AN ARITHMETIC PROCESSOR FOR ROBOTICS

by

JOSEPH KIN-SHING POON

B.Sc.(Hon.), Queen's University, 1983

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF APPLIED SCIENCE

in

THE FACULTY OF GRADUATE STUDIES

(Department of Electrical Engineering)

We accept this thesis as conforming
to the required standard.

THE UNIVERSITY OF BRITISH COLUMBIA

May, 1985

© Joseph Kin-Shing Poon, 1985
In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Department of Electrical Engineering

The University of British Columbia
1956 Main Mall
Vancouver, Canada
V6T 1Y3

Date June 27, 1985
ABSTRACT

An arithmetic processor chip for use in robotics has been designed in 4μm CMOS technology. The chip is intended to function as a slave processor to a general purpose microprocessor host and be able to perform robot arm coordinate transformation calculations for use in real-time control applications. A parallel-processing, multi-pipelined architecture has been used to produce a 45mm$^2$ chip for which a machine cycle time of 200ns appears possible. The general nature of the architecture of this microprogrammable processor renders it useful for a range of computational tasks in robotics in addition to coordinate transformation.
Table of Contents

Abstract ........................................................................................................................................ ii
Acknowledgement ......................................................................................................................... viii
1. Introduction ................................................................................................................................. 1
2. Coordinate Transformations in Robotics ..................................................................................... 3
3. Choosing an Appropriate Architecture ....................................................................................... 8
   3.1 Options for High Speed Computation .................................................................................. 10
      3.1.1 Systolic Array for Band Matrix Multiplication ......................................................... 10
      3.1.2 Data Flow .................................................................................................................... 13
      3.1.3 Control Flow ............................................................................................................... 17
4. Implementation Considerations .................................................................................................... 18
   4.1 Constraints of the Fabrication Process ............................................................................... 18
   4.2 Floating-Point vs Fixed-Point Implementation ................................................................. 20
   4.3 On-Chip Memory vs Off-Chip Memory .............................................................................. 23
   4.4 Multiplication ....................................................................................................................... 24
   4.5 32 Bit vs 24 Bit Representation ......................................................................................... 24
   4.6 Reduced Instruction Set ....................................................................................................... 25
   4.7 Outline of Possible Architecture ....................................................................................... 25
5. The APR Architecture .................................................................................................................. 26
   5.1 The Register File ................................................................................................................ 26
   5.2 The Adder Section .............................................................................................................. 28
   5.3 The Multiplier Section ....................................................................................................... 29
   5.4 The Clock Phases .............................................................................................................. 30
   5.5 Fixed-Point Mode ............................................................................................................. 31
   5.6 Input/Output and Control ................................................................................................. 33
6. Programming and Scheduling ..................................................................................................... 35
List of Figures

2.1 A Six Degree of Freedom Manipulator .................................................. 3
2.2 Format of Transformation Matrix .......................................................... 4
3.1 Example of Transformation Matrix .......................................................... 8
3.2 A Band Matrix of Width w ......................................................................... 10
3.3 Systolic Array for Matrix Multiplication .................................................. 10
3.4 Simplistic Implementation of a Data Flow Machine .................................... 14
3.5 The MIT Data Flow Computer ................................................................... 15
4.1 Yield Characteristics of ISOCMOS ............................................................... 18
4.2 Number Format for 16-bit Fixed-Point Representation ............................... 20
4.3 Error Estimation of Number Format .......................................................... 22
5.1 Block Diagram of APR in Floating-Point Mode ......................................... 26
5.2 Clock Phases Required in APR ................................................................. 30
5.3 Block Diagram of APR in Fixed-Point Mode ............................................ 31
5.4 Instruction Pipeline .................................................................................... 33
7.1 Communication in Adder Mantissa Section .............................................. 37
7.2 Inefficient Routing of Adder Mantissa Section ......................................... 37
7.3 Final Floor Plan of Adder Mantissa Section .............................................. 38
7.4 A 3-Port Register Cell ................................................................................ 41
7.5 4-out-of-7 Barrel Shifter ............................................................................ 44
7.6 Barrel Shifter Connected as Up/Down Shifter .......................................... 44
7.7 4-bit Leading Zero Checker ....................................................................... 47
7.8 Dynamic Regenerative Adder .................................................................... 48
7.9 Static Regenerative Adder ......................................................................... 49
7.10 4-Bit Adder with Partial Carry Look-Ahead ............................................ 51
7.11 Circuit Diagram of a Par-Add Block

7.12 End-Around-Carry Adder with Domino CMOS

7.13 Shift Register for Multi-Phase Clock Generation

7.14 Signals from Clock Generation Shift Register

7.15 Two-Phase Clock with Feedback to Guarantee Non-Overlapping

7.16 PLA for Stack Controller

7.17 2-bit by 2-level Stack

8.1 Floor Plan of APR

9.1 Block Diagram of APRj

9.2 Floor Plan of APRj

9.3 Instruction Formats

9.4 Block Diagram of the APRj Subsystem
ACKNOWLEDGEMENT

I would like to thank my supervisors, Dr. D.L. Pulfrey and Dr. P.D. Lawrence for their invaluable guidance and encouragement throughout this project.

In addition, I would like Dr. P. van Halen for his help in the use of the pen-plotter.

This research has been supported by the Natural Sciences and Engineering Research Council of Canada, and Robotic Systems International, Sidney, British Columbia.
1. INTRODUCTION

In order to control the motion of a robot manipulator, it is necessary to know what the position of the manipulator is, given the angles at the joints of the manipulator, and vice versa. For real time applications, such calculations cannot be performed fast enough by a general purpose microprocessor. A mainframe computer or a minicomputer with an attached array processor may be able to accomplish the task. But it is much more desirable to have a smaller unit, preferably one that would fit on the bus of a microprocessor system, that can perform the job. This would reduce cost, space, and power consumption.

In this thesis, a special purpose processor, designed to compute coordinate transformations in robotics and implemented in LSI, is described. The processor was designed taking into consideration factors such as available technology, speed, number of supporting chips required, ease of communication with a host processor, ease of programming, and power consumption.

Two versions of the processor are discussed, namely the APR chip which supports both floating-point and fixed-point arithmetic, and the APRjr chip which can only handle fixed-point numbers. Both these chips were designed to compute the direct transform of an industrial manipulator [6], in a time short enough for the solution to be used to obtain the inverse transformation by iteration for use in real time trajectory planning. To obtain the necessary speed of operation, the designs embody the features of parallel processing, independent operation of the multiplier and adder sections, non-interlocking pipelines and a reduced instruction set. Because the APR chip can handle floating-point numbers, it has a potentially wider range of applications than APRjr. However, for the specific task of fast computation of the direct transform, APRjr is adequate and, in this initial investigation, its design was carried forward to the chip fabrication stage.
A brief introduction to coordinate transformations in robotics is presented in section 2. Section 3 investigates the various ways of performing coordinate transformations at high speed. A framework of a feasible design taking into account the various constraints and requirements is given in section 4. In section 5, the architecture of APR is described in detail. Since scheduling multiple resources is both an interesting topic in parallel computer architectures and an integral part of the APR design, section 6 is devoted to a discussion of how the computing resources on APR can be scheduled efficiently. The detailed circuit design and layout of the more important cells is the topic of section 7. Section 8 deals with routing and floor planning of APR. The architecture and design of APRjr is presented in section 9. In section 10, the procedures that can be used to test the chip design are outlined. Finally, in section 11, the contributions of this research are summarized, and suggestions for further research outlined.
2. COORDINATE TRANSFORMATIONS IN ROBOTICS

A manipulator, or robot arm, consists of a number of rigid links connected by movable joints. Each joint has one degree of freedom, i.e., can revolve about one axis if it is a revolute joint or can slide along one axis if it is a prismatic joint. For example, Figure 2.1 illustrates a manipulator with six degrees of freedom (six revolute joints). The axes about which the links rotate are also indicated.

In order to control the movement of the manipulator, it is necessary to know the positions of the various links of the manipulator, in particular the position of the end effector (the free end of the manipulator). It is possible to have sensors at each joint to read the angles and send them back to the controller. The task of determining in cartesian coordinates the position and the orientation of the end effector, given the angles of the joints (or the length of a prismatic joint), is called the direct transform in robotics. Conversely, the process of determining the magnitudes of the angles given the position and orientation of the end effector is called the inverse transform.

The direct transform can be computed by multiplication of 4 by 4 matrices. Each joint can be assigned a cartesian coordinate system, which can be related to the coordinate system of the immediate previous joint by a homogeneous transformation matrix (4 by 4) [27]. To compute the transformation matrix of the end effector relative to the base, it is only necessary to multiply the transformation matrices of all the joints together. For example, for the manipulator in Figure 2.1, if \( A_i \) is the transformation matrix relating the coordinate system of joint 2 to joint 1, and \( A_j \) is that which relates the coordinate system of joint 3 to joint 2, etc., then the coordinate system of the end effector with respect to the base of the manipulator is given by

\[
T = A_1A_2A_3A_4A_5A_6
\]
Fig. 2.1: A Six Degree of Freedom Manipulator
The format of the matrix thus obtained is as shown in Figure 2.2. The first three columns specify the orientation of the end effector and the last column specifies the position of the end effector, all in cartesian coordinates.

\[
\begin{bmatrix}
  n_x & o_x & a_x & p_x \\
  n_y & o_y & a_y & p_y \\
  n_z & o_z & a_z & p_z \\
  0 & 0 & 0 & 1 \\
\end{bmatrix}
\]

Fig. 2.2: Format of Transformation Matrix

It should be pointed out that these coordinate transformations are only meaningful if the manipulator is rigid. A rigid manipulator is one which does not deform significantly even under loading conditions.

It can be seen that although the method of obtaining the direct transform is quite straightforward, the amount of computation involved is very large. It is possible to multiply the matrices out algebraically and obtain the equations for the four required vectors, which can then be simplified by factoring. However, these equations are still quite complicated for real-time use. For example, the simplified equations of the position vector of a simple manipulator [6], which has a construction similar to the one shown in Figure 2.1, is
\[ p_x = C_1[a_2 C_2 - d_4(S_{23} C_3 - C_{23} S_3)] + S_4(d_1 + d_4 S_3 S_2) \]
\[ p_y = S_1[d_4(C_2 S_{23} - C_{23} C_3) - a_1 C_2] - C_4(d_1 + d_4 S_4 S_3) \]
\[ p_z = d_4(S_{23} C_4 S_1 + C_{23} C_3) + a_3 S_1 + d_1 \]

where, for example, \( C_{23} = \cos(\theta_2 + \theta_3) \), \( S_{23} = \sin(\theta_2 + \theta_3) \), and the \( d \)'s and \( a \)'s are related to the dimensions and geometry of the manipulator.

The inverse transform is considerably more complicated than the direct transform. Not only is the form of the inverse transform very different for different manipulators, but it is also not unique (more than one set of joint angles can give rise to the same end effector position), extremely complicated (involving inverse trigonometric functions, divisions, square roots, etc.), and very often cannot even be found in closed form! However, it is possible to obtain the inverse transform by iterative application of the direct transform. In other words, a guess can be made of what the joint angles should be, the position of the end effector can then be computed using the direct transform with the guessed angles, and the error in the actual position of the end effector should enable a better estimation to be made the next time.

The direct transform is also useful in end-point servoing. Since the direct transforms can compute the end-point position and orientation, no TV camera or end-point sensor is required. Only the joint sensors are required.

It is very desirable that the direct transform can be calculated as quickly as possible. Present day manipulators do not usually move very fast. Typical sampling rates are less than 100Hz. So it may be possible to calculate the inverse transform by iterative application of the direct transform, using a high performance general purpose microprocessor with an attached special purpose arithmetic processor. When computing trajectories in real-time, this approach is far more practical than to try to compute the inverse transform directly by evaluating the inverse transform equations. These
equations are very difficult to obtain, and they are so complicated that there are few special properties in them that can be exploited by employing special hardware. The requirement of divisions and large numbers of transcendental function calls also makes evaluation of these equations very slow. For example, even with modern high-speed arithmetic coprocessors, a transcendental function takes the order of hundreds of micro-seconds to compute. In contrast, a high-speed multiplier takes about 50ns to complete a 16-bit multiplication.
3. CHOOSING AN APPROPRIATE ARCHITECTURE

Because the present project is to design a chip for a special application, namely computing the direct transform in robotics, it is possible, and necessary, to choose an architecture that is most suitable for this particular application. The choice of the correct architecture is very important. Since raw processing speed is very much limited by the integrated circuit technology, such as CMOS or TTL, significant improvement in throughput can only be achieved with parallel processing and careful circuit design. However, as the degree of parallelism is increased, the amount of hardware will generally be increased, since parallel processing is essentially spreading out computation in processing elements instead of in time. Also, increase in parallelism almost invariably leads to decrease in generality of application. So the tradeoffs must be considered very carefully.

In order to decide what architecture is best suited to the present application, the properties of the problem must be analyzed, and then the hardware and software that will take best advantage of these properties can be designed. The following observations about the direct transform in robotics can be readily made:

1. It is obtained by multiplication of 4 by 4 matrices.
2. The matrices are usually quite sparse, with the last row always containing (0, 0, 0, 1), usually about 6 or 7 other zero entries, and a 1 or -1 entry somewhere. An example of a typical transformation matrix relating 2 adjacent links is shown in Figure 3.1.
3. The overall transformation matrix is of the form shown in Figure 2.2, where \( \vec{n} \), \( \vec{d} \), and \( \vec{a} \) are orthogonal vectors specifying the orientation of the end effector, and \( \vec{p} \) is the position vector of the end effector relative to the base of the manipulator.
4. The computation requires addition, subtraction, multiplication, and sines and
cosines of angles. For a manipulator with 6 degrees of freedom, there are 6 angles and so at least 12 trigonometric functions calls are required.

5. If the transformation matrices are multiplied out algebraically, equations can be obtained for $\vec{n}$, $\vec{o}$, $\vec{a}$, and $\vec{p}$. These equations typically contain a number of common terms. For example, the term 

$$(C_1C_3 - C_1S_3)C_4 + S_1S_4$$

appears in the $x$ component of $\vec{n}$, $\vec{o}$, $\vec{a}$, and $\vec{p}$ in the Excalibur manipulator [6]. This is a consequence of the fact that these vectors are obtained by matrix multiplication. Unless a dedicated matrix multiplication engine is used, this is the best way to compute the direct transform.

\[
\begin{bmatrix}
C_4 & 0 & S_4 & 0 \\
S_4 & 0 & C_4 & 0 \\
0 & 1 & 0 & d_4 \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

Fig. 3.1: Example of Transformation Matrix
3.1 Options for High Speed Computation

In this section, various options for high speed computation are described with their general performance advantages. Then the pros and cons in applying each option to the present problem are evaluated. Although there is no strict specification as to what the computation system should look like, it is reasonable to assume that the entire system must be able to fit in a number of (preferably one) PC boards that would fit into a card cage connected to a common microprocessor bus. The power requirements of the system must be kept to a minimum. The system will be controlled by the host(s) on the bus. Therefore, options such as using a VAX with an array processor can be ruled out. In fact, because of the general purpose nature of these large computing systems, they are not necessarily much faster than a specially designed small system in solving this particular problem.

