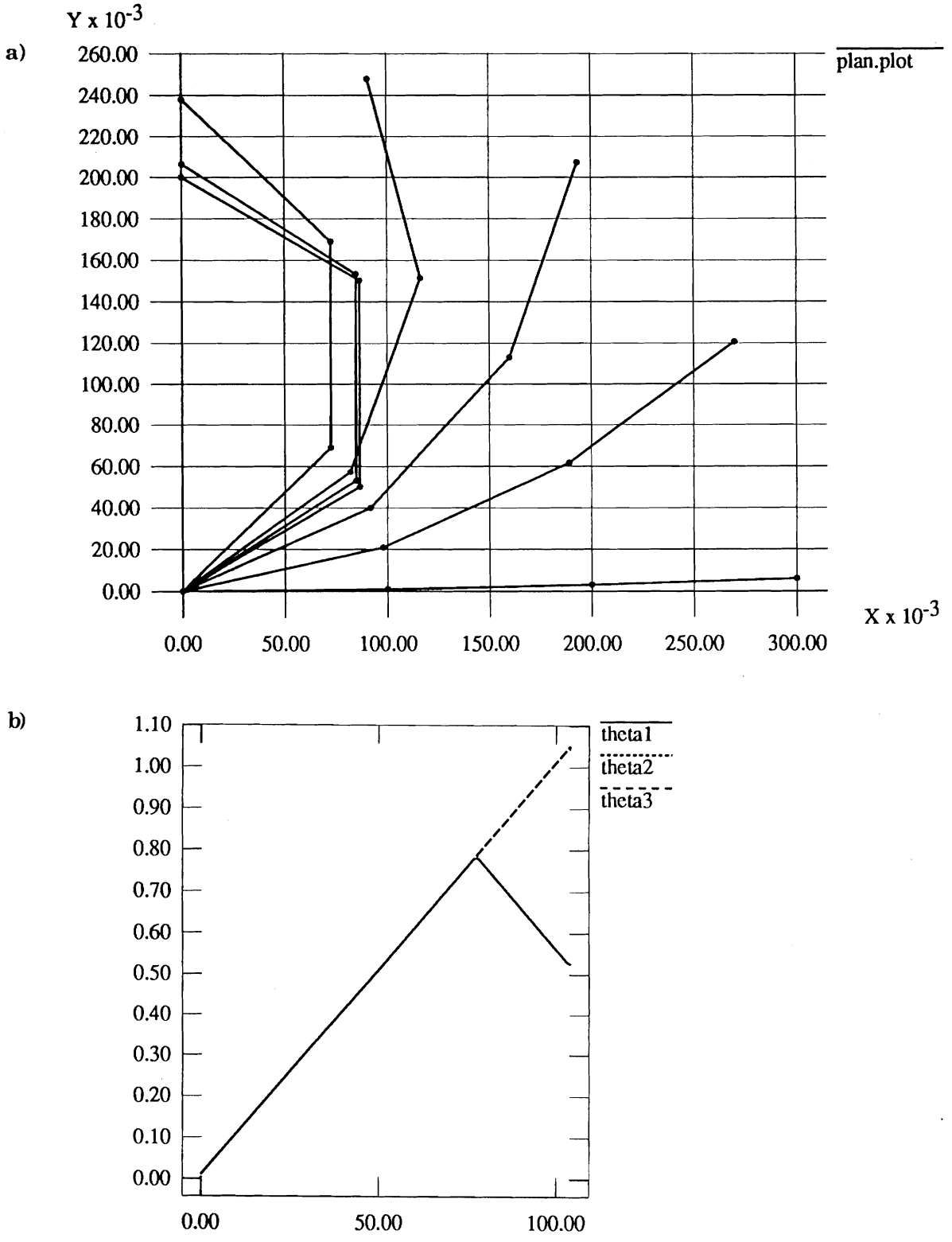


Figure 7.4.1. a) Trajectory generated for motion between (0.3,0.0) and (0.0, 0.2). Every 20th point along the trajectory is shown in cartesian space.
b) Graph of joint angles versus steps along the trajectory for the same path.

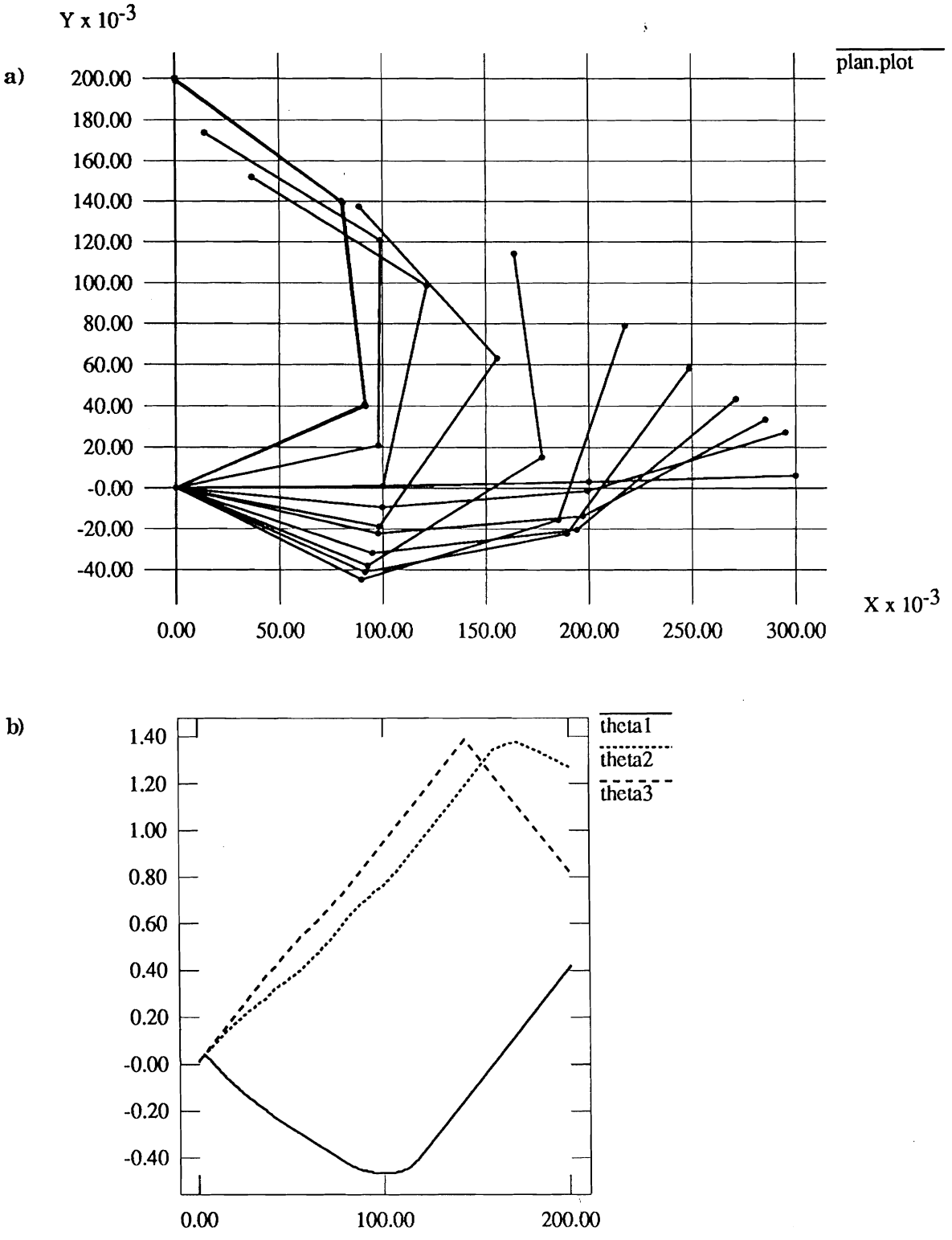


Figure 7.4.2. a) Trajectory generated for motion between (0.3,0.0) and (0.0, 0.2) with the constraint that the motion remain within 0.02 units of a straight line Every 20th point along the trajectory is shown in cartesian space.
b) Graph of joint angles versus steps along the trajectory for the same path.

In the second test the system was required to move between (0.3,0.0) and (-0.3,0.0). Again it was required that points be chosen to minimise the cartesian distance at each step. The resulting trajectory is shown in Figure 5.4.3.

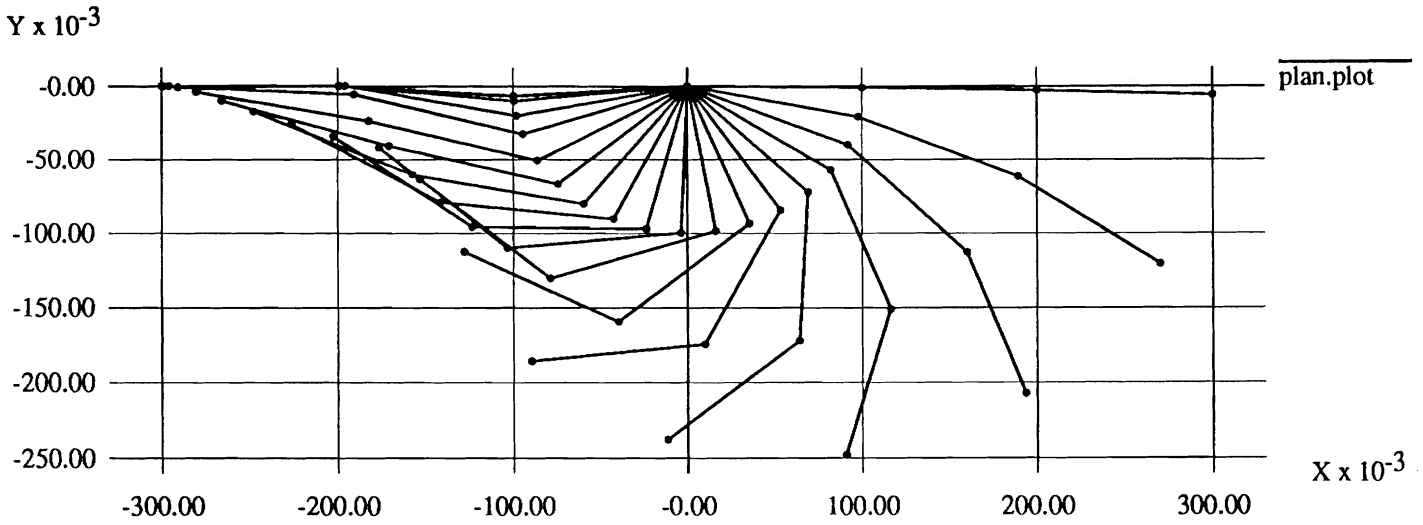
One might expect that the manipulator could move between the initial and target points using only joint one while joints two and three remained stationary. However, while this would accomplish the same end result the intermediate points along the trajectory would be further way from the target point than in this generated trajectory. This system therefore appears to offer some advantage over moving along a straight line in joint-space.

