APPLICATION SPECIFIC PROCESSORS FOR NUMERICAL ALGORITHMS

by

Lars Erik Thon

Memorandum No. UCB/ERL M92/139

11 December 1992

APPLICATION SPECIFIC PROCESSORS FOR NUMERICAL ALGORITHMS

by

Lars Erik Thon

Memorandum No. UCB/ERL M92/139

11 December 1992

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

APPLICATION SPECIFIC PROCESSORS FOR NUMERICAL ALGORITHMS

by

Lars Erik Thon

Memorandum No. UCB/ERL M92/139

11 December 1992

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

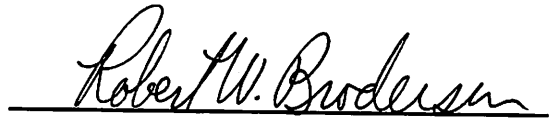
Application Specific Processors for Numerical Algorithms

by

Ph.D. **Lars Erik Thon** **Department of EECS**

Abstract

The development of Application Specific Integrated Circuits (ASICs) has historically been driven to a large extent by applications within the areas of Digital Signal Processing (DSP) and Communication Networks. An open question is whether the gains and advantages that have been observed by applying ASIC technology in these areas can be duplicated in the Numerical Processing application domain. The standard DSP-inspired approaches are not always applicable to design tools, architectures, simulation and circuit design for Numerical Processing (NP), because NP is different from DSP in terms of data types, data organization, data access and pipelining margins. This work builds on the advances made in the traditional application areas and expands the technology into the NP application domain. Two applications are used as test cases for evaluating the ASIC/NP combination. The first case is a geometric computation problem, and the other involves solving nonlinear equations, ultimately leading to a core problem of matrix computations that are common to a variety of NP applications. An automated silicon compilation approach as well as manual designs and methodologies have been developed and applied to the test cases.



Robert W. Brodersen

Chairman of Committee

Application Specific Processors for Numerical Algorithms

Lars Erik Thon

Acknowledgments

This page is for the people who made my life enjoyable while I struggled my way through Graduate School in Berkeley. There are many people to thank, and I hope I have remembered at least everyone I collaborated with on a daily basis and the many others that made a difference in my private life.

Let me start out by thanking Bob Brodersen for supporting me during the long years here at Berkeley. It has been quite a trip. I would also like to thank:

Brian Richards for being the all around wizard and author of numerous CAD tools, and for coaching me on all aspects of chip design. Without Brian I would never have gotten even one chip out the door. Kirk Thege and Kevin Zimmerman for helping out with the computers and making me a trusted member of the root community. Mani Srivastava for answering all my beginners questions about UNIX and many other topics just too numerous to mention, and helping out with Dpp maintenance and hacking. Rajeev Jain for getting LagerIV off the ground. Erik Lettang for getting all my pads routed. Sam Sheng for being such a helpful guy and always answering questions and especially for making MakeThorSim work. Andy Burstein and Monte Mar for being great cubicle-mates, and (Andy) for fixing bugs in (or is it adding features to?) ext2spice. Bill Baringer for being very inspiring and helpful when I was learning to draw my first transistors. Susan for completely changing my life. Nancy for being such a good friend, even afterwards. Judy for being an inspiration during my last 2 years in Berkeley. Arlene for being the most fun to flirt with. Jan Rabaey and Seungjeun Lee for helping out with Flint. Sigvor for providing a home away from home for all the Norwegian students. Bertrand Irissou and John Wawrzynek for introducing me to high-speed circuit design and helping out with the cell library, and Bertrand for being a great pal. Ken Rimey for teaching me LISP and writing the first ever bug-free compiler. Edward Wang for helping out when Ken was not available. Lars "Johnny" Svensson for exploring all the architectures that I had no time to deal with, and writing the first version of MakeThorSim. Markus Thaler and his wife for writing the longest useful CSH script ever (DMpost).

Clara Chang and Jean Souza for taking care of me when I first came to Berkeley. Mark Davis and Jeff Bradt for being best friends and partners in crime. Jennifer, Herb, Robert, Stuart and Janet for lots of fun during my first two years in Berkeley and later. Ann Irschick and Duncan Irschick for being the best landpersons one could wish for. Michael Coleman for being a great roommate during my stressful dissertation-writing months. Nils and Steve for pitching in with help on my project. All the fun people at UCBD for providing R&R when I needed it badly. Susan, Beth and Rebecca for being the most fun dance partners, and Jerry and Jeff for not always stealing them away. Sunny, Sue and Al for teaching the most fun PE classes. Greg for keeping me company at the RSF. Richard Stallman, Larry Wall and the Free Software Foundation for providing the software I used the most. I'm pretty sure I could never have finished without it. Richard Muller, Richard White, Shankar Sastry, Ted van Duzer, Martin Graham and Paul Gray for being a very friendly and helpful bunch of professors. Prof. Desoer for teaching the must educational class I had at Berkeley.

I would like to thank the members of my dissertation and qualifying exam committee, Professors Brayton, Hald and Fearing. Your help is appreciated. The administrative support from Tom Boot, Carole Frank and Peggye Brown has been excellent, and I thank you for your speedy response to all "emergencies".

Last, I would like to thank my mother and my late father, who never pressured me about school at all, with the result that I pressured myself all the more. It worked well! Thanks for everything.

Table of Contents

List of Figures

List of Tables

CHAPTER 1

INTRODUCTION

The late 1980s produced a tremendous development in the area of Application Specific Integrated Circuits (ASICs) and systems. Consumer electronic products such as Compact Disc (CD), Digital Audio Tape (DAT) players and video camcorders typically contain several ASIC parts. Telecommunication products such as mobile radios and telephones contain ASICs because of the requirements of compactness, low weight and low power consumption. A list of common applications is shown in Table 1-1.

What should be noted from this table is that the computationally intensive portions of these applications fall within the domain of Digital Signal Processing (DSP). There has been little development of ASICs that can be classified as belonging to the general area of Numerical Processing (NP). We conclude that the development of computationally intensive ASICs in the 1980s has largely been driven by DSP applications.

Common to most application specific systems is that they are *embedded systems,* meaning that they are intended to be self-sufficient and self-contained. This means, in particular, that they

Application area	Product	Acronyms
Consumer	Digital Audio (CD, DAT, DCC) Video products (camcorders, editing) High Definition Television (HDTV)	Compact Disk Digital Audio Tape Digital Comp. Cass
Telecom	Mobile radio and telephony Cellular radio Wireless computer networks Video telephone Voice-band data modems Switching systems (SDH, ATM) Encryption (DES, RSA public key) Speech recognition Speech synthesis	SDH=Standard Digital Hierarchy, ATM=Asynchrono us transmission mode, DES=Data Encryption Stan-dard, RSA=Rivest-Shamir-Adelman.
Workstations and computer products	Graphics processors Video compression (JPEG, MPEG) Multimedia support (audio, image) Vector processing units Disk drive read/write channel	JPEG=Joint Pho-tography Expert Group, MPEG=-Motion Picture Expert Group.
Imaging	Medical imaging Image analysis and reconstruction	

Table 1-1 Use of application specific integrated circuits and systems

cannot and do not rely on outside computational power from a general-purpose computer, typically because of size, power, economical and communication constraints. For systems requiring numerical processing, the solutions has traditionally been to build special purpose boards based on off-the-shelf microprocessor or DSP chips [chen86][gagli86][nara86][nara88]. The reliance on ASIC solutions has been much lower than in embedded DSP systems, partly because the design of numerical ASICs is often a more complex problem than the design of DSP ASICs and partly because the benefits were unclear. However, the demands on size, power and performance pertain to all embedded systems, and it will be shown that ASIC solutions for Numerical Processing will also result in considerable advantages.

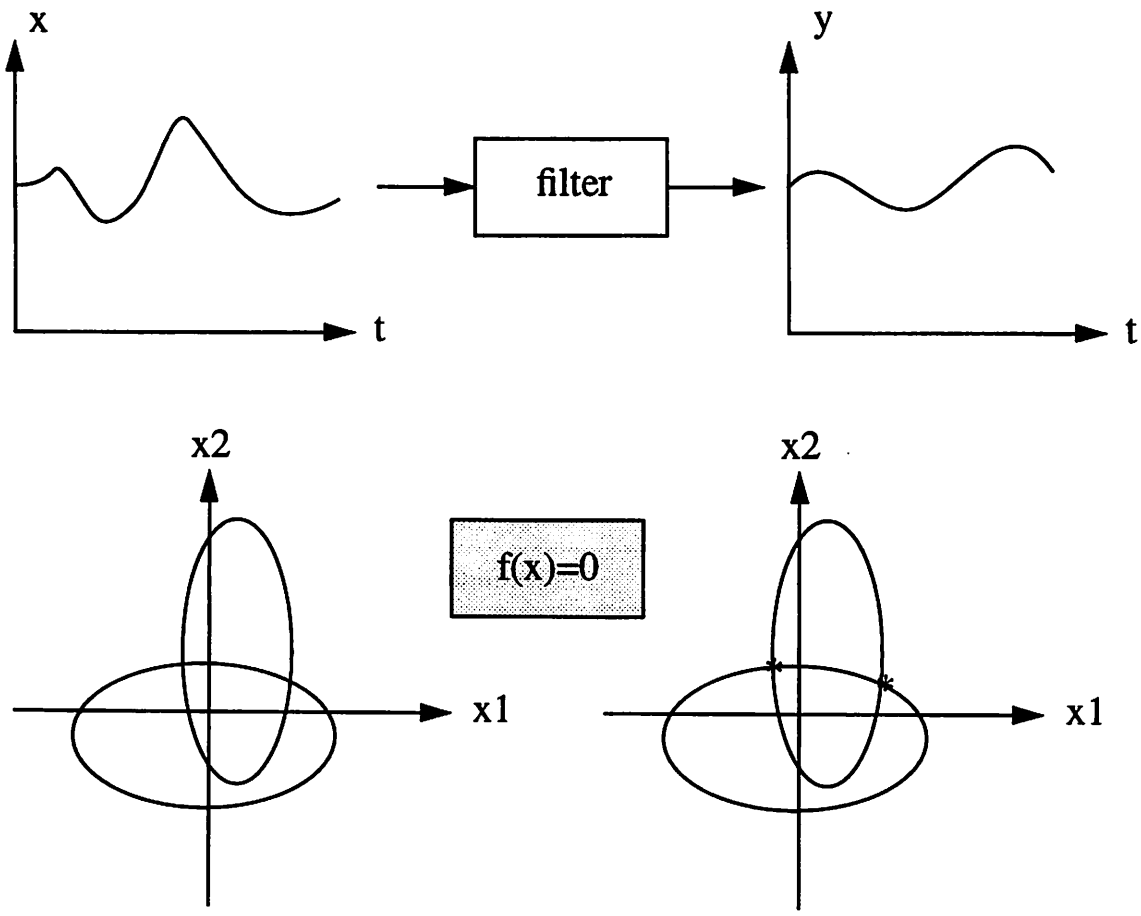


Figure 1-1 DSP versus Numerical Processing (a) Digital filter (b) Solving nonlinear equations

1.1 What is Numerical Processing?

The differentiation of Numerical Processing from DSP is not obvious. After all, both areas necessarily involve digital computation in some form. A typical example of DSP versus NP is illustrated in Figure 1-1. The digital filtering example has many of the characteristics of a DSP problem, such as an indeterminate data stream and real-time flow-through processing. The numerical example, on the other hand, consists of finding the solution(s) of two 2nd degree polynomial equations in two unknowns, corresponding to the intersection of two ellipses.

Digital Signal Processing	Numerical Processing
Indeterminate length data stream	Finite data set
Data originates as an analog real-world *signal*	Data is an equation that needs to be solved
Modifying data stream	Computing solution to equation
Closed form algorithm	Iterative algorithm
Scalar and vector operations: Add, Multiply, Delay, Accumulate	Matrix operations: Gaussian elimination, LU decomposition
Flow-through processing (data streams through pipelined datapath)	Data is shuffled back and forth between memory and datapath multiple times
Circular buffers, FIFO buffers	Matrix indexing, pivoting, permutations

Table 1-2 Typical characteristics of DSP and NP algorithms

Table 1-2 is an attempt to contrast DSP and NP by making a more complete list of some their opposing characteristics. The table itself is just a list of differences, and it does not attempt to explain the ramifications of these differences, but these will become clearer in later chapters as examples are considered in detail. The next section presents examples that will help establish the distinctive features of Numerical Processing.

1.2 Examples of Numerical Processing

In this section we will introduce two applications of numerical processing that will be used as design examples later on. Both examples are taken from the field of robotics, but this is mostly a coincidence. Under the surface, the examples contain elements that are common to many NP problems, and this means that they can serve as interesting design examples in this dissertation.

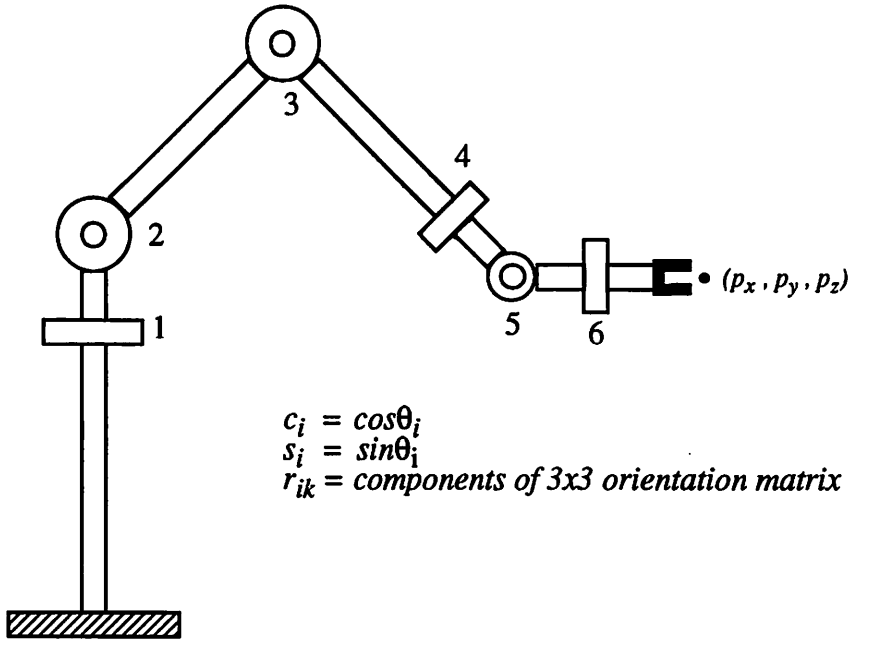


Figure 1-2 A fully articulated robotic arm

1.2.1 Inverse Position-Orientation problem for the PUMA robot

This application comes from the field of robot control and path planning, and is known as the inverse position-orientation (IPO) problem[1]. The background is as follows: The most advanced industrial robots have 6 revolute joints (6R) driven by independent actuators. A typical such robot is shown in Figure 1-2.

The robot is controlled by executing a particular position/speed/acceleration profile (over time) for each separate joint, and employing feedback to correct deviations from the given profile. However, the robot task is more naturally described in cartesian space than in joint space. Hence, we need to be able to compute a set of joint angles that correspond to a given position and orientation of the robot hand in the cartesian workspace. Our task is to compute the solutions of the IPO for the

1. Also known as the Inverse Kinematics problem

Puma 560 robot.

The input to the IPO algorithm for the PUMA 560 [craig86] is a data set that specifies a goal position and orientation for the robot end-effector. The algorithm itself is a closed form, but repetitive, calculation that executes 4 times to find all the solutions to the problem (there are 8 solutions total and the algorithm computes 2 of them at a time). The nature of the computation is geometrical, and the algorithm depends heavily on trigonometrical function evaluations.

1.2.2 Inverse Position-Orientation algorithm for general 6R robots

A closed form solution to the IPO problem for 6R robots is known only for the case when the last three joint axes intersect in a point [pieper68]. If the robot has a more general form, the solution cannot be found in closed form and the problem has to be attacked using an iterative numerical procedure. A goal position and orientation is commonly described in terms of a 4x4 matrix **T** which contains a 3x3 submatrix **R** that specifies the orientation and a 3x1 vector **p** that specifies the position. The equations that must be solved come about in the following way:

The functional relation between the angles $\theta=(\theta1,\theta2,\theta3,\theta4,\theta5,\theta6)$ and the resulting position and orientation $\mathbf{T=(R,p)}$ can be derived easily, so that $\mathbf{T=f(\theta)}$. Our problem is to solve the equation $\mathbf{f(\theta)=T}$, with **T** given, with respect to θ. If we consider $c_i=\cos(\theta_i)$ and $s_i=\sin(\theta_i)$ to be our basic unknowns, then the equation can be arranged so that it has the form of a system of n polynomial equations in n variables (unknowns), with terms up to 2nd degree present [tsai84]. The number of equations and variables to be included can also vary (with the extraneous variables being computable from the ones contained in the equations). One particular system with 8 complex-valued variables (4 sines and 4 cosines) will be used in this work. This means that we have a system of 8 complex equations in 8 variables. When the system is "realified", that is, the real and imaginary parts of the variables are treated separately, we can turn it into a system with n=16 variables and equations.

The preceding paragraph implies that the IPO problem has been refined into a problem of computing all solutions of a nxn system of 2nd degree polynomial equations. This has long been considered a difficult problem in numerical mathematics, and only in the last 10-15 years have robust methods been developed. The biggest problem is the requirement that ***all*** solutions to the system must be found. It is often possible to find at least one solution using a simple numerical method, but finding al the solutions is much more difficult. Two different approaches [tsai84][morgan86] [manocha92] have been successfully developed during the 1980s, and we will be concentrating on the method developed by Morgan and others.