3.1.1 Systolic Array for Band Matrix Multiplication

Because the direct transform is obtained by multiplication of 4 by 4 homogeneous matrices, a dedicated matrix multiplication structure might be suitable. Much work has been done on ways of making highly parallel computing structures for the multiplication of matrices [22]. In the most general case, a systolic array can be used to multiply two \( n \times n \) band matrices with band width \( w = p + q - 1 \). A matrix that is not banded is a special case where \( p = q = n \). Figure 3.2 shows a band matrix with \( p = 3 \) and \( q = 2 \). The multiplication of two band matrices of widths \( w1 \) and \( w2 \) can be implemented with a systolic array of \( w1 \times w2 \) processing elements (Figure 3.3) [22]. The time required for multiplication is \( 3n + \min(w1, w2) \) cycles. During each cycle each processing element multiplies its two input operands, adds the product to the previous accumulated value, and passes the result to the next element in the next cycle.
Fig. 3.2: A Band Matrix of Width $w$

\[
\begin{pmatrix}
a_{11} & a_{12} & a_{13} & \cdots & 0 \\
a_{21} & a_{22} & a_{23} & \cdots & a_{2w} \\
& a_{32} & a_{33} & \cdots & a_{3w} \\
& & & \ddots & \ddots \\
& & & & a_{w1} & a_{w2} & a_{w3} & \cdots & a_{ww}
\end{pmatrix}
\]

$w = p + q - 1$

Fig. 3.3: Systolic Array for Matrix Multiplication
The advantages of such an arrangement are that the processing elements are identical, very little control is required, the interconnection is regular, communication is local, and the multiplication can be completed in a very short time.

Applying this system to the coordinate transformation problem, it can be observed that

\[
\begin{align*}
  n &= 4 \\
  p &= 4 \\
  w_1 = w_2 &= (4 + 3 - 1) = 6.
\end{align*}
\]

Hence a total of \(6 \times 6 = 36\) processing elements are needed. The number of cycles required would be \(3(4) + 6 = 18\) for multiplying two matrices. Therefore, to obtain the overall transformation matrix for a six degree of freedom manipulator, \(18 \times 5 = 90\) cycles will be needed.

Because of the simplicity of the processing elements, there are a number of ways that they can be implemented:

1. A custom designed chip. Required on chip are:
   - a multiplier–accumulator (MAC).
   - a simple control section.
   - a few storage registers.

   It is possible to put all these on a single chip. The advantage of this design is that a small number of chips is required (if 36 processors can be considered small!) The disadvantage is that the on–chip MAC will be slow (more than 200ns multiply time).

2. Use an off–the–shelf high speed MAC and design an LSI controller or use MSI off–the–shelf components to build the controller. This approach requires a much larger chip count (at least \(2 \times 36\)). The high speed multipliers will also dissipate a lot of power. Thirty-six high–speed MAC's will dissipate around 25–50W.
3. Use an off-the-shelf signal processing chip with on chip MAC and memory. Candidate chips include the TMS320 [21], the NEC µPD7720 [30], or the HITACHI HSP [38]. These chips do not have the proper I/O pins required for communication if they are to be used as the processors in the systolic array. Therefore operation will have to be slowed down to get the operands in and the results out.

In summary, a systolic array implementation would take a large number of chips and a large amount of PC board area. The system will be very fast, but will only work for multiplying matrices. The only property of the problem that can be made use of is that the direct transform can be obtained by matrix multiplication. Extra processors will also be necessary to control information flow to and from the host, and to look up sines and cosines.

3.1.2 Data Flow

One of the relatively new computer architectures that facilitate parallel processing is data flow [35,36]. So it is also worth investigating.

An instruction in a data flow computer is carried out as soon as all the operands for the instruction are available. For example, the equation

\[ a = (b + 1) \times (b - c) \]

can be implemented with the following data flow instructions:

i1: (+ () 1 i3/1) ; instruction 1. Read an operand. The '(' represents an operand. Then add the constant 1 to it, and pass the result to operand 1 of instruction 3.

i2: (- () () i3/2) ; read 2 operands, subtract the second operand from the first, and pass the result to operand 2 of instruction 3.
i3: (* () () a) ;read 2 operands, multiply them together, and pass the result to the variable a.

the order in which these instructions are arranged in the stored program is immaterial (as opposed to a conventional control-flow program). The information of the flow of data and control is embedded in the instructions themselves. In i1, it is specified that an add operation will be performed on the first (only) operand and the constant 1 when the operand is available, and after the operation, the result is to be passed to the first operand of instruction i3. A 'natural' implementation of this set of instructions is with three processors (Figure 3.4). The supply of b and c and the retrieval of a will be controlled by some other processors. The three processors shown operate on their data whenever all the data is available, independent of the other processors. Although in this simple example data flow does not increase throughput (the '*' block

Fig. 3.4: Simplistic Implementation of a Data Flow Machine
is a bottle-neck), in the general case a data flow computer is faster than a control flow computer with the same number of processors. It is not necessary that the processors be synchronous. In fact, the system is more efficient if they are fully asynchronous.

It is not possible to have a separate processor for each instruction in a complicated program. Therefore, in an actual data flow machine, the computing resources will have to be shared. An example (the MIT data flow computer [35]) is shown in Figure 3.5. In this system, instructions that have become ready are picked up by the arbitration network, and then sent to the processing section to be processed. The processing section contains a number of processing elements. The results are then sent to the control and distribution networks, passing the data to the specified instructions. In this way, other instructions will become activated and then picked up by the arbitration network. This sequence then repeats itself.

For calculating the direct transform, the matrices can be multiplied out and the equations for $\bar{n}$, $\bar{o}$, $\bar{a}$, and $\bar{p}$ obtained. This is the best way to calculate these vectors; matrix multiplication is only suitable for a systolic array implementation. The equations can then be implemented by writing an appropriate data flow program. The program is easier to write than a comparable one in a conventional computer because it is more natural. But the program only has to be written once for each manipulator, so this is not a major issue. The main advantage that a multiple processor data flow machine has over a multiple processor von Neumann machine is that of scheduling. In a data flow machine, scheduling is trivial: the programmer only needs to embed the proper execution sequence of the instructions in the instructions themselves, and the data flow computer will take care of the proper sequencing itself. In a control flow machine, the programmer has to make sure that the instructions are sequenced properly by suitable ordering of the instructions.
On the other hand, implementing a data flow computer will present many difficulties. First of all, it is not possible to integrate everything on one chip, especially if multiple processors are desired. It is possible to integrate the control, distribution, and arbitration networks, but because of the bottle-neck nature of these networks, communication will be too heavy for a single chip to handle. Also, the frequent memory accesses in a data flow computer may make it inferior to a control flow computer in this application. In a control flow computer, high speed on-chip registers can reduce the number of memory accesses for data.

Fig. 3.5: The MIT Data Flow Computer
3.1.3 Control Flow

From the previous observations, it seems that the conventional control flow architecture is probably still the best way to meet the present requirements. There are many ways to build a high speed control flow machine. A custom architecture can be built using bit slice processors, or a large number of suitably connected advanced general purpose microprocessors can be used. But since the size of the overall computation system should be kept small, the best approach is to design a custom large scale integration processor. This processor will be called APR, for Arithmetic Processor for Robotics.
4. IMPLEMENTATION CONSIDERATIONS

The idea of a special purpose arithmetic processor for robotics is not new. APAC, a powerful processor similar to and more ambitious than this work, has been described in [21]. APAC has been estimated to require the integration of over 150K transistors using an unspecified "mid-80's" technology. Even though it is now 1985, such a technology is still not widely-available through foundry services. However, foundries able to produce chips with 4\(\mu\)m design rules and yields in excess of 60% for 50mm\(^2\) die do exist [8]. Such specifications should be sufficient to enable the fabrication of a meaningful special purpose chip and to allow an examination of its effectiveness in a real system. In this section, various implementation constraints of such a processor are considered, and an architecture that would meet these constraints with present day IC technology is outlined.

4.1 Constraints of the Fabrication Process

The process that is available is a 4\(\mu\)m ISOCMOS process from GTE [8]. At the present time the GTE process is capable of producing chips of area around 25mm\(^2\) with a defect density close to 5.4 Defects/cm\(^2\). The yield for such chips is around 80%, which is very close to that predicted by the widely used SEEDS model [31]. Figure 4.1 shows the appropriate curve for this model. The curve suggests that the yield will only decrease to about 20% on increasing the chip size to about 75mm\(^2\). However, in practice, at least as regards GTE's foundry, the curve of Figure 4.1 is not followed beyond about 50mm\(^2\) and it has been found that for chip sizes above this figure the yield rapidly tends to zero [17].

The GTE process is not a particularly advanced process. Processes that have a minimum feature size of 1.5\(\mu\)m are not uncommon these days. Also, quite a few of the recently designed powerful microprocessors, such as RISC II [7,16,24,26,32,33,34] and
Fig. 4.1: Yield Characteristics of ISOCMOS

MIPS [9,10,11,12], have fairly large die sizes (RISC II measures 10.3mm x 5.8mm [16], MIPS measures 7.5mm x 8.4mm [9], both use 4μm nMOS technology).

The GTE ISOCMOS process is a p-well process, with a single metal layer and two polysilicon layers. The second poly is used for making capacitors. Capacitors are useful in analog circuits, but are seldom used in digital circuits. Again, such a technology is not very advanced. Many processes nowadays have two metal layers. With two metal layers, not only can layouts be made a lot more compact, but the circuits
can be made to run much faster. This is because polysilicon layers usually have a sheet resistance orders of magnitude larger than metal, and the RC delay caused by long polysilicon lines is usually the cause of timing bottlenecks in a digital integrated system. The second polysilicon layer is useless to the present project. Because it is deposited after the field oxide is deposited, transistors cannot be made with this layer. Moreover, polyl cannot be crossed with poly2, since wherever poly2 runs over polyl, a fairly large capacitor is formed, greatly increasing the RC delay and causing crosstalk.

A rough estimate of the number of transistors that can be integrated on a chip of size 7mm x 7mm with this 4μm CMOS process is about 20,000. This estimate assumes that the design is fairly irregular, such as a microprocessor, as opposed to a regular layout such as a memory chip.

4.2 Floating-Point vs Fixed-Point Implementation

Should the data be represented in fixed point or floating point format? No definite answer can be given before a thorough simulation is done. However, rough estimations can be made. If all distances are represented with a 16-bit fixed point as shown in Figure 4.2, then numbers from about -16 to +16 can be represented with a resolution of $2^{-11} = 4.8 \times 10^{-4}$. If metres are taken as the unit of measurement, then distances of up to 16m can be represented with a precision of about 0.5mm. By shifting the position of the binary point, larger numbers can be represented with a coarser precision, or smaller numbers with a finer precision. The effects of error propagation must also be considered. Consider for example $\overline{p}$ of the Excalibur manipulator,

$$p_x = C_1[a_2C_7 - d_4(S_{12}C_3 C_{12}C_4S_5)] + S_1(d_5 + d_4S_4)$$

Normally the accuracies of the sines and cosines of the angles are limited by the
accuracies of the sensors rather than the data format. Typically the accuracy of a position sensor is around 0.1% to 10%, much poorer than what is offered by the above format. Because the effect of the errors introduced by the fixed-point format is to be investigated, it is assumed that the sensors are as accurate as the data format. Also, one can almost always find more accurate sensors, although they would undoubtedly cost a lot more. With this assumption, a maximum of $\pi$ can be represented by the same format shown in Figure 4.2 with a resolution of $\pi/2048$, using only the lower 11 bits to represent the angle. The higher order bits can be used for memory partitioning. Such an arrangement would allow off-the-shelf sine/cosine generators such as the AM29526/27 and AM29528/29 to be used. If the host uses straight binary numbers to represent the angles, then a conversion is easily performed by multiplying the angles by a suitable conversion factor. This can be done by APR.

When sines or cosines are multiplied together, the error will only diminish because sines and cosines of any angle always have a magnitude less than one. Only when terms containing lengths are multiplied together will the errors be magnified.
Notice also that in this equation (and all the others in the direct transform of the Excalibur manipulator), it is never necessary to take the difference of two terms with magnified errors and then multiply it with a big number (larger than one). Assuming that the maximum length of each link is 4 meters (which is a more than reasonable assumption for most present day manipulators), the worst case error estimation is shown in Figure 4.3. It can be seen that there is an error of only about 1mm in the x-component of the position vector of the end effector. Error estimation is the same for the other vector components. The actual errors will be bigger because of the errors introduced by the sensors.

Since a 16-bit fixed-point number format seems to suffice, is there any reason why a floating-point processor should be preferred?

Floating-point representation requires more hardware. In VLSI, increasing the amount of hardware implies increased design cost and potentially slower operation

\[ P_x = C_1 \left[ a_2 C_2 - d_6 \left( S_{23} C_5 - C_{23} C_4 S_5 \right) \right] + S_1 \left( d_3 + d_6 S_4 S_5 \right) \]

\[ e=0.0001 \quad e=0.0007 \quad e=0.0005 \quad e=0.0003 \]

\[ e=0.003 \]

\[ e=0.004 \]

\[ e=0.001 \]

Fig. 4.3: Error Estimation of Number Format
speed, in addition to increased chip size. Floating-point arithmetic also takes a longer computation time than fixed-point arithmetic. But this difference in speed is diminished with the use of pipelined systems. Also, in possible applications of APR other than computing direct transforms, such as robot dynamics and control, computer graphics, and digital filtering, floating-point arithmetic is often necessary. Floating-point arithmetic can also increase the accuracy of direct transform computations due to reduction in error accumulation. Therefore, floating-point arithmetic is highly desirable.

Since the hardware for fixed-point arithmetic is basically a subset of the hardware for floating-point arithmetic, a processor designed for floating-point arithmetic can easily be extended to handle fixed-point arithmetic also.