The trajectory is sub-optimal (in terms of the total number of steps required). This appears to be the result of excessive motion by joint three.

Figure 7.4.4 shows motion between the same points but with the additional constraint that the end-effector remain within ±0.02 of a straight line cartesian trajectory. This is a more difficult task due to the need to pass close to the origin.

As for the previous case there is the predictable result that considerably more steps are required to specify the trajectory when straight-line motion is required.

a)



b)

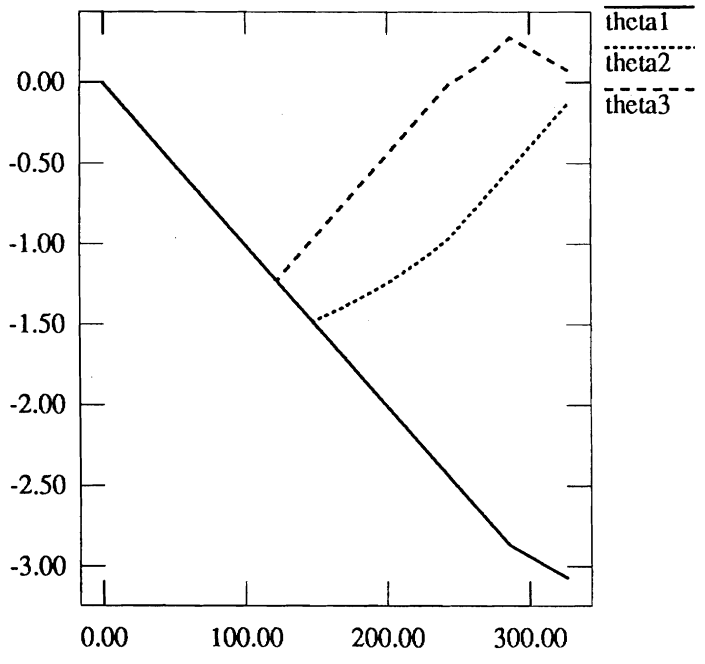
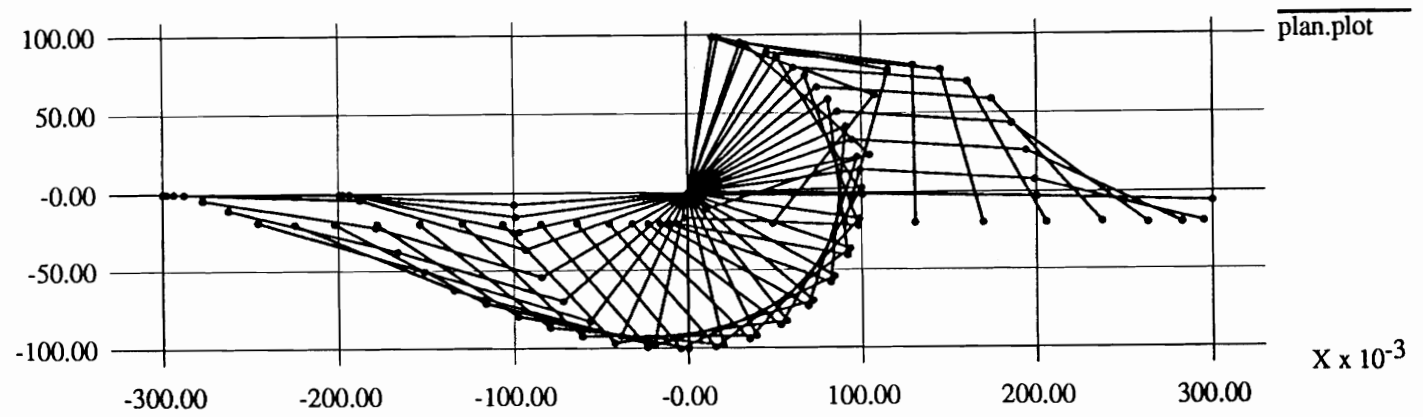


Figure 7.4.3. a)Trajectory generated for motion between (0.3, 0.0) and (-0.3, 0.0). Every 20th point along the trajectory is shown in cartesian space
b) Graph of joint angles versus steps along the trajectory for the same path.

a)



b)

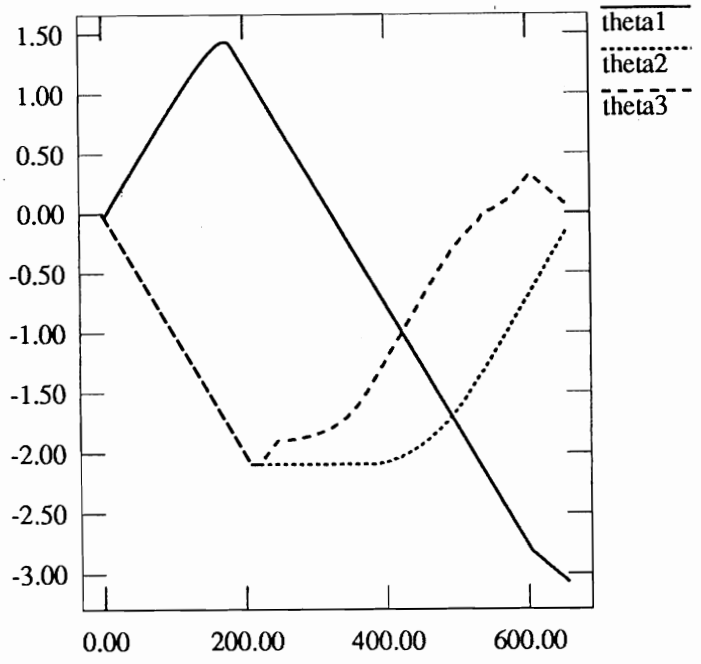


Figure 7.4.4. a) Trajectory generated for motion between (0.3, 0.0) and (-0.3, 0.0) with the constraint that motion must be within 0.02 units of a straight line. Every 20th point along the trajectory is shown in cartesian space.
b) Graph of joint angles versus steps along the trajectory for the same path.

It was hypothesized that the sub-optimality of the solutions in the previous examples was a result of the system being under-constrained. To examine this possibility the tests were repeated with the additional constraint that the difference between the angles of joints two and three should be minimised (if there were several possibilities then the one which minimised the cartesian distance to the target point was chosen). The results of this are shown in Figures 7.4.5 and 7.4.6.

This appears to provide a smoother motion than in the previous example since it prevents the arm from adopting a "kinked" posture (where joints two and three have opposite sign). It also provides solutions which require fewer steps.

It is interesting that even in the case where straight line motion near the origin was required, the system chose trajectory values where joints two and three were exactly equal. It should be noted that this need not always be the case. For example, if the allowable deviation from a straight-line cartesian motion were to be made very small then the system would be forced to choose different values for joints two and three in order to keep the end-effector within the desired path.

In summary it is clear from these example trajectories that, provided the solution is sufficiently well constrained, the system can produce trajectories which are near-optimal in terms of the number of steps required.

There is currently no guarantee that the system will successfully find an appropriate trajectory for every possible cartesian path. Indeed it is likley that, particularly when motion near joint limits is concerned, the system may fail to find a suitable path. Further research is required to characterise cases where this situation might occur.

At present the system only produces trajectories in joint-space. Further work is needed to generate appropriate motor torque values. It would be interesting to see if dynamic considerations could be incorporated into the algorithm (perhaps by varying the amount each joint can move at each step) to enable it to produce solutions which were near-optimal in terms of travelling time.