Morgan's method is called *homotopy continuation*, and is based upon finding and solving an "easy" set of equations that is "close" to the "difficult" system that we really want to solve. Theory has been established which proves that there exists, under mild restrictions, a path from the solutions of the easy system to the solutions of the difficult system, and that one can track these paths using an iterative scheme, typically involving Newton's method [dahlquist74].

Applying Newtons method to solving a nonlinear nxn system $\mathbf{h(x)=0}$ involves three major parts, namely computing $\mathbf{h(x)}$, the jacobian matrix $\mathbf{Dh(x)}$, and solving a linear system of nxn equations. Of these three steps, the first two can easily be parallelized and computed on one or more general purpose DSP chips. Solving the systems of linear equations is the bottleneck in the process, and is what we will concentrate on as our core numerical problem for this particular application. In fact, most (if not all) multivariable iterative numerical problems will, if examined in detail, boil down to solving linear systems or performing other matrix operations. In other words, *linear algebra* is the key, and will be the focus of the second design case considered in this dissertation (Chapter 5 - Chapter 8).

Other applications involving polynomial systems

The IPO computation for the general 6R robot is just one example of an application where the problem essentially is computing all solutions of a system of polynomial equations. This

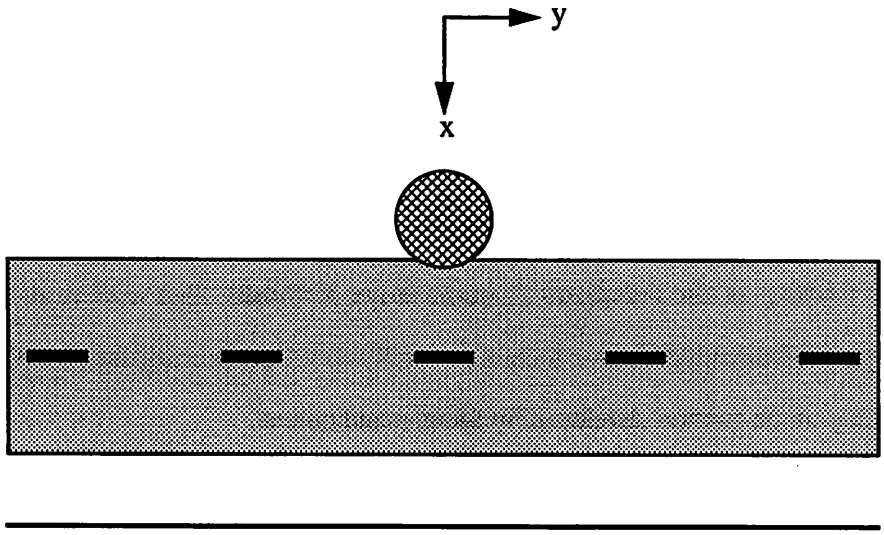


Figure 1-3 Cylindrical object making contact with elastic layer with stress sensors (from [pati88])

numerical problem arises in a number of other applications, including 3D graphics, kinetics of chemical reactions, and solid modelling [morgan87a].

1.2.3 Sensor inversion problems

Sensor inversion problems are common in systems that use arrays of sensor to measure some physical quantity and then need to translate the measured quantity into useful information. One example is from the area of tactile perception, where a mechanical hand is equipped with pressure (stress) sensors on its fingers, and one wants to compute the force and/or the shape of the contact area between the hand and the object that the hand is gripping. Such shape and force information is needed to be able to perform dextrous manipulation of the object by the hand. This problem has been studied in [pati88][fearing91], among others. Figure 1-3 shows the model used in [pati88]. The equations governing this system is the integral equation

$$\varepsilon_x(y) = \int_{-\infty}^{\infty} t(x, y - y_0) f_x(y_0) dy_0 \qquad (1\text{-}1)$$

where ε() is the resultant strain, t() is the "impulse response" of the elastic material and f () is the

applied force (or line pressure). The problem now is to compute the line force f from the measured ε(). This can be achieved by discretizing the equation over the region of interest, resulting in a system of linear equations

$$\varepsilon = Tf \qquad or \qquad Tf = \varepsilon \tag{1-2}$$

where we can solve for the force vector f. Depending on the sensor and hand configuration, the size if the linear system will vary but should remain fairly small (n<16) for most applications. A multifingered hand may have many sensor arrays and will need considerable amount of numerical processing power to compute all the forces on all the fingers in real time (100Hz sampling rate). The force (and possibly shape) information could then be fed to the control system to achieve dextrous manipulation of the object. It should be mentioned that inversion problems of this type are often somewhat ill-conditioned and may require Least Squares solution methods, which nevertheless will amount to linear equation solving in the end.

1.2.4 Other numerical applications

- Control systems: (recursive) state estimation, (recursive) parameter estimation, system identification, Newton-Euler equations for robot dynamics.
- Radar, antenna array processing, multivariable signal processing.
- Real-time optimization, flexible structure control.

1.3 Implementation alternatives for NP systems

Because of differences between DSP and NP, such as the ones shown in Table 1-2, the architectures and design methods used for DSP systems are not always directly applicable to NP design problems. We will now discuss some of the building blocks, design tools and methods used for DSP ASIC design, and indicate some of their shortcomings with respect to Numerical Processing problems.

1.3.1 DSP building blocks

The first distinction we need to make is between systems based (partly) on off-the-shelf building blocks and systems based on ASIC (full custom) solutions. Table 1-3 shows some of the most common DSP building blocks. The question is whether these processors can be efficiently applied to numerical problems. The answer is that in many cases they are in fact very attractive candidates: All of them support floating-point arithmetic and they work at quite decent clock speeds by today's standards. However, for certain numerical problems, such as matrix computations, the processors are not ideally suited. As will be explained in detail later (Chapter 7), none of the processors are capable of executing the inner loop of a Gaussian elimination (or LU-decomposition) at full pipeline speed. Table 1-4 shows the performance of some common DSP chips on the inner loop of LU decomposition. It turns out that all the standard DSP chips have been designed with one goal in mind, namely single-cycle execution of the inner loop of FIR and IIR filter programs. In essence, the problem is that FIR/IIR filters needs multiply-accumulate (MAC) instructions whereas LU decomposition needs multiply-add(-store) (MADD) instructions. The MADD instructions involve

Building block	Manufacturer	Clock frequency	Instruction latency
DSP32C	ATT	50.0 MHz	80ns
TMS320C30	Texas Instruments	33.4 MHz	60ns
MC96002	Motorola	40.0MHz	50ns

Table 1-3 Some standard programmable DSP building blocks

Processor	Icycles/iteration	Time/iteration	Ideal time
DSP32C	1.5	120ns	20ns
TMS320C30	2.0	120ns	60ns
MC96002	2.0	100ns	50ns

Table 1-4 Performance of DSP chips for LU-decomposition (no pivoting)

3 memory references (2 read, 1 write) instead of 2, and none of the commercial DSP architectures are designed with this requirement in mind.

In addition, there is the problem of *pivoting* [dahlquist74]. For all but trivial-sized LU problems, it is necessary to rearrange the rows of the matrix after each major elimination step, so as to preserve the numerical stability of the procedure. This rearrangement can be done by outright swapping of rows in the memory or by using a level of indirection in the addressing so that the row address is passed through a permutation table which holds the information about how the rows are currently rearranged. The DSP chips are even less able to deal with this type of situation. An extra level of indirection will lead to additional stall of the floating-point pipeline, and decrease the performance of the inner loop by another factor of 2 or 3. This means that the chips may run at only 1/5 of their peak rate, which is clearly not very good utilization of their high-speed floating-point units.

A third problem with programmable DSP chips is that they are not generally available as chip cores, that is, a designer cannot take one of the DSP chips and use it as a core around which to put other functional blocks, all on one chip: It is often the case in e.g. telecom applications that a designer would like to have the basic functionality of a programmable DSP chip, but will also need to integrate it with other more specialized functions on a single chip.

Finally, even if the DSP is available as a chip core, one will often see that only certain parts of the core are really useful, meaning that there is a lot of wasted area on the chip. If the designer needs a programmable chip solution, it would be better to have architectural flexibility so that the designer can include exactly those features and functional blocks that are needed. This theme will be explored in Chapter 3 on C-to-Silicon compilation.

1.3.2 General purpose computers

The scope of this dissertation is embedded, real-time, application specific systems. As indicated earlier, such systems are typically constrained in size, weight, power consumption and communication capability. This means that general purpose computers (workstations,

supercomputers, etc.) are generally not a viable option. One could consider building systems based on commercial microprocessors (μP) and floating point coprocessors, but this alternative involves complications such as the need for external memories, caching schemes, memory management overhead and system busses. These complications are even more a concern for general purpose uP chips than for DSP chips, since DSP chips always have a fair amount of on-board memory and do not support virtual memory. The conclusion is that most uP chips are not a good alternative for embedded NP systems. However, to be fair, Chapter 7 contains an extensive and often detailed evaluation of a number of alternative implementation vehicles, including several DSP chips, RISC Microprocessors, Vector processors and other more specialized architectures.

1.3.3 ASIC DSP design systems

There has been considerable work done in the area of ASIC DSP automatic design tools. Currently, the most popular form of these toolkits is the *integrated CAD environment*, meaning a "complete" system which supports all aspects of the design process. Many such toolkits are now available either from academia or from commercial vendors. The toolkits are typically distinguished by application area (specialization), and what level of detail the input description is at. For example, there are systems that use structure descriptions, RTL descriptions and behavior descriptions as their main input form. Some of the systems will be discussed below, mainly HYPER [rabaey91], Cathedral-II [rabaey88], McDAS [hoang92], PADDI [chen92], LAGER, FIRGEN [jain91], BLIS [whitcomb92], Olympus [micheli90] and C-to-Silicon [thon92][rb92].

Most of the work in this dissertation is based on the LAGER [rabaey85] [shung89] [shung91][rb92] design tools, either by direct application of the tools or building on top of them. The core of LAGER is a silicon assembler that allows a high-level structure specification of an ASIC in terms of parameterized functional blocks. A limited behavior specification capability is also supported in terms of boolean equations. LAGER comes with a fairly extensive collection of predesigned library blocks which the user can call up, parameterize (personalize) and interconnect at will. The purely structural approach is most often used when performance is critical so that

manual design of the structure of the datapaths, memory and control is necessary.

LAGER is the common denominator of several higher-level synthesis systems and architecture exploration tools. HYPER [rabaey91] is a synthesis system which converts a dataflow algorithm specification into a hardwired ASIC implementation. The algorithm is specified as a data flowgraph using the Silage language, and HYPER goes through several steps such as resource allocation, datapath synthesis, operation scheduling and control synthesis to produce the structure description that is passed on to LAGER. HYPER also can target other implementation forms than ASIC chips. McDAS [hoang92] and PADDI [chen92] are two subparts of HYPER that target multiprocessor DSP implementations and field-programmable DSP architectures, respectively. However, common to all HYPER tools is that they target medium data-rate applications and that they generate hardware implementations with flow-through pipelined processing, low levels of resource sharing and fairly simple hardwired control. This of course reflects upon the basic design decision which was to base the tools on dataflow-type algorithm descriptions. Many DSP applications fit well into this pattern, but this is unfortunately not the case for a typical Numerical Processing problem.

C-to-Silicon [thon92] [rb92] is an architecture exploration and design system for programmable ASIC DSP chips. It allows a procedural description of the users' algorithm in the RL language (which is a subset of C extended with a fixed-point datatype [rimey89]), and generates a microprogrammed ASIC, using a progressive refinement of architecture descriptions provided by the user. The strength of the C-to-Silicon system is that the user only needs to provide a very high-level description of the architecture in order to compile the algorithm and get accurate information on program size and execution speed. Only after the appropriate architecture has been determined is it necessary to develop the more detailed structure description. The RL compiler is retargetable by the architecture description and can thus cover a wide variety of possible architectures and performances. C-to-Silicon is typically better suited for Numerical Processing than is dataflow synthesis programs, since numerical algorithms tend to contain more control (conditional

execution) and a much higher degree of resource sharing and iterative reuse of data. Chapter 3 and Chapter 4 of this dissertation describes the C-to-Silicon system and how it was applied to design the PUMA chip. However, C-to-Silicon is certainly not suited for all kinds of numerical algorithms: Matrix operations, matrix addressing, and pipelined vector processing are not easily expressed in C, resulting in a considerable performance penalty. For such purposes it will be necessary to apply more direct, specialized and less automated methods.

Another tool built on top of LAGER is FIRGEN [jain91]. FIRGEN generates FIR filter layout from a frequency domain specification, by first generating a structure description from the behavior description, and then let the LAGER tools and cell libraries create the layout. In fact, [sriva92] points out that this appears to be a common theme for the higher-level design tools: There are two phases of the design; one that generates an architecture (structure description) from a behavior specification, and the other that generates the physical layout from the architecture. The two phases communicate via a well-defined structure description interface.

Other ASIC DSP design systems include Cathedral-II [rabaey88], Bit Serial Silicon Compiler (BSSC) [jassica85], BLIS [whitcomb92] and Olympus [micheli90]. Cathedral-II is somewhat similar to HYPER and is also geared towards medium rate DSP applications rather than Numerical Processing. Bit serial arithmetic, as in BSSC, is not practical for numerical applications, which most often require floating point hardware.

BLIS (Behavior-to-Logic Interactive Synthesis System) is a high-level synthesis system suited for control-dominated designs, such as cache controllers, microprocessors, communication chips, etc. BLIS supports functional-level synthesis from the ELLA language. Olympus is a synthesis system based on the Hardware-C language. It supports multilevel synthesis, technology mapping and simulation. BLIS and Olympus assume that there exists hardware primitives, typically at the gate level, that can be used to assemble the layout. While these are indeed impressive design systems, assembling the hardware from primitives (Standard cell, Sea-Of-Gates, gate array) most often will not suffice when targeting high-performance Numerical Processor design. BLIS/Olympus are

reasonable alternatives for less performance-oriented designs such as the PUMA chip (which was based on the C-to-Silicon system, Chapter 4). However, the C-to-Silicon system is by no means restricted to low sample rates or clock rates, as the user can obtain high levels of performance by providing the necessary high-speed building blocks. One could argue that BLIS/Olympus can solve this problem as well by supplying them with the same building blocks, but that would in a sense defeat the purpose of the systems, considering that their main goal is exactly to *synthesize* the blocks that are needed in the design. Finally, the synthesis systems, by nature, offer less control over the resulting architecture, and it is not as easy to perform architecture explorations as in the C-to-Silicon system. The lack of direct support for fixed-point (or other non-bitvector-like datatypes) should also be noted.

1.3.4 DSP board-level design tools

Although the scope of this dissertation is Application Specific Integrated *Circuits* (ASICs), as opposed to Application Specific Integrated *Systems* (ASISs), it seems relevant also to consider some tools that are used for board-level design, since board-level systems certainly can be built for the purpose of Numerical Processing as well as DSP.

Ptolemy [ptolemy91] is a DSP block-diagram simulator and to lesser extent a system implementation tool. It supports multiple computation models such as synchronous dataflow, dynamic dataflow and event-driven simulation. The computation models are called *domains*. The synchronous dataflow domain has a code-generation capability which allows generation of code fragments that can be tied together and executed on commercial DSP chips such as the Motorola MC56000. This approach combines some of the dataflow concepts as seen in HYPER with the use of commercial DSP building blocks. The drawbacks with respect to Numerical Processing are the same as mentioned earlier. Being mostly a simulation environment, Ptolemy does not have any support for the actual board design or assembly of the hardware. The code fragments can be plugged into the processor(s) but the designer must add on the code and hardware necessary to allow data communication and other interaction between the processor(s) and the outside world.

Vulcan-II [gupta92a][gupta92b] is a board-level design system which is under development at Stanford University. It provides the ability to map an algorithm described in Hardware-C [micheli90] to multiple ASICs and one software-controlled microprocessor, such that part of the functionality is implemented in software. Vulcan itself performs the *partitioning* subtask, with the chip designs being carried out in the Olympus framework and the software being developed on a workstation or a uP/DSP development system The novelty of Vulcan is that the partitioning is taking place at the *algorithm description* level, as opposed to the more common hardware module partitioning. Vulcan is helpful in determining a reasonable partitioning of the algorithm onto different hardware blocks. Other systems typically rely on the expertise of the designer to perform this task. Sometimes a designer can also do a good job at algorithm-level partitioning, given the expert knowledge about the purpose and nature of the computation. Manual partitions often are done along natural functional boundaries. An example of functional partitioning is presented in Chapter 5, where it is shown that one part of a problem is suited for a chip implementation, and the remainder can be implemented on a generic programmable processor.

Siera

Siera is a board-level system design tool under development at UC Berkeley [sriva92]. The system uses a *layered* approach to system design (Table 1-5), where the layers represent increasing specialization, communication bandwidth and ability to meet real-time constraints. The board-layout generation facilities of Siera were modelled after LAGER, using the SDL language and parameterizable modules as the main features.

Siera is a flexible and powerful system that can be used for a variety of applications, including board-level design of Numerical Processing systems.

1.4 Problems in Numerical Processor design

Examining the above examples has revealed that many of the architectures, design tools and design techniques used in the DSP world are not necessarily well suited for NP chip designs. The

requirements of NP applications vary widely, as will be demonstrated by the relatively large differences between the PUMA chip and the SMAC (Chapter 8) architecture. Still, there are certain common features that need to be supported, even if they are not present in all NP tasks. They are:

- **Architecture.** Emphasize architectures that support iterative processing as opposed to flow-through processing. Support multiport memory access and efficient addressing and address generation techniques, especially for 2-dimensional addressing (matrices). Support pivoting for efficient row operations in linear algebra problems. Use deep pipelining to achieve high-speed vector processing.
- **Tools.** Emphasize the ability to explore a variety of architectures so that it becomes easy to select a cost-effective implementation. Support high-level algorithmic input and the proper arithmetic datatypes for numerical tasks.
- **Circuit techniques.** Pipelining and the resulting high clock rates make the latches and the clocking schemes critical. Consider using newer clocking schemes such as TSPC (true single phase clocking) [yuan87][yuan89][afghahi90]to cut down on the number of clock wires and to reduce clock skew problems.