4.3 On-Chip Memory vs Off-Chip Memory

The program for computing the transformation equations is static, i.e., it never changes for the same manipulator. Therefore, it is possible to put the program on chip as a ROM. RAM is not desirable because then the chip will have to be bootstrapped by writing to the RAM before any computation can start. The lower packing density of RAM also places a severe limitation on program size.

Since the instructions are likely to be very wide (32 bits), with the program on chip 32 pins can be saved. Instruction fetch cycles can also be faster with an on-chip ROM. But instruction fetch cycles are not the potential bottlenecks of the processor. Because of the special purpose nature of the chip, not too many will be made for any particular application. With the program stored in an off-chip PROM, program development is a lot easier. Larger programs are also more feasible this way. Therefore, the decision was made that the program would not be integrated on-chip.

As for data memory, they must be implemented in RAM. With the available ISOCMOS process, it is not possible to have a large on-chip RAM. A 4 Kbit RAM
requires 23 mm$^2$ with this process. Therefore, it is not feasible to put all the data memory on-chip. The best that can be done is to have a relatively large register file of about 32 units. Most general purpose processors have 16 general purpose registers or less. A large register file can reduce the number of memory accesses. Off-chip RAM must be used for the main data memory.

4.4 Multiplication

In computing the direct transform, many multiplications are required. Multiplications can be performed in computers by either shift and add, or by the use of a parallel multiplier. Performing multiplication by shift and add requires minimal hardware, but is very slow. Multiplying two 16-bit numbers together by shift and add requires about 6.4μs with a 4μm CMOS process. However, with suitable pipelining and parallel processing, high throughput can be attained in some applications [3]. Parallel multipliers, on the other hand, are a lot faster than their sequential counterparts. Off-the-shelf 16-bit integer multipliers today have multiply times of around 50 or 60ns (e.g. Weitek WTL 2010). But parallel multipliers require a lot of silicon real estate. A 16-bit parallel multiplier will require about 3mm x 3mm in a 4μm CMOS process. In the present project, it is not possible to effect high throughput by pipelining slow serial multipliers. It is also not possible to integrate a parallel multiplier on-chip. The best alternative then is to use an off-the-shelf high speed integer multiplier.

4.5 32 Bit vs 24 Bit Representation

Since the largest high speed integer multipliers available today are 16-bit multipliers, mantissa multiplication is fixed to 16 bits. However, it is still possible to have a full 32-bit (23 bit mantissa) IEEE floating-point standard [4] adder and full 32-bit registers. This, for example, is the approach taken by the TMS320 [21]. For
In the present application, a 16-bit mantissa appears sufficient. Besides, there are so many multiplications in this application (about twice the number of additions) that the extra accuracy offered by the 32-bit adder will probably not offer much better overall accuracy. Using a sign–magnitude representation for the mantissa (the IEEE standard), the magnitude of the mantissa can be represented with 16 bits on-chip, with another bit representing the sign, thus effectively giving a 25-bit representation. This extra 1-bit accuracy can only be preserved when operands are kept on chip. For communication with external memory or the host, it is better to keep the number format to 24-bits, which is more convenient because it is a multiple of 8.

4.6 Reduced Instruction Set

Reduced instruction set computers have become very popular recently, the most notable designs being the RISCs and MIPSs. A reduced instruction set implies a simpler control section, which, in turn, means reduction in chip area requirement, reduced design time, less chances of design errors, and faster execution speed if the instructions are chosen properly. Since APR is designed to be a special purpose arithmetic processor, its instruction set should therefore support only the functions that are necessary for this purpose, namely coordinate transformation in robotics.

4.7 Outline of Possible Architecture

In summary, then, APR should handle 32-bit floating-point and 16-bit fixed-point arithmetic. A parallel multiplier is needed, which will be an off-the-shelf high speed integer multiplier. The program memory is to be stored in an off-chip PROM. Except for a large high speed register file on chip, data memory is stored in an off-chip RAM. The control section must be made as simple as possible, implementing only instructions that are needed most often.
5. THE APR ARCHITECTURE

A block diagram of the overall architecture of APR is shown in Figure 5.1. APR can handle both 24-bit floating-point numbers and 16-bit fixed-point numbers. Figure 5.1 shows APR in its floating-point mode. This high speed floating-point unit design is a conventional design similar to that described in [1]. Both the adder section and the multiplier section are divided into a mantissa half and an exponent half. A 25-bit by 31 word 3-port register file is shared by the two sections. The internal representation of the mantissa in APR is always 17 bit sign-magnitude. The exponent is represented by 8 bits with a $+127$ bias. This format is close to the IEEE floating-point standard and will thus allow host processors that use this IEEE standard to send data to APR directly, albeit with a loss of 8 bits of accuracy because of the shorter mantissa used in APR.

5.1 The Register File

The register file has two read buses and one write bus. During every cycle, the register file operates twice. Firstly it supplies the two operands to the multiplier, while at the same time storing back the current result from the multiplier section. Secondly, it does the same for the adder section. Because N-channel pass transistors are used for the read and write enable switches in the register cells, the read buses are precharged before every run. The write bus does not need to be precharged because it is sourced by full CMOS circuits. More will be said about the register file in section 7.
Fig. 5.1: Block Diagram of APR in Floating-Point Mode
5.2 The Adder Section

In floating-point mode, the adder section is a 5 stage pipeline, similar to that described for APAC [20]. Operands are fetched from the register file in the first stage and, in the next stage, the exponents are compared. The larger exponent is passed to the next stage in the exponent half, and the mantissa of the smaller number is shifted down the appropriate amount to align the binary point. The mantissas are then added or subtracted in the next stage, while the exponent goes through a one stage delay. In the fourth stage, the mantissa is normalized and the exponent updated. Overflows and underflows are checked in this stage, and the appropriate flags set. In the final stage, the result is stored back in the register file. If the hard limit flags are set, the number stored will be set to the maximum magnitude representable with the same sign, or zero, depending on whether overflow or underflow has occurred respectively. The adder mantissa section consists of a fraction alignment 16-bit barrel shifter, a fast 18-bit adder, a leading zero checker, and a normalization 16-bit barrel shifter. The exponent half consists of an exponent comparison subtractor, and an exponent adjustment adder. The use of sign-magnitude format, together with the requirement that the mantissa must never overflow (overflow and underflow must be handled in the normalization stage), forces the use of an 18-bit adder to handle the 17-bit numbers. Also, the mantissa adder must be implemented with an end-around-carry adder. This is necessary because in sign-magnitude representation, the magnitude of a number must always be positive. Therefore, when the difference between two mantissas is negative, the magnitude of the mantissa must be kept positive with the sign changed to indicate a negative number. This is best done with an end-around-carry adder using 1's complement to represent negative numbers [1]. However, this in effect requires the adder to operate twice during each clock cycle, so a fast carry look-ahead scheme must be employed. The "par-add" scheme developed by Brent and Kung [2] and recently implemented in
Both barrel shifters use the design described in RISC II [34]. The fraction alignment shifter is a down shifter, while the normalization shifter is an up shifter. The leading zero checker detects the number of leading zeros in the adder result and controls the normalization shifter so that the mantissa always comes out normalized in floating-point mode. It also feeds the shift amount to the exponent adjust adder, which adjusts the exponent of the result accordingly. In order to synchronize the mantissa and exponent halves, one extra delay stage has to be added in between the exponent comparison subtractor and the exponent adjustment adder. Both these adders are regenerative Manchester carry adders [14]. The exponent comparison subtractor is an 8-bit adder, whereas the exponent update adder needs to be 9 bits wide to distinguish between overflow and underflow. When either overflow or underflow occurs, the result is hard limited to the largest magnitude representable or zero, respectively. Because it is not known which input to the exponent comparison adder will be the larger, the difference may come out to be either positive or negative. This is handled by a PLA that maps both a positive and a negative difference to a positive shift amount which is then used to control the first barrel shifter.

5.3 The Multiplier Section

The multiplier section is a 4-stage pipeline. In the first stage, the operands are fetched and, in the next stage, the mantissas are multiplied by the parallel multiplier. The exponents just go through a delay in this stage. In the third stage, the result mantissa is normalized by the barrel shifter. The exponent section in this stage is split into two sub-stages. In the first sub-stage, the two exponents are added together. In the second sub-stage, the exponent bias is restored by subtracting one bias amount from the result. The effect of shifting the mantissa is taken care of by either feeding a 1 or 0 to the carry in of the exponent adder. For the same reason presented in
the adder section, this exponent update adder in the multiplier section must be 9 bits wide. The correct result is then stored back in the register in the fourth stage.

In floating-point multiplications the resulting mantissa never needs to be shifted more than one position, so there is no need for a barrel shifter and a 16-bit leading zero checker. These two blocks are needed, however, in fixed-point mode, when the numbers are not normalized.

5.4 The Clock Phases

In order to synchronize the above operations, a clock with more than two phases is required. The required clock phases are shown in Figure 5.2, and are

![Clock Phases Diagram]

Fig. 5.2: Clock Phases Required in APR
generated as described in section 7.5. The operation of the multiplier section in floating-point mode is as follows: during $\phi_1$, the register file is precharged. In $\phi_3$, operands to the multiplier are read from the register file. The tri-state pads connecting APR to the external multiplier are enabled as output pads during all of $\phi_2$, so that the operands are driven out through the pads with all the duration of $\phi_2$ available. During the next $\phi_1$, the operands are fed to the inputs of the multiplier. The multiplier operates in the ensuing $\phi_2$, and the product is sent back to APR in $\phi_1$. The product is normalized in $\phi_2$, and then stored back in the register file in $\phi_3$, after a delay in $\phi_1$.

In the adder section, the operands are read from the register file in $\phi_4$. The fractions are then aligned during the next $\phi_2$, after a simple register transfer in the $\phi_1$ between. Then the aligned fractions are added in the next $\phi_2$. The sum is normalized in another $\phi_2$, and the normalized sum finally stored back in the register file in the next $\phi_4$.

It should be noted that APR basically operates with a two phase clock consisting of $\phi_1$ and $\phi_2$. The phases $\phi_3$, $\phi_4$, and $\phi_5$ are needed so that the register file can be operated twice every machine cycle.

5.5 Fixed-Point Mode

Figure 5.3 shows APR in fixed-point mode. In this mode, the two barrel shifters in the adder section are bypassed, so that the adder section becomes a 3-stage pipeline. The multiplier section stays as a 4-stage pipeline, but the shift amount is now a constant controlled by the control unit. The location of the binary point is controlled by the program, and the barrel shifter in the multiplier section is used to realign the binary point after a multiplication. The exponent half of the adder section is completely shut off in fixed point mode, as is the register file and the
Fig. 5.3: Block Diagram of APR in Fixed-Point Mode

- Barrel Shifter
- Leading Zero Checker
- Multiplier (off-chip)
- Register File for Mantissa
- Adder
- Data/Address
- Data I/O
- CCR
- Control
- IR
- PC
- Adder
- Instruction

- Shift amount
- Control signals
- Required no. of leading zeros

Program Address
exponent half in the multiplier section. However, the adder in the multiplier section is used to check for overflows. This is achieved by subtracting the number of leading zeros in the result from the multiplier from a preset value and flagging any negative differences.

5.6 Input/Output and Control

For communication with external memory, APR can perform both direct and indirect read and write with 24-bit words. In order to make APR compatible with 8-bit wide system data buses, APR can also do direct read and write with byte wide words. No indirect read/write instructions are available for byte wide data.

Table look-up for sines and cosines, as required in the computation of direct transforms, and for DMA to the system memory is provided by appropriate memory mapping. The condition code register has 9 flags. The overflow, zero, and negative flags are set by the data path in the usual way. Of the six remaining flags, 3 are input flags which are set by external circuits, and 3 are output flags that can be set by the APR control unit. These flags can be used for interrupt requests, bus grant, and other control signals.

The program counter is 16 bits wide, and is not connected to the data path. Only absolute direct jumps are allowed. Conditional jumps are possible, using the zero flag as the jump condition. In order to allow simple subroutine calls, a two level stack is also included in the PC section.

Because of the simple instructions required, the control section comprises only a simple PLA. Figure 5.4 shows the instruction fetch and execution pipeline. Every cycle a new instruction is fetched from the program memory and decoded in the following cycle. Execution of the instruction is then started in the following cycle, although the instruction may take more than one cycle to finish. As in RISC and MIPS, and for
reasons clearly described in [9], the instruction fetch and execution pipelines are not interlocked. In other words, when a conditional branch is encountered, the prefetched instructions are always executed, regardless of the outcome of the condition test. However, unlike the RISC and MIPS design, there is no internal forwarding [7] in the data path. This makes the design of the control section quite simple, provided that the speed penalty due to the extra delays can be reduced by suitable rearrangement of the instructions.
6. PROGRAMMING AND SCHEDULING

The efficiency of APR depends entirely on the efficiency of the program. Because much of the processing speed of APR comes from its multi-pipelined architecture, its throughput can be widely different for programs that can keep the pipelines full at most times and programs that cannot. There are three major pipelines in APR: the adder pipeline, the multiplier pipeline, and the instruction fetch and execution pipeline. The block diagrams of these pipelines have been shown in the previous section.

An operation can be started in the adder or multiplier pipeline only when both operands are in the register file. APR does not make any effort to check whether the operands are valid or not. This task is left to the programmer. Therefore, as an example, if the following two instructions are to be executed with APR in floating-point mode:

   i1: C = A + B
   i2: D = C + 2

then instruction i2 can be started only after at least five cycles from the time i1 was started. But other instructions that do not require 'C' can be started immediately after i1. Therefore, the instructions must be arranged in such a way that the pipelines are filled most of the time. The instruction fetch and execution pipeline is easier to handle. The only complication arises when there is a conditional branch in the program. APR does not provide any facility to flush the instruction pipeline when a branch occurs. This is what is called non-interlocked pipelines in MIPS. The instructions that are prefetched into APR will always be executed, regardless of the outcome of any conditional branches. The programmer must therefore make sure that the two instructions after a conditional branch that are prefetched into APR will not cause any errors when executed. It may be possible to rearrange the instructions so
that these two instructions will actually do some useful work. Otherwise, NOP (no operation) instructions must be used to fill up this gap. As pointed out in [9], this requirement is often more of a blessing than a curse. Not only is the design of the processor easier with non-interlocking pipelines, but the two prefetched instructions which are wasted in a pipeline that can be flushed can now be made to do useful work.