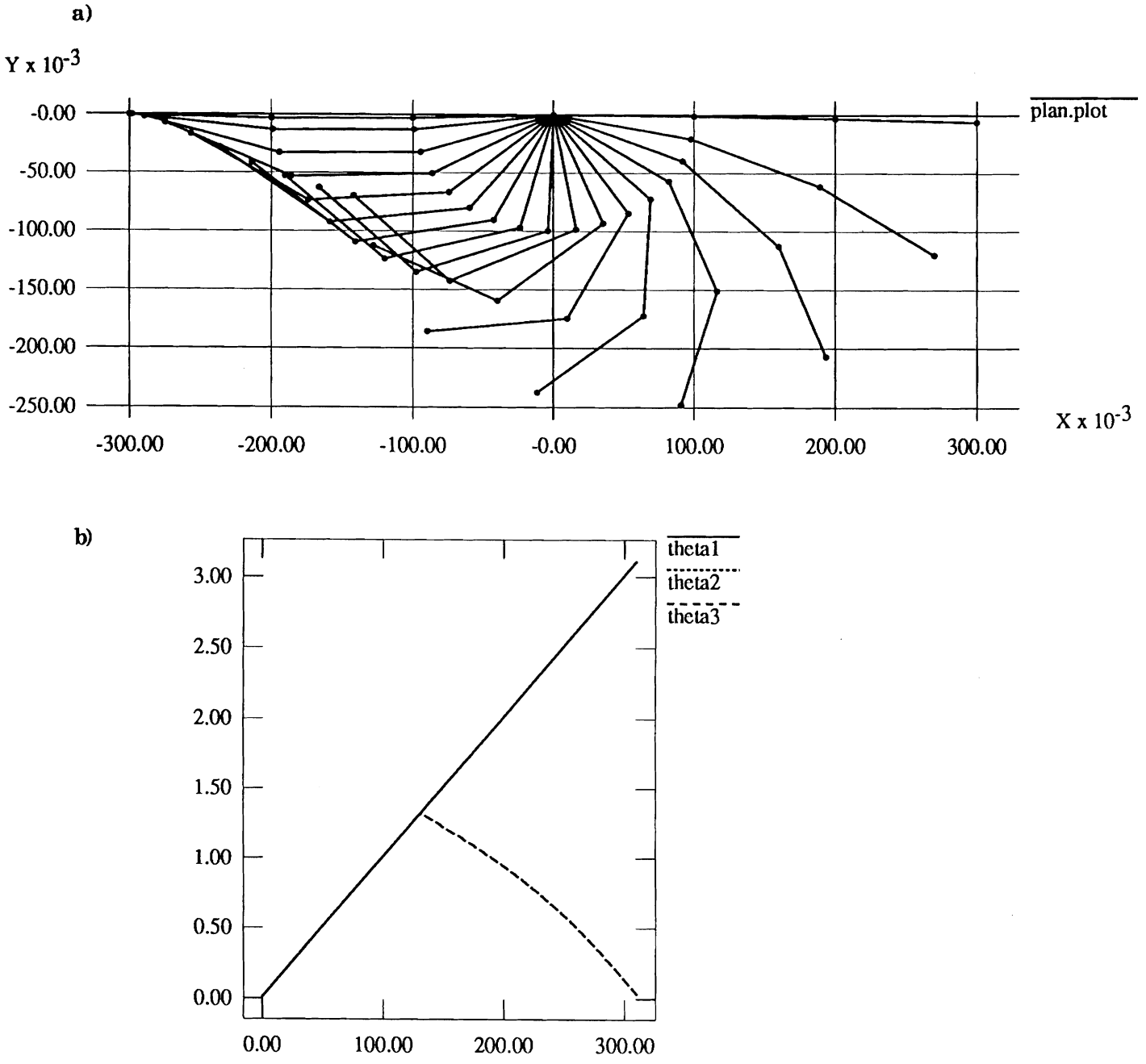
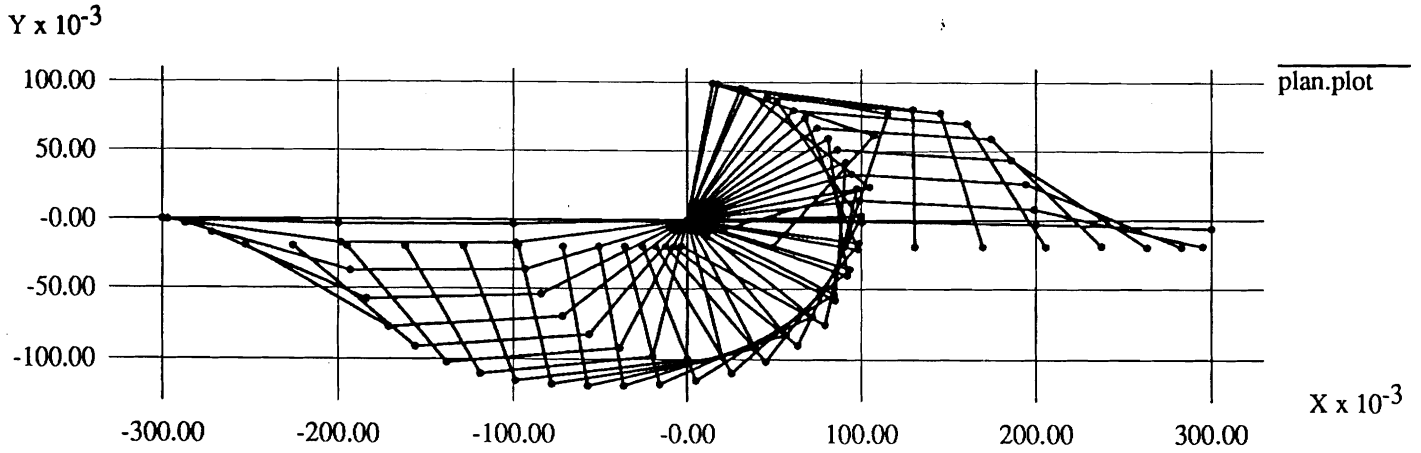


Figure 7.4.5. a) Trajectory generated for motion between (0.3, 0.0) and (-0.3, 0.0) and with the additional constraint that joints two and three should have values as close as possible. Every 20th point along the trajectory is shown in cartesian space.
b) Graph of joint angles versus steps along the trajectory for the same path.

a)



b)

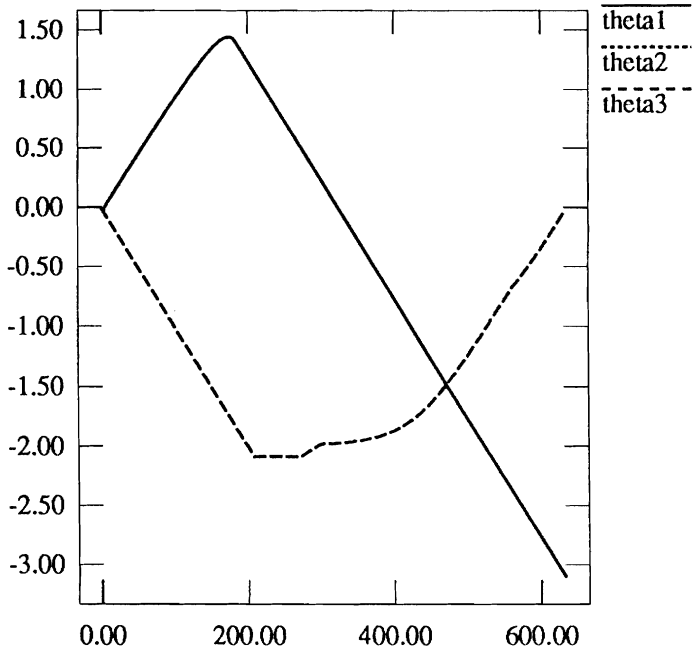


Figure 7.4.6. a) Trajectory generated for motion between (0.3, 0.0) and (-0.3, 0.0) with the constraint that motion must be within 0.02 units of a straight line and with the additional constraint that joints two and three should have values as close as possible. Every 20th point along the trajectory is shown in cartesian space.
b) Graph of joint angles versus steps along the trajectory for the same path.

8. Discussion and Suggested Design Modifications

This section presents a discussion of some suggested changes for future hardware designs as well as some desirable features for future software implementations.

8.1 Hardware

The current implementation of the processor only supports fixed-point operations. The result is that considerable time must be spent examining every algorithm to ensure that no input, intermediate or final value exceeds the storage capability of the machine. In the case of complicated algorithms this is a non-trivial task. It is recommended that future implementations employ a floating-point format (this has previously been suggested by Zhang[22]).

At present the processing elements only support the four basic arithmetic operations and a sort function. The incorporation of additional functions (for example bitwise operations) would increase the number of situations in which the co-processor could be utilized.

The sort function in each processing element went some way towards reducing the problem posed by the lack of conditional statements. However, while the sorter can be used to find the maximum/minimum of two numbers it can not indicate which was the larger/smaller. What is needed is a combined sort/move operation which sorts two registers and then swaps the contents of two other registers if, and only if, the numbers to be sorted were in ascending order.

Such a function would be particularly useful for implementing code of the form:

---

22 Zhang, Y., Redundancy Control of a Robot Manipulator using a Systolic Array Processor, PhD Thesis, The Department of Mechanical Engineering and Applied Mechanics, The University of Pennsylvania, 1989, p87.

```
if( x < TOLLERANCE )
      y = 0;
else
      y = f(x)
```

Since this could be implemented as:

```
Store 0 in register B0
Compute f(x) and store in register A0
Sort ( x,TOLLERANCE ) and swap A0 and B0 iff x < TOLLERANCE
This leaves the result in A0.
```

Note that by use of such a function several instances which would otherwise have required if/then type constructs could be implemented.

Another possibility is to provide support for conditional operations in a similar way to that used in early IBM programming languages (and presumably replicated in hardware). The basic idea was to have comparison operations set/clear indicator bits. For example: " compare b and c and set indicator 43 iff b=c". Subsequent operations could then be predicated on combinations of these bits. For example: "perform the following operation iff indicator 12 is clear and indicator 87 is set". Such a scheme would seem to be well suited to a parallel implementation. A single register within each processing element could be used to store up to 32 indicator bits and these could then be set/reset/tested using standard bitwise operations. Each processing element could compare its indicator bits to a programmable register and thereby decide whether or not to perform the current instruction.

If this system of indicator bits were implemented then it may be possible to remove the necessity of specifying a 32-bit mask field within each instruction. For example, consider the case where an additional register were provided within each processing element which, when read, returned the unique position of that element within the array. Each processing element could then use this information in conjunction with input data to set/clear its own indicator bits during program execution. By predicating subsequent operations on those bits any possible combination of processing elements could be enabled.

At present it is not possible to create an on-line debugger for the Hughes processor. While it would be possible to write a program on the host which single-stepped the co-processor, there is currently no way to print out the internal status of each processing element (the dynamic registers would decay long before they could be read). One possible solution would be to implement the output registers within each functional unit as static registers. This would not only make de-bugging easier it would also remove the restriction that results must be used within a short time of their computation.

8.2 Software

8.2.1 The Assembler

The assembler performed well during testing however there were a few additional features which would have been desirable:

Some form of improved error-checking would be helpful. For example it would be useful to have warnings for cases where dynamic registers are read after their contents have decayed or before their contents have been fully computed (this has previously been suggested by Hartholtz [23]).

Some support for nested loops would also be helpful although this would be complicated to implement due to the way program memory addresses are handled in hardware.

It is particularly difficult to write code at the assembler level due to the number of factors which must constantly be kept in mind. The result is that one tends to become overwhelmed with the details of programming and the overall algorithm becomes lost in the static. Some higher-level form of programming the processor would be of great benefit. For example a program which could convert a series of arithmetic expressions into optimised assembly language to run on a processing element would be particularly helpful.

---

23 Hartholz, M.A., The Systolic/Cellular System Assembler: User's Guide, Master's Thesis, The University of Pennsylvania, August, 1988, p95.

8.2.2 The Simulator

During testing of the simulator the author found that virtually every desirable feature was already included.

The only significant difficulty found in practice was that the simulator did not give exactly the same mathematical results as the co-processor. However, since the differences were due primarily to the limited accuracy of the co-processor it is difficult to see how such a feature could be implemented.

8.2.3 The Run time system

At present the loader program and the library of C functions only support the use of the processor with one program at a time. Since the size of the average program is considerably smaller than the program memory there is some justification for considering the possibility of having several programs resident in memory simultaneously. It is possible that such a scheme could be implemented although it would be a non-trivial process due to the need to re-map the addresses in the fifo queues.

9. Conclusions

A loader program and library of C-callable routines have been developed and tested. The loader program was effective in the testing of simple programs while the library of C-callable routines was used successfully in the development of systems which required a significant level of interaction between the co-processor and the host machine.

Routines to compute square root, sine and cosine functions using the array co-processor have been created and their performance has been evaluated. The implementation of these routines was complicated by the inability of the co-processor to perform range reduction on the input. It was found that the accuracy of the square root routine was limited by errors in the arithmetic used by the co-processor; while the accuracy of the sine and cosine functions was limited by the form of polynomial approximation employed.