Layer	Implementation level	Example Hardware/Software implementation
1	Workstation	Sparc 2 SUN OS 4.1
2	Single-Board Computer	Heurikon HKV-30 (MC68020) on LAN Vx Works kernel
3	Processor Module	TMS320C30 on VME bus SPOX kernel
4	ASIC Slave Processors	Pulse Width Modulator (Motor Control) Slave Bus

Table 1-5 SIERA uses a layered approach to system implementation

1.5 Summary

The purpose of this chapter is to establish that current design tools and methods for DSP ASICs and systems are not always sufficient when transplanted into the domain of Numerical Processing. The remainder of this dissertation describes two cases of Numerical Processor tool and chip development, starting out with a concrete numerical problem, some algorithmic alternatives and previous designs of ASICs for DSPs as the background setting. The algorithms have been analyzed in detail to establish their architectural and implementation requirements, and tools as well as chips have been developed.

The first case is the development of the C-to-Silicon design system (Chapter 3) and its use to design the PUMA chip (Chapter 4). C-to-Silicon is a high level design system that supports the design path from a C program algorithm description down to silicon implementation, while providing powerful tools for architecture experimentation, performance estimation and numerical verification. The PUMA chip is the prototype design for the C-to-Silicon system.

The second case is the design of the Small MAtrix Computer (SMAC) architecture, and the design of a set of hardware building blocks that can be used to implement SMAC. The SMAC chapters also presents details of high-speed circuit design, addressing architecture and pipelined floating-point design.

Both PUMA and SMAC have their background in the robot IPO problem introduced in this chapter. PUMA implements the simpler case, whereas as SMAC is aimed at the general case as well as other numerical problems involving matrix computations.

The remainder of this dissertation is organized as follows:

- Chapter 2 is a survey of the robot IPO computation problem.
- Chapter 3 presents the C-to-Silicon system.
- Chapter 4 describes the architecture exploration and hardware development of the PUMA chip.

- Chapter 5 surveys background material on the homotopy continuation method used for solving nxn polynomial systems, in particular as applied to the general 6R robot IPO problem. A program package (ConsolC) is developed to experiment with different algorithms and to establish some numerical properties of the algorithms. It is found that the computational bottleneck is to solve small systems of linear algebraic equations.
- Chapter 6 discusses available algorithms for solving linear equations. An investigation of the properties and computational requirements of the algorithms are presented.
- Chapter 7 is an evaluation and comparative study of commercial computing architectures with respect to their efficiency in solving linear equations. The basic architectural requirements for an Application Specific Processor are identified.
- Chapter 8 describes the SMAC architecture, and the design, simulation and testing of a collection of high-speed building blocks for SMAC.
- Chapter 9 is the conclusion.

CHAPTER 2

INVERSE POSITION-ORIENTATION (IPO) COMPUTATION

The Inverse Position-Orientation (IPO) computation is a classical numerical problem that will be used as an example in this work. It is ideally suited for an investigation of Numerical Processing, because there exists a variety of algorithms for the IPO problem, ranging from fairly simple to quite complex. This means that we can draw upon several different approaches to IPO to illustrate various aspects of Numerical Processing while at the same time staying within the same general application domain. This chapter will introduce IPO problems at several levels of complexity and generality and explain how they can be solved using Numerical Processing techniques. The algorithms will to some extent be explained in detail, based on the work of Morgan [morgan87a][morgan87b]. Though much of this chapter is mathematical in nature, the conclusions are simple and have immediate implications with respect to Application Specific Processors, which is our primary focus. We will start out by describing the general IPO problem.

2.1 Kinematics of mechanisms

A general mechanism (robotic or otherwise) consists of rigid members (links) connected by joints allowing relative motion of the links. Joints can be either prismatic (sliding, translational) or rotational. Figure 2-1 shows a general geometrical model (the **Denavit-Hartenberg** model) of a link-joint mechanism. A general link ($Link_{i+1}$) and the one joint ($Joint_{i+1}$) rigidly attached to it can be uniquely described for kinematic purposes by a 4-tuple of parameters (a_i, d_i, α_i, θ_i) known as the (length, offset, twist, joint angle) of the link. The parameters are derived from the geometry of the link as seen relative to the plane which is uniquely determined by the lines through the joint axis of the previous link (z_i) and the joint axis of the current link (z_{i+1}). Translational links have

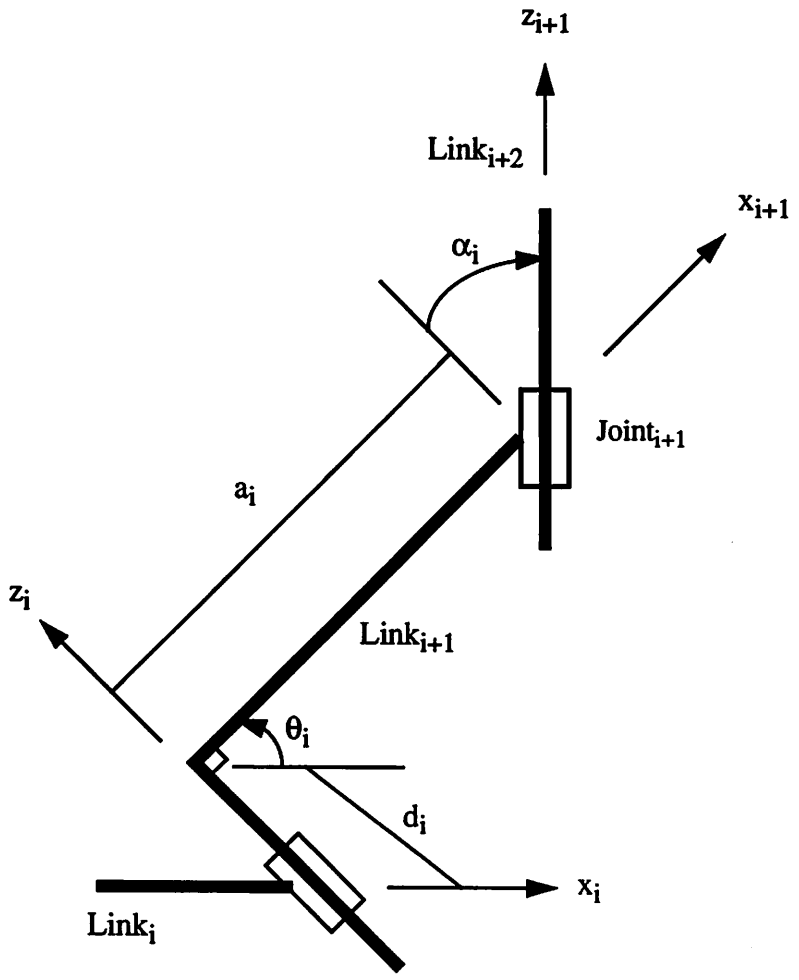


Figure 2-1 Denavit-Hartenberg link parameters (adapted from [morgan87a])

variable values for the offset d_i and rotational links have variable values for the joint angle θ_i.

One of the basic problems in mechanism theory is to compute the position and orientation of the joint at the end of the last link in a series of links, given the constant and/or variable values for the Denavit-Hartenberg parameters. This problem is known as (forward) kinematics. It is assumed that each joint has a coordinate system rigidly attached to it, with the origin of the system defining the joint position and the directions of the unit coordinate vectors defining the orientation of the joint. The position and orientation are usually expressed in the form

$$\mathbf{p} = \begin{bmatrix} p_x \\ p_y \\ p_z \end{bmatrix} \qquad \mathbf{R} = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix} \tag{2-1}$$

where p is the position vector and R is a 3x3 orientation matrix. The 3 column vectors of R give the coordinates of the unit vectors as seen from the coordinate system on the previous joint (or some arbitrary base system in case the link in question is the first one in the chain).

Transformation matrices

The mathematical construct for handling this type of geometrical problem is known as the *homogenous transform.* Suppose that we have three coordinate systems (0,1,2) placed arbitrarily in space, with 0 considered the base coordinate system. Also suppose that we know the orientation $\mathbf{R}^{12}$ of system 2 (in 1-coordinates) and the orientation $\mathbf{R}^{01}$ of system 1 (in 0-coordinates). Suppose further that we know the vector $\mathbf{p}^{01}$ (in 0-coordinates) and the vector $\mathbf{p}^{12}$ (in 1-coordinates). We would like to compute the position vector $\mathbf{p}^{02}$ and the orientation $\mathbf{R}^{02}$ (in 0-coordinates). Now, a given vector **p** may be expressed in several different coordinate systems. For example, if we know the coordinates ${}^1\mathbf{p}=(p_x,p_y,p_z)$ of **p** in system 1 then we can find the coordinates ${}^0\mathbf{p}$ if we know the coordinates of the unit vectors of system 1 as seen from system 0. The transformation consists simply of replacing the unit vectors (e_x,e_y,e_z) by their coordinates as seen from system 0. In other words,

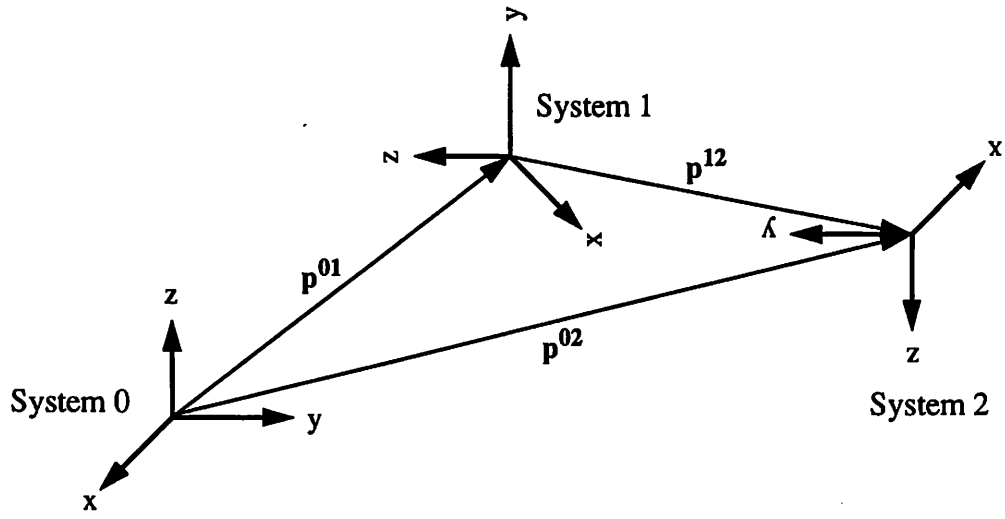


Figure 2-2 Transformation of coordinates between systems

$$ {}^{1}\mathbf{p} = \begin{bmatrix} p_x & p_y & p_z \end{bmatrix} \Rightarrow \mathbf{p} = p_x\mathbf{e}_x + p_y\mathbf{e}_y + p_z\mathbf{e}_z \Rightarrow {}^{0}\mathbf{p} = p_x \begin{bmatrix} e_{x1} \\ e_{x2} \\ e_{x3} \end{bmatrix} + p_y \begin{bmatrix} e_{y1} \\ e_{y2} \\ e_{y3} \end{bmatrix} + p_x \begin{bmatrix} e_{z1} \\ e_{z2} \\ e_{z3} \end{bmatrix} \qquad (2\text{-}2) $$

This means that the coordinates of **p** in system 0 are given by a simple matrix multiplication with the rotation matrix **R**, that is,

$$ {}^{0}\mathbf{p} = \begin{bmatrix} e_{x1} & e_{y1} & e_{z1} \\ e_{x2} & e_{y2} & e_{z2} \\ e_{x3} & e_{y3} & e_{z3} \end{bmatrix} \begin{bmatrix} p_x \\ p_y \\ p_z \end{bmatrix} = \mathbf{R}^{01} \cdot {}^{1}\mathbf{p} \qquad (2\text{-}3) $$

By applying this relation both to the vectors $\mathbf{p}^{01}$ and $\mathbf{p}^{12}$ and to the vectors making up the orientation matrices $\mathbf{R}^{01}$ and $\mathbf{R}^{02}$ we can deduce that

$$ \mathbf{p}^{02} = \mathbf{p}^{01} + \mathbf{R}^{01}\mathbf{p}^{12} \qquad \text{and} \qquad \mathbf{R}^{02} = \mathbf{R}^{01} \cdot \mathbf{R}^{12} \qquad (2\text{-}4) $$

It is customary just to use the simplified symbols ($\mathbf{p}$, $\mathbf{p}_1$, $\mathbf{p}_2$, $\mathbf{R}$, $\mathbf{R}_1$, $\mathbf{R}_2$) instead of ($\mathbf{p}^{02}$, $\mathbf{p}^{01}$, $\mathbf{p}^{12}$, $\mathbf{R}^{02}$, $\mathbf{R}^{01}$, $\mathbf{R}^{12}$). The equations can then be written

$$ \mathbf{p} = \mathbf{p}_1 + \mathbf{R}_1\mathbf{p}_2 \qquad \text{and} \qquad \mathbf{R} = \mathbf{R}_1 \cdot \mathbf{R}_2 \qquad (2\text{-}5) $$

Since generally we are interested in both **R** and **p**, a notation has been devised where **R** and **p** are

made into components of a 4x4 matrix T, as follows:

$$T = \begin{bmatrix} R & p \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & p_x \\ r_{21} & r_{22} & r_{23} & p_y \\ r_{31} & r_{32} & r_{33} & p_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2-6}$$

Using this notation, the coordinate transformation task can be expressed compactly as

$$T = \begin{bmatrix} R & p \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} R_1 \cdot R_2 & p_1 + R_1 p_2 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} R_1 & p_1 \\ 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} R_2 & p_2 \\ 0 & 1 \end{bmatrix} = T_1 T_2 \tag{2-7}$$

This notation is known as the 4x4 homogenous transform notation.

Homogenous transform for a general link

A general link with given Denavit-Hartenberg parameters (a_i, d_i, α_i, θ_i) has the following transformation matrix relating the position/orientation of the joint at the end of the link to the position/orientation of the joint at the beginning of the link:

$$T_i = \begin{bmatrix} R & p \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} c_i & -s_i\lambda_i & s_i\mu_i & a_i c_i \\ s_i & c_i\lambda_i & -c_i\mu_i & a_i s_i \\ 0 & \mu_i & \lambda_i & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2-8}$$

where

$$c_i = \cos\theta_i \quad s_i = \sin\theta_i \quad \lambda_i = \cos\alpha_i \quad \mu_i = \sin\alpha_i \tag{2-9}$$

This relation can be derived by breaking down the action of the link into 2 translations and 2 rotations, corresponding to the 4 parameters, and multiplying the corresponding matrices together ([craig86] contains one derivation but uses somewhat different parameter definitions than [morgan87a]). For a 6 link robot arm, the implication is that the position and orientation of the endpoint of the arm is given by the product of 6 matrices, that is,

$$T = T_1 T_2 T_3 T_4 T_5 T_6 = f(\theta_1, \theta_2, \theta_3, \theta_4, \theta_5, \theta_6) = f(\theta) \tag{2-10}$$

By nature of the matrix multiplication, Each element of T is a complicated *polynomial* expression over some subset of the variables c_i and s_i (i=1:6). The matrix equation (2-10) is known as the forward kinematic equation for the robot, meaning that given the angles one can plug in the numbers and compute the position and orientation of the endpoint. We could also call it the Forward Position-Orientation (FPO) equation.

2.2 IPO computation

The Inverse Position-Orientation problem is the opposite of the task described in the preceding paragraph. This time, we are given the desired goal T and want to compute all values of θ that produce the desired T. This means, in general, that we have to solve a complicated set of nonlinear equations and find *all* possible solution points. The equations do not in general have a closed form solution, meaning that some iterative numerical method has to be employed.

This causes a multitude of problems. A common approach such as Newton's method [dahlquist74] does not always converge to a solution, and even if it does, it will only find *one* solution. The process can be repeated from different starting points and possibly leading to other solutions. However, there is no way to know beforehand exactly how many solutions there are (the answer may range from none to infinitely many for any given case), so that another problem is to know how many solutions to look for.

There are several ways to get around these problems. One is to design the robot so that the last three joint axes intersect. In this case, a closed form solution can be derived for the IPO equations exists. This result was derived by Pieper and is described in [craig86] p.112-119. This type of solution is described in the next section (2.3) and is the basis for the PUMA chip design. For the general case, the problem has remained largely unsolved until the 1980s, when Morgan and others developed new numerical methods based on the *homotopy continuation* principle. This method and some of the theory behind it will be explained in section 2.4. Recently, other methods have also been developed [manocha92], but we will concentrate on using Morgan's results as the main

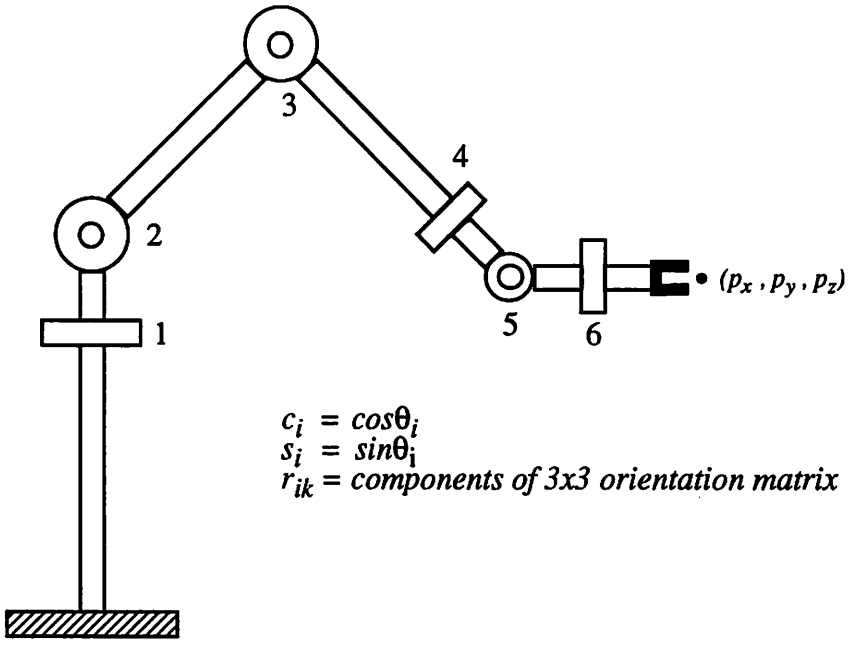


Figure 2-3 Stick diagram of the PUMA 560 industrial robot

source of Numerical Processing algorithms in this work.