It is rather difficult to write an efficient program to schedule a number of computing resources, even though the present case has only two resources, namely the adder section and the multiplier section. The subject of scheduling multiple resources has been studied extensively in the area of micro-programming. In fact, the programs for APR are somewhere between assembly language programs and micro-programs. As an aid to the programmer, the assembler is written to take into account the possible parallel operation of the adder and the multiplier sections, and the various pipeline relationships. The branch and bound algorithm with list scheduling heuristic [19] is used in the assembler. It requires the programmer to arrange the addition and multiplication operations as a data dependency graph [19]. The close to optimal program is then generated automatically in a very short time. It is rather difficult to include the effects of a finite number of registers in a scheduling algorithm, so the effects of the limited number of registers available on-chip is largely ignored in the present assembler. The scheduling algorithm is described in more detail in Appendix D.
7. DESIGN OF CELLS

It is well known that in VLSI, communication is far more expensive than computation [22]. However, the notion that communication must be modular and local is rather futuristic. Classical von Neumann architecture does not lend itself easily to systems requiring little or no global communication. Also, the complexity and density of most VLSI chips today have not reached the point where multiple self-contained systems can be integrated on one chip. Therefore, it is conceivable that most, if not all, processor designs today utilize global signals that run across the span of the chip.

Nevertheless, in VLSI, even local communication must be designed carefully. Take, for example, the adder mantissa section of APR (see Figure 7.1). The register file, barrel shifters, leading zero checker, and adder are all of different heights and widths in their most compact form. If the layout is designed following exactly the arrangement of the block diagram, a lot of area is wasted (see Figure 7.2). Although

![Fig. 7.1: Communication in Adder Mantissa Section](image-url)
Figure 7.2 is only symbolic, it is obvious from it that a lot of silicon real estate is used up by the wires. A better design is to stretch all the building blocks to the largest height so that they can be butted together snugly. Although this implies less compact layouts for the smaller blocks, the savings in routing area will usually lead to a much more compact overall design. Also, since the adder design adopted in APR is much more compact when the sum is generated on the same side as the inputs, the overall design will be more compact if the adder is placed at the end of the data path. The final design is shown in Figure 7.3. Notice that there are no 'bare' busses in this arrangement. Such considerations were taken into account in the design of the basic cells. The design of the more important cells is described in this section. The following set of rules of thumb is used as a guideline in the design of the layouts:

1. Extensive layout compaction is applied only to cells that have a high repetition factor, such as the register cells and the rest of the data path. For cells that are not repeated a lot, logic gates from a standard cell library are used. This
Fig. 7.3: Final Floor Plan of Adder Mantissa Section

reduces design time while compromising little in terms of overall layout compactness.

2. Full CMOS is used as much as possible in order to reduce power consumption, except in places where the use of pseudo-nMOS can result in significant reduction in real estate requirement.

3. P-channel transistor are made 2.5 times the size of their corresponding N-channel transistors, in order to have symmetrical rise and fall times.

4. Minimum size transistors (with p-channel transistors 2.5 times the size of N-channel transistors) are used everywhere except where high current drive is required. This not only keeps the area required to a minimum, but also reduces the loading on the drivers that drive the transistors, which in turn reduces the overall delay.

5. Metal wires are used for signal communication as much as possible, with polysilicon used only for short distances or crossunders.
6. Power and ground wires are always routed in metal. When signal lines cross the power lines, the signal lines must cross under the metal power lines in polysilicon. In rare cases where power must cross ground, an N+ diffusion layer inside a P-well is used as crossunder for the ground wire. N+ diffusion layers are preferable to polysilicon because N+ diffusion has a smaller sheet resistance than polysilicon, but has a larger capacitance. Therefore N+ diffusion is better than polysilicon for conducting ground lines but worse for connecting signal lines, since the signal level in the ground lines never has to change, so the increased capacitance has no serious effects.

7. Manhattan geometry is used except where significant savings in area can be realized with the use of 45 degree geometry. This makes a conversion of the design to a process supporting only Manhattan geometry easier, although there are currently no plans to do that.

The designs were done on a Metheus λ750 VLSI work station. The designs were simulated extensively wherever possible using the simulation tools available. These tools include:

1. SPICE: a circuit level simulator. For accurate analog simulations of small circuits consisting of up to 20 transistors.

2. Hg: a switch level simulator. For digital simulations, using a switch model for MOS transistors. Circuits containing a few hundred transistors can be simulated conveniently with this simulator.

3. HILO: a gate level simulator. For digital simulations of large complex digital systems. With functional simulations [13], theoretically whole chips and systems can be simulated. But because it cannot be used with the circuit extractor, it was only used sparingly.

The layouts have been thoroughly checked for design rule violations using the DRC (Design Rule Checker) on the Metheus VLSI workstation.
7.1 Multi-port Register Design

In almost any digital processor design, a number of multi-port registers are needed. A register is basically a RAM word. The difference (if any) between what is usually called a register file and what is called a RAM is that a register usually has more than one port, with a small number of registers in a register file, and the file is usually placed in the data path itself. This is necessary because by design, registers are used very often, typically once every machine cycle. A multi-port register file would allow the processor to run faster. It is not practical to have a very large register file, because larger files require longer access time [24]. Historically, there is a big difference between registers and RAMs in that registers are integrated in the same chip as the processor, whereas RAMs are in separate packages. With the advent of VLSI, on-chip RAMs are becoming more and more common, and the difference between registers and RAMs is fading. In fact, in RISC II [34], the terms are used interchangeably.

The register cell design used in APR is shown in Figure 7.4. This is a 3-port register cell, with two read busses and one write bus. During every cycle, two words can be read from the register file and one word can be written to it. A write operation proceeds as follows. For every register cell, when write is high, M5 is on and M0 is off. The write bus is then connected to the input of the inverter formed by M1 and M3. The signal on the write bus is thus stored in the gate capacitance of this inverter. When the write control signal returns to low, M5 is turned off and M0 is turned on. The write bus is then disconnected from the register cell. Since M0 is on, the output of the inverter formed by M2 and M4 is connected to the input of the inverter formed by M1 and M3, forming the familiar cross-coupled static memory cell, and the information stored can be retained indefinitely. This completes a write cycle.
For a read cycle, either ReadBus0 or ReadBus1 can be used. When Read0 goes high, M6 is turned on. This connects ReadBus0 to the output of the inverter formed by M2 and M4, and the logic level of ReadBus0 will follow the output logic level of this inverter. The same description applies for ReadBus1. With two read busses and one write bus, two words can be read from the register file and one word written to it every cycle. This is very desirable because both the adder and the multiplier requires two operands and produce one result.
In order to save area, pass transistors must be used instead of transmission gates for the read and write enable switches (M5, M6 and M7). In CMOS, either N- or P-channel transistors can be used for pass transistors. P-channel transistors are typically 2 to 3 times slower than N-channel transistors, because of the lower mobility of holes compared with that of electrons. However, in a P-well process, N-channel transistors reside in a P-type tub of slightly higher doping density than the substrate. This worsens the body effect, and N-channel pass transistors lose more than a gate threshold in passing a high signal. If one end of an N-type pass transistor is held at 5V, the other end only reaches about 3V. On the other hand, a P-channel transistor can pass a low signal (0V) to around 1.1V.

N-channel transistors have been chosen for implementing the register cell because of their superior speed. In order to combat the threshold losses, the read busses are precharged. When the register file is not used, the read and write control lines are driven to zero volts, and the read busses are precharged to Vdd. When the register file becomes active in $\phi_2$, the capacitance of the read busses will keep the voltage of the busses close to Vdd. Therefore if the register cell contains a $I$, it will not have to pull the bus high, since the bus is already high. On the other hand, if the cell contains a $0$, the bus will have to be pulled low. This presents no problems since N-channel transistors can pass a low signal without any loss.

The SPICE simulations of the register cell are shown in sections B.1 to B.6 of appendix B.

- In B.1, a $0$ is written to the register.
- In B.2, a $1$ is written.
- In B.5, it is sourcing a $0$.
- In B.6, the register is sourcing a $1$ to a read bus.

The case that presents the most concern is B.2, since the signal after the write enable
pass transistor M5 barely reaches 3V, just slightly above the threshold voltage of 2.5V for a standard inverter. The noise margin has been increased by making the N-channel pull down transistor M1 bigger than the P-channel pull-up M3. In appendix B.3, the write signal is reduced to 4V, and it is shown that the register cell can still record the high signal properly. This marginal state exists only when the register is being written to. As soon as writing is complete, M5 will be turned off and M0 will be turn on, restoring the logic level to much safer values.

The register file is divided into two sections, consisting of 31 x 16 and 31 x 9 units of this register cell. The bigger file is for the mantissa, while the smaller one is for the sign and the exponent. Driving the register file are three 5-to-32 decoders, one for each bus. The decoders are basically structured like a PLA, but with no OR planes, since they are not needed. From simulations, the time required to disable the read/write switches and precharge the busses is found to be about 20ns. The time to decode an address is about 10ns. And the time for a register cell to be enabled and sink a 1 from the write bus is about 30ns (there are other possible activities, such as sourcing a 1, but they are not worst cases). Therefore, the register file can be run at a minimum cycle time of 60ns. See Appendix A.1, 2, and B.1-7.

7.2 Barrel Shifter

The design of a versatile and compact barrel shifter has been thoroughly discussed in [34]. The circuit diagram of a 4-out-of-7 barrel shifter is shown in Figure 7.5. This barrel shifter design can serve as either an up shifter or a down shifter (it can also rotate a word, but this capability is not utilized in APR). Figure 7.6a shows the barrel shifter connected as a down shifter, and Figure 7.6b shows it working as a up shifter. In Figure 7.6a, the input 4-bit word is fed to ports In0 to In3, where the MSB is connected to In3. If sh3 is high and sh0 to sh2 are all low, then Out3 is connected to In6, and Out0 is connected to In3, etc. Therefore, the MSB
of the input word becomes the LSB of the output, and Out1 to Out3 are filled with 0's. This has the effect of shifting the input word down by three positions. To shift the input word up by two positions, sh2 is set to a 1 state. Then Out3 is connected to In5, and Out0 is connected to In2, etc. When sh0 is high, the input word is passed directly to the output.

Fig. 7.5: 4-out-of-7 Barrel Shifter
In Figure 7.6b, the input word is connected to \textit{In3} through \textit{In6}. The operation is very similar to that of the down shifter. Except that now enabling \textit{sh0} will shift the input word up by 3 positions and enabling \textit{sh3} will pass the input word to the output directly.

Since the barrel shifter is an nMOS pass transistor design, the threshold drop problem is also present. This implies that the output lines of the barrel shifter must also be precharged.
7.3 Leading Zero Checker

A 4-bit leading zero checker design is shown in Figure 7.7. This circuit is basically a PLA with no OR plane. The position of the first non-zero bit can be obtained both along the direction of the data path or at right angles to it, making it

Fig. 7.7: 4-bit Leading Zero Checker
particularly suitable for part of a data path. Simulations show that the worst case delay for a 16-bit leading zero checker is about 6ns. The number of leading zeros in binary code can also be obtained with the addition of an OR plane.

7.4 Adders

There are two adder designs used in APR. For the small adders and the program counter adder, regenerative Manchester carry adders were used. For the fast mantissa adder, a carry look-ahead adder was used. The regenerative Manchester carry scheme [14] is basically a Manchester carry scheme [22]. The $P, G$ and $K$ signals are generated as in normal Manchester carry adders and a pass transistor is also used for the carry propagation chain. The difference is that the carry signals are regenerated at every stage. In a dynamic regenerative adder (see Figure 7.8), each section of the carry chain is pulled down to GND in the precharge phase. This is equivalent to a carry kill at every bit. No kill ($K$) signal needs to be generated. In the next (active) phase, if propagate ($P$) is high, which implies generate ($G$) is low, the carry-in signal is passed to carry-out. In a normal Manchester carry adder, the level of the signal at carry-out will not be the same as that at carry-in. This is because there is a potential drop across M0. However, in the regenerative Manchester carry, if the carry-in signal is high, M1 will pull carry-out all the way to Vdd. If the carry-in signal is low, M1 will be off, and the section of the carry chain will stay at GND, since it is initially set to GND. The logic levels can be restored at every stage in the normal Manchester carry scheme too, but the two extra gate delays introduced by the doubly inverting buffers will more than offset the speed improvement gained by the restored logic levels.

When $P$ is low, the adder behaves as follows. If $G$ is high, a carry of 1 is generated by M1. If $G$ is low, M0 and M1 are both off, and the carry-out stays at GND. This is the same as killing the carry.
A static regenerative adder scheme is also possible (Figure 7.9). This is done by adding some circuitry to the dynamic design to implement a carry kill. Static designs are more versatile, since no clock is required. Therefore they can be operated asynchronously. Although the static design has a longer cycle time than the dynamic design, when the adder is used as part of a larger design, the overall speed of the design may be faster with a static adder. For example, in one machine cycle in APR, the number of leading zeros in a number must be determined and then added to the original exponent in the multiplier of APR. With a dynamic adder, the machine cycle must be split into two sections. The first section must be long enough for the leading zero checker to reach a stable (correct) result. And the second section must be long enough for the adder to complete the addition. If a static adder is used, then it is
only necessary that the total time required for the leading zero checker and the adder is less than the period of one machine cycle. The static scheme does have some drawbacks, though. The transistor M7 presents extra load on the carry chain. Also, when G or K is low, they may have to work against the carry-in signal, so transistor sizing is very critical in a static design. The static regenerative adder design was chosen for all the adders on APR except the high-speed 16-bit mantissa adder. Simulations show that the carry propagation time for one stage of the static regenerative adder is about 7ns. See Appendix B, section B.9. Therefore, an 8-bit regenerative adder will require about 60ns to perform an addition.

Fig. 7.9: Static Regenerative Adder
A carry look-ahead scheme that is particularly suitable to VLSI has been described in [2]. The "Par-Add" scheme, as it is called in [37], is a very regular and modular design, making it very suitable for an algorithmic design, i.e., by specifying only the size (number of bits) of an adder, its layout can be generated automatically by a program. Par-Add is essentially a partial carry look-ahead tree across groups of two bits. With TTL components, carry look-ahead is usually across groups of four bits. But MOS transistors have less current drive than TTL circuits. So even carry look-ahead across groups of four bits can significantly increase delays due to large capacitive loads on the outputs.