A trajectory planning algorithm for redundant manipulators has been implemented. This algorithm generates a joint space trajectory from a cartesian space trajectory by using the co-processor to compute in parallel a large number of forward kinematics solutions. The system was tested using a three-link planar manipulator and it was found to provide solutions which were near-optimal (in terms of the number of trajectory points required for each planned path).

While implementing the forward kinematics solution on the co-processor it was found that considerable efficiency gains could be realised by treating the array as a SIMD machine rather than as a systolic array.

The hardware and software have been analysed and a number of recommendations have been made concerning both future hardware implementations and improvements to the existing software.

Appendix A - The Testing Process.

Some time was spent in the initial phases of this research in testing and debugging the existing hardware and software. At the time this research was started the assembler had only been used in conjunction with the simulator and the first stage of the testing process was therefore to develop a loader program which could take the assembler output and execute it on the array co-processor. This program has wider application than just within the testing process and is described more fully within the main body of the report.

Once the loader program was written the assembler/ loader/ hardware combination was tested by writing assembly language versions of routines used in the pre-existing hardware test programs. These routines could then be assembled, loaded and executed; with the results being compared to those produced by the existing test programs.

This process was complicated by the fact that there were errors in the assembler, loader, co-processor hardware and original test programs. These errors are perhaps to be expected given the complexity of the systems involved.

The only error found with the assembler was that the phase one and phase two instruction op-codes were reversed in the output file. This was corrected by reversing the phases in the loader program.

Errors in the loader program were largely a result of misunderstandings concerning the detailed operation of the hardware and these were corrected with reference to the pre-existing test routines.

Errors in the co-processor hardware proved particularly difficult to characterise. The symptoms being that the machine would produce incorrect mathematical results when certain input data was used, but correct results when different inputs were employed. These problems were traced to deficiencies in the processing elements and were largely corrected by slowing the system clocks. The author is indebted to Charles Martin for his assistance in this regard.

One side-effect of this change in speed is that multiplication and division operations now require fewer instruction cycles. Only three NOP (or equivalent length) instructions are now needed during multiplication and only ten are required during division.

At the present time all known bugs have been removed from the software development system. Most of the significant errors in the hardware have also been corrected although some inaccuracy is still present in the multiplier and divider units (see Section 2.3).

Appendix B - Software for the Square Root Routine

```
{
       Program - square_root.a
       Author  - Craig Sayers
       Date    - 3 Nov 1990

       Purpose - This program calculates the square root of a number
                 using an initial linear approximation and then
                 improving that by the use of 8 itterations of
                 Newton's method.
}

{
Register Map:


A0        N, the number to be square-rooted.
B0        Constant A
B1        Constant B,   initial approx = A.N+B
AB0       Constant 0.5
AB1       Temporary storage and final result
AB2       Temporary storage

}

DEFQUEUE Q1  64;                    Constants and input numbers
DEFQUEUE Q2  16;                    Output Numbers

       NOP;
       READQ Q1;
       WRITEQ Q2;

L0:    GETNR( B0, B0);              Read in A constants
       LOOP 15 L0;

L1:    GETNR( B1, B1);              Read in B constants
       LOOP 15 L1;

L2:    GETNR( AB0, AB0);            Read in 0.5 constants
       LOOP 15 L2;

L3:    GETNR( A0, A0);              Read in values for N
       LOOP 15 L3;

       {------- GENERATE INITIAL APPROXIMATION ------}

       MULTFD( A0,B0 : A0,AB0 );    { PROD1A = N*A : PROD2B = N*0.5 }
       NOP;
```

```
NOP;
NOP;
NOP;
NOP;
MULTSD;

MOV( : PROD2B, AB1 );          { AB1 = N*0.5 }

ADD2( PROD1A, B1 );            { SUM2B = N*A + B    }
MOV( : SUM2B, AB2 );           { AB2 = first_guess }

{------ FIRST ITERATIVE STEP ------}

DIVF( AB2, AB1 );              { -+- QUOTA = N*0.5 / first_guess  }
DIVS;                          {  |                               }
NOP;                           {  |                               }
MULTF2( AB0, AB2 );            {  |  -+- PROD2B = 0.5*first_guess }
NOP;                           {  |   |                           }
NOP;                           {  |   |                           }
NOP;                           {  |   |                           }
NOP;                           {  |   |                           }
NOP;                           {  |   |                           }
MULTS2;                        {  |  -/                           }
NOP;                           {  |                               }
NOP;                           {  |                               }
NOP;                           {  |                               }
NOP;                           { -/                               }

ADD2( QUOTA, PROD2B);          { SUM2B = first_guess*0.5 + N*0.5/first_guess }
MOV( : SUM2B, AB2 );           { AB2 = new result                 }

{------ SECOND ITERATIVE STEP ------}

DIVF( AB2, AB1 );              { -+- QUOTA = N*0.5 / result       }
DIVS;                          {  |                               }
NOP;                           {  |                               }
MULTF2( AB0, AB2 );            {  |  -+- PROD2B = 0.5*result      }
NOP;                           {  |   |                           }
NOP;                           {  |   |                           }
NOP;                           {  |   |                           }
NOP;                           {  |   |                           }
NOP;                           {  |   |                           }
MULTS2;                        {  |  -/                           }
NOP;                           {  |                               }
NOP;                           {  |                               }
NOP;                           {  |                               }
NOP;                           { -/                               }

ADD2( QUOTA, PROD2B);          { SUM2B = result * 0.5 + N*0.5/ result }
MOV( : SUM2B, AB2 );           { AB2 = new result                 }

{------ THIRD ITERATIVE STEP ------}
```

```
DIVF( AB2, AB1 );           { -+- QUOTA = N*0.5 / result       }
DIVS;                       {  |                               }
NOP;                        {  |                               }
MULTF2( AB0, AB2 );         {  |  -+- PROD2B = 0.5*result      }
NOP;                        {  |   |                            }
NOP;                        {  |   |                            }
NOP;                        {  |   |                            }
NOP;                        {  |   |                            }
NOP;                        {  |   |                            }
MULTS2;                     {  |  -/                            }
NOP;                        {  |                               }
NOP;                        {  |                               }
NOP;                        {  |                               }
NOP;                        { -/                               }

ADD2( QUOTA, PROD2B);       { SUM2B = result * 0.5 + N*0.5/ result }
MOV( : SUM2B, AB2 );        { AB2 = new result                 }

{------ FOURTH ITERATIVE STEP ------}

DIVF( AB2, AB1 );           { -+- QUOTA = N*0.5 / result       }
DIVS;                       {  |                               }
NOP;                        {  |                               }
MULTF2( AB0, AB2 );         {  |  -+- PROD2B = 0.5*result      }
NOP;                        {  |   |                            }
NOP;                        {  |   |                            }
NOP;                        {  |   |                            }
NOP;                        {  |   |                            }
NOP;                        {  |   |                            }
MULTS2;                     {  |  -/                            }
NOP;                        {  |                               }
NOP;                        {  |                               }
NOP;                        {  |                               }
NOP;                        { -/                               }

ADD2( QUOTA, PROD2B);       { SUM2B = result * 0.5 + N*0.5/ result }
MOV( : SUM2B, AB2 );        { AB2 = new result                 }

{------ FIFTH ITERATIVE STEP ------}

DIVF( AB2, AB1 );           { -+- QUOTA = N*0.5 / result       }
DIVS;                       {  |                               }
NOP;                        {  |                               }
MULTF2( AB0, AB2 );         {  |  -+- PROD2B = 0.5*result      }
NOP;                        {  |   |                            }
NOP;                        {  |   |                            }
NOP;                        {  |   |                            }
NOP;                        {  |   |                            }
NOP;                        {  |   |                            }
MULTS2;                     {  |  -/                            }
NOP;                        {  |                               }
```

```
NOP;                          {  |                                    }
NOP;                          {  |                                    }
NOP;                          { -/                                    }

ADD2( QUOTA, PROD2B);         { SUM2B = result * 0.5 + N*0.5/ result }
MOV( : SUM2B, AB2 );          { AB2 = new result                      }

{------ SIXTH ITERATIVE STEP ------}

DIVF( AB2, AB1 );             { -+- QUOTA = N*0.5 / result            }
DIVS;                         {  |                                    }
NOP;                          {  |                                    }
MULTF2( AB0, AB2 );           {  |  -+- PROD2B = 0.5*result           }
NOP;                          {  |   |                                }
NOP;                          {  |   |                                }
NOP;                          {  |   |                                }
NOP;                          {  |   |                                }
NOP;                          {  |   |                                }
MULTS2;                       {  |  -/                                }
NOP;                          {  |                                    }
NOP;                          {  |                                    }
NOP;                          {  |                                    }
NOP;                          { -/                                    }

ADD2( QUOTA, PROD2B);         { SUM2B = result * 0.5 + N*0.5/ result }
MOV( : SUM2B, AB2 );          { AB2 = new result                      }

{------ SEVENTH ITERATIVE STEP ------}

DIVF( AB2, AB1 );             { -+- QUOTA = N*0.5 / result            }
DIVS;                         {  |                                    }
NOP;                          {  |                                    }
MULTF2( AB0, AB2 );           {  |  -+- PROD2B = 0.5*result           }
NOP;                          {  |   |                                }
NOP;                          {  |   |                                }
NOP;                          {  |   |                                }
NOP;                          {  |   |                                }
NOP;                          {  |   |                                }
MULTS2;                       {  |  -/                                }
NOP;                          {  |                                    }
NOP;                          {  |                                    }
NOP;                          {  |                                    }
NOP;                          { -/                                    }

ADD2( QUOTA, PROD2B);         { SUM2B = result * 0.5 + N*0.5/ result }
MOV( : SUM2B, AB2 );          { AB2 = new result                      }

{------ EIGTH ITERATIVE STEP ------}

DIVF( AB2, AB1 );             { -+- QUOTA = N*0.5 / result            }
DIVS;                         {  |                                    }
NOP;                          {  |                                    }
```

```
        MULTF2( AB0, AB2 );          {  |  -+- PROD2B = 0.5*result      }
        NOP;                         {  |   |                           }
        NOP;                         {  |   |                           }
        NOP;                         {  |   |                           }
        NOP;                         {  |   |                           }
        NOP;                         {  |   |                           }
        MULTS2;                      {  |  -/                           }
        NOP;                         {  |                               }
        NOP;                         {  |                               }
        NOP;                         {  |                               }
        NOP;                         { -/                               }

        ADD2( QUOTA, PROD2B);        { SUM2B = result * 0.5 + N*0.5/ result }
        MOV( : SUM2B, AB1 );         { AB1 = final result               }

L4:     GETNW( AB1, AB1);            { Output results  }
        LOOP 15 L4;

        STOP;
        END;
```