2.3 Special case IPO computation: The PUMA robot

The PUMA is a standard industrial robot with 6 revolute joints, as shown in Figure 2-3. The joints are driven by independent actuators (motors). The robot is controlled by executing a particular position/speed/acceleration profile (over time) for each separate joint, and employing feedback to correct deviations from the given profile. However, the robot task is more naturally described in cartesian space than in joint space. Hence, we need to be able to compute a set of joint angles that correspond to a given position and orientation of the robot hand in the cartesian workspace.

The closed form IPO solution derived by Pieper [craig86] is shown in Figure 2-4. The equations are listed in the order they should be evaluated when computing the solutions. Because the formulas contain two binary alternatives in the form of alternative signs ±sqrt(), most of the computation has to be repeated 4 times to produce 4 different solutions.

$$\theta_1 = \text{atan2}(p_y, p_x) - \text{atan2}(d_3, \pm \text{sqrt}(p_x^2 + p_y^2 - d_3^2)) \quad (2\text{-}11)$$

$$K = (p_x^2 + p_y^2 + p_z^2 - a_2^2 - a_3^2 - d_3^2 - d_4^2) / (2\, a_2) \quad (2\text{-}12)$$

$$\theta_3 = \text{atan2}(a_3, d_4) - \text{atan2}(K, \pm \text{sqrt}(a_3^2 + d_4^2 - K^2)) \quad (2\text{-}13)$$

$$\theta_{23} = \text{atan2}\left((-a_3 - a_2 c_3)\, p_z - (c_1 p_x + s_1 p_y)(d_4 - a_2 s_3),\right. \quad (2\text{-}14)$$

$$\left.(a_2 s_3 - d_4)\, p_z + (a_3 + a_2 c_3)(c_1 p_x + s_1 p_y)\right) \quad (2\text{-}15)$$

$$c_4 s_5 = -r_{13} c_1 c_{23} - r_{23} s_1 c_{23} + r_{33} s_{23} \quad (2\text{-}16)$$

$$s_4 s_5 = -r_{13} s_1 + r_{23} c_1 \quad (2\text{-}17)$$

$$\theta_4 = (c_4 s_5 c_4 s_5 + s_4 s_5 s_4 s_5 < \varepsilon) \ ? \ \theta_4^{old} : \text{atan2}(s_4 s_5, c_4 s_5); \quad (2\text{-}18)$$

$$s_5 = -r_{13}(c_1 c_{23} c_4 + s_1 s_4) - r_{23}(s_1 c_{23} c_4 - c_1 s_4) + r_{33} s_{23} c_4 \quad (2\text{-}19)$$

$$c_5 = -r_{13} c_1 s_{23} - r_{23} s_1 s_{23} - r_{33} c_{23} \quad (2\text{-}20)$$

$$\theta_5 = \text{atan2}(s_5, c_5) \quad (2\text{-}21)$$

$$s_6 = -r_{11}(c_1 c_{23} s_4 - s_1 c_4) - r_{21}(s_1 c_{23} s_4 + c_1 c_4) + r_{31} s_{23} s_4 \quad (2\text{-}22)$$

$$c_6 = r_{11}((c_1 c_{23} c_4 + s_1 s_4) c_5 - c_1 s_{23} s_5) \quad (2\text{-}23)$$

$$+ r_{21}((s_1 c_{23} c_4 - c_1 s_4) c_5 - s_1 s_{23} s_5) \quad (2\text{-}24)$$

$$- r_{31}(s_{23} c_4 c_5 + c_{23} s_5) \quad (2\text{-}25)$$

$$\theta_6 = \text{atan2}(s_6, c_6) \quad (2\text{-}26)$$

The number of solutions is doubled (from 4 to 8) by applying the modifications:

$$\theta_4 = \theta_4 + \pi \qquad \theta_5 = -\theta_5 \qquad \theta_6 = \theta_6 + \pi \quad (2\text{-}27)$$

Figure 2-4 Closed form solution to IPO equations for the PUMA 560 robot

The input to the IPO algorithm for the PUMA 560 [craig86] is a data set that specifies a goal position and orientation for the robot end-effector. The nature of the computation is geometrical, and the algorithm depends heavily on trigonometrical function evaluations. Otherwise the computation is straightforward and can easily be programmed into a general purpose or special purpose computer.

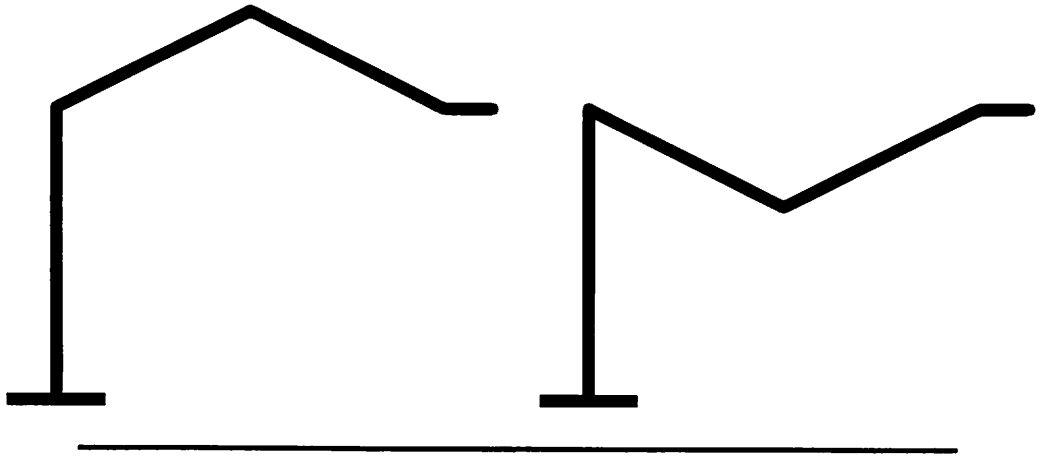


Figure 2-5 Example of multiple solutions to the IPO problem

All immobile robots have a limited workspace, meaning that only certain positions and orientations are reachable. In fact, by *reachable workspace* [craig86] is meant the volume of space that can be reached with at least one orientation of the hand. The *dextrous workspace* is defined as the volume of space which can be reached with all orientations. The dextrous workspace is a subset of the reachable workspace. For PUMA, the reachable workspace is bordered by two concentric spheres, with all positions in between the spheres reachable. Inside most of the reachable workspace, any orientation is possible, whereas near minimum and maximum reach, more limited sets of orientations are possible. For example, at full reach, only the radial orientation is possible.

In addition to workspace considerations, there is also the question of how many solutions exist to the problem. It was mentioned that there is up to 8 solutions for a given position and orientation of the PUMA robot. Figure 2-5 illustrates multiple ways of reaching a given point and orientation. This particular multiplicity is known as the elbow up/down multiplicity of joint 3. There is also the possibility of the robot turning its "back" on the goal point and reaching around over its shoulder, resulting in 2 more solutions. Finally, there is another elbow up/down multiplicity of joint 5, resulting in a total of 8 solutions. When an elbow (joint 3 or 5) is straight, there is no difference between elbow up and elbow down, meaning that the two normally different solutions degenerate into just one. This is an example of a *singularity*. The technical definition of a singularity involves

the Jacobian matrix of the FPO equations, but generally a singularity corresponds to a multiple root of the IPO equations.

The concepts of workspace and singularities appear naturally as part of the IPO equations (2-11) to (2-26). In equation (2-11), a negative argument in the square root corresponds to a point that is too close to the robot. In equation (2-13), a negative argument corresponds to a point that is outside the reachable workspace. The notation a ? b : c used in equation (2-18) has the same meaning as in the C programming language. Equation (2-18) handles the joint 5 elbow singularity. This singularity is more severe than the joint 3 elbow singularity, because when joint 5 is straight, there is an infinite number of solutions for the angles θ_4 and θ_6 (only the difference θ_4-θ_6 is significant). The solution chosen is to keep whatever is the current value of θ_4 and compute θ_6 to match it. The joint 3 elbow singularity is less severe in that it is simply a solution of multiplicity two. No particular computational precautions are needed.

2.4 IPO for general 6R robots

This section describes how the system (2-10) can be reduced into a system of 8 polynomial equations with 8 unknowns and 2nd degree terms, following the reduction work performed by Morgan and Tsai [tsai84][morgan87a]. The first step in the reduction is to write (2-10) in the form

$$\mathbf{T}_3\mathbf{T}_4\mathbf{T}_5 = \mathbf{T}_2^{-1}\mathbf{T}_1^{-1}\mathbf{T}\mathbf{T}_6^{-1} \tag{2-28}$$

Here it should be mentioned that the inverse of a T-type matrix is a simple function of the original contents, namely

$$\mathbf{T}_i = \begin{bmatrix}\mathbf{R}_i & \mathbf{p}_i \\ 0 & 1\end{bmatrix} \Rightarrow \mathbf{T}^{-1} = \begin{bmatrix}\mathbf{R}_i^T & -\mathbf{R}_i^T\mathbf{p}_i \\ 0 & 1\end{bmatrix} = \begin{bmatrix} c_i & s_i & 0 & -a_i \\ -s_i\lambda_i & c_i\lambda_i & \mu_i & -d_i\mu_i \\ s_i\mu_i & -c_i\mu_i & \lambda_i & -d_i\lambda_i \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2-29}$$

where the last equality holds especially for the case when T is a link transformation matrix. Also note that the system (2-28) does not fully describe the system without the additional constraints

that each c_i and s_i (i=1:6) are cosine/ sines pairs of the same angle θ_i. This constraint can be equivalently expressed as

$$c_i^2 + s_i^2 = 1 \qquad i = 1 \rightarrow 6 \tag{2-30}$$

Now let

$$\mathbf{P} = \mathbf{T}_3\mathbf{T}_4\mathbf{T}_5 \qquad\qquad \mathbf{Q} = \mathbf{T}_2^{-1}\mathbf{T}_1^{-1}\mathbf{T}\mathbf{T}_6^{-1} \tag{2-31}$$

and consider the matrix equations

$$\mathbf{R} = \mathbf{P} - \mathbf{Q} = \mathbf{0} \qquad\qquad \mathbf{S} = \mathbf{P}^T\mathbf{P} - \mathbf{Q}^T\mathbf{Q} = \mathbf{0} \tag{2-32}$$

Morgan studied these equations using a symbolic algebra program and found that the degree of the various elements of the equations are

$$\deg(\mathbf{R}) = \begin{bmatrix} 3 & 3 & 3 & 3 \\ 3 & 3 & 3 & 2 \\ 3 & 3 & 2 & 2 \\ 0 & 0 & 0 & 0 \end{bmatrix} \quad \text{and} \quad \deg(\mathbf{S}) = \begin{bmatrix} 0 & 0 & 0 & 3 \\ 0 & 0 & 0 & 3 \\ 0 & 0 & 0 & 2 \\ 3 & 3 & 2 & 2 \end{bmatrix} \tag{2-33}$$

He also found that even though all the variables c_i and s_i (i=1:6) occur in *some* elements, a given element of R or S typically contains only a subset of all the variables. In fact, the following set of 8 equations (picked from the elements of R=0 and S=0) contain only the 8 unknowns (c_i, s_i, i=1,2,4,5):

$$\begin{gathered} \mathbf{R}_{33} = \mathbf{0} \qquad \mathbf{R}_{34} = \mathbf{0} \qquad \mathbf{S}_{43} = \mathbf{0} \qquad \mathbf{S}_{44} = \mathbf{0} \\ c_i^2 + s_i^2 = 1 \qquad i = 1, 2, 4, 5 \end{gathered} \tag{2-34}$$

The remaining unknowns (c_i, s_i, i=3,6) are determined by the other variables and can be computed in closed form ([tsai84] p14). The system (2-34) is a system of 8 polynomial equations in 8 unknowns, with each equation having terms of degree no higher than 2. In Chapter 5 it will be described how this system is solved using a numerical method. First, however, we are going to describe the general method that Morgan applied to this problem.

2.5 Systems of polynomial equations

A system of n polynomial equations in n unknowns is an equation of the form

$$\mathbf{f}(\mathbf{x}) = \mathbf{0} \qquad \text{where} \qquad \mathbf{f}: \mathbf{C}^n \rightarrow \mathbf{C}^n \tag{2-35}$$

and **f** is a polynomial function. In other words, **f** is a vector valued function of the vector **x**, with both **x** and **f** (**x**) being complex vectors with n variables. Moreover, each component f_i of **f** must have the form

$$f_i(x) = \sum_{t=1}^{t_i} a_{it} x_1^{m_{it1}} x_2^{m_{it2}} \ldots x_n^{m_{itn}} \tag{2-36}$$

By definition, the degree of each term of the polynomial f_i is defined as the sum of the exponents of all the variables occurring in the term. Similarly, the degree of each polynomial f_i is defined as the highest degree occurring among the terms:

$$\deg(f_i) = \max\{m_{it1} + m_{it2} + \ldots + m_{itn} \mid t \in (1 \ldots t_i)\} \tag{2-37}$$

The *total degree* of the *system* **f** (**x**) =0 is defined as the product of the degrees of the individual equations:

$$\deg(\mathbf{f}) = \deg(f_1)\deg(f_2)\ldots\deg(f_n) \tag{2-38}$$

Example

The system $\mathbf{f}: \mathbf{C}^2 \rightarrow \mathbf{C}^2$ given by

$$\begin{aligned} f_1(\mathbf{x}) &= a_{11}x_1^2 + a_{12}x_1x_2 + a_{13}x_2^2 + a_{14}x_1 + a_{15}x_2 + a_{16} = 0 \\ f_2(\mathbf{x}) &= a_{21}x_1^2 + a_{22}x_1x_2 + a_{23}x_2^2 + a_{24}x_1 + a_{25}x_2 + a_{26} = 0 \end{aligned} \tag{2-39}$$

is a simple example of a polynomial system. In the general case, there are n equations, n variables and an arbitrary number of terms of arbitrary degree in each equation. For example:

$$\begin{aligned} f_1(\mathbf{x}) &= a_{11}x_1^2x_3 + a_{12}x_2^6 + a_{13}x_2^5x_1 + a_{14}x_2^4 + a_{15}x_2^3 + a_{16}x_2^2 + a_{17}x_2 + a_{18} = 0 \\ f_2(\mathbf{x}) &= a_{21}x_1^2x_2 + a_{22}x_1^2x_3 + a_{23}x_1x_2 + a_{24}x_2^5 + a_{25}x_2^4 + a_{26}x_2^3 + a_{27}x_2^2 + a_{28}x_2 + a_{29} = 0 \\ f_3(\mathbf{x}) &= a_{11}x_1^2 + a_{12}x_1x_3 + a_{13}x_2 + a_{14} = 0 \end{aligned} \tag{2-40}$$

It is easy to imagine that solving such a system is a nontrivial task, considering that even one univariate polynomial equation cannot be solved (by radicals, in closed form) for degrees higher than 4 [fraleigh83].

2.6 Finding all solutions of a polynomial system

This section is based mainly on results by Morgan [tsai84][morgan87a][morgan87b]. The purpose of the section is to extract the relevant parts of Morgan's work, which has developed considerably over the last 10 years, and present the essential parts in an as simple manner as possible.

The task is the following: Given a polynomial system, we want to find **all** the solutions to the system. As we shall see, classical methods such as Newton ([dahlquist74] p249) or Elimination ([canny88] Chap.3) cannot guarantee to find all solutions (Newton) or are numerically very tricky (Elimination). Before we go any further, some theory is required about how many roots a polynomial can have.

Definition 2.1 [solutions at infinity]: Given a polynomial **f**, construct a polynomial **F** by setting the coefficients of all lower degree terms to zero. Any nonzero solutions of $\mathbf{F} = \mathbf{0}$ are called *solutions at infinity* for the original polynomial. ■

For example, the solutions at infinity of the example (2-39) above are given by the system

$$\begin{aligned} F_1(\mathbf{x}) &= a_{11}x_1^2 + a_{12}x_1x_2 + a_{13}x_2^2 = 0 \\ F_2(\mathbf{x}) &= a_{21}x_1^2 + a_{21}x_1x_2 + a_{31}x_2^2 = 0 \end{aligned} \tag{2-41}$$

and for the example (2-40) they are given by

$$\begin{aligned} F_1(\mathbf{x}) &= a_{12}x_2^6 = 0 \\ F_2(\mathbf{x}) &= a_{24}x_2^5 = 0 \\ F_3(\mathbf{x}) &= a_{11}x_1^2 + a_{12}x_1x_3 = 0 \end{aligned} \tag{2-42}$$

The importance of the concept of solutions at infinity is clear from the following theorem:

Theorem 2.1 [Bezout]: Let **f** be a polynomial system of total degree d. Then:

- 1. The total number of geometrically isolated solutions and solutions at infinity of **f = 0** is no more than d.
- 2. If **f = 0** has neither an infinite number of solutions nor an infinite number of solutions at infinity, then it has exactly d solutions and solutions at infinity altogether, counting multiplicities. ■

This theorem is one explanation why Newton's method is not a reliable algorithm for solving polynomial equations: We generally do not know how many solutions there are unless we know the number of solutions at infinity, and to find out we have to solve an equation that is just as complicated as the one we started out with. We may find *some* solutions using Newton's method and random starting points, but we don't generally know *when* and *if* we have found all the solutions. The method of resultants is numerically very hard because it typically requires an extreme amount of precision, say 100-200 decimal digits. This makes it hard to implement in an efficient manner. In 1977, Garcia and Zangwill [garcia77], and Drexler [drexler77] independently developed a theory that opened the way for a new method for solving polynomial equations using the method of homotopy continuation. The method has since been refined by Chow, Mallet-Paret and Yorke [chow78], and expanded substantially by Morgan [morgan87a][morgan87b]. It has developed into the numerical algorithm that will be used as a design example in this work.