A block diagram of a 4-bit par-add adder is shown in Figure 7.10, where A

Fig. 7.10: 4-Bit Adder with Partial Carry Look-Ahead
and B are the numbers to be added, and \( S \) is the sum. For the blocks labeled FA (for Full Adder),

\[
\begin{align*}
P_{\text{out}} &= A \cdot B \\
G_{\text{out}} &= A \cdot B \\
S &= A + B + C
\end{align*}
\]

For the PA (for Par-Add) blocks,

\[
\begin{align*}
P_{\text{out}} &= P_0 \cdot P_1 \\
G_{\text{out}} &= G_1 + (P_1 \cdot G_0) \\
C_i &= G_0 + (P_0 \cdot \text{Cin}) \\
C_0 &= \text{Cin}
\end{align*}
\]

Notice that all the PA blocks implement the same logic. This is why the circuit is modular and regular. If \( N \) is the width of the adder in number of bits, then the speed of the adder decreases as \( \log_2 N \) as opposed to \( N \) in an adder without carry look-ahead, such as a Manchester carry adder. The original design in [37] is for an nMOS process. Since APR is to be implemented in CMOS, the design has to be modified. With CMOS, the PA blocks cannot be implemented in full complementary MOS. Full CMOS requires about four times the amount of fan-out of pseudo-nMOS or domino CMOS [18]. Domino CMOS is particularly suitable for this application since it works with non-inverting logic (i.e. the basic building blocks are AND and OR gates, rather than NAND and NOR gates), which is exactly what the PA blocks require. Therefore, domino CMOS was chosen to implement the mantissa adder, which must be much faster than the other adders because it is 18 bits long, and is an end-around-carry adder. The circuit diagram of a PA block in domino CMOS is shown in Figure 7.11. Because domino CMOS is dynamic, making an end-around-carry adder using domino CMOS presents extra problems. The carry-in of an end-around-carry adder does not become stable until about half way through the
Fig. 7.11: Circuit Diagram of a Par-Add Block
overall add time of the adder. But dynamic circuits require their inputs to be stable before they start operating, otherwise incorrect results will occur. This requirement is handled in the mantissa adder by precharging the adder during $\phi_0$, and operating the adder during $\phi_3$ and $\phi_4$. During $\phi_3$, the carry-out is established and latched. During $\phi_4$, the carry-in is stable and correct, and the sum generated is valid. Figure 7.12 shows the schematic of this arrangement.

Fig. 7.12: End-Around-Carry Adder with Domino CMOS
7.5 The Clock

The clock phases required by APR are shown in Figure 5.2, on page 30. Basically, each machine cycle is split into four phases, two relatively short phases (the \( \phi_0 \)'s), and two relatively long phases (\( \phi_3 \) and \( \phi_4 \)). The reason that these phases are required has been described in section 5.4. This section is concerned with how such phases can be generated in VLSI.

In most processors, multi-phase (typically two-phase) clocks are generated from a single phase clock. The single phase clock can be generated either off-chip or on-chip. Depending on the accuracy required of the clock, or the amount of clock frequency drift that can be tolerated, such as due to temperature changes and aging, a crystal or an RC circuit may be used to provide the oscillation. The subject about generation of single phase clocks has been discussed in detail in [22].

APR runs on its own clock, asynchronous with that of the system clock, if there is any. This is necessary to make full use of the speed of APR. For a prototype chip, it is not wise to use a totally internally generated clock using a ring oscillator [22]. The reason is that the timing estimations may be inaccurate. For example, at present the minimum width of \( \phi_2 \) is estimated to be 160ns. If a ring oscillator is used to generate the base clock, the number of stages in the ring oscillator must be adjusted so that \( \phi_2 \) is around but above 160ns. If the minimum phase width for \( \phi_2 \) is longer than estimated, i.e. longer than 160ns, then APR will not work, and there is no way to change the clock frequency after the chip is fabricated. On the other hand, if the minimum phase width is shorter than expected, then APR will run at less than its maximum speed. Therefore, an external oscillator is much more desirable for generating the base clock. With an external oscillator, using either a crystal or an RC circuit, the clock frequency can easily be changed. From here on it is assumed a single phase symmetrical square wave of selectable
frequency is available from off-chip sources.

Given a symmetrical single phase clock, the problem now is how to generate the phases $\phi_0$ to $\phi_4$ shown in Figure 5.2. Apart from the obvious relationships shown, the following requirements should be noted:

1. $\phi_1$ is high only when $\phi_0$ is high.
2. $\phi_1$ and $\phi_2$ must not overlap.
3. $\phi_3$ and $\phi_4$ should be about two or three times the width of $\phi_0$.
4. $\phi_3$ and $\phi_4$ must not overlap $\phi_0$.

A method to generate a two phase non-overlapping clock from a single phase clock is described in [22]. To generate the clock phases required in APR, a more sophisticated scheme is necessary. The heart of the scheme used is a dynamic shift register (see Figure 7.13). The function of the N-channel pull-down transistors will be explained later in this section. The shift register is controlled by a two phase non-overlapping clock, generated from the base clock (clkin in Figure 7.13) using the method in [22]. With the base clock running, $s0 = 1$, and $s1$ to $s5 = 0$ on startup, the signals $s0$ to $s5$ will take the shapes shown in Figure 7.14. From these signals, $\phi_0$ to $\phi_4$ can be generated as follows:

1. $\phi_0 = s0 + s3$
2. $\phi_1 = s0$
3. $\phi_2 = s1 + s2 + s3 + s4 + s5$
4. $\phi_3 = s1 + s2$
5. $\phi_4 = s4 + s5$

However, there is still one problem. The non-overlapping requirements are not guaranteed by the shift register. It is possible to estimate the delay of the slowest clock path and then insert enough number of idle base clock periods in between the different clock phases. For example, if the signals $s0$ to $s5$ are to be used to generate a two phase non-overlapping clock, then the phases can be generated by the
Fig. 7.13: Shift Register for Multi-Phase Clock Generation
Fig. 7.14: Signals from Clock Generation Shift Register

following logic equations:

\[ \phi_1 = s_1 + s_2 \]
\[ \phi_2 = s_4 + s_5 \]

The duration of \( s_0 \) and \( s_3 \) provide the necessary non-overlap periods. If more overlapping period is required, the number of idle periods can be increased.

A better approach is to use the feedback technique described in [22]. An example to generate a two phase non-overlapping clock is shown in Figure 7.15. It should be noted that the signal that is being fed back to the inputs of the generator must be from the end of the critical path (slowest path). Otherwise non-overlapping cannot be guaranteed. Using this technique, the corresponding signals that are fed to
Fig. 7.15: Two-Phase Clock with Feedback to Guarantee Non-Overlapping

the various generated signals as shown in the following table.

<table>
<thead>
<tr>
<th>Signal Generated</th>
<th>Signal Fed Back</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\phi_0$</td>
<td>$\phi_3 + \phi_4$</td>
</tr>
<tr>
<td>$\phi_1$</td>
<td>$\phi_2 + \overline{\phi_0}$</td>
</tr>
<tr>
<td>$\phi_2$</td>
<td>$\phi_1$</td>
</tr>
<tr>
<td>$\phi_3$</td>
<td>$\phi_0$</td>
</tr>
<tr>
<td>$\phi_4$</td>
<td>$\phi_0$</td>
</tr>
</tbody>
</table>

There is still one potential problem in this clock generator. The proper functioning of the shift register clock generator depends on the fact that at any time, one and only one of the $s0$ to $s5$ signals is high. Although it is a relatively simple task to set the $s0$ to $s5$ signals to the proper state on start-up by using the reset line, measures must still be taken to ensure that if there is more than one high
signal or no high signal at all in $s0$ to $s5$, due to noise perhaps, then the shift register must be able to correct itself after at most one machine cycle. The N-channel pull-down transistors shown in Figure 7.13 are included for this purpose. When $s0$ is high, $s1$ to $s5$ are forced to low. On the other hand, when all of $s1$ to $s5$ are low, $s0$ is forced to a high.

7.6 Programmable Logic Array

There are six programmable logic arrays (PLA) used in APR:

1. Instruction decoder.
2. Stack controller.
3. Fraction alignment barrel shifter controller.
4. Register file decoder.
5. Leading zero checker in the adder section.
6. Leading zero checker in the multiplier section.

All these PLA's use a static pseudo-nMOS design. An example of such a design is shown in Figure 7.16, which is the stack controller used in APR. The logic equations that are implemented are:

\[
\begin{align*}
\text{out}_0 &= \text{in}_0 \cdot \text{in}_1 \cdot \overline{\text{in}_3} + \text{in}_0 \cdot \text{in}_1 \cdot \overline{\text{in}_2} + \overline{\text{in}_1} \cdot \text{in}_4 \\
\text{out}_1 &= \text{in}_0 \cdot \text{in}_1 \cdot \overline{\text{in}_4} + \overline{\text{in}_0} \cdot \overline{\text{in}_4} \\
\text{out}_2 &= \overline{\text{in}_0} \cdot \overline{\text{in}_4} \\
\text{out}_3 &= \overline{\text{in}_1} \cdot \overline{\text{in}_2}
\end{align*}
\]

Pseudo-nMOS static PLA's are preferable to dynamic PLA's or full CMOS static PLA's. Dynamic PLA's must be clocked, placing limitations on when inputs can be fed to the PLA and when outputs can be obtained from it. Full CMOS PLA's are more than twice the size of pseudo-nMOS PLA's. Since the speed of a PLA decreases as its size increases, the saving in power consumption is usually not worth the sacrifice in speed and area.
Fig. 7.16: PLA for Stack Controller
7.7 Stack

The stack design described in [22] is used for the program counter stack in APR. The only modification made is that P-channel transistors are used instead of N-channels for the pass transistors. The circuit diagram of a 2-bit by 2-level stack is shown in Figure 7.17.

Fig. 7.17: 2-bit by 2-level Stack
8. FLOOR PLAN OF APR

The logic design of APR has been carried to completion. A floor plan of APR has been investigated (Figure 8.1). With due allowances made for routing channels, the chip is estimated to measure approximately 8.5mm x 8.5mm. There are 83 pads on the periphery, including the power and ground pads. It can be seen in Figure 8.1 that a significant portion of the chip is devoted to routing. This exemplifies the statement in section 7 that communication in VLSI is more expensive than computation. The design of an efficient floor plan is thus of primary importance for a compact chip layout. With a single metal layer and a single polysilicon layer for interconnections, the routing problem is further complicated because polysilicon wires must be used as scarcely as possible, or significant deterioration in processing speed will result. Unexpected RC delays caused by long polysilicon wires are probably one of the most common causes of chip design failures. Until better simulation software becomes available, all that can be done to increase the chance of a successful chip implementation is to keep the design simple and regular, and to be extremely careful in every stage of the design.

Although chip sizes larger than 70mm² are not uncommon these days (see section 3.1), such a large chip is not feasible with the available ISOCMOS process due to poor yield. Also, if a 2µm process is available, APR will measure only about 4.3mm x 4.3mm. Therefore, although the APR design is definitely feasible with present day IC technology, it cannot be fabricated with a high degree of confidence using the technology available to the author. Because APR is actually more powerful than is required for coordinate transformations in robotics, it is, however, possible to fabricate a chip that contains only the subset of the functions of APR which are essential for the computation of coordinate transformations. Not only can such a chip serve as a testing vehicle for the APR, but it can also be used immediately in industrial robots.
1. Pad frame
2. Support circuitry for multiplier section
3. Register file for mantissa
4. Decoders and drivers for mantissa register file
5. Drivers for exponent register file
6. Register file for exponent and sign
7. Adder section
8. PLA for pre-shift control
9. Pipeline registers for mul. sect. exponent
10. Adder for mul. sect. exp.
11. Exponent adjust adder in adder sect.
12. Shift registers for register sinks
13. Shift registers for register sources
14. Instruction decoder
15. Program counter
16. Clock generator
17. Latches for literals and other constants
18. Exponent comparison adder in adder sect.
19. I/O multiplexer for mantissa
20. OA-plane for L2C in mul. sect.
21. I/O multiplexer for exponent
applications. This simplified design is called APRjr.
9. THE APRJR

The essential difference between APR and APRjr is that the latter only handles fixed-point numbers. For the specific task in mind, namely the computation of the direct transform, a fixed-point representation should be adequate for most robot arms. The analysis concerning the adequacy of fixed-point representation in the computation of the direct transform has been discussed in section 3.2. The circuitry for performing the fixed-point calculations in APRjr is the same as described for APR. Figure 9.1 shows a block diagram of APRjr. The APRjr design is a subset of the APR design. The entire exponent part, the two barrel shifters and the leading zero checker in the adder section are deleted. In addition, the width of the program counter is reduced to 10 bits, and the number of flags, and hence pins, is reduced by two. The rest of the design is essentially the same as in APR, with the numbers being represented in 16-bit 2's complement format instead of sign-magnitude. The sign-magnitude representation is used in APR because it is the IEEE standard and it also makes normalization simpler to handle. In APRjr, there is no IEEE standard to adhere to, and the numbers do not have to be normalized, so there is no reason to use sign-magnitude representation. Using 2's complement representation also removes the need for an end-around-carry adder for the adder section. A simple 2's complement adder is used instead. Because both APR and APRjr are heavily pipelined, APRjr does not run any faster than APR in terms of the length of each machine cycle. However, the reduced number of pipeline stages in the adder section in APRjr will enable it to complete a computation faster in cases where it cannot run in streamline mode (i.e., when operands can be fed to the execution pipelines continuously). This relatively small improvement in speed at the expense of reduced versatility was the reason why the APR design was attempted originally. Nevertheless, APRjr is meant to be a testing prototype for APR that is both useful and can be fabricated with the available technology, not an improved or alternative design.
Fig. 9.1: Block Diagram of APRji
1. Pad frame
2. Support circuitry for multiplier section
3. Register file
4. Adder section
5. Decoders for register file
6. Clock generator
7. I/O unit
8. Shift registers for register file
9. Instruction register
10. Program counter
11. Instruction decoder
12. Shift registers for instructions
13. Flags
14. PLA for stack control
15. Registers for barrel shifter shift amount
The floor plan of APRjr is shown in Figure 9.2. Its layout has been completed and has been sent for fabrication. The overall chip size is about 6.5mm x 6.9mm.

The following sections describe some of the more global aspects of APRjr, such as processing speed and the use of APRjr in a microprocessor system. Most of the description is applicable to APR also. The differences are pointed out whenever they are important.

9.1 Speed Considerations

In APRjr, the possible timing bottlenecks are the multiply stage of the multiplier section, and the register decode and fetch stage. APR has another potential timing bottleneck, namely the exponent comparison and fraction alignment stage. Depending on the speed of the off-the-shelf integer multiplier used, either of these two stages will become the ultimate bottleneck. With super-fast multipliers such as the Weitek WTL 2010, the speeds of these two stages are comparable. Although the multiply time of these fast integer multipliers is very short compared with the time required to read and write the register file twice, the overall multiply time is much longer because the signals have to go off-chip. The register read and write cycle is slow mainly because it has to work twice every clock cycle, once for the adder and once for the multiplier. It may be possible to run APRjr faster, and at the same time reduce the size and complexity of the chip, by not running the adder and multiplier in parallel, as, in this case, the register file only has to work once every cycle. However, operating the register file twice in one cycle is preferred here, especially as it allows for the possibility of incorporating a multiplier on chip at some later date. It is to be appreciated that the estimated speed of a 16-bit parallel multiplier in 4μm CMOS is about 300ns, which is much slower than the register file.
The time required to run the register file twice is about 160 ns. The adder in
the data path requires 50 ns. Shifting a number with the barrel shifter takes 60 ns.
Using the Weitek WTL 2010, which has a multiply time of 65 ns, total multiply time
is estimated to be 160 ns. Instruction decoding and program address generation are not
in the critical path.