Appendix C - Software for the Sin/Cos Routine

```
{
        Program - sincos.a
        Author  - Craig Sayers
        Date    - Dec 1990

        Purpose - This routine calculates the sine and cosine
                  of an input number.  The input is in the
                  range -1 <= a <= 1, where a is some fraction of
                  PI radians: Angle in radians = PI.a
}

{
       Register Map

       AB0      Input and Output
       AB1      Temporary Storage

       A0       b1 constant
       A1       c3 constant
       A2       c5 constant
       A3       c7 constant
       A4       c9 constant
       A5       c11 constant
       A6       c13 constant
       A7       c15 constant

       B0       C0 constant
       B1       C2 constant
       B2       C4 constant
       B3       C6 constant
       B4       C8 constant
       B5       C10 constant
       B6       C12 constant
       B7       C14 constant

}

DEFQUEUE Q1     272;    Input Queue
DEFQUEUE Q2     16;    Output Queue

       NOP;
       READQ Q1;
       WRITEQ Q2;

L0:    GETNR( AB0, AB0 );          Read in input
       LOOP 15 L0;

L10:   GETNR( A0, A0 );            Read in b1
```

```
        LOOP 15 L10;

L11:    GETNR( A1, A1 );            Read in c3
        LOOP 15 L11;

L12:    GETNR( A2, A2 );            Read in c5
        LOOP 15 L12;

L13:    GETNR( A3, A3 );            Read in c7
        LOOP 15 L13;

L14:    GETNR( A4, A4 );            Read in c9
        LOOP 15 L14;

L15:    GETNR( A5, A5 );            Read in c11
        LOOP 15 L15;

L16:    GETNR( A6, A6 );            Read in c13
        LOOP 15 L16;

L17:    GETNR( A7, A7 );            Read in c15
        LOOP 15 L17;

L20:    GETNR( B0, B0 );            Read in c0
        LOOP 15 L20;

L21:    GETNR( B1, B1 );            Read in c2
        LOOP 15 L21;

L22:    GETNR( B2, B2 );            Read in c4
        LOOP 15 L22;

L23:    GETNR( B3, B3 );            Read in c6
        LOOP 15 L23;

L24:    GETNR( B4, B4 );            Read in c8
        LOOP 15 L24;

L25:    GETNR( B5, B5 );            Read in c10
        LOOP 15 L25;

L26:    GETNR( B6, B6 );            Read in c12
        LOOP 15 L26;

L27:    GETNR( B7, B7 );            Read in c14
        LOOP 15 L27;

        MULTF2( AB0, AB0 );         PROD2B = a^2
        NOP;
        NOP;
        NOP;
        NOP;
```

```
NOP;
MULTSD;

MOV( :PROD2B, AB1);          AB1 = a^2

MULTFD( AB1, B7: A7, AB1 );  PROD2B = c15.a^2
NOP;                         PROD1A = c14.a^2
NOP;
NOP;
NOP;
NOP;
MULTSD;

ADD2( A6, PROD2B);
MOV(: SUM2B, AB2);           AB2 = c15.a^2+c13

ADD2( PROD1A, B6);
MOV(: SUM2B, AB3);           AB3 = c14.a^2+c12

MULTFD( AB3,AB1: AB2,AB1 );  PROD2B = c15.a^4+c13.a^2
NOP;                         PROD1A = c14.a^4+c12.a^2
NOP;
NOP;
NOP;
NOP;
MULTSD;

ADD2( A5, PROD2B);
MOV(: SUM2B, AB2);           AB2 = c15.a^4+c13.a^2+c11

ADD2( PROD1A, B5);
MOV(: SUM2B, AB3);           AB3 = c14.a^4+c12.a^2+c10

MULTFD( AB3,AB1: AB2,AB1);   PROD2B = c15.a^6+c13.a^4+c11.a^2
NOP;                         PROD1A = c14.a^6+c12.a^4+c10.a^2
NOP;
NOP;
NOP;
NOP;
MULTSD;

ADD2( A4, PROD2B);
MOV(: SUM2B, AB2);           AB2 = c15.a^6+c13.a^4+c11.a^2+c9

ADD2( PROD1A, B4);
MOV(: SUM2B, AB3);           AB3 = c14.a^6+c12.a^4+c10.a^2+c8

MULTFD( AB3,AB1: AB2,AB1);   PROD2B = c15.a^8+c13.a^6+c11.a^4+c9.a^2
NOP;                         PROD1A = c14.a^8+c12.a^6+c10.a^4+c8.a^2
NOP;
NOP;
NOP;
```

```
NOP;
MULTSD;

ADD2( A3, PROD2B);
MOV(: SUM2B, AB2);          AB2 = c15.a^8+c13.a^6+c11.a^4+c9.a^2+c7

ADD2( PROD1A, B3);
MOV(: SUM2B, AB3);          AB3 = c14.a^8+c12.a^6+c10.a^4+c8.a^2+c6

MULTFD( AB3,AB1: AB2,AB1);  PROD2B = c15.a^10+c13.a^8+c11.a^6+c9.a^4+c7.a^2
NOP;                        PROD1A = c14.a^10+c12.a^8+c10.a^6+c8.a^4+c6.a^2
NOP;
NOP;
NOP;
NOP;
MULTSD;

ADD2( A2, PROD2B);
MOV(: SUM2B, AB2);          AB2 = c15.a^10+c13.a^8+c11.a^6+c9.a^4+c7.a^2+c5

ADD2( PROD1A, B2);
MOV(: SUM2B, AB3);          AB3 = c14.a^10+c12.a^8+c10.a^6+c8.a^4+c6.a^2+c4

MULTFD( AB3,AB1: AB2,AB1);  PROD2B =
                            c15.a^12+c13.a^10+c11.a^8+c9.a^6+c7.a^4+c5.a^2
NOP;                        PROD1A =
                            c14.a^12+c12.a^10+c10.a^8+c8.a^6+c6.a^4+c4.a^2
NOP;
NOP;
NOP;
NOP;
MULTSD;

ADD2( A1, PROD2B);
MOV(: SUM2B, AB2);          AB2 =
                            c15.a^12+c13.a^10+c11.a^8+c9.a^6+c7.a^4+c5.a^2+c3

ADD2( PROD1A, B1);
MOV(: SUM2B, AB3);          AB3 =
                            c14.a^12+c12.a^10+c10.a^8+c8.a^6+c6.a^4+c4.a^2+c2

MULTFD( AB3,AB1: AB2,AB1);  PROD2B =
                            c15.a^14+c13.a^12+c11.a^10+c9.a^8+c7.a^6+c5.a^4+c3
                            .a^2
NOP;                        PROD1A =
                            c14.a^14+c12.a^12+c10.a^10+c8.a^8+c6.a^6+c4.a^4+c2
                            .a^2
NOP;
NOP;
NOP;
NOP;
MULTSD;
```

```
        MOV( PROD1A, AB3:);

        MULTF2( AB0, PROD2B );      PROD2B =
                                    c15.a^15+c13.a^13+c11.a^11+c9.a^9+c7.a^7+c5.a^5+c3
                                    .a^3
        NOP;
        NOP;
        NOP;
        NOP;
        NOP;
        MULTSD;

        ADD2( AB0, PROD2B);         AB2 =
                                    c15.a^15+c13.a^13+c11.a^11+c9.a^9+c7.a^7+c5.a^5+c3
                                    .a^3+a

        MULTFD( A0,AB3: A0,SUM2B ); PROD1A = b1.cos_sum

                                     PROD2B = b1.sin_sum
        NOP;
        NOP;
        NOP;
        NOP;
        NOP;
        MULTSD;

        MOV( PROD1A, AB3:);

        MULTFD( A0,AB3: A0,PROD2B );

                                    PROD1A = b1.b1.cos_sum   PROD2B = b1.b1.sin_sum
        NOP;
        NOP;
        NOP;
        NOP;
        NOP;
        MULTSD;

        MOV(:PROD2B, AB0);          AB0 = result for sin

        ADD2( PROD1A, B0);
        MOV(: SUM2B, AB3);          AB3 =
                                    PI.(c14.a^14+c12.a^12+c10.a^10+c8.a^8+c6.a^6+c4.a^
                                    4+c2.a^2)+c0

L30:    GETNW( AB0, AB0);           Write results for sin
        LOOP 15 L30;

L31:    GETNW( AB3, AB3);           Write results for cos
        LOOP 15 L31;
```