2.6.1 Homotopy continuation

The basic idea of the continuation method is the following: Suppose you want to solve the polynomial system **f(x)=0**. Assume that you have another polynomial system **g(x)=0** for which you know all the solutions, and that **g=0** has the same number of solutions as **f=0**. For example, one can use the system

$$g_i(\mathbf{x}) = p_i x_i^{d_i} + q_i \qquad d_i = \deg(f_i) \qquad i=1{:}n \tag{2-43}$$

where p_i, q_i are random complex numbers. This system is easy to solve by hand and has exactly

$d=d_1 d_2 \ldots d_n$ (nonsingular) solutions. It has no solutions at infinity. Now consider the system

$$\mathbf{h}(\mathbf{x}, t) = (1-t)\,\mathbf{g}(\mathbf{x}) + t \cdot \mathbf{f}(\mathbf{x}) \qquad t \in [0, 1] \qquad \text{(2-44)}$$

It is easy to see that $\mathbf{h}(\mathbf{x},0)=\mathbf{g}(\mathbf{x})$ and $\mathbf{h}(\mathbf{x},1)=\mathbf{f}(\mathbf{x})$. In fact, $\mathbf{h}(\mathbf{x},\bullet)$ is a family of functions that depends continuously on the parameter t. Such a family is called a **homotopy**. The system $\mathbf{g}=\mathbf{0}$ is referred to as the *start system*, and $\mathbf{f}=\mathbf{0}$ is referred to as the *target system*. The idea behind homotopy continuation is the following: Since we know the solutions of $\mathbf{g} = \mathbf{0}$, we can start at $t = 0$ with a solution of $\mathbf{g} = \mathbf{0}$ and gradually transform it into a solution of $\mathbf{f} = \mathbf{0}$ by increasing t in small steps and "dragging" the solutions with us by using a local numerical method (for example Newton's method). By trying all the roots of $\mathbf{g} = \mathbf{0}$ as starting points, we hope to be able to reach all the solutions of $\mathbf{f} = \mathbf{0}$. So far, this is of course only speculation, but it is confirmed by the following theorem, as worded by Morgan (p60). Let $\mathbf{p} = (p_i) \in \mathbf{C}^n$ and $\mathbf{q} = (q_i) \in \mathbf{C}^n$ denote the random parameters of $\mathbf{g}$.

Theorem 2.2 ([morgan87a] p60): Given $\mathbf{f}$, there are sets of measure zero, A_p and A_q, in $\mathbf{C}^n$ so that if $\mathbf{p} \notin A_p$ and $\mathbf{q} \notin A_q$, the following holds:

- 1. The solution set $\{ (\mathbf{x}, t) \in C \times [0, 1) \mid \mathbf{h}(\mathbf{x}, t) = \mathbf{0} \}$ is a collection of d non-overlapping smooth paths.
- 2. Each path extends from t=0 to t=1 without backtracking in t.
- 3. Each geometrically isolated solution of $\mathbf{f} = \mathbf{0}$ of multiplicity m has exactly m paths leading to it.
- 4. A continuation path can diverge to infinity only as $t \to 1$. If $\mathbf{f} = \mathbf{0}$ has no solutions at infinity, all the paths remain bounded. If $\mathbf{f} = \mathbf{0}$ has a solution at infinity, at least one path will diverge as $t \to 1$. Each geometrically isolated solution at infinity of multiplicity m will generate exactly m diverging continuation paths. ■

This is a very powerful result that allows us to find *all* solutions of polynomial systems. The process of computing a continuation path from t=0 to t=1 is referred to as *path tracking*. The method used is typically some variation on Newton's method, which we will now review for

completeness sake. Assume we are starting at t=0 with the solution x_0, We then want to make a small step to t=Δt and find the corresponding solution $x_1=x_0+\Delta x$, i.e. so that $\mathbf{h}(x_1, \Delta t)= \mathbf{0}$. By Taylor's theorem we have

$$\mathbf{0} = \mathbf{h}(x_0 + \Delta x, \Delta t) = \mathbf{h}(x_0, \Delta t) + \mathbf{Dh}_x(x_0, \Delta t)\Delta x + O(\|\Delta x\|^2) \quad (2\text{-}45)$$

This leads naturally to Newton's method, which consists of ignoring the higher order terms (on the assumption that a small Δt results in a small Δx). In other words, we pretend that the approximation

$$\mathbf{0} \approx \mathbf{h}(x_0, \Delta t) + \mathbf{Dh}_x(x_0, \Delta t)\Delta x \quad (2\text{-}46)$$

is exact and use it to solve for Δx. Then we update x with Δx and repeat the procedure (with t=Δt fixed) until Δx (known as the **residual**) becomes ignorably small. We have then found x_1 and are ready to start a new round of iterations with t=2Δt. There are many small variations to Newton's method, such as using more sophisticated predictions of the next point. It is also common practise to use an adaptive stepsize that adjusts Δt whenever a step fails to converge. For the sake of our discussion there is no need to go into further details. It suffices to note that the method of continuation boils down to repeatedly executing the following steps:

- Evaluate the function **h**
- Evaluate the Jacobian $\mathbf{Dh}_x$
- Solve the nxn linear system $\mathbf{Dh}_x(x_0, \Delta t)\Delta x = -h(x_0, \Delta t)$

This list will be used later as the basis for our development of a processor architecture.

2.6.2 Problems with the continuation method

The continuation method is not without its share of problems. We shall now identify some of the more common problems and describe how to handle them. Morgan [morgan87a] has described measures for neutralizing all the common problems, and it is my experience that these measures are very efficient in practice. Let us start with a list of real and potential problems:

- path crossings (singularity for $t=t_0<1$)
- solutions at infinity for $t=t_0<1$ (paths diverging in the middle)
- singular solutions at $t=1$
- solutions at infinity for $t=1$ (paths diverging towards the end)
- too many paths to track
- ill-conditioned systems (badly scaled systems)

Path crossings will occur if, for some $0<t_0<1$, the system $\mathbf{h}(\mathbf{x}, t_0)= \mathbf{0}$ has a singular solution (also known as a multiple root). This case is already covered by Theorem 2.2. There is of course the practical problem that paths may get so close to each other that the tracker may jump from one path onto the other due to numerical error. This *can* happen but has not been a problem in my experience. Likewise, Theorem 2.1 guarantees that no path will diverge at any $0<t_0<1$.

Singular path endpoints is a problem that cannot be avoided, but is not necessarily a big problem. The main implication is that Newton's method tends to converge slowly when the solution point is singular (because the Jacobian becomes almost singular when you get close to the point). This results in linear instead of quadratic convergence towards the point. While not critical, this may lead to reduced accuracy of the obtained solution. One common way of dealing with singular roots is to do extra iterations at the endpoint.

Paths diverging when $t \rightarrow 1$ is serious problem. The reason is that it is hard to decide whether a path is really divergent (in which case we may as well abandon it), or will turn back and converge when $t \rightarrow 1$. What happens is usually that the stepsize must be decreased repeatedly, and that we

eventually reach the minimum stepsize (MINSTEP) or the limit on the number of steps (MAXNS) allotted. Later on we will describe how an arbitrary system can be transformed using the *projective transform*. This will prevent solutions at infinity altogether.

Another common problem is that the system we want to solve has a high total degree. For example, the IPO system in its crudest form has degree d=2073600. This is clearly too many paths to track, especially considering that the problem has at most 16 physical solutions [primrose86]. The IPO system was rearranged by Tsai and Morgan [tsai84] to yield a degree of 256. This is still somewhat high. Morgan and Sommese [morgan87b] has since developed a so-called 2-homogenous form of the equations, reducing the degree to 96. They have also developed a theory called *the method of the generic case*, which makes it possible to eliminate 32 of the 96 paths, leaving us with 64 paths to track. The method of the generic case is a theory about the structure of the solutions of the polynomial equations. The random starting system **g** is replaced by a special (and less random) starting system that has the same "structure" (in a technical sense) as the target system. Morgan has shown that if **g** is chosen so that its solution set has "generic" structure, we can compute once and for all which starting points lead to singular solutions at infinity. These paths will lead to singular solutions at infinity for *any* IPO-type target system, and these starting points can hence be excluded from our computation. Finally, Wampler and Morgan [wampler89] have developed an 11×11 system of degree 1024 for which only 16 paths have to be tracked, again due to the method of the generic case. However, we do not make use of this result here.

The final item on our list of problems is ill-conditioned systems. This refers to systems that have coefficients and/or solution components that are spread out over a wide range, from very small (but nonzero) to very large. Such systems can sometimes be improved by equation scaling and/or variable scaling. As it turns out, the IPO system is typically well conditioned, so we will not discuss this problem any further here.

2.6.3 Non-problems with the continuation method

Morgan also mentions several examples of pathological path behavior that sometime occur in general homotopies, but not in polynomial homotopies. They are:

- paths that end at some $0<t_0<1$.
- paths that remain bounded but never reach t=1 (e.g. a spiral).
- paths that branch out into several paths or into a surface.
- paths that turn back in t (become multiple-valued).

Theorem 2.1 guarantees that none of the above will happen for a polynomial continuation of the form described.

2.6.4 Homogenization

A polynomial is called **homogenous** if all the terms have the same degree. It is easy to see that

$$\mathbf{f} \text{ is homogenous and } \mathbf{f}(\mathbf{x}) = \mathbf{0} \quad \Rightarrow \quad \forall r \in C \quad \mathbf{f}(r\mathbf{x}) = \mathbf{0} \tag{2-47}$$

Given a polynomial system f, it can be homogenized into a new system $\hat{\mathbf{f}}$ as follows:

Definition 2.2 [homogenization] Let $\mathbf{f}: C^n \to C^n$ be a polynomial system. The *homogenization* $\hat{\mathbf{f}}: C^{n+1} \to C^n$ of f is defined by the relations

$$\hat{\mathbf{f}}(y, y_{n+1}) = y_{n+1}^{d_i} f_i\left(\frac{y_1}{y_{n+1}}, \frac{y_2}{y_{n+1}}, \ldots, \frac{y_n}{y_{n+1}}\right) \tag{2-48}$$

This definition is just a precise way of saying that each term of f_i is multiplied by the appropriate power of y_{n+1} so that all terms get degree d_i. Before we go on, remember that the solutions of a homogenous system are unique only up to a factor. Let us now look at correspondence between solutions of $\mathbf{f} = \mathbf{0}$ and the homogenization $\hat{\mathbf{f}}$. The solutions of $\mathbf{f} = \mathbf{0}$ determine a subset of the solutions of $\hat{\mathbf{f}} = \mathbf{0}$: Since $\hat{\mathbf{f}}(y, 1) = \mathbf{f}(y)$ it is clear that if y is a solution of $\mathbf{f} = \mathbf{0}$, then (y,1) is a solution of $\hat{\mathbf{f}} = \mathbf{0}$. On the other hand, the solutions of $\hat{\mathbf{f}} = \mathbf{0}$ determine both the solutions of $\mathbf{f} = \mathbf{0}$ *and* the solutions of $\mathbf{f} = \mathbf{0}$ *at infinity*, in the following way: Suppose we have a (nonzero) solution of

$\hat{\mathbf{f}} = \mathbf{0}$ with $y_{n+1} = 0$. Then this solution is a solution at infinity of **f**, because

$$\mathbf{0} = \hat{\mathbf{f}}(\mathbf{y}, 0) = \mathbf{F}(\mathbf{y}) \tag{2-49}$$

In fact, $\hat{\mathbf{f}}(\mathbf{y}, 0)$ is exactly **f** with all the lower degree terms set to zero (by the effect of $y_{n+1} = 0$), i.e. the system **F** that defines the solutions at infinity for **f**. On the other hand, suppose $\hat{\mathbf{f}}(\mathbf{y}, y_{n+1})$, with $y_{n+1} \neq 0$. Then obviously from (2-48),

$$f_i\left(\frac{y_1}{y_{n+1}}, \frac{y_2}{y_{n+1}}, \ldots, \frac{y_n}{y_{n+1}}\right) = 0 \tag{2-50}$$

so that $\mathbf{x} = \mathbf{y}/y_{n+1}$ is a solution of $\mathbf{f} = \mathbf{0}$. In summary, we can recover both the solutions and the solutions at infinity from the solutions of $\hat{\mathbf{f}} = \mathbf{0}$.

Since $\hat{\mathbf{f}} = \mathbf{0}$ itself does not have any (additional) solutions at infinity (no homogenous system does), it seems that it may be easier to solve $\hat{\mathbf{f}} = \mathbf{0}$ than to solve $\mathbf{f} = \mathbf{0}$. However, there is the uniqueness problem: The solutions of $\hat{\mathbf{f}} = \mathbf{0}$ are unique only up to a factor, i.e. they are lines in C^{n+1}. This also means that they are singular and hard to find. In fact, one would expect to wander randomly along the solution line as one gets closer to it. Not to mention the fact that the Jacobian is non-square ($n \times (n+1)$), which causes an under-determined linear system to appear in Newton's method.

2.6.5 The projective transform

The solution to this problem is the projective transform (p-transform). The projective transform of $\hat{\mathbf{f}} = \mathbf{0}$ is a new system derived from $\hat{\mathbf{f}} = \mathbf{0}$ by adding the extra constraint

$$y_{n+1} = L(\mathbf{y}) = \alpha^T \mathbf{y} + \beta = \sum_{i=1}^{n} \alpha_i y_i + \beta \tag{2-51}$$

where $\alpha \in C^n$ and $\beta \in C$ are random constants. By abuse of notation we will use $\hat{\mathbf{f}}(\mathbf{y}, L(\mathbf{y}))$ to denote the projective transform of $\hat{\mathbf{f}}$. We will also use the notation $\hat{\mathbf{f}}^L$. It should be noted that the projective transform is no longer a homogenous system (there will be lower degree terms when we

substitute the affine expression for y_{n+1} into $\hat{\mathbf{f}}$. This means that we may potentially have another system that has solutions at infinity, which would bring us back to square one. However, we have the following theorem:

Theorem 2.3 [morgan87a], p56): Assume that f =0 has neither an infinite number of solutions nor an infinite number of solutions at infinity. Then for almost all $\alpha \in C^n$ and $\beta \in C$, the projective transform $\hat{\mathbf{f}}(\mathbf{y}, L(\mathbf{y}))$ has no solutions at infinity. ■

One important question remains to be answered: Can the solutions of **f** be recovered from the solutions of $\hat{\mathbf{f}}^L$ This is by no means obvious. It is clear that if $\hat{\mathbf{f}}(\mathbf{y}, L(\mathbf{y})) = \mathbf{0}$ then $\mathbf{x}=\mathbf{y}/y_{n+1}$ is a solution of $\mathbf{f} = \mathbf{0}$. But what about the other way around? That is, given that $\mathbf{f}(\mathbf{x})=\mathbf{0}$, does there exist a **y** such that $\mathbf{x} = \mathbf{y}/L(\mathbf{y})$? This was not proven by Morgan[1], but the following result should suffice:

Theorem 2.4 : Assume that $\alpha \in C^n$ and $\beta \in C$ are random constants. Then

$$\mathbf{f}(\mathbf{x}) = \mathbf{0} \quad \Rightarrow \quad \exists \mathbf{y} \text{ such that } \hat{\mathbf{f}}(\mathbf{y}, L(\mathbf{y})) = \mathbf{0} \text{ and } \mathbf{x} = \frac{\mathbf{y}}{L(\mathbf{y})} \tag{2-52}$$

with probability one. That is, the "bad" values of α and β constitute sets of measure 0.

Proof: We will try to construct **y** from **x** by solving the equation $\mathbf{x}=\mathbf{y}/L(\mathbf{y})$ or $\mathbf{y}=L(\mathbf{y})\mathbf{x}$. This equation expands into

$$\mathbf{y} = (\alpha^T \mathbf{y} + \beta)\mathbf{x} \tag{2-53}$$

which in turn can be written as

$$(I - \mathbf{x}\alpha^T)\mathbf{y} = \beta\mathbf{x} \tag{2-54}$$

When does this system have a solution? Consider the matrix $I - \mathbf{x}\alpha^T$. We apply the Sherman-Morrison formula ([luenberger84], p269)

---

1. Probably because he found it to be obvious.

$$[A + ab^T]^{-1} = A^{-1} - \frac{A^{-1}ab^TA^{-1}}{1 + b^TA^{-1}a} \quad (2\text{-}55)$$

which yields

$$[I - x\alpha^T]^{-1} = I + \frac{x\alpha^T}{1 - \alpha^T x} \quad (2\text{-}56)$$

This inverse exists as long as $1 - \alpha^T x \neq 0$, i.e. $\alpha^T x \neq 1$. For a given x, what is the probability that the random variable $\alpha^T x$ is equal to 1? If α consists of independent and identically distributed continuous random variables, then $\alpha^T x$ is another continuously distributed variable, and hence the probability that it takes on any particular value is zero.

It also follows immediately that if we consider a finite collection $x_1, \ldots, x_d$ of solutions to $f = 0$, the probability is zero that any product $\alpha^T x_i$ is 1. We conclude that the projective transform will produce all the solutions of $f = 0$, with probability one. ■

2.6.6 m-Homogenous systems

The theory of m-homogenous polynomials provides a way of reducing the number of continuation paths by exploiting special structure that may occur in some systems of polynomial equations. The IPO system can be cast as a 2-homogenous polynomial, and this reduces the number of paths to track from 256 to 96. Let us first define what an m-homogenous polynomial is.