9.2 Programming

APRjr has twenty 32-bit wide instructions, which are divided into two groups:
multiply/add and ETC. Figure 9.3 shows the formats of the two types of instructions.
The first bit of each instruction specifies whether the instruction is a multiply/add or
an ETC instruction. In a multiply/add instruction, 1 bit determines whether the adder
is to perform addition or subtraction, and 5 bits are used to represent each of the
sources and sinks of the multiplier and adder. In the ETC group, 5 bits are used as
the op-code. These instructions include memory reads and writes, program branching,
loading constants, etc. The complete set of instructions for APRjr is listed in Appendix C.

The instruction set for APR is more extensive, consisting of instructions to switch from fixed-point to floating-point mode, and converting 2's complement mantissas to sign-magnitude representations.

9.3 The APRjr Subsystem

Figure 9.4 shows how APRjr can be used in an arithmetic subsystem for a microprocessor system. Besides APRjr and its associated integer multiplier, the subsystem contains a program ROM, a data RAM, and a data ROM for table look-ups. Part of the data address space of APRjr is mapped to the system memory. The two data busses are usually separate, however, so that APRjr can run in parallel with the host processor.

When the host processor has collected (or guessed) a new set of joint coordinates, and wants the APRjr subsystem to compute the direct transform, it signals to the subsystem by writing to InFlag0 of APRjr. This port can either be memory mapped or I/O mapped. Upon receipt of this signal, APRjr will request the use of the system bus from the host by sending an interrupt from its OutFlag0. When the bus-grant signal is received (using the other input port on APRjr, InFlag1), the portion of the data space in APRjr that is mapped to the system memory is used to read the joint coordinates directly from the host's memory. When all the data is read, APRjr relinquishes the system bus, and proceeds to compute the direct transform. At this time, the host is free to perform other tasks. After APRjr has completed the transform, it sends an interrupt to the host, using OutFlag1. The host acknowledges the interrupt by writing to InFlag0 of APRjr and relinquishing the system bus. APRjr then writes the result of the transform in the common memory area. Upon finishing,
Fig. 9.4: Block Diagram of the APRjr Subsystem
it sends a signal from $OutFlag0$, and returns to the idle state, waiting for another task from the host.
10. TESTING

A test chip comprising one register cell and one barrel shifter cell was designed, fabricated in 5\(\mu\)m ISOCMOS at GTE's foundry and tested at UBC. These cells were chosen for testing because they contain violations of the design rules. The violations were deliberately incurred in the interests of saving space but were not deemed to be electrically unimportant. This was confirmed by the testing as the chip was found to be fully functional. Testing for timing (operation speed) is not possible because of the extra capacitive load of the pads on the chip and the probes of the testing station.

Testing APRj is will be a lot more difficult when it is eventually fabricated. The APRj design does not contain any testability enhancement circuits such as LSSD (Level Sensitive Scan Detection) [5]. This is because inclusion of such circuitry will make the design up to 30% bigger, and there is not much room to spare in the present design. Testing can be divided into 2 parts: prototype testing and production testing.

10.1 Prototype Testing

When the first chips come back, the chips may still contain design errors. Therefore, a step by step procedure is necessary to isolate any errors so that proper corrections can be made in future runs. Prototype testing must start with chip level testing. The following steps should be followed:

1. None of the pins should be shorted when no power is supplied to APRj.
2. Power should then be connected to APRj. With all other input pads grounded and output and tri-state pads floating, the clock input should be connected to a clock generator. A data generator such as the HP8180A can be used. The reset pad on APRj should be connected to a debounced switch. When reset is
high, the internal clock should stay at $\phi_0$. When \textit{reset} goes low, the internal clock should start running. The internally generated clock phases, $\phi_0$ to $\phi_4$ and their complements should be monitored on a data analyzer such as the HP8182A. A high bandwidth oscilloscope should also be used to observe the exact shapes of these waveforms.

If the clock phases prove to be as expected, then the next testing step can be carried out. However, if the clock phases are faulty, then measures must be taken so that other building blocks can be tested without the internally generated clock phases. This can be done by forcing the required clock phases by using the probes in a VLSI probing station to connect the output of a data generator to the appropriate clock wires. Although the tips of the probes are very tiny, the rest of the probes are rather bulky. As a result, in practice only about 6 probes can be used at any one time. Hence, if some of the probes have to be used to force signals onto the chip and some to monitor signals from the chip, then only very small sections of the chip can be tested one at a time.

The following steps can be applied to test any faulty building block:

a. Using the probes in a probing station, check all power supplies to the block.

b. Check all clock inputs, if any.

c. Starting from the inputs to the block (e.g. for the clock generator block, the inputs come from the input single-phase clock), check the output of each gate. The inputs and outputs of each stage can be monitored on a data analyzer. Stages that are faulty can then be identified by comparing the observed waveforms with the expected ones. This can be repeated until the final outputs are reached.
Where necessary, a stage can be isolated from its inputs by forcing signals on its inputs with the probes connected to a data generator. Because the data generator has a much larger drive than the signals generated by APRjr, the inputs will take on the values from the data generator. It is also possible to isolate a block from other circuits by physically cutting the metal wires on the chip with the probes. This is useful when some circuits connected to a multiplexed line are faulty.

3. The next function to test is address sequencing. With the clock running (assuming it works), the reset line should be held high. This forces the program counter to 0 and the rest of APRjr to a NOP state. A NOP instruction (all 0's) should be connected to the program address/data pads. The reset line should then be allowed to go low. This starts program execution in APRjr. The program counter should increment one step every machine cycle until it reaches 3FF. It should then go back to 0.

4. With the program running, instructions that affect the program counter, such as JMP and RTN, can be tested. The proper instructions should be forced to the program address/data pads using the data generator, while the program counter is monitored on the data analyzer.

5. The next instructions that can be tested are I/O instructions. An input datum can be connected to the data address/data pads (the pads for the data are multiplexed to communicate either the data address or the data themselves, hence the term "data data pads") by hard wiring. A memory read instruction should then be supplied by the data generator followed by a memory write instruction, using any register on APRjr to temporarily store the memory content. If the instructions are executed properly, then the signals on the data address/data pads should equal the read address supplied by the read instruction during \(\phi_3\) of the read instruction. In \(\phi_1\) of the write instruction, the
write address should be observed on the pads, with the data appearing in the following \( \phi_4 \).

6. The read flags and set flags instructions can be tested next.

7. Finally, the most important instructions: add and multiply, can be tested. This can be done by having the data generator generate a short program that reads in some data and then have APRjr operate on the data using either the adder section or the multiplier section. The results can then be written out for checking. The multiplier section can be tested without actually using a parallel multiplier. The multiplier 'result' can be supplied by the data generator.

If APRjr passes all these tests, then testing on the chip level is complete. APRjr must then undergo testing at the board level. The following steps are best followed with the use of a wire-wrap board:

1. With APRjr connected as in the last section, connect a WTL 2010 16-bit fixed-point parallel multiplier, or any multiplier that is pin-for-pin compatible with it, and run the same test program as the last step above on the data generator again. This time, the multiplier result should be correct.

2. Next a data RAM can be added to the system. A number of memory read/write instructions can then be executed to check if data can be read from and written to the data RAM.

3. The program ROM can now be added. Testing programs can be burned in an EPROM and then connected to APRjr. By observing the outputs of APRjr, it should be possible to decide whether the programs are executed correctly or not.

4. The final step in board level testing is the addition of the necessary glue logic required to interface the ARPjr sub-system to a microprocessor bus. Then a complete test program (one that will be used to compute the direct transform, for example) can be run. Again, if the results of the computation are correct,
then it can be assumed that the program has been executed correctly.

10.2 Production Testing

After APRjr has been thoroughly tested in the prototype stage, volume production can be started. Because the yield of chips is never 100%, testing is required to determine which chips are faulty due to processing defects. This is best done by putting every APRjr in its board sub-system and then running a test program, as was done in the last step in prototype testing. Chips that produce the correct results can then be considered defect free.
11. CONCLUSIONS

After extensive analysis of robot manipulator applications, it is concluded that a 16-bit fixed-point number format is adequate for most present-day applications.

By considering the properties of the computations involved in computing coordinate transformations in robotics, and by investigating all the new computer architecture concepts, it is decided that a general purpose single processor system is still the best approach. However, all the latest innovations in computer architecture are applied to this processor. These innovations include parallel processing (adder and multiplier can run in parallel), non-interlocked pipeline instruction fetch and execution cycles, reduced instruction set, large register file, and Harvard architecture.

Following these decisions, an arithmetic processor has been designed. The chip, called APR, supports both floating-point and fixed-point arithmetic. Floating-point operations are included in order to broaden the potential applications of APR. A test chip called APRj r has also been designed. APRjr only supports fixed-point arithmetic. It will be both useful as a practical chip and as a test vehicle for APR. The layout of APRjr has been carried to completion and it was submitted for fabrication with a 4μm CMOS process in late January, 1985. Based on extensive simulations, a machine cycle time of 200ns is expected.

11.1 Contributions of this Thesis

The APR project has contributions in both the VLSI and robotics areas. In the VLSI area, the APR project has stimulated the interest in large scale processor designs here at the University of British Columbia. The experience accumulated during the design of APR and APRjr will benefit future VLSI digital designs. Through hands-on experience, fundamental digital designs in LSI such as registers, adders, shifters, and clock generation have been investigated in detail. Hence, a large number of library
cells have been created. These designs can be used in future designs, or as the starting point for improved designs. At the beginning of this project, no library cells were available.

The attempt to design a large scale chip has also revealed a lot of deficiencies in the CAD tools currently available. The possible improvements are outlined in the next section.

In the robotics area, APR can make real time trajectory possible and economical. Because of the general nature of the architecture, APR can be used for other computations in robotics other than coordinate transformations, such as real-time control of manipulators [20].

In addition, APR can be used for other applications such as computer graphics, digital signal processing, and any other tasks that require high speed but only moderate precision arithmetic.

11.2 Suggestions for Further Research

VLSI is only an emerging field today, particularly in the University of British Columbia. The ambition of the present project is greatly restricted by the lack of an advanced fabrication process, the high cost and long turn around time of fabrication, and the lack of a sizable research team working actively in VLSI. When all or most of these difficulties are overcome, the APR design can be improved in quite a few ways.

APR has had to be designed in one man–year, therefore, the designer will be more than satisfied if the chip turns out to be functional. Given more time and design talents, the author believes that the APR design can be improved in terms of compactness, regularity, and timing. A more refined floor plan can significantly compact
the design, and at the same time make the layout more regular. Regularity not only renders the design easier and thus less error prone, but it also facilitates any future modifications. A careful study of the delays of all the signal paths in the design will locate exactly where the bottlenecks in the design are. With this information, the design can be fine-tuned by reducing the resistance in the critical paths, by buffering to reduce the load on each stage, and by modifying the architectural design.

On the system level, the computation sub-system can be made more compact and probably faster by designing a multiple chip set to replace the other components, such as the integer multiplier and the program and data memory. For example, the multiplier and the data memory may be integrated on one chip.

When a more advanced process becomes available, the APR layout can be completed and the chip fabricated. It may even be possible to make APR more powerful in a number of ways:

- The parallel multiplier can be integrated on chip.
- A full 32-bit data path can be implemented, with full 32-bit registers and a 23-bit parallel multiplier.
- The size of the register file can be increased; 128 registers would be ideal. Larger register files will slow down the machine cycle so much that the gain in having fewer memory accesses will not be worthwhile [24].
- A more powerful instruction set can be designed so that APR is made a lot more versatile, with little or no deterioration in processing speed.

A lot of research is also required in the CAD area. Many of the following required CAD tools already exist in some form or another, however, they are not available at the University of British Columbia. Also, most of the existing tools are still at a rudimentary stage, and improvements are not only possible but necessary.

- Simulation from layouts. Presently only small sections of a large design (of the
size of APRjr) can be simulated. The circuit simulator, SPICE, is not only extremely slow, but it has convergence problems with its level 2 MOSFET model. The switch level simulator, \( h_g \), or RNL (\( H_g \) is a Metheus modification of RNL), performs reasonably well with moderate size circuits consisting of around 100 transistors. But it has problems dealing with certain circuits [29]. This means that \( h_g \) cannot be run on any design without scrutiny.

There is probably very little that can be done about the speed of simulation with these simulators, unless hardware accelerators are available. However, it will be extremely helpful if the simulators can be invoked interactively using a windowing facility. In other words, the designer can call the simulators to work on a certain portion of his design, without any modifications to the design. At present, in order to simulate a section of a design, the designer must make that portion that he wants to be simulated a self-contained cell.

A timing analyzer [15,23] is also helpful in filling the gap between precise circuit or switch level simulation and functional simulation on the chip level.

Hierarchical simulation system. When a digital design is formulated, it is best to simulate the design in a top-down fashion. For example, the design is specified first by its input and output characteristics. The design is then simulated on the system (or processor-memory) level. After making sure that the designs works in this level, it is expanded one step down the hierarchy, namely the register-transfer level. The design is then simulated again in this level, making sure that the inputs and outputs are as specified in the processor-memory level model. Next the design is detailed in the gate (logic) level, then the circuit level, and finally the layout level. A design in a lower
level is started only after the present level is proven to be compatible with the level above it using the proper simulation tools. At present, only gate level (HILO) and circuit level (Hg and SPICE) simulators are available.

- **Auto-Router.** Routing a VLSI chip is extremely painstaking and error prone, simply because the designer cannot see what he is doing even on a high-resolution graphics terminal. Multiple windows would help, but it would be much more convenient if an auto-router is available. The ideal router should be able to operate in several modes. In manual mode, the designer can direct the router interactively on the screen. The general direction that the route should take is supplied by the designer, while the router will take care of small details such as separation between layers and crossovers. This mode is useful when the routing channels are tight. In semi-auto mode, the router should be able to route a set of ports together without aid, given the location of the ports and the building blocks. In full-auto mode, it should be able to rearrange the building blocks slightly, such as shifting a little bit to either side, so that a complete routing is obtained. The most efficient routing is then obtained by the application of a suitable combination of these modes.