Appendix D - Software for Trajectory Planning

```
{
        Program - plan.a
        Author  - Craig Sayers
        Date    - Dec 1990

        Purpose - This routine calculates the forward kinematics
                  for a three-link planar manipulator.
                  The input is three angles in the range
                  -1 <= a <= 1, where a is some fraction of
                  PI radians.

                  The routine also calculates the cartesian distance
                  between the end-effector and a specified target
                  position.  ( This calculation is inaccurate for small
                  distances due to inaccuracies in the square root
                  routine - see report on square root routine for
                  further details ).
}

{
        Register Map for calculating sine and cosine

        AB0     Input joint angles
        AB1     Temporary Storage
        AB2     sin3
        AB3     cos3
        AB4     sin2
        AB5     cos2
        AB6     sin1
        AB7     cos1

        A0      b1 constant
        A1      c3 constant
        A2      c5 constant
        A3      c7 constant
        A4      c9 constant
        A5      c11 constant
        A6      c13 constant
        A7      c15 constant

        B0      C0 constant
        B1      C2 constant
        B2      C4 constant
        B3      C6 constant
        B4      C8 constant
        B5      C10 constant
        B6      C12 constant
        B7      C14 constant
```

```
        Register map for calculating forward kinematics

        AB0     Temporary storage
        AB1     L2
        AB2     sin3
        AB3     cos3
        AB4     sin2
        AB5     cos2
        AB6     sin1
        AB7     cos1

        A0,A1,A2,B0,B1,B2 Temporary storage

        B4      Final result for x
        B5      Final result for y

        B6      L1
        B7      L3

        Register map for calculating cartesian distance

        AB0     Constant 0.5
        AB1     Temporary storage and final result
        AB2     Temporary storage

        A0       N, the number to be square-rooted.

        B0       Constant A
        B1       Constant B,   initial approx = A.N+B
        B4      Calculated value for x
        B5      Calculated value for y

}

DEFQUEUE Q1  480;                   Input Queue
DEFQUEUE Q2  64;                    Output Queue

        NOP;
        READQ Q1;
        WRITEQ Q2;

L10:    GETNR( A0, A0 );            Read in b1
        LOOP 15 L10;

L11:    GETNR( A1, A1 );            Read in c3
        LOOP 15 L11;

L12:    GETNR( A2, A2 );            Read in c5
        LOOP 15 L12;

L13:    GETNR( A3, A3 );            Read in c7
```

```
        LOOP 15 L13;

L14:    GETNR( A4, A4 );            Read in c9
        LOOP 15 L14;

L15:    GETNR( A5, A5 );            Read in c11
        LOOP 15 L15;

L16:    GETNR( A6, A6 );            Read in c13
        LOOP 15 L16;

L17:    GETNR( A7, A7 );            Read in c15
        LOOP 15 L17;

L20:    GETNR( B0, B0 );            Read in c0
        LOOP 15 L20;

L21:    GETNR( B1, B1 );            Read in c2
        LOOP 15 L21;

L22:    GETNR( B2, B2 );            Read in c4
        LOOP 15 L22;

L23:    GETNR( B3, B3 );            Read in c6
        LOOP 15 L23;

L24:    GETNR( B4, B4 );            Read in c8
        LOOP 15 L24;

L25:    GETNR( B5, B5 );            Read in c10
        LOOP 15 L25;

L26:    GETNR( B6, B6 );            Read in c12
        LOOP 15 L26;

L27:    GETNR( B7, B7 );            Read in c14
        LOOP 15 L27;

        {-------------------- calculate sin/cos for theta1 --------------------}

L30:    GETNR( AB0, AB0 );          Read in theta1
        LOOP 15 L30;

        MULTF2( AB0, AB0 );         PROD2B = a^2
        NOP;
        NOP;
        NOP;
        NOP;
        NOP;
        MULTSD;

        MOV( :PROD2B, AB1);         AB1 = a^2
```

```
MULTFD( AB1, B7: A7, AB1 );  PROD2B = c15.a^2
NOP;                         PROD1A = c14.a^2
NOP;
NOP;
NOP;
NOP;
MULTSD;

ADD2( A6, PROD2B);
MOV(: SUM2B, AB2);           AB2 = c15.a^2+c13

ADD2( PROD1A, B6);
MOV(: SUM2B, AB3);           AB3 = c14.a^2+c12

MULTFD( AB3,AB1: AB2,AB1 );  PROD2B = c15.a^4+c13.a^2
NOP;                         PROD1A = c14.a^4+c12.a^2
NOP;
NOP;
NOP;
NOP;
MULTSD;

ADD2( A5, PROD2B);
MOV(: SUM2B, AB2);           AB2 = c15.a^4+c13.a^2+c11

ADD2( PROD1A, B5);
MOV(: SUM2B, AB3);           AB3 = c14.a^4+c12.a^2+c10

MULTFD( AB3,AB1: AB2,AB1);   PROD2B = c15.a^6+c13.a^4+c11.a^2
NOP;                         PROD1A = c14.a^6+c12.a^4+c10.a^2
NOP;
NOP;
NOP;
NOP;
MULTSD;

ADD2( A4, PROD2B);
MOV(: SUM2B, AB2);           AB2 = c15.a^6+c13.a^4+c11.a^2+c9

ADD2( PROD1A, B4);
MOV(: SUM2B, AB3);           AB3 = c14.a^6+c12.a^4+c10.a^2+c8

MULTFD( AB3,AB1: AB2,AB1);   PROD2B = c15.a^8+c13.a^6+c11.a^4+c9.a^2
NOP;                         PROD1A = c14.a^8+c12.a^6+c10.a^4+c8.a^2
NOP;
NOP;
NOP;
NOP;
MULTSD;

ADD2( A3, PROD2B);
```

```
MOV(: SUM2B, AB2);          AB2 = c15.a^8+c13.a^6+c11.a^4+c9.a^2+c7

ADD2( PROD1A, B3);
MOV(: SUM2B, AB3);          AB3 = c14.a^8+c12.a^6+c10.a^4+c8.a^2+c6

MULTFD( AB3,AB1: AB2,AB1);  PROD2B = c15.a^10+c13.a^8+c11.a^6+c9.a^4+c7.a^2
NOP;                        PROD1A = c14.a^10+c12.a^8+c10.a^6+c8.a^4+c6.a^2
NOP;
NOP;
NOP;
NOP;
MULTSD;

ADD2( A2, PROD2B);
MOV(: SUM2B, AB2);          AB2 = c15.a^10+c13.a^8+c11.a^6+c9.a^4+c7.a^2+c5

ADD2( PROD1A, B2);
MOV(: SUM2B, AB3);          AB3 = c14.a^10+c12.a^8+c10.a^6+c8.a^4+c6.a^2+c4

MULTFD( AB3,AB1: AB2,AB1);  PROD2B =
                            c15.a^12+c13.a^10+c11.a^8+c9.a^6+c7.a^4+c5.a^2
NOP;                        PROD1A =
                            c14.a^12+c12.a^10+c10.a^8+c8.a^6+c6.a^4+c4.a^2
NOP;
NOP;
NOP;
NOP;
MULTSD;

ADD2( A1, PROD2B);
MOV(: SUM2B, AB2);          AB2 =
                            c15.a^12+c13.a^10+c11.a^8+c9.a^6+c7.a^4+c5.a^2+c3

ADD2( PROD1A, B1);
MOV(: SUM2B, AB3);          AB3 =
                            c14.a^12+c12.a^10+c10.a^8+c8.a^6+c6.a^4+c4.a^2+c2

MULTFD( AB3,AB1: AB2,AB1);  PROD2B =
                            c15.a^14+c13.a^12+c11.a^10+c9.a^8+c7.a^6+c5.a^4+c3
                            .a^2
NOP;                        PROD1A =
                            c14.a^14+c12.a^12+c10.a^10+c8.a^8+c6.a^6+c4.a^4+c2
                            .a^2
NOP;
NOP;
NOP;
NOP;
MULTSD;

MOV( PROD1A, AB3:);
```

```
        MULTF2( AB0, PROD2B );        PROD2B =
                                      c15.a^15+c13.a^13+c11.a^11+c9.a^9+c7.a^7+c5.a^5+c3
                                      .a^3
        NOP;
        NOP;
        NOP;
        NOP;
        NOP;
        MULTSD;

        ADD2( AB0, PROD2B);           AB2 =
                                      c15.a^15+c13.a^13+c11.a^11+c9.a^9+c7.a^7+c5.a^5+c3
                                      .a^3+a

        MULTFD( A0,AB3: A0,SUM2B );   PROD1A = b1.cos_sum

                                       PROD2B = b1.sin_sum
        NOP;
        NOP;
        NOP;
        NOP;
        NOP;
        MULTSD;

        MOV( PROD1A, AB3:);

        MULTFD( A0,AB3: A0,PROD2B);   PROD1A = b1.b1.cos_sum   PROD2B = b1.b1.sin_sum
        NOP;
        NOP;
        NOP;
        NOP;
        NOP;
        MULTSD;

        MOV(:PROD2B, AB6);            AB6 = result for sin( theta1)

        ADD2( PROD1A, B0);            SUM2B =
                                      PI.(c14.a^14+c12.a^12+c10.a^10+c8.a^8+c6.a^6+c4.a^
                                      4+c2.a^2)+c0
        MOV(: SUM2B, AB7);            AB7 = result for cos( theta1)

        {-------------------- calculate sin/cos for theta2 --------------------}

L31:    GETNR( AB0, AB0 );            Read in theta2
        LOOP 15 L31;

        MULTF2( AB0, AB0 );           PROD2B = a^2
        NOP;
        NOP;
        NOP;
        NOP;
        NOP;
```

```
MULTSD;

MOV( :PROD2B, AB1);          AB1 = a^2

MULTFD( AB1, B7: A7, AB1 );  PROD2B = c15.a^2
NOP;                         PROD1A = c14.a^2
NOP;
NOP;
NOP;
NOP;
MULTSD;

ADD2( A6, PROD2B);
MOV(: SUM2B, AB2);           AB2 = c15.a^2+c13

ADD2( PROD1A, B6);
MOV(: SUM2B, AB3);           AB3 = c14.a^2+c12

MULTFD( AB3, AB1: AB2, AB1 );PROD2B = c15.a^4+c13.a^2
NOP;                         PROD1A = c14.a^4+c12.a^2
NOP;
NOP;
NOP;
NOP;
MULTSD;

ADD2( A5, PROD2B);
MOV(: SUM2B, AB2);           AB2 = c15.a^4+c13.a^2+c11

ADD2( PROD1A, B5);
MOV(: SUM2B, AB3);           AB3 = c14.a^4+c12.a^2+c10

MULTFD( AB3,AB1: AB2,AB1);   PROD2B = c15.a^6+c13.a^4+c11.a^2
NOP;                         PROD1A = c14.a^6+c12.a^4+c10.a^2
NOP;
NOP;
NOP;
NOP;
MULTSD;

ADD2( A4, PROD2B);
MOV(: SUM2B, AB2);           AB2 = c15.a^6+c13.a^4+c11.a^2+c9

ADD2( PROD1A, B4);
MOV(: SUM2B, AB3);           AB3 = c14.a^6+c12.a^4+c10.a^2+c8

MULTFD( AB3,AB1: AB2,AB1);   PROD2B = c15.a^8+c13.a^6+c11.a^4+c9.a^2
NOP;                         PROD1A = c14.a^8+c12.a^6+c10.a^4+c8.a^2
NOP;
NOP;
NOP;
NOP;
```

```
MULTSD;

ADD2( A3, PROD2B);
MOV(: SUM2B, AB2);          AB2 = c15.a^8+c13.a^6+c11.a^4+c9.a^2+c7

ADD2( PROD1A, B3);
MOV(: SUM2B, AB3);          AB3 = c14.a^8+c12.a^6+c10.a^4+c8.a^2+c6

MULTFD( AB3,AB1: AB2,AB1);  PROD2B = c15.a^10+c13.a^8+c11.a^6+c9.a^4+c7.a^2
NOP;                        PROD1A = c14.a^10+c12.a^8+c10.a^6+c8.a^4+c6.a^2
NOP;
NOP;
NOP;
NOP;
MULTSD;

ADD2( A2, PROD2B);
MOV(: SUM2B, AB2);          AB2 = c15.a^10+c13.a^8+c11.a^6+c9.a^4+c7.a^2+c5

ADD2( PROD1A, B2);
MOV(: SUM2B, AB3);          AB3 = c14.a^10+c12.a^8+c10.a^6+c8.a^4+c6.a^2+c4

MULTFD( AB3,AB1: AB2,AB1);  PROD2B =
                            c15.a^12+c13.a^10+c11.a^8+c9.a^6+c7.a^4+c5.a^2
NOP;                        PROD1A =
                            c14.a^12+c12.a^10+c10.a^8+c8.a^6+c6.a^4+c4.a^2
NOP;
NOP;
NOP;
NOP;
MULTSD;

ADD2( A1, PROD2B);