Definition 2.3 [m-homogenous]: Let $f: C^p \rightarrow C^n$ be polynomial. Assume that the variables $z_1, z_2, \ldots, z_p$ can be grouped into m nonempty disjoint sets $Z_1, Z_2, \ldots, Z_m$ in such a way that (for all i=1:n)

$$f_i(Z_1, 1, 1, \ldots, 1) \quad (2\text{-}57)$$

$$f_i(1, Z_2, 1, \ldots, 1) \quad (2\text{-}58)$$

$$\ldots. \quad (2\text{-}59)$$

$$f_i(1, 1, 1, \ldots., Z_m) \quad (2\text{-}60)$$

are all homogenous polynomials. Then f is called an *m-homogenous* polynomial. ■

The definition means that if we consider one group Z_i of variables at a time, each of the polynomials f_i should be homogenous, considered as a function of only the variables in Z_i. Note that if a polynomial is m-homogenous it is also necessarily homogenous, which is again the same as 1-homogenous. The opposite is not generally true. The power of m-homogenous theory derives from a special version of Bezout's theorem that holds for m-homogenous polynomial systems.

Definition 2.4 [Bezout number]: Let $d_{ij} = \deg(f_i, Z_j)$ denote the degree of component f_i with respect to the variable set Z_j (i=1:n and j=1:m). Also let k_j denote the number of variables in the set Z_j, less 1. That is, the variables in the set are $z_{0j}, z_{1j}, \ldots, z_{k_j j}$, for a total of k_j+1 variables. Then the Bezout number *d* of **f** is defined as the coefficient in front of the term

$$\prod_{j=1}^{m} \alpha_j^{k_j} \tag{2-61}$$

where α_j is the placeholder variable in the combinatorial product

$$\prod_{i=1}^{n} \left(\sum_{j=1}^{m} d_{ij} \alpha_j \right) \tag{2-62}$$

■

The importance of the Bezout number stems from the Theorem 2.5 (below), which states that when **f** is m-homogenous, $\mathbf{f} = \mathbf{0}$ can have no more than *d* geometrically isolated solutions. However, since each solution is a line through the origin in C^p (cf. non-uniqueness for homogenous polynomials, discussed earlier), we need to be a little careful about how the theorem is stated. The machinery needed is called *projective space.*

Definition 2.5 [projective space]: The k-dimensional complex projective space P^k is defined as the set of lines through the origin in C^{k+1}. This is sometimes written

$$P^k = \{ [\mathbf{z}] \,|\, \mathbf{z} \in C^{k+1} - \{\mathbf{0}\} \} \tag{2-63}$$

where [**z**] denotes the line determined by **z** and the origin. ■

Since a line in C^{k+1} becomes a point in P^k, the solution lines in C^p will become solution points in

$$P^k = P^{k_1} \times P^{k_2} \times \ldots \times P^{k_m} \qquad (2\text{-}64)$$

and we can talk about unique solution points of an m-homogenous system, with the understanding that these are points in P. We then have the following theorem:

Theorem 2.5 [m-Bezout]: Let $\mathbf{f} = \mathbf{0}$ be an m-homogenous system, as defined above. Then $\mathbf{f} = \mathbf{0}$ has no more than d geometrically isolated solutions in P. If $\mathbf{f} = \mathbf{0}$ does not have an infinite number of solutions in P, then it has exactly d solutions, counting multiplicities. ■

This settles how many solutions the system may have. It remains to explain how this leads to a method that uses only d paths, where d is the m-homogenous Bezout number. Morgan ([morgan86], p5) presents the following result: Let g be a system chosen to have the same m-homogenous structure as f, except for the choice of the coefficients.

Theorem 2.6 : Let g be a system chosen to have the same m-homogenous structure as f, except for the choice of the coefficients. Assume that $\mathbf{g} = \mathbf{0}$ has exactly d (the Bezout number) nonsingular solutions, and define the homotopy

$$\mathbf{h}(z, t) = (1-t) \cdot \gamma \cdot \mathbf{g}(z) + t \cdot \mathbf{f}(z) \qquad \gamma \in C \qquad t \in [0, 1] \qquad (2\text{-}65)$$

Suppose $\gamma = re^{i\theta}$ for some real r>0. Then for all except a finite number of θ, the following holds: $\mathbf{h}^{-1}(\mathbf{0})$ consist of smooth paths over $C \times [0, 1)$, and every geometrically isolated solution of $\mathbf{f} = \mathbf{0}$ (in projective space P) has a path converging to it. In fact, if m_0 is the multiplicity of a geometrically isolated solution z_0, then there are exactly m_0 paths converging to z_0. Furthermore, the paths are all strictly increasing in t, and in fact dt/ds>0 on paths for $t \in [0, 1)$, where s denotes the arc length parameter of the path. ■

2.7 Summary

This chapter has introduced the IPO computation problem and its solution, both in closed form (for special cases) and in general using homotopy continuation, as outlined in Section 2.6.1. The IPO computation will be used, in various forms, as the main example in the remainder of this dissertation. IPO computations can range from fairly simple to quite complex, depending on whether one considers special cases or the general case. The next two chapters will describe the C-to-Silicon design system and show how the simple version of the IPO application was used as a the first design example for C-to-Silicon.

CHAPTER 3

C-TO-SILICON COMPILATION

C-to-Silicon is a system for designing *programmable* Application Specific Processors, either for DSP applications, or as will be demonstrated in Chapter 4, Numerical Processing applications. In particular, it is a system for designing an integrated circuit that will execute a given "C" language [kernighan78][harbison87][rimey89] program specified by the designer.

There is a variety of reasons why such a system is useful. First of all, it is important to note that in general it is more flexible and area efficient to design an ASIC which is programmable and uses a time-multiplexed datapath than to design a completely hardwired datapath for a given function. Some computations (especially of the numerical variety) also do not lend themselves easily to the hardwired datapath approach, as noted in Chapter 1.

Given that we want a programmable processor, it is possible to purchase standard off-the-shelf programmable DSP chips. These chips generally have high performance, at least for traditional DSP applications. The main incentive for building an ASIC programmable DSP/NP processor is that the design can be *optimized, minimal* and *embeddable.* Optimized means that the architecture

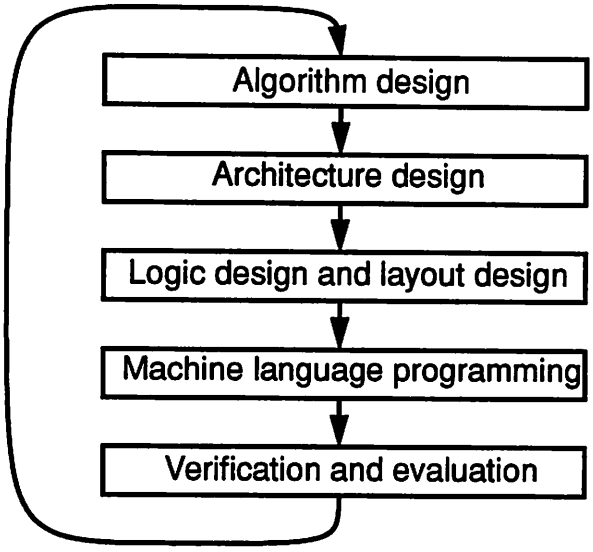


Figure 3-1 Design process for programmable Application Specific Processor

can be tailored to the task at hand. Minimal means the opportunity to design a bare-bones (lowest cost and power consumption) architecture that does not contain unnecessary or unusable features. Embeddable means that the processor can be placed on the same chip as other related circuitry, be it analog or digital system functions. This is important especially for portable systems, where a high level of integration is synonymous with light weight and low power consumption. Portable communication systems is one important example.

Some DSP (TMS320) and RISC (SPARC, MIPS) chips are being offered commercially as building blocks that can also be embedded with other circuitry. In both cases, but more pronounced for the RISC chips, there will be overhead and architectural mismatches. Memory management units and on-chip caches are prime examples of unwanted or unnecessary features. Data wordlengths and data types are examples of architectural mismatches.

3.1 Why C-to-Silicon compilation?

Having discussed the need for designing programmable ASPs, I would now like to turn the

attention to the need for a C-to-Silicon design system. Let us start out by taking a look at the steps involved in such a design (Figure 3-1). Independent of the level of automation, there are certain design steps that have to be carried out. The design goal is a processor that performs one particular computation, so the design starts at the algorithm level. Once a suitable algorithm has been chosen, an architecture has to be designed that can execute the algorithm efficiently. Once the architecture (and hence instruction set) has been determined, the next step is to go through the logic design and the layout design, and to code the algorithm in the machine language of the processor. Finally, the functionality of the complete design (both the software and the hardware) must be verified through simulation, and the performance assessed. The performance metrics normally are area, speed (cycle time), program code size, numerical soundness and total execution time (cycles).

The process just described is a long and complex one, and it has several problems. One obvious problem is the need for machine language programming. This step is particularly error-prone when the instruction set is new or under development. Also, it can be difficult to determine whether a problem is due to an error in the program code or an error in the architecture.

A major concern is that it is very hard to get accurate performance numbers for the design without finishing all steps of the design process: Program code size can be estimated at the architecture level, but only after writing out the code in considerable detail. Total execution time is very hard to estimate accurately except through simulation of the execution, either at the logic level or at the layout level. Speed and area can be estimated at a high level, but the estimates may not be accurate.

The implication of the above problems is that one must carry out all of the design steps in considerable or complete detail before being able to determine the quality of a particular algorithm/architecture combination. Since the design process is tedious and error prone, the designer can only afford one or a few design iterations. This means that *architecture exploration* is generally not possible. Exploration means to consider several major or (more often) minor architectural variations and accurately evaluate the costs and benefits of the variations.

3.2 Goals of the C-to-Silicon system

The goal of the C-to-Silicon system is to lessen some of the design burdens mentioned above, while at the same time providing an environment that supports easy architecture exploration. Specifically, the goals are to:

- Use a high-level "C" language for algorithm specification
- Allow architecture exploration *without* detailed hardware design
- Separate the hardware implementation from algorithm and hardware design
- Simplify concurrent design of hardware/architecture/software
- Eliminate machine language coding altogether
- Support simulation at all abstraction levels
- Provide accurate performance data without detailed hardware design

The system does not by itself provide area estimates or cycle time estimates unless the designer proceeds down to the layout level. Estimating the impact on area and speed from architecture changes is currently left to the expertise of the designer, but can conceivably be made part of the system in future versions. Area and speed estimates depend on the cell library used, and will require that each cell in the library has an associated area function and speed function that can compute the unknowns when given the cell parameters (such as wordlength or memory size and configuration). Also, the user would have to specify what hardware blocks (macrocells) to use, as there may be more than one version of each type of functional block. Using a default cell type for each type of function is another possibility.

3.3 The C-to-Silicon system

The remaining sections of this chapter describe the various features of the C-to-Silicon system and provide some details about how the features were implemented. The main features of the system are

- Retargetable compilation
- High-level simulation
- Architecture exploration

The logic and layout level design and simulation tools will also be described in some detail. Finally, there is a short summary of the syntax and semantics of the RL programming language (from [rb92]).

3.4 Retargetable compilation

By *compilation* is meant the process shown in the right half of Figure 3-2. The user (chip designer) provides an algorithm described in the RL language. RL [rimey89] is a subset of C extended with a fixed point datatype, and will be described in some more detail in section 3.14. The RL compiler[rimey89] compiles the program and generates *symbolic microcode*. The microcode is passed through the microcode assembler (**Mass**), which turns the microcode into layout parameters such as PLA contents and ROM contents. Finally, the LAGER Silicon Assembly system [rb92] is used to generate the physical layout of the chip.

By *retargetable* is meant the process shown in the left half of the figure, which indicates how the user provides a sequence of stepwise refined architecture descriptions. The RL compiler is retargetable to a wide variety of architectures by this process. The highest level architecture description is the *machine description file* (md-file). This file describes an architecture at the block-diagram level. The refinements of the md-file are the *microoperation file* (mo-file) and the hierarchical structure description, which is written in the SDL (structure description) language.

One major strength of the C-to-Silicon system is that the designer needs only write the (high-level) md-file in order to evaluate different architectures. Both program code size and total execution time (cycles) can be exactly predicted by providing the md-file for a given architecture. The next three sections will describe the md-file, the mo-file and the SDL file in more detail by means of some simple examples.

The machine description file

Figure 3-3 shows a simple address computation unit and the corresponding machine description file. The address unit can perform immediate, indirect and indexed addressing. Immediate addressing is performed by passing the value *addr* straight through the unit (addr is an immediate constant field in the instruction). Indirect addressing is performed by storing a memory address in one of the index registers (X[0-2]) and then combining it with a 0 immediate constant. Indirect addressing with a fixed index is also possible by using a non-zero immediate address. Finally, indexed addressing is available by using the immediate constant as a base address and one of the

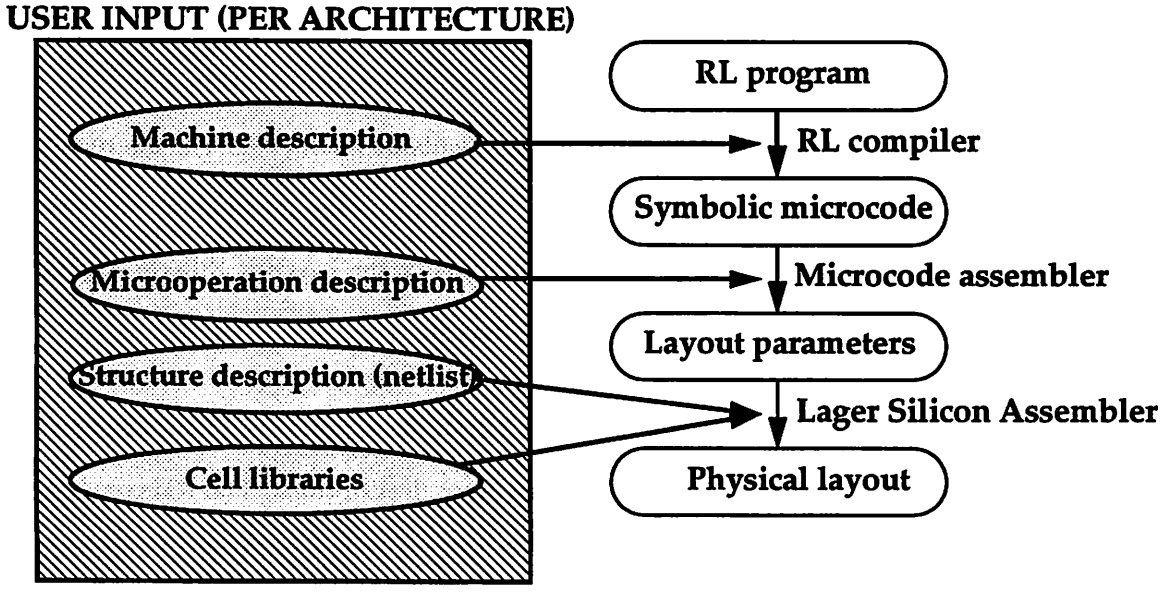


Figure 3-2 Retargetable C-to-Silicon compilation

index registers as the index. This is useful for array element access.