- **Layout compactor.** A layout compactor is useful in compacting small cells. The designer specifies the circuit, the topology of the connections, and the dimensions of the desired cell, and the layout compactor will attempt to fit the design in the specified area.

- **Floor planner.** A floor planner can give the designer estimates of the overall size of the chip given the rough floor plan of the building blocks and the routing requirements. Perhaps the floor planner can even be made smart enough to come up with some close to optimum floor plans with minimal help from the designer.

- **A more powerful circuit extractor.** The circuit extractor that is available now
only recognizes transistors and parasitic capacitors. What is more desirable is a hierarchical circuit extractor that can recognize the layouts in the gate level, register-transfer level, and even the processor-memory level. Recognizing logic gates from a layout should be easy, especially with the use of artificial intelligence. Register-transfer level and processor-memory level recognition is more difficult, requiring considerable programming effort and a sizable database.

- A better scheduling program. The scheduling program which is in use now is still very difficult to use. The user has to break up the individual operations that are required to compute the equations, and then arrange the operations in the form of a data dependency graph. The program can be improved so that the user is only required to enter the equations in a more convenient form (reverse Polish notation, RPN, is a prime candidate). Another possibility for improvement is to include in the scheduling algorithm the effects of the finite number of registers available.
BIBLIOGRAPHY

Cited References


[6]. Excalibur Arm (Model No. RT3), Robotic Systems International, Sidney, B.C., V8L 3S1, Canada.


[8]. GTE Microcircuits, Tempe, Arizona 85281.


Caltech Conf. on VLSI, R. Bryant (ed), Computer Science Press, Rockville, Maryland, pp.57–69, 1983.


[34]. R. Sherburne, "Data Path Design for RISC", Proc. MIT Conf. on Advanced


General References

In addition to the above cited literature, the following references will be found useful for anyone interested in VLSI or coordinate transformations in robotics:


Robots", *13th Symposium on Industrial Robots*, pp.16-1,16-8, 1983.


APPENDIX A: LAYOUTS

In this appendix, the layouts of some of the more important cells are shown. The design rules used are those for the GTE 4μm ISOCMOS process. The rules are confidential and so cannot be disclosed here. The following legend is used in all the figures:

- Diffusion
- Polysilicon
- Metal
- Contact Cut
- N+ Diffusion
- P+ Diffusion
- P-Well
A.1: 3-Port Register Cell

See also pp. 41–44, 92, 110
A.2: 2 x 2 Register File

Notice how the symmetry of the register cell is taken advantage of in the register file.

See also pp. 41–44, 91, 105–110
A.3: One Barrel Shifter Cell

See also pp. 44–47, 94
A.4: 3-out-of-5 Barrel Shifter

See also pp. 44–47, 93
A.5: 4-Bit Leading Zero Checker

See also pp. 47, 111, 112
A.6: Stack Cell

See also pp. 62, 97, 115
A.7: 2-Bit by 2-Level Stack

See also pp. 62, 96, 115
A.8: 2-Bit Par-Add Adder

See also pp. 51-54, 113
A.9: 1-Bit Static Regenerative Manchester Carry Adder

See also pp. 48-50, 114
A.10: PLA for Stack Control

See also pp. 60, 61, 117
A.11: Clock Generator

The circuit for the following layout is shown in Figure 7.13.

See also pp. 55–60, 116
A.12: Flags

The following circuit and corresponding layout is used to store the flags in APRjr.
A.13: Barrel Shifter Controller for APRjr

The following circuit and corresponding layout is used for controlling the barrel shifter in the multiplier section of APRjr, and for testing the number of leading zeros in the multiplier result.
APPENDIX B: SIMULATIONS

In this appendix, some of the more important simulations performed are presented. Two simulators have been used in this project, namely the circuit level simulator, SPICE, and the switch level simulator, Hg. For small circuits, SPICE with level 1 MOSFET parameters is used. Level 2 is only used for very small circuits, not only because of the slow execution speed of level 2, but also because it has frequent convergence problems. Hg is used for larger circuits, consisting of over 20 transistors. Where necessary, the circuit used in the simulation is shown with the simulation output.
B.1: Writing a Zero to a Register

See also pp. 41–44, 92
B.2: Writing a One to a Register

See also pp. 41–44, 92
B.3: Writing a One to a Register with Noise

The signal on the write bus is set to only 4V in this simulation. Notice the response of the register cell is almost the same as in Appendix B.2.

See also pp. 41-44, 92
B.4: Writing a One to a Register with Noise, Level 2

MOSFET model 2 is used in this simulation to investigate the reliability of model 1. Observe that the response is very close to, although not exactly the same as, Appendix B.3.

See also pp. 41–44, 92
B.5: Register Sourcing a Zero

See also pp. 41–44, 92
B.6: Register Sourcing a One

See also pp. 41-44, 92
B.7: Leading Zero Checker

See also pp. 47, 95
GROUP 1: V(izout), V(init), V(i3)
B.8: One Par-Add Logic Block

One of the Par-Add logic blocks, the circuit of which is shown below, is simulated to estimate the speed of a Par-Add adder. Proper loads, corresponding to the next stages, have been added in the simulation.

See also pp. 51-54, 98
B.9: Carry Propagation in One Static Regenerative Manchester Carry Adder Cell

See also pp. 48–50, 98
B.10: 2-Bit by 2-Level Stack

A '0' is pushed in bit 1, then a '1' is pushed in bit 1. The data are then popped back out. Bit 0 is not used.

See also pp. 62, 96, 97
B.11: Clock Generator

The clock phases that are generated can be compared with those given in Figure 5.2, except that here the complements of the phases are shown. The glitches will probably not cause any problems because of the large capacitances that the clock has to drive, although improvements can probably be made to remove them.

See also pp. 55–60, 101
B.12: PLA for Stack Control

Notice that the outputs occasionally take on a wrong value before finally settling down to the correct values. This causes no problems in a static PLA, but may be disastrous if a dynamic design is used.

See also pp. 60, 61, 100
APPENDIX C: INSTRUCTION SET

The complete set of instructions for APRjr is listed below. The instructions have been grouped together according to the functions they perform. For each instruction, a three-letter mnemonic is given, followed by the 32-bit op-code. The number of machine cycles required to execute each instruction is also given. Since APRjr uses non-interlocked pipelines, a new instruction can be started whether the previous instruction has finished executing or not. It is therefore up to the user, or the compiler, to make sure that the instructions are executed in the proper sequence. For example, in a series of Multiply/Add instructions, an instruction that uses a previous multiplication result can be started only after at least three cycles, and an instruction that uses a previous addition result can be started only after at least two cycles. See Appendix D for some detailed examples.

The full name of the instruction, definitions of the various fields in the op-code, and the action performed by the instruction is given in the remarks section. In the opcode descriptions, the following convention is used:

1, 0: Binary digits which must be entered as is;
?–F0–?: A sub-field, containing a certain number of bits, called F0;
x–5–x: A field of don't care bits, in this particular case 5 bits long;
MULTIPLY/ADD GROUP

Mnemonic: MAD

Opcode: 10 ?-F5-? ?-F4-? ?-F3-? ?-F2-? ?-F1-? ?-F0-?

No. of cycles: 4 for multiply portion, 3 for add portion

Remarks: Simultaneous Multiply/Add;

F0: 5-bit field, sink for multiplier;
F1: 5-bit field, source 0 for multiplier;
F2: 5-bit field, source 1 for multiplier;
F3: 5-bit field, sink for adder;
F4: 5-bit field, source 0 for adder;
F5: 5-bit field, source 1 for adder;

The contents of the registers specified by F1 and F2 are multiplied and the result will be stored in the register specified by F0.

At the same time, the contents of the registers specified by F4 and F5 are added together and the result will be stored in the register specified by F3;

Mnemonic: MSB

Opcode: 11 ?-F5-? ?-F4-? ?-F3-? ?-F2-? ?-F1-? ?-F0-?

No. of cycles: 4 for multiply portion, 3 for add portion

Remarks: Simultaneous Multiply and Subtract;

Action similar to MAD, except that the contents of the register specified by F4 are subtracted from the contents of the register specified by F5;
ETC GROUP

Program Counter Control Sub-Group

Mnemonic: JMP
Opcode: 000001 ?-F0-? x-10-x
No. of cycles: 1
Remarks: Unconditional Jump;
F0: 16-bit field, new program address, where bit 10 is the LSB;
The contents of the register specified by F0 is forced into the program counter;

Mnemonic: JOF
Opcode: 000010 7-F0-? x-10-x
No. of cycles: 1
Remarks: Jump on Flag Set;
F0: 16-bit field, new program address, where bit 10 is the LSB;
PC is set to new value if the main flag (set or cleared by TSF instruction) is set. Otherwise, PC is incremented as usual;

Mnemonic: JSR
Opcode: 000011 ?-F0-? x-10-x
No. of cycles: 1
Remarks: Jump to Subroutine;
F0: 16-bit field, jump address;
The present PC is pushed onto the hardware stack (2 levels deep), and the PC is then set to the new value;
Mnemonic: RTN
Opcode: 000100 x-26-x
No. of cycles: 1
Remarks: Return from Subroutine;
The contents of the top of the hardware stack is transferred to
the PC. The original PC will be lost;

I/O Sub-Group

Mnemonic: LCD
Opcode: 000101 ?-F1-? ?-F0-? x-5-x
No. of cycles: 1
Remarks: Load Constant Data;
F0: 5-bit field, data sink;
F1: 16-bit field, data word;
The data word given in F1 is stored in the register specified by F0;

Mnemonic: WWD
Opcode: 000110 ?-F1-? ?-F0-? x-5-x
No. of cycles: depends on external memory speed, minimum = 1
Remarks: Write Word Direct;
F0: 5-bit field, data source;
F1: 16-bit field, data address;
The contents of the register specified by $F0$ is written to the
data memory location specified by $F1$;
When 2 or more memory write instructions are executed in
sequence, care must be taken such that the external memory has
enough time to respond. Other instructions, such as a MAD or
JMP instruction, can be executed immediately after a WWD;

Mnemonic: WWI
Opcode: 000111 x-11-x ?-F1-? ?-F0-? x-5-x
No. of cycles: depends on external memory speed, minimum = 1
Remarks: Write Word Indirect;
F0: 5-bit field, data address source;
F1: 5-bit field, data source;
The contents of the register specified by $F1$ is written to the
memory location specified by the contents of the register
specified by $F0$;
See remarks in WWD;

Mnemonic: RDW
Opcode: 001000 x-21-x ?-F0-?
No. of cycles: 1
Remarks: Read Data Word;

F0: 5-bit field, sink for data;

All 16 bits of the data bus is copied to the register specified by \( F0 \);

Mnemonic: RUB
 Opcode: 001001 \( x-26-x \)
 No. of cycles: 1

Remarks: Read Upper Data Byte;

The upper 8 bits of the data bus is latched on-chip. Note that the data is not written to the register file;

Mnemonic: RLB
 Opcode: 001010 \( x-21-x \) \( ?-F0-? \)
 No. of cycles: 1

Remarks: Read Lower Data Byte;

F0: 5-bit field, data sink;

The lower 8 bits of the data bus is latched on-chip. The contents of the whole 16-bit latch is then written to the register specified by \( F0 \);

Mnemonic: WUB
 Opcode: 001011 \( ?-F1-? \) \( ?-F0-? \) \( x-5-x \)
 No. of cycles: 1
Remarks: Write Upper Data Byte;
F0: 5-bit field, data source;
F1: 16-bit field, data address;
The upper byte of the register specified by F0 is written to the memory location specified by F1;

Mnemonic: WLB
Opcode: 001100 ?-F1-? ?-F0-? x-5-x
No. of cycles: 1
Remarks: Write Lower Data Byte;
F0: 5-bit field, data source;
F1: 16-bit field, data address;
The lower byte of the contents of the register specified by F0 is written to the memory location specified by F1;

Mnemonic: SAD
Opcode: 001101 ?-F0-? x-10-x
No. of cycles: 1
Remarks: Send Memory Address, Direct;
F0: 16-bit field, memory address;
The address specified by F0 is driven onto the data address bus.
Used to prepare for a direct read from the data memory;

Mnemonic: SAI
Opcode: 001110 x-16-x ?-F0-? x-5-x
No. of cycles: 1
Remarks: Send Data Address, Indirect;
F0: 5-bit field, address source;
The contents of the register specified by \( F0 \) is driven onto the data address bus. Used to prepare for an indirect read from the data memory;

Miscellaneous Sub-Group

Mnemonic: LDF
Opcode: 001111 \( x-9-x \ ?-F0-? \ x-10-x \)
No. of cycles: 1
Remarks: Load Flags;
F0: 7-bit field, new flags;
The flags in the condition code register (CCR) are set or cleared according to the new flags supplied. The flags are:

\[
\text{Out0 Out1 In0 In1 V N Z}
\]

Mnemonic: TSF
Opcode: 010010 \( x-9-x \ ?-F0-? \ x-10-x \)
No. of cycles: 1
Remarks: Test Flags;
F0: 7-bit field, test template;
The data in the template is ANDed with the CCR, and the main flag is set or cleared according to the result. Only JOF uses the main flag;
The V(overflow), N(negative), and Z(zero) reflects only the condition of the results of the adder section. There is no way to monitor the results of the multiplier section, they are hard limited when an overflow occurs. The In0 and In1 flags are external conditions which can be tested. See section 9.3 for the meaning of these 2 flags.

Mnemonic: SSH
Opcode: 01000 ?-F0-? 0 x-9- x
No. of cycles: 1
Remarks: Set Shift Amount;
F0: 17-bit field, shift amount, only 1 bit should be non-zero in these 17 bits;
The shift amount is written to the barrel shifter controller. All subsequent results from the multiplier will then be shifted by the specified amount;
For example, to shift all results from the multiplier up by 2 bits, the instruction is:
01000 00100000000000000 xxxxxxxxx

Mnemonic: SLZ
Opcode: 01000 ?-F0-? 1 x-9- x
No. of cycles: 1

Remarks: Set Required Number of Leading Zeros;

F0: 17-bit field, required number of leading zeros. See examples below;

The required number of leading zeros is written to the barrel shifter controller. All subsequent results from the multiplier which have less than this number of leading zeros will be flagged as overflows;

For example, to require all results from the multiplier to have at least 2 leading zeros, the instruction is:

01000 0011111111111111 x x x x x x x x x

and for 5 leading zeros the instruction is:

01000 0000011111111111 x x x x x x x x

Mnemonic: NOP

Opcode: 000000 x-26- x

No. of cycles: 1

Remarks: No Operation;

No action is performed. The flags in the CCR are not affected;
APPENDIX D: SCHEDULING THE ADDER AND THE MULTIPLIER SECTIONS

Since the adder and multiplier sections can run in parallel, the speed of a program depends a lot on the sequence of the instructions. A program which keeps the adder and the multiplier running most of the time will run a lot faster than a program which does not. To illustrate this, consider programming the following simple equation:

\[ y = a \cdot b \cdot c - (b \cdot d + a + b \cdot c) \]

A possible schedule for the adder and multiplier is shown below. For the sake of simplicity, the delays required because of the various pipelines is ignored. In other words, it is pretended that there are no pipeline delays.