MOV(: SUM2B, AB2);          AB2 =
                            c15.a^12+c13.a^10+c11.a^8+c9.a^6+c7.a^4+c5.a^2+c3

ADD2( PROD1A, B1);
MOV(: SUM2B, AB3);          AB3 =
                            c14.a^12+c12.a^10+c10.a^8+c8.a^6+c6.a^4+c4.a^2+c2

MULTFD( AB3,AB1: AB2,AB1);  PROD2B =
                            c15.a^14+c13.a^12+c11.a^10+c9.a^8+c7.a^6+c5.a^4+c3
                            .a^2
NOP;                        PROD1A =
                            c14.a^14+c12.a^12+c10.a^10+c8.a^8+c6.a^6+c4.a^4+c2
                            .a^2
NOP;
NOP;
NOP;
NOP;
MULTSD;
```

```
        MOV( PROD1A, AB3:);

        MULTF2( AB0, PROD2B );      PROD2B =
                                    c15.a^15+c13.a^13+c11.a^11+c9.a^9+c7.a^7+c5.a^5+c3
                                    .a^3
        NOP;
        NOP;
        NOP;
        NOP;
        NOP;
        MULTSD;

        ADD2( AB0, PROD2B);         AB2 =
                                    c15.a^15+c13.a^13+c11.a^11+c9.a^9+c7.a^7+c5.a^5+c3
                                    .a^3+a

        MULTFD( A0,AB3: A0,SUM2B ); PROD1A = b1.cos_sum

                                     PROD2B = b1.sin_sum
        NOP;
        NOP;
        NOP;
        NOP;
        NOP;
        MULTSD;

        MOV( PROD1A, AB3:);

        MULTFD( A0,AB3: A0,PROD2B); PROD1A = b1.b1.cos_sum   PROD2B = b1.b1.sin_sum
        NOP;
        NOP;
        NOP;
        NOP;
        NOP;
        MULTSD;

        MOV(:PROD2B, AB4);          AB4 = result for sin( theta2)

        ADD2( PROD1A, B0);          SUM2B =
                                    PI.(c14.a^14+c12.a^12+c10.a^10+c8.a^8+c6.a^6+c4.a^
                                    4+c2.a^2)+c0
        MOV(: SUM2B, AB5);          AB5 = result for cos( theta2)

        {-------------------- calculate sin/cos for theta3 --------------------}

L32:    GETNR( AB0, AB0 );          Read in theta3
        LOOP 15 L32;

        MULTF2( AB0, AB0 );         PROD2B = a^2
        NOP;
        NOP;
        NOP;
```

```
NOP;
NOP;
MULTSD;

MOV( :PROD2B, AB1);          AB1 = a^2

MULTFD( AB1, B7: A7, AB1 );  PROD2B = c15.a^2
NOP;                         PROD1A = c14.a^2
NOP;
NOP;
NOP;
NOP;
MULTSD;

ADD2( A6, PROD2B);
MOV(: SUM2B, AB2);           AB2 = c15.a^2+c13

ADD2( PROD1A, B6);
MOV(: SUM2B, AB3);           AB3 = c14.a^2+c12

MULTFD( AB3,AB1: AB2,AB1);   PROD2B = c15.a^4+c13.a^2
NOP;                         PROD1A = c14.a^4+c12.a^2
NOP;
NOP;
NOP;
NOP;
MULTSD;

ADD2( A5, PROD2B);
MOV(: SUM2B, AB2);           AB2 = c15.a^4+c13.a^2+c11

ADD2( PROD1A, B5);
MOV(: SUM2B, AB3);           AB3 = c14.a^4+c12.a^2+c10

MULTFD( AB3,AB1: AB2,AB1);   PROD2B = c15.a^6+c13.a^4+c11.a^2
NOP;                         PROD1A = c14.a^6+c12.a^4+c10.a^2
NOP;
NOP;
NOP;
NOP;
MULTSD;

ADD2( A4, PROD2B);
MOV(: SUM2B, AB2);           AB2 = c15.a^6+c13.a^4+c11.a^2+c9

ADD2( PROD1A, B4);
MOV(: SUM2B, AB3);           AB3 = c14.a^6+c12.a^4+c10.a^2+c8

MULTFD( AB3,AB1: AB2,AB1);   PROD2B = c15.a^8+c13.a^6+c11.a^4+c9.a^2
NOP;                         PROD1A = c14.a^8+c12.a^6+c10.a^4+c8.a^2
NOP;
NOP;
```

```
NOP;
NOP;
MULTSD;

ADD2( A3, PROD2B);
MOV(: SUM2B, AB2);          AB2 = c15.a^8+c13.a^6+c11.a^4+c9.a^2+c7

ADD2( PROD1A, B3);
MOV(: SUM2B, AB3);          AB3 = c14.a^8+c12.a^6+c10.a^4+c8.a^2+c6

MULTFD( AB3,AB1: AB2,AB1);  PROD2B = c15.a^10+c13.a^8+c11.a^6+c9.a^4+c7.a^2
NOP;                        PROD1A = c14.a^10+c12.a^8+c10.a^6+c8.a^4+c6.a^2
NOP;
NOP;
NOP;
NOP;
MULTSD;

ADD2( A2, PROD2B);
MOV(: SUM2B, AB2);          AB2 = c15.a^10+c13.a^8+c11.a^6+c9.a^4+c7.a^2+c5

ADD2( PROD1A, B2);
MOV(: SUM2B, AB3);          AB3 = c14.a^10+c12.a^8+c10.a^6+c8.a^4+c6.a^2+c4

MULTFD( AB3,AB1: AB2,AB1);  PROD2B =
                            c15.a^12+c13.a^10+c11.a^8+c9.a^6+c7.a^4+c5.a^2
NOP;                        PROD1A =
                            c14.a^12+c12.a^10+c10.a^8+c8.a^6+c6.a^4+c4.a^2
NOP;
NOP;
NOP;
NOP;
MULTSD;

ADD2( A1, PROD2B);
MOV(: SUM2B, AB2);          AB2 =
                            c15.a^12+c13.a^10+c11.a^8+c9.a^6+c7.a^4+c5.a^2+c3

ADD2( PROD1A, B1);
MOV(: SUM2B, AB3);          AB3 =
                            c14.a^12+c12.a^10+c10.a^8+c8.a^6+c6.a^4+c4.a^2+c2

MULTFD( AB3,AB1: AB2,AB1);  PROD2B =
                            c15.a^14+c13.a^12+c11.a^10+c9.a^8+c7.a^6+c5.a^4+c3
                            .a^2
NOP;                        PROD1A =
                            c14.a^14+c12.a^12+c10.a^10+c8.a^8+c6.a^6+c4.a^4+c2
                            .a^2
NOP;
NOP;
NOP;
NOP;
```

```
        MULTSD;

        MOV( PROD1A, AB3:);

        MULTF2( AB0, PROD2B );        PROD2B =
                                      c15.a^15+c13.a^13+c11.a^11+c9.a^9+c7.a^7+c5.a^5+c3
                                      .a^3
        NOP;
        NOP;
        NOP;
        NOP;
        NOP;
        MULTSD;

        ADD2( AB0, PROD2B);           AB2 =
                                      c15.a^15+c13.a^13+c11.a^11+c9.a^9+c7.a^7+c5.a^5+c3
                                      .a^3+a

        MULTFD( A0,AB3: A0,SUM2B );   PROD1A = b1.cos_sum

                                       PROD2B = b1.sin_sum
        NOP;
        NOP;
        NOP;
        NOP;
        NOP;
        MULTSD;

        MOV( PROD1A, AB3:);

        MULTFD( A0,AB3: A0,PROD2B);   PROD1A = b1.b1.cos_sum   PROD2B = b1.b1.sin_sum
        NOP;
        NOP;
        NOP;
        NOP;
        NOP;
        MULTSD;

        MOV(:PROD2B, AB2);            AB2 = result for sin( theta3)

        ADD2( PROD1A, B0);            SUM2B =
                                      PI.(c14.a^14+c12.a^12+c10.a^10+c8.a^8+c6.a^6+c4.a^
                                      4+c2.a^2)+c0
        MOV(: SUM2B, AB3);            AB3 = result for cos( theta3)

        {-------------------- calculate forward kinematics ----------}

L40:    GETNR( B7, B7 );              Read in L1
        LOOP 15 L40;

L41:    GETNR( AB1, AB1 );            Read in L2
        LOOP 15 L41;
```

```
L42:    GETNR( B6, B6 );              Read in L3
        LOOP 15 L42;

        MULTFD(AB5,AB7 : AB4,AB6);    PROD1A = cos1.cos2   CPROD2A = -sin1.sin2
        NOP;
        NOP;
        NOP;
        NOP;
        NOP;
        MULTSD;

        MOV( CPROD2A, AB0:);

        ADDD( PROD1A, AB0 );          SUM1A = cos1.cos2-sin1.sin2
        MOV( SUM1A, A0:);             A0 = r11, r22

        MULTFD( AB7, B7 : A0, AB1);   PROD1A = cos1.L1
                                      PROD2B = L2.( cos1.cos2-sin1.sin2 )
        NOP;
        NOP;
        NOP;
        NOP;
        NOP;
        MULTSD;

        ADDD( PROD1A, PROD2B);        SUM2B = cos1.L1 + L2.( cos1.cos2-sin1.sin2 )
        MOV( SUM1A, A1:);             A1 = r14

        MULTFD(AB5,AB6 : AB4,AB7);    PROD1A = cos1.sin2   PROD2B = sin1.cos2
        NOP;
        NOP;
        NOP;
        NOP;
        NOP;
        MULTSD;

        ADDD( PROD1A, PROD2B );       SUM2B = cos1.sin2+sin1.cos2
        MOV( :SUM2B, B0);             B0 = -r12, r21

        MULTFD( AB6, B7 : AB1,B0 );   PROD1A = sin1.L1
                                      PROD2B = L2.( cos1.sin2+sin1.cos2 )
        NOP;
        NOP;
        NOP;
        NOP;
        NOP;
        MULTSD;

        ADD2( PROD1A, PROD2B);        SUM2B =  sin1.L1 +  L2.( cos1.sin2+sin1.cos2 )
        MOV( :SUM2B, B1);             B1 = r24
```

```
        MULTFD(A0, AB3 : AB2, B0);   PROD1A = cos3.r11
                                     CPROD2A = sin3.r12
        NOP;
        NOP;
        NOP;
        NOP;
        NOP;
        MULTSD;

        MOV( CPROD2A, AB0:);

        ADDD( PROD1A, AB0 );         SUM1A = cos3.r11 + sin3.r12
        MOV( SUM1A, A2:);            A2 = cos3.r11 + sin3.r12

        MULTFD(AB3, B0 : A0, AB2);   PROD1A = cos3.r21
                                     PROD2B = sin3.r22
        NOP;
        NOP;
        NOP;
        NOP;
        NOP;
        MULTSD;

        ADDD( PROD1A, PROD2B );      SUM2B = cos3.r21 + sin3.r22
        MOV( SUM1A, A3 :);           A3 = cos3.r21 + sin3.r22

        MULTFD( A3, B6 : A2, B6);    PROD1A = L3.(cos3.r21 + sin3.r22)
        NOP;                         PROD2B = L3.(cos3.r11 + sin3.r12)
        NOP;
        NOP;
        NOP;
        NOP;
        MULTSD;

        ADD2( A1, PROD2B);           SUM2B = r14 + L3.(cos3.r11 + sin3.r12)
        MOV( :SUM2B, B4);            B4 = result for x
        MOV( :SUM2B, B2);

        ADD2( PROD1A, B1);           SUM2B = r24 + L3.(cos3.r12 + sin3.r22)
        MOV( :SUM2B, B5);            B5 = result for y
        MOV( :SUM2B, B3);

L50:    GETNW( B2, B2);              Write results for x
        LOOP 15 L50;

L51:    GETNW( B3, B3);              Write results for y
        LOOP 15 L51;

        {--------------------calculate cartesian distance----------}

L60:    GETNR(A4 , A4);              Read in -(desired x)
        LOOP 15 L60;
```

```
L61:    GETNR( A5, A5);              Read in -(desired y)
        LOOP 15 L61;

        ADDD( A4, B4);               SUM1A = ( calc_x - desired_x )
        MOV(SUM1A, AB0:);

        ADD2( A5, B5);               SUM2B = ( calc_y - desired_y )
        MOV(:SUM2B, AB1);

        MULTFD( AB0,AB0 : AB1,AB1); PROD1A = ( calc_x - desired_x )^2 = dx^2
        NOP;                         PROD2B = ( calc_y - desired_y )^2 = dy^2
        NOP;
        NOP;
        NOP;
        NOP;
        MULTSD;

        ADDD( PROD1A, PROD2B);       SUM1A = dx^2 + dy^2
        MOV( SUM1A, A0:);            A0 = dx^2 + dy^2

        { now find square-root of A0 }

L70:    GETNR( B0, B0);              Read in A constants
        LOOP 15 L70;

L71:    GETNR( B1, B1);              Read in B constants
        LOOP 15 L71;

L72:    GETNR( AB0, AB0);            Read in 0.5 constants
        LOOP 15 L72;

        {------- GENERATE INITIAL APPROXIMATION ------}