The corresponding machine description file consists of two main parts: The first part declares the *hardware resources*, meaning the buses (**bus**) and the storage elements (**file**). The second part defines the *instruction set* (or microoperation set) for the architecture, and also implicitly defines the interconnectivity of the functional blocks such as the adder and the multiplexer. Note the close correspondence between the textual description and the block diagram. In fact, we could say that the machine description file is nothing more than a textual version of the block diagram

The microoperation file

The microoperations defined by the md-file generally are executed by setting an appropriate group of control signals to some particular values. Figure 3-4 shows the microoperation file

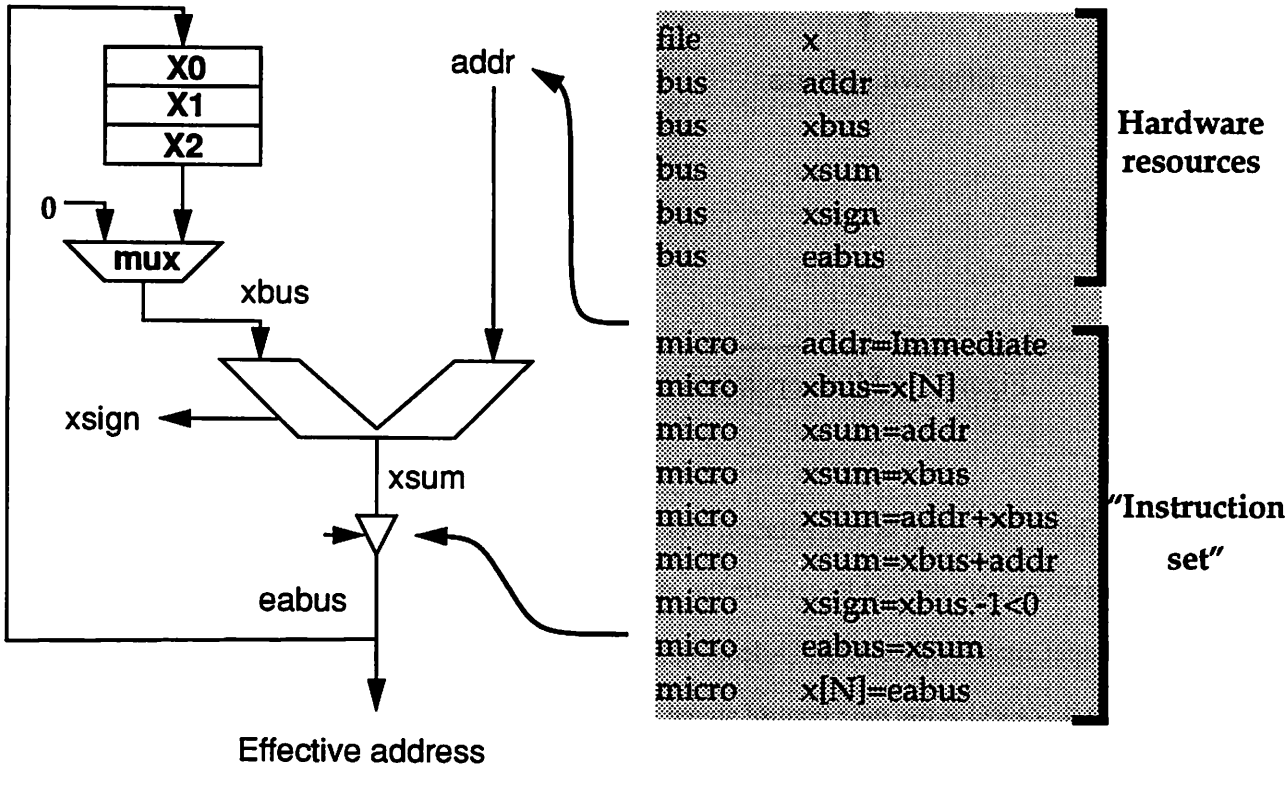
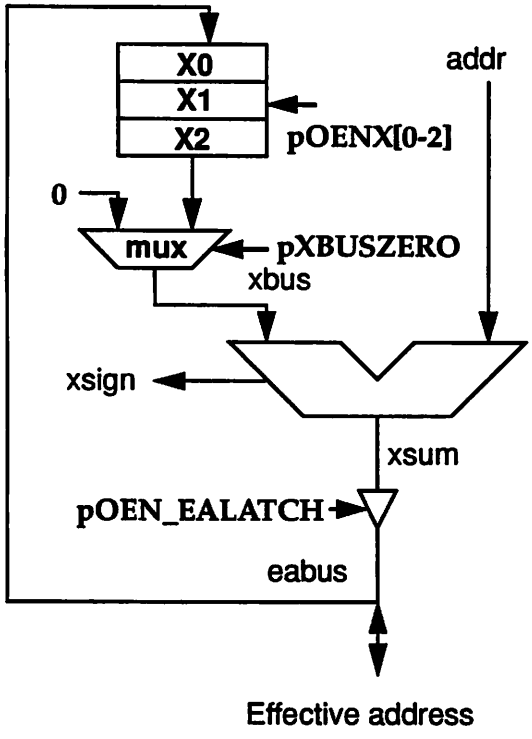


Figure 3-3 Machine description file for a simple address computation unit




```
file      x
bus       addr, xbus,xsum, xsign,eabus

micro     addr=Immed
          {addr=Immediate}
micro     xbus=x[N]
          {pOENX[N]=1,pXBUSZERO=0}
micro     xsum=addr
          pXBUSZERO=1}
micro     xsum=xbus
          {pXBUSZERO=0, addr=0}
micro     xsum=addr+xbus
          {pXBUSZERO=0 }
micro     xsum=xbus+addr
          {pXBUSZERO=0}
micro     xsign=xbus.-1<0
micro     eabus=xsum
          {pOEN_EALATCH=1,
          pMBUS2EABUS=0}
micro     x[N]=eabus
          {pLOADX[N]=1}
```

Figure 3-4 Microoperation file for address computation unit

corresponding to the address computation unit. The block diagram has been annotated with (some of) the control signals needed to execute the microoperations. For example, the signal pOENX[0-2] is an output enable signal which selects which one of the registers X[0-2] should be read. The signal pXBUSZERO is the mux control signal, but with a mnemonic name to indicate its function. The mo-file is just a refinement of the md-file, in that it lists for each operation also the control signals that need to be set to effect that operation.

Parameterized structure description

Once the designer has chosen an appropriate architecture, a structure description (netlist) of the architecture must be provided in order to generate the physical layout. The input format used by LAGER is SDL, or Structure Description Language. The complete design is typically described by a hierarchy of SDL files, where each SDL file declares subcells, their interconnections and what nets should be brought out to terminals. The SDL file is in turn used as a subcell at a higher level of the hierarchy. A block defined by an SDL file will also typically have *parameters* that affect the specific layout of the block. Examples of parameters are wordlengths, register bank sizes, ROM contents, PLA contents, conditional net flags and conditional subcell flags.

Figure 3-5 shows an example of how layout parameters can be used to control the layout of a specific block. The example used is again the address unit, where the wordlength and the number of registers can be controlled by parameters. The actual SDL file for the address unit is 201 lines

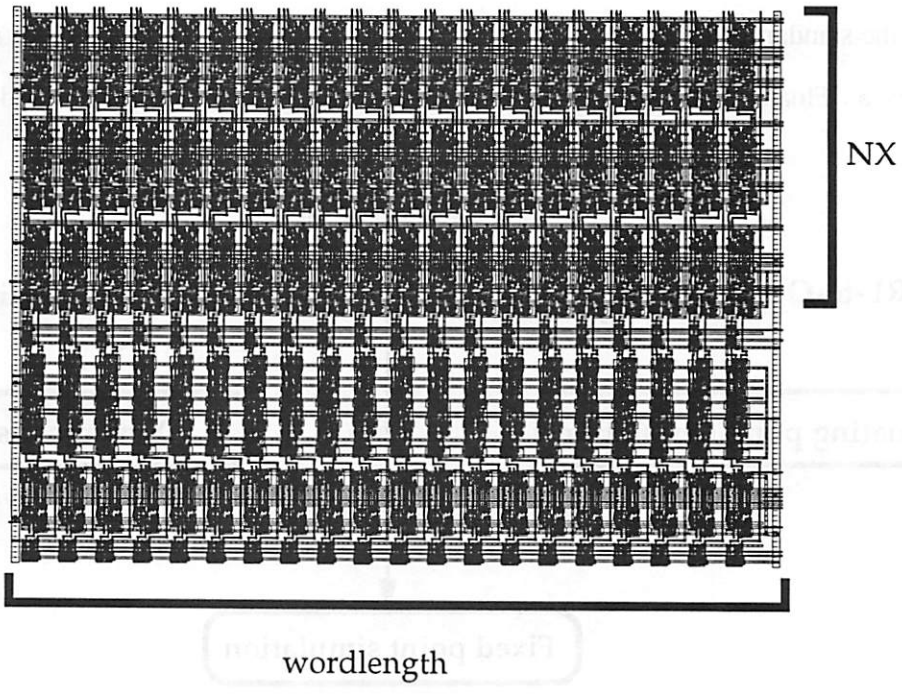


Figure 3-5 Examples of layout parameters

long. An excerpt of the SDL file is shown in Figure 3-6. The parameter N controls the wordlength and the parameter NX controls the number of address registers.

3.5 High-level simulation

It is important that a design system supports simulation at all levels of detail of the design. This section describes the high-level simulation support in the C-to-Silicon system

After creating an RL program for the desired algorithm, the user typically wants to check the basic soundness of the program by running it on some examples. Since there is no RL compiler targeting general purpose computers, and the Application Specific Processor has not yet been designed, the system provides for translating the RL code into standard C code (Figure 3-7). The program can then be compiled on a workstation and executed on some appropriate set of input data. Floating-point simulation is carried out by a straightforward translation of the RL program into C. The special functions `in()` and `out()` in the RL language are replaced by C routines that read/write using the standard i/o channels. The functions `in()` and `out()` are part of the standard library `libkt.a`. Floating-point simulation provides a first soundness check and is used to make certain

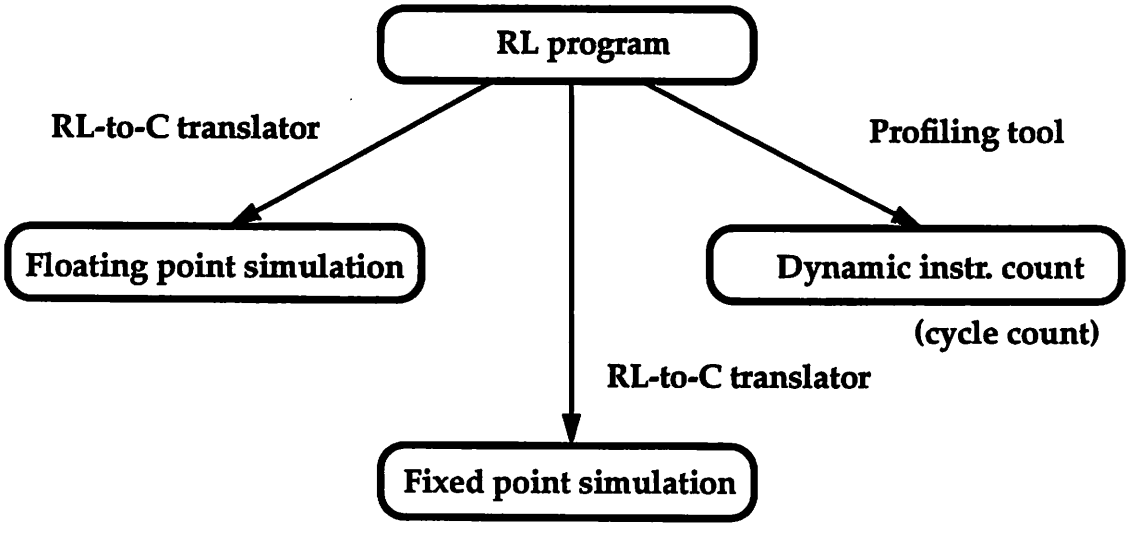


Figure 3-7 High-level simulation of algorithm and architecture

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Name : apudpM.sdl
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(parent-cell apudpM)
(layout-generator Flint a)
(structure-processor dpp)
(parameters N NX) ;;number of X registers

(dotimes (i NX)(subcells
(scanreg2Port REGX ((N N) ))))

(subcells
 (isozero ISOZERO ((N N)))
 (adder ADDER ((N N)))
 (scanlatch_ph1 EALATCH ((N N)))
 (trist_inverter EAGATE ((N N))))

;;; NETS
(net Vdd (NETTYPE SUPPLY)
 ((parent Vdd)(ISOZERO Vdd)(ADDER Vdd)(EALATCH Vdd)(EAGATE Vdd)))

(net GND (NETTYPE SUPPLY)
 ((parent GND)(ISOZERO GND)(ADDER GND)(EALATCH GND)(EAGATE GND)))

;;; DATA NETS
(instance REGX ((DATAIN eabus)(DATAOUT xbus1)(Vdd Vdd)(GND GND)))

(net xbus1((parent RegShiftamount) (ISOZERO IN)))
(net xbus2((ISOZERO OUT) (ADDER IN1)))

;;; CONTROL NETS
(instance REGX (
 (LOAD loadx (net-index i)); same i as in the dotimes
 (OEN oenx (net-index i))
 (SHIFTIN shiftin (net-index i))
 (SHIFTOUT shiftout (net-index i))
 (SCANIN scanchain (net-index i))
 (SCANOUT scanchain (net-index (1+ i))))