<table>
<thead>
<tr>
<th>Multiplier</th>
<th>Adder</th>
</tr>
</thead>
<tbody>
<tr>
<td>b \cdot d</td>
<td>--</td>
</tr>
<tr>
<td>b \cdot c</td>
<td>--</td>
</tr>
<tr>
<td>a \cdot (b \cdot c)</td>
<td>--</td>
</tr>
<tr>
<td>(b \cdot d + a + b \cdot c) \cdot d</td>
<td>a + (b \cdot c)</td>
</tr>
<tr>
<td>(a \cdot b \cdot c) - (b \cdot a + a + b) \cdot d</td>
<td>--</td>
</tr>
</tbody>
</table>

And the corresponding APRjr program is, again ignoring pipeline delays:

SAD &A ;send address of variable A out,
;to prepare for reading A.
;&A stands for address of A,
;as in the C programming language.
RDW R1 ;read in value of A, and place it in register 1
SAD &B ;prepare to read B
RDW R2 ;put B in register 2
The schedule can be rearranged to reduce the number of instructions by 1:

<table>
<thead>
<tr>
<th>multiplier expression</th>
<th>adder expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>$b\cdot c$</td>
<td>$a + (b\cdot c)$</td>
</tr>
<tr>
<td>$b\cdot d$</td>
<td>$(b\cdot d) + (a + b\cdot c)$</td>
</tr>
<tr>
<td>$a - (b\cdot c)$</td>
<td>$a - (b\cdot c)$</td>
</tr>
<tr>
<td>$(b\cdot d + a + b\cdot c)\cdot d$</td>
<td>$(a\cdot b\cdot c - ((b\cdot d + a + b\cdot c)\cdot d)$</td>
</tr>
</tbody>
</table>
It can be seen that even in this simple example, the execution speed of the program depends on the proper scheduling of the multiplier and the adder. The problem of scheduling multiple processing elements efficiently has been explored quite thoroughly in the micro-programming area. For the present application, the branch and bound (BAB) algorithm with list scheduling heuristic is used. See [19] for more information on scheduling algorithms and [28] for why BAB is chosen among other algorithms.

In the BAB algorithm with list scheduling heuristic, each operation is assigned a weight, equal to the number of operations that is dependent on the operation. Then the schedule is determined by simply picking the operation that has the heaviest weight first. If two operations that can be scheduled have the same weight, then an arbitrary choice is taken. The algorithm is:

*Compute weight of each operation;*

*From the list of operations that can be scheduled,*

    *schedule in descending order of weights;*

*Repeat until all operations scheduled.*

During scheduling, no attention is paid to the registers that are needed. Only the sequence of execution is established. After scheduling is complete, registers are allocated to store the intermediate results of each operation. The register that is assigned to an operation is freed whenever all the sons of the operation have been assigned a register.

A simplified (pipeline delays ignored) schedule generated for the computation of p̄ of the Excalibur Arm is given below. The program listing of the BAB algorithm is also included for completeness.
Data Dependency Graph for $\tilde{p}$ of Excalibur Arm
## The Resulting Schedule

<table>
<thead>
<tr>
<th>multiplier operations</th>
<th>register assigned</th>
<th>adder operations</th>
<th>register assigned</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>3</td>
<td>10</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td>5</td>
<td>11</td>
<td>6</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>14</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>12</td>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>17</td>
<td>7</td>
</tr>
<tr>
<td>15</td>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>20</td>
<td>8</td>
<td>18</td>
<td>9</td>
</tr>
<tr>
<td>33</td>
<td>4</td>
<td>32</td>
<td>10</td>
</tr>
<tr>
<td>23</td>
<td>5</td>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>22</td>
<td>4</td>
<td>26</td>
<td>9</td>
</tr>
<tr>
<td>16</td>
<td>5</td>
<td>25</td>
<td>3</td>
</tr>
<tr>
<td>28</td>
<td>1</td>
<td>19</td>
<td>4</td>
</tr>
<tr>
<td>27</td>
<td>2</td>
<td>31</td>
<td>3</td>
</tr>
<tr>
<td>24</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>21</td>
<td>4</td>
<td>29</td>
<td>5</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>30</td>
<td>1</td>
</tr>
</tbody>
</table>

- operation no. = 0 → unit idle
- registers are reused as soon as possible.
- the sine's and cosines of the angles are assumed to have been read into the registers already.
- pipeline delays are ignored.
program BABHeuristic;
{-----------------------------}
{
   Branch and Bound with Heuristic
}
{-----------------------------}

const
MaxOp = 50;
MaxReg = 50;
Occupied = 1;
UnOccupied = 0;
Null = 0;
Mul = 1;
Add = 2;
Infinity = 10000;

type
SetOfParents = array [1..10] of integer;
SetOfSons = array [1..10] of integer;
SetOfDes = array [1..100] of integer;
OpRec = record
   Reg: integer;
   Op: integer;
   Wt: integer;
   NumParents: integer;
   RemParents: integer;
   Parents: SetOfParents;
   NumSons: integer;
   Sons: SetOfSons;
end;
OneNode = record
   OpNum: integer;
   Next: PtrNode;
end;
PtrTimeSlot = @TimeSlot;
TimeSlot = record
   OpNum : integer;
   Prev: PtrTimeSlot;
end;
var
   ShortestSched, SchedLen: integer;
   RdyNull, RdyMul, RdyAdd: PtrNode;
   NextRdyNull: PtrNode;
   BestSched: PtrTimeSlot;
   LastSlot: PtrTimeSlot;
   Op: integer;
   RegSet: array [1..MaxReg] of integer;

function FindFreeReg: integer;
var
   i: integer;
begin
   i := 1;

while RegSet[i] = Occupied do
  i := i + 1;
  RegSet[i] := Occupied;
end; {func}

procedure AllocateReg(PrevTimeSlots: PtrTimeSlot);
var
  CurParent, i, NumSons: integer;
  GreatParent: integer;
  Abort: boolean;
  CurOp: PtrTimeSlot;
begin
  CurOp := PrevTimeSlots;
  if CurOp^.Prev <> nil then
    begin
      AllocateReg(CurOp^.Prev);
    end; {if}
  if (CurOp^.OpNum <> 0) then
    begin
        begin
          NumSons := OpSet[CurParent].NumSons - 1;
          OpSet[CurParent].NumSons := NumSons;
          if NumSons = 0 then
            begin
              if OpSet[CurParent].Op = Null then
                begin
                  GreatParent := CurParent;
                  Abort := false;
                  while (OpSet[GreatParent].Op = Null) and not Abort do
                    begin
                      if OpSet[GreatParent].NumParents <> 0 then
                      else
                        Abort := true;
                    end;
                end;
              if not Abort then
                begin
                  NumSons := OpSet[GreatParent].NumSons - 1;
                  OpSet[GreatParent].NumSons := NumSons;
                  if NumSons = 0 then
                    begin
                    end;
                end;
            end
          else
        end;
    end;
  write(CurOp^.OpNum);
  if CurOp^.OpNum = 0 then
    write(0)
  else
    write(OpSet[CurOp^.OpNum].Reg);
if Op = Add then
  begin
    writeln;
    Op := Mul;
  end
else
  Op := Add;
end; {proc}

procedure PrintSched;
var
  i: integer;
begin
  for i:=1 to MaxReg do
    RegSet[i] := Unoccupied;
  Op := Mul;
  AllocateReg(LastSlot);
end; {proc}

procedure ConstructGraph;
var
  i, OpNum, NumSons, Son, NumParents: integer;
begin
  for i:=1 to MaxOp do
    OpSet[i].NumParents := 0;
  read(OpNum);
  while OpNum <> 0 do
    begin
      read(OpSet[OpNum].Op);
      NumSons := 0;
      repeat
        read(Son);
        NumSons := NumSons + 1;
        if Son <> 0 then
          begin
            NumParents := OpSet[Son].NumParents + 1;
            OpSet[Son].NumParents := NumParents;
            OpSet[Son].RemParents := NumParents;
          end; {if}
      until Son = 0;
      read(OpNum);
    end; {while}
end; {proc}

{ This procedure updates the ready-null, ready-mul, and ready-add lists. }
procedure UpdateLists(OpNum: integer);
var
  i, NumSib, CurOp, CurWt: integer;
  Siblings: SetOfSons;
Node, RefNode1, RefNode2: PtrNode;
begin
  NumSib := OpSet[OpNum].NumSons;
  Siblings := OpSet[OpNum].Sons;
  for i := 1 to NumSib do
    begin
      CurOp := Siblings[i];
      if OpSet[CurOp].RemParents = 0 then
        begin
          new(Node);
          Node@.OpNum := CurOp;
          case OpSet[CurOp].Op of
            Null : begin
              Node@.Next := NextRdyNul;
              NextRdyNul := Node;
            end;
            Mul : begin
              if RdyMul = nil then
                begin
                  RdyMul := Node;
                  Node@.Next := nil;
                end
              else
                begin
                  CurWt := OpSet[CurOp].Wt;
                  if CurWt > OpSet[RdyMul@.Opnum].Wt then
                    begin
                      Node@.Next := RdyMul;
                      RdyMul := Node;
                    end
                  else
                    begin
                      RefNode1 := RdyMul;
                      RefNode2 := RefNode1@.Next;
                      while ((RefNode2 <> nil) and (OpSet[RefNode2@.OpNum].Wt > CurWt)) do
                        begin
                          RefNode1 := RefNode2;
                          RefNode2 := RefNode2@.Next;
                          end; {while}
                    end; {if}
                end; {else}
            end; {clause}
            Add : begin
              if RdyAdd = nil then
                begin
                  RdyAdd := Node;
                  Node@.Next := nil;
                end
              else
                begin
                  CurWt := OpSet[CurOp].Wt;
                  if CurWt > OpSet[RdyAdd@.Opnum].Wt then
                    begin
                      Node@.Next := RdyAdd;
                      RdyAdd := Node;
                    end
                end}
end
else
begin
RefNode1 := RdyAdd;
RefNode2 := RefNode1@.Next;
while ((RefNode2 <> nil) and (OpSet[RefNode2@.OpNum].Wt > CurWt)) do
begin
RefNode1 := RefNode2;
RefNode2 := RefNode2@.Next;
end; {while}
RefNode1@.Next := Node;
Node@.Next := RefNode2;
end: {if}
end; {case}
end; {for}
end; {proc}

procedure Schedule;
var
NewSlot: PtrTimeSlot;
MulOp, AddOp: PtrNode;
begin
NextRdyNull := nil;
new(NewSlot);
NewSlot@.Prev := LastSlot;
LastSlot := NewSlot;
MulOp := RdyMul;
if RdyMul = nil then
NewSlot@.OpNum := 0
else
begin
RdyMul := RdyMul@.Next;
NewSlot@.OpNum := MulOp@.OpNum;
end; {if}
new(NewSlot);
NewSlot@.Prev := LastSlot;
LastSlot := NewSlot;
AddOp := RdyAdd;
if RdyAdd = nil then
NewSlot@.OpNum := 0
else
begin
RdyAdd := RdyAdd@.Next;
NewSlot@.OpNum := AddOp@.OpNum;
end; {if}
if MulOp <> nil then
UpdateLists(MulOp@.OpNum);
if AddOp <> nil then
UpdateLists(AddOp@.OpNum);
while RdyNull <> nil do
begin
UpdateLists(RdyNull@.OpNum);
RdyNull := RdyNull@.Next;
end; {while}
RdyNull := NextRdyNull;
end; (proc)

procedure Union(var ParentDes: SetOfDes;
    var NumPDes: integer;
    SonDes: SetOfDes;
    NumSDes: integer);
var
    i, j, k: integer;
    found: boolean;
begin
    k := NumPDes;
    for i:=1 to NumSDes do
        begin
            found := false;
            j := 1;
            while ((j <= NumPDes) and (not found)) do
                begin
                    if SonDes[j] = ParentDes[j] then
                        found := true
                    else
                        j := j + 1;
                end; {while}
            if not found then
                begin
                    k := k + 1;
                    ParentDes[k] := SonDes[i];
                end; {if}
        end; {for}
    NumPDes := k;
end; (proc)

procedure MergeDescendants(var Descendants: SetOfDes;
    var NumDes: integer;
    OpNum: integer);

procedure AssignWeights(var Descendants: SetOfDes;
    var NumDes: integer;
    OpNum: Integer);
var
    NumSons, MyNumDes, i: integer;
    Sons: SetOfSons;
    MyDes: SetOfDes;
begin
    NumSons := OpSet[OpNum].NumSons;
    Sons := OpSet[OpNum].Sons;
    MyNumDes := 1;
    for i:=1 to NumSons do
        AssignWeights(MyDes, MyNumDes, Sons[i]);
    Union(Descendants, NumDes, MyDes, MyNumDes);
end; (proc)
procedure ComputeWeights;
    var
        DummyA: SetOfDes;
        DummyC: integer;
    begin
        DummyC := 0;
        AssignWeights(DummyA, DummyC, 1);
    end;

procedure MergeDes;
    var
        i: integer;
        found: boolean;
    begin
        if OpSet[OpNum].NumSons = 0 then
            begin
                found := false;
                while ((i <= NumDes) and (Not found)) do
                    begin
                        if Descendents[i] = OpNum then
                            found := true
                        else
                            begin
                                i := i + 1;
                            end;
                    end;  (while)
                if not found then
                    begin
                        Descendents[i] := OpNum;
                        NumDes := i;
                    end;
            end;
        else
            AssignWeights(Descendents, NumDes, OpNum);
        end;  (proc)
    begin
        ConstructGraph;
        ComputeWeights;
        RdyNull := nil;
        RdyMul := nil;
        RdyAdd := nil;
        LastSlot := nil;
        UpdateLists(1);
        while ((RdyNull <> nil) or (RdyMul <> nil) or (RdyAdd <> nil)) do
            Schedule;
        PrintSched;
    end.