        MULTFD( A0,B0 : A0,AB0 );    { PROD1A = N*A : PROD2B = N*0.5 }
        NOP;
        NOP;
        NOP;
        NOP;
        NOP;
        MULTSD;

        MOV( : PROD2B, AB1 );        { AB1 = N*0.5 }

        ADD2( PROD1A, B1 );          { SUM2B = N*A + B   }
        MOV( : SUM2B, AB2 );         { AB2 = first_guess }

        {------ FIRST ITERATIVE STEP ------}

        DIVF( AB2, AB1 );            { -+- QUOTA = N*0.5 / first_guess  }
        DIVS;                        {  |                               }
        NOP;                         {  |                               }
```

```
MULTF2( AB0, AB2 );          {  |  -+- PROD2B = 0.5*first_guess }
NOP;                         {  |   |                           }
NOP;                         {  |   |                           }
NOP;                         {  |   |                           }
NOP;                         {  |   |                           }
NOP;                         {  |   |                           }
MULTS2;                      {  |  -/                           }
NOP;                         {  |                               }
NOP;                         {  |                               }
NOP;                         {  |                               }
NOP;                         { -/                               }

ADD2( QUOTA, PROD2B);        { SUM2B = first_guess*0.5 + N*0.5/first_guess }
MOV( : SUM2B, AB2 );         { AB2 = new result                 }

{------ SECOND ITERATIVE STEP ------}

DIVF( AB2, AB1 );            { -+- QUOTA = N*0.5 / result       }
DIVS;                        {  |                               }
NOP;                         {  |                               }
MULTF2( AB0, AB2 );          {  |  -+- PROD2B = 0.5*result      }
NOP;                         {  |   |                           }
NOP;                         {  |   |                           }
NOP;                         {  |   |                           }
NOP;                         {  |   |                           }
NOP;                         {  |   |                           }
MULTS2;                      {  |  -/                           }
NOP;                         {  |                               }
NOP;                         {  |                               }
NOP;                         {  |                               }
NOP;                         { -/                               }

ADD2( QUOTA, PROD2B);        { SUM2B = result * 0.5 + N*0.5/ result }
MOV( : SUM2B, AB2 );         { AB2 = new result                 }

{------ THIRD ITERATIVE STEP ------}

DIVF( AB2, AB1 );            { -+- QUOTA = N*0.5 / result       }
DIVS;                        {  |                               }
NOP;                         {  |                               }
MULTF2( AB0, AB2 );          {  |  -+- PROD2B = 0.5*result      }
NOP;                         {  |   |                           }
NOP;                         {  |   |                           }
NOP;                         {  |   |                           }
NOP;                         {  |   |                           }
NOP;                         {  |   |                           }
MULTS2;                      {  |  -/                           }
NOP;                         {  |                               }
NOP;                         {  |                               }
NOP;                         {  |                               }
NOP;                         { -/                               }
```

```
ADD2( QUOTA, PROD2B);        { SUM2B = result * 0.5 + N*0.5/ result }
MOV( : SUM2B, AB2 );         { AB2 = new result                     }

{------ FOURTH ITERATIVE STEP ------}

DIVF( AB2, AB1 );            { -+- QUOTA = N*0.5 / result           }
DIVS;                        {  |                                   }
NOP;                         {  |                                   }
MULTF2( AB0, AB2 );          {  |  -+- PROD2B = 0.5*result         }
NOP;                         {  |   |                               }
NOP;                         {  |   |                               }
NOP;                         {  |   |                               }
NOP;                         {  |   |                               }
NOP;                         {  |   |                               }
MULTS2;                      {  |  -/                               }
NOP;                         {  |                                   }
NOP;                         {  |                                   }
NOP;                         {  |                                   }
NOP;                         { -/                                   }

ADD2( QUOTA, PROD2B);        { SUM2B = result * 0.5 + N*0.5/ result }
MOV( : SUM2B, AB2 );         { AB2 = new result                     }

{------ FIFTH ITERATIVE STEP ------}

DIVF( AB2, AB1 );            { -+- QUOTA = N*0.5 / result           }
DIVS;                        {  |                                   }
NOP;                         {  |                                   }
MULTF2( AB0, AB2 );          {  |  -+- PROD2B = 0.5*result         }
NOP;                         {  |   |                               }
NOP;                         {  |   |                               }
NOP;                         {  |   |                               }
NOP;                         {  |   |                               }
NOP;                         {  |   |                               }
MULTS2;                      {  |  -/                               }
NOP;                         {  |                                   }
NOP;                         {  |                                   }
NOP;                         {  |                                   }
NOP;                         { -/                                   }

ADD2( QUOTA, PROD2B);        { SUM2B = result * 0.5 + N*0.5/ result }
MOV( : SUM2B, AB2 );         { AB2 = new result                     }

{------ SIXTH ITERATIVE STEP ------}

DIVF( AB2, AB1 );            { -+- QUOTA = N*0.5 / result           }
DIVS;                        {  |                                   }
NOP;                         {  |                                   }
MULTF2( AB0, AB2 );          {  |  -+- PROD2B = 0.5*result         }
NOP;                         {  |   |                               }
NOP;                         {  |   |                               }
NOP;                         {  |   |                               }
```

```
        NOP;                        {  |   |                          }
        NOP;                        {  |   |                          }
        MULTS2;                     {  |  -/                          }
        NOP;                        {  |                              }
        NOP;                        {  |                              }
        NOP;                        {  |                              }
        NOP;                        { -/                              }

        ADD2( QUOTA, PROD2B);       { SUM2B = result * 0.5 + N*0.5/ result }
        MOV( : SUM2B, AB2 );        { AB2 = new result                }

        {------ SEVENTH ITERATIVE STEP ------}

        DIVF( AB2, AB1 );           { -+- QUOTA = N*0.5 / result      }
        DIVS;                       {  |                              }
        NOP;                        {  |                              }
        MULTF2( AB0, AB2 );         {  |  -+- PROD2B = 0.5*result     }
        NOP;                        {  |   |                          }
        NOP;                        {  |   |                          }
        NOP;                        {  |   |                          }
        NOP;                        {  |   |                          }
        NOP;                        {  |   |                          }
        MULTS2;                     {  |  -/                          }
        NOP;                        {  |                              }
        NOP;                        {  |                              }
        NOP;                        {  |                              }
        NOP;                        { -/                              }

        ADD2( QUOTA, PROD2B);       { SUM2B = result * 0.5 + N*0.5/ result }
        MOV( : SUM2B, AB2 );        { AB2 = new result                }

        {------ EIGTH ITERATIVE STEP ------}

        DIVF( AB2, AB1 );           { -+- QUOTA = N*0.5 / result      }
        DIVS;                       {  |                              }
        NOP;                        {  |                              }
        MULTF2( AB0, AB2 );         {  |  -+- PROD2B = 0.5*result     }
        NOP;                        {  |   |                          }
        NOP;                        {  |   |                          }
        NOP;                        {  |   |                          }
        NOP;                        {  |   |                          }
        NOP;                        {  |   |                          }
        MULTS2;                     {  |  -/                          }
        NOP;                        {  |                              }
        NOP;                        {  |                              }
        NOP;                        {  |                              }
        NOP;                        { -/                              }

        ADD2( QUOTA, PROD2B);       { SUM2B = result * 0.5 + N*0.5/ result }
        MOV( : SUM2B, AB1 );        { AB1 = final result }

L74:    GETNW( AB1, AB1);           { Output results  }
```

```
LOOP 15 L74;

STOP;
END;
```