(instance parent (
 (LOADX loadx (width NX))
 (OENX oenx (width NX))
 (SHIFTIN shiftin (width NX))
 (SHIFTOUT shiftout (width NX))
 (SCANIN scanchain (net-index 0))
 (SCANOUT scanchain (net-index NX)))
;;; Lots of other stuff deleted ...
(end-sdl)
```

Figure 3-6 SDL file (main parts) for address datapath with variable number of registers

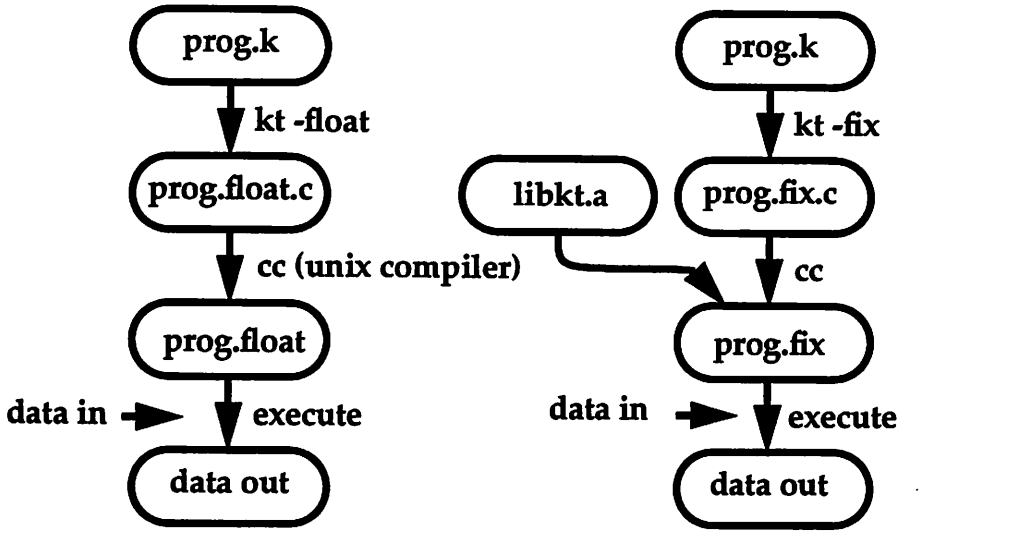


Figure 3-8 Implementation of floating- and fixed point simulation

that there are no serious mistakes in the RL program.

The C-to-Silicon system also provides the capability of fixed point simulation. For fixed point simulation, the fix variables are replaced by integer variables and the arithmetic operations are simulated on the integers by special fixed point library routines that replace the usual integer operators. Fixed point simulation is used to check the numerical soundness of the algorithm, including wordlength and scaling considerations.The floating- and fixed point simulation tools are implemented as shown in Figure 3-8. The fixed point routines in libkt.a are from [svensson90].

Finally, the C-to-Silicon system contains a *profiling* tool based on the standard UNIX profiling system. On a UNIX system, profiling is used to determine how much time is spent in each subroutine of a program (and how many times each subroutine is called). In C-to-Silicon, profiling means to count how many times each basic block of microcode is executed, thereby giving an exact execution cycle count for any given combination of architecture, algorithm (RL code) and input data. Profiling is a crucial part of the architecture exploration process described in the next section.

The profiling is implemented by inserting the special `MARK(blocknumber)` assembly code

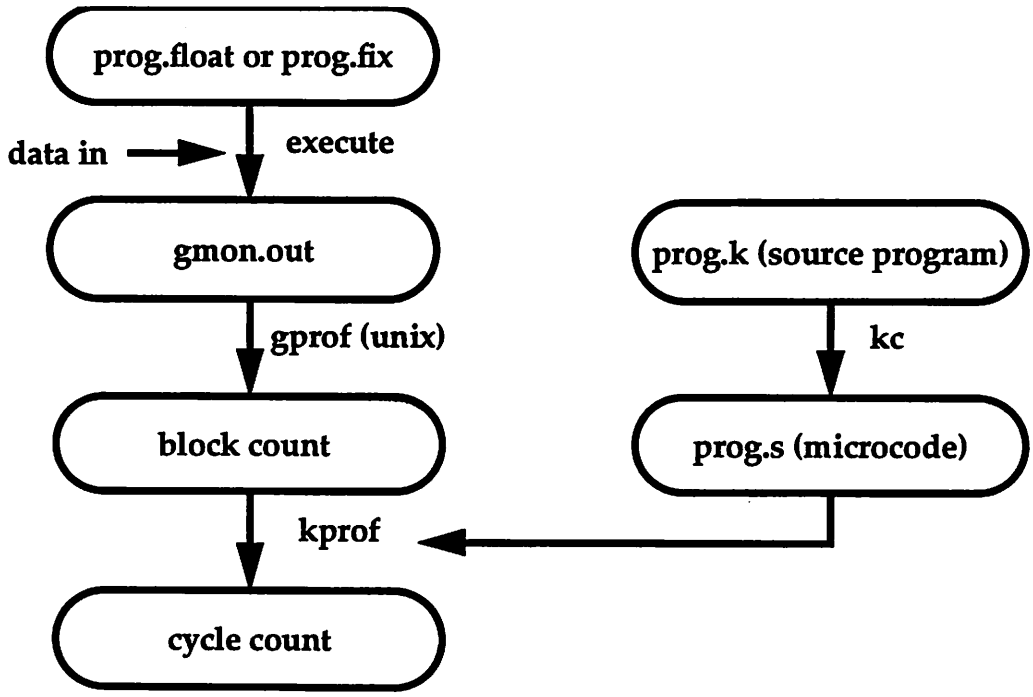


Figure 3-9 Implementation of profiling tool

macros in between the basic blocks, as they appear in the translated C program. Upon execution, the program will generate a file (called gmon.out) which will contain a count of how many times each block was executed. The profiler script will extract the block count from gmon.out and the block size from the symbolic microcode and compute the total number of instructions (cycles) executed.

3.6 Architecture exploration

By architecture exploration is meant the process shown in Figure 3-10. The designer can start out creating a brand new architecture and then write the corresponding machine description file. Or, more commonly, a suitable architecture already exists and the designer only needs to perform small modifications to an existing md-file. Once the md-file has been created, the algorithm (RL program) can be compiled onto the architecture, and the designer will immediately see the size of

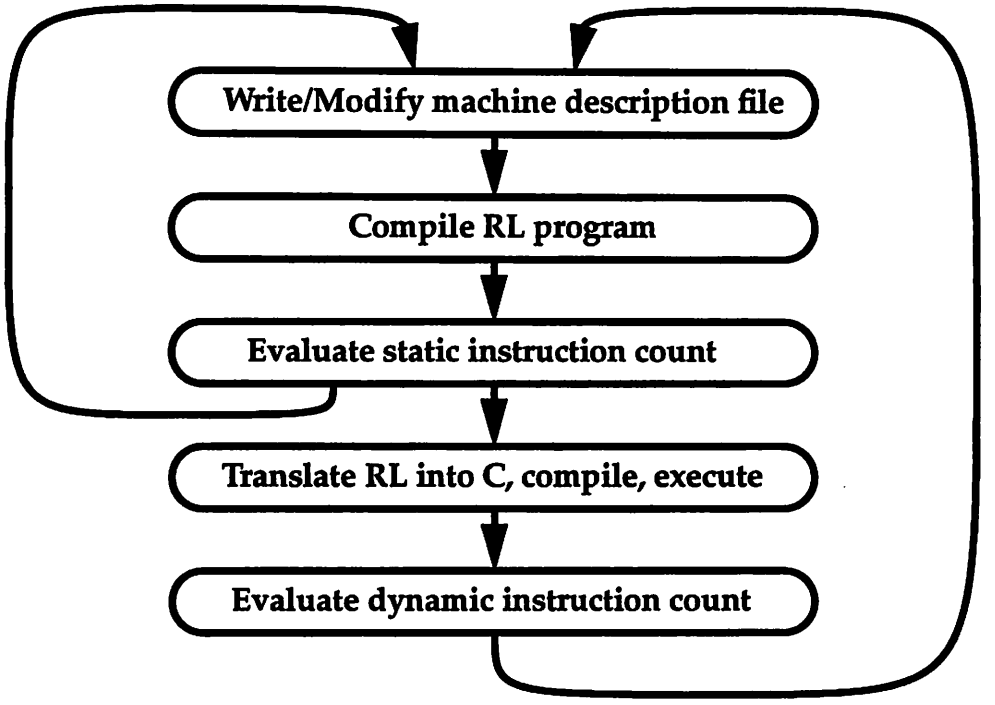


Figure 3-10 The architecture exploration process

the code (static instruction count). He can then repeat the process as often as necessary, or continue and obtain the cycle count using the profiling tool described in the previous section. The profiling information will likely lead to new design iterations, and the designer can "tune" the architecture until a satisfactory design has been found.

The process only involves writing and modifying machine description files, and there is no logic design or layout design involved. This means that the C-to-Silicon serves as a powerful tool for architecture exploration. Chapter 4 contains a fairly large example of the architecture exploration process, as it occurred during the development of the architecture for the PUMA chip.

3.7 Architecture examples

One important question regarding the C-to-Silicon system is what range of different architectures that the system can gracefully handle. The constraints are mainly within the RL compiler itself,

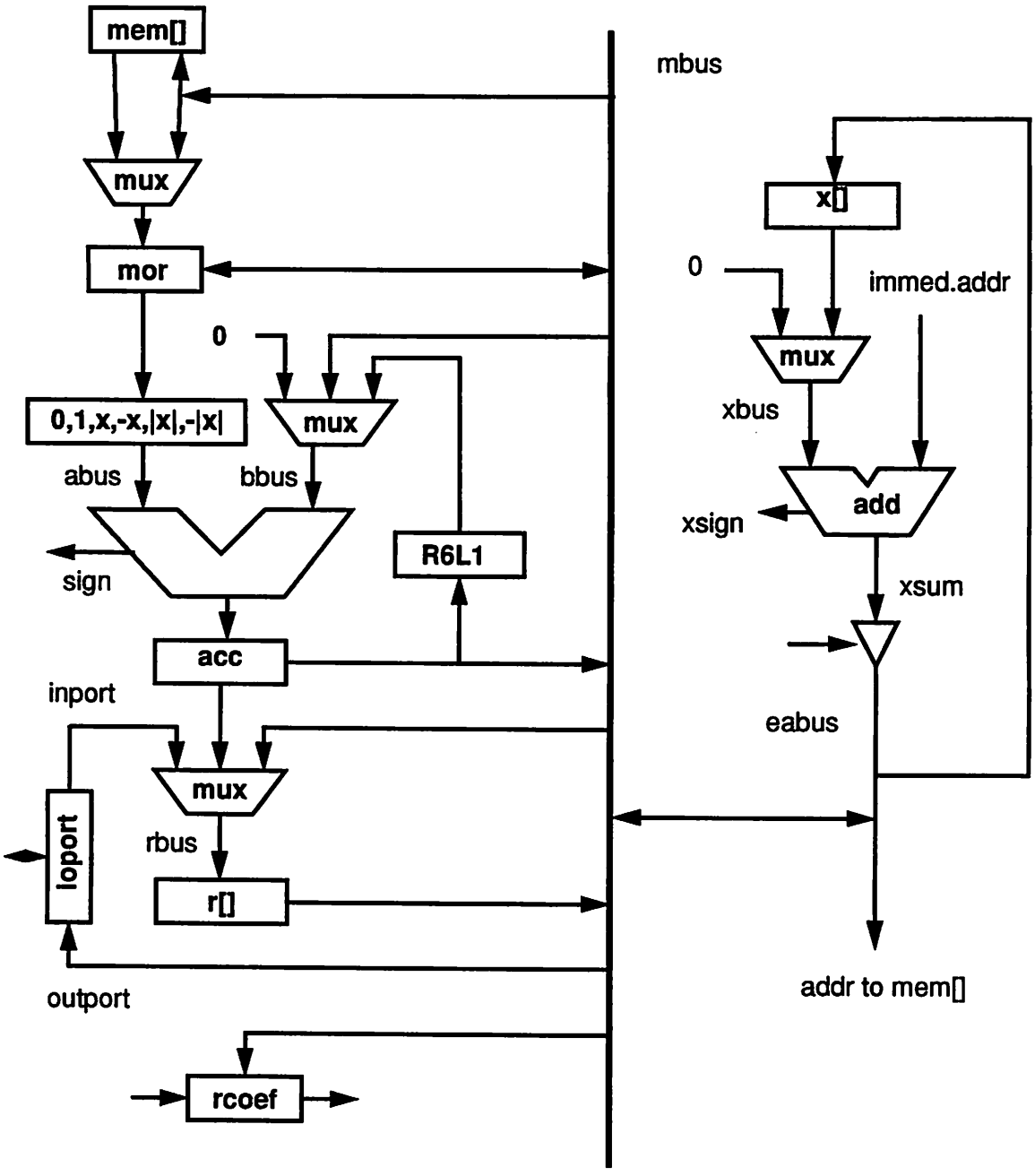


Figure 3-11 Example of an architecture suitable for the C-to-Silicon system

since the layout system can handle just about any architecture and the assembler is relatively easier to update for different architectures. Figure 3-11 shows one example of an architecture that is suitable for the system. This particular architecture (known as Kappa) [azim88] was used as an initial model of the types of architectures that C-to-Silicon should be able to handle. The characteristics of the architecture is that it contains a somewhat irregular datapath with several distributed storage locations. The Kappa architecture was based on earlier successful architectures [pope84][ruetz86] for speech and audio applications, and was therefore considered a good candidate for what the C-to-Silicon system should target.

As the C-to-Silicon system developed, additional architectures were designed and tested on the system. One example is the PUMA architecture described in Chapter 4. Another example was an architecture for a Decision Feedback Equalizer (DFE) for mobile radio [svensson90]. The DFE architecture is more complex than both the Kappa and PUMA architectures, in that it contains dual address units and two independent RAM blocks. The DFE datapath with dual memories is shown in Figure 3-12. This datapath is more regular than the PUMA datapath and quite complex. The address unit is shown in Figure 3-13. It is considerably more complex than the address units of Kappa or PUMA. It was found that the RL compiler had no problems compiling for the DFE architecture, indicating that the C-to-Silicon system is quite flexible with respect to the range of architectures allowed.

3.8 Execution model

By *execution model* we mean the basic method for instruction decoding and sequencing in a processor. The previous section showed that the C-to-Silicon system accepts a variety of datapath architectures. The execution model requirements are more strict, in that a certain basic functionality is always needed. This section is about the requirements on the execution model which the C-to-Silicon system (again, mostly the compiler) assumes. There are two main restrictions:

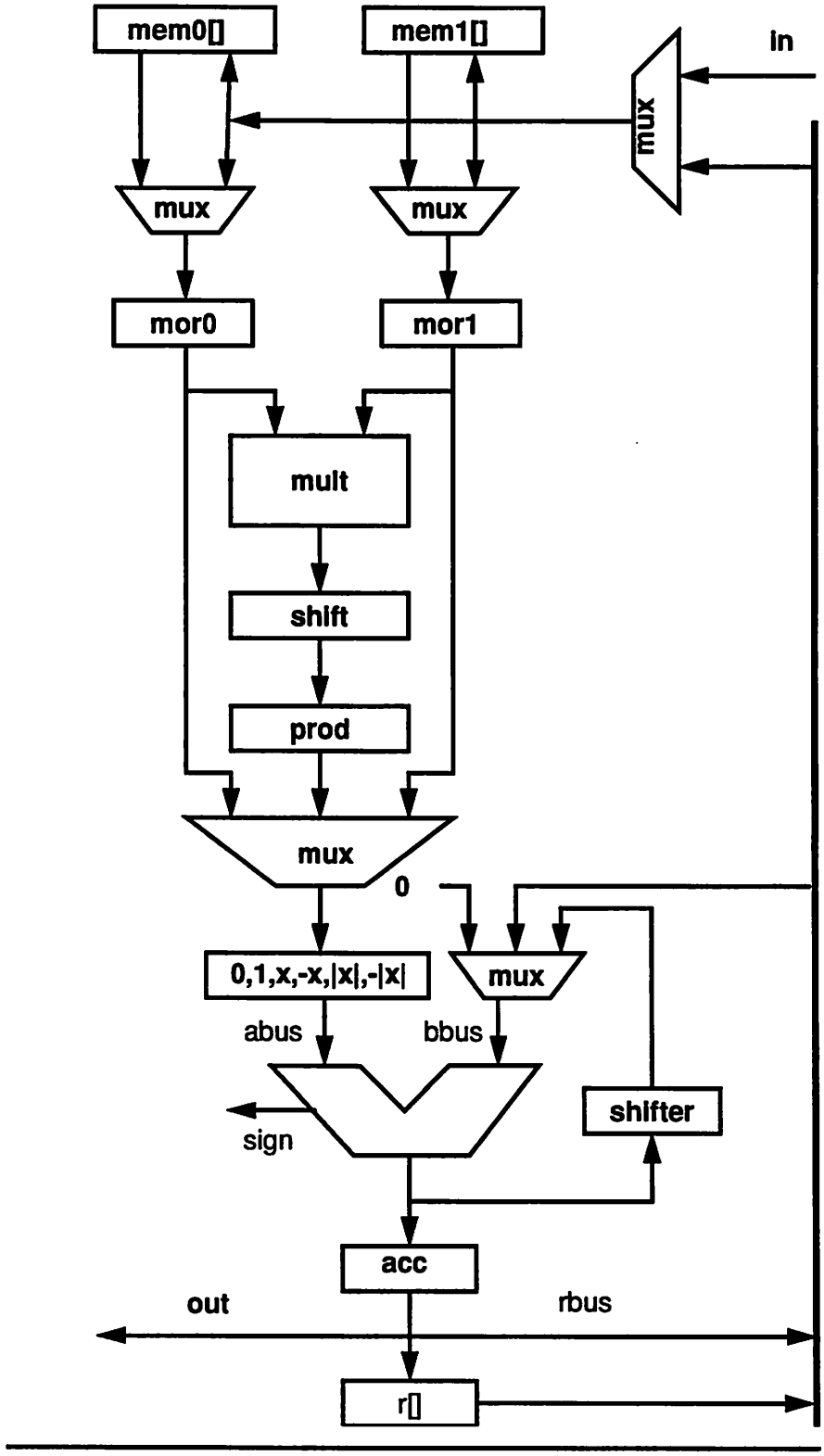


Figure 3-12 Datapath for Decision Feedback Equalizer [svensson90]

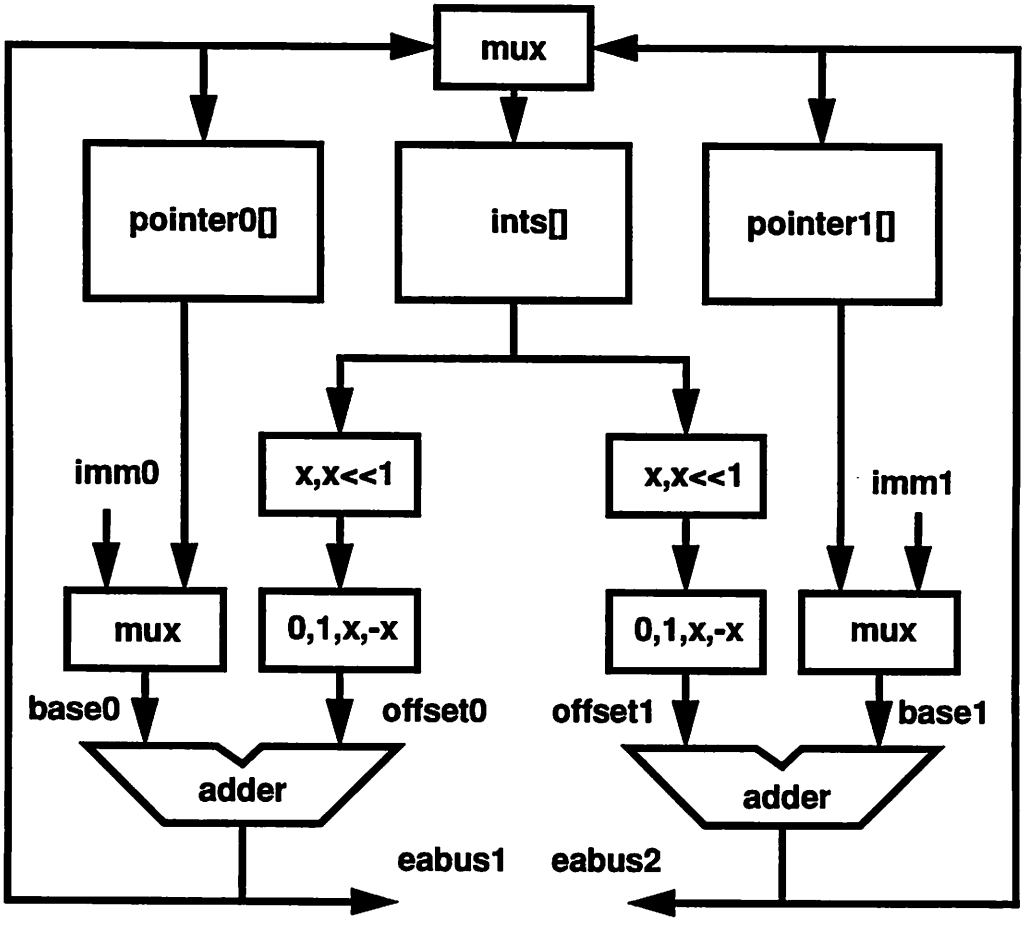


Figure 3-13 Address unit for Decision Feedback Equalizer [svensson90]

- The processor must be programmable in microcode with no or little restrictive encoding. This means that if the block diagram of the architecture indicates that two (or more) particular micro*operations* can be performed in parallel without resource conflicts, then there should be no encoding preventing both operations from being specified in the same micro*instruction* (remember that according to our terminology, a microinstructions consist of one or more micro-operations performed in parallel). This means that the processor will typically have a wide instruction word with separate and independent control bits for all (or most of) the hardware

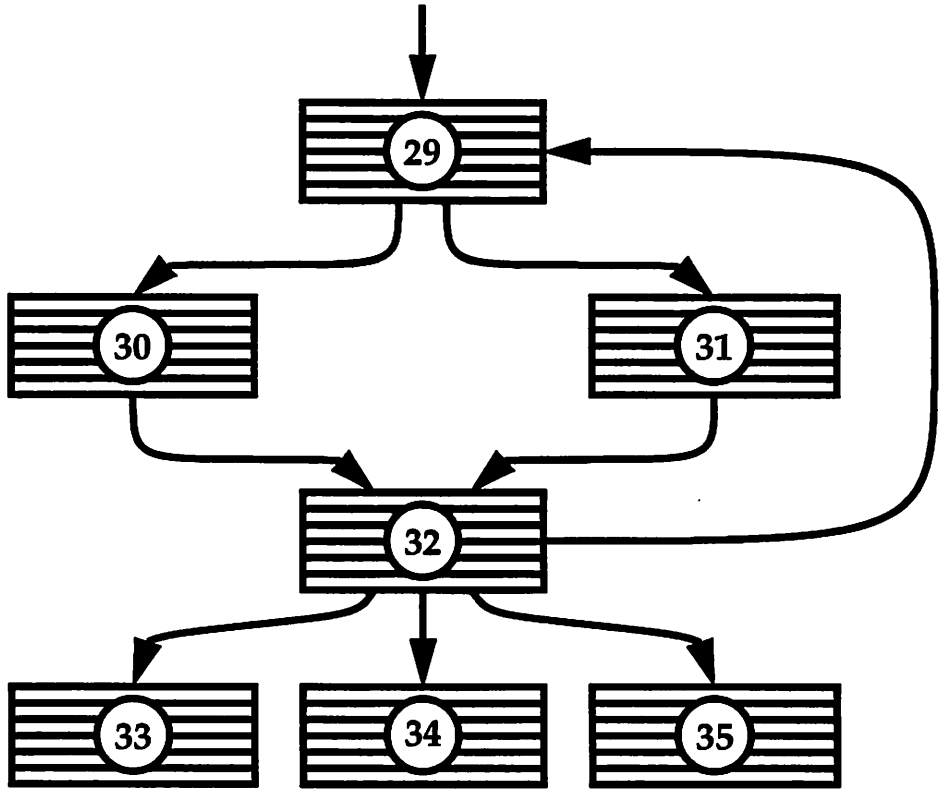


Figure 3-14 The execution model is based on straight-line blocks of code (numbered above) separated by arbitrary multiway branches

resources.

- The processor must be able to execute straight-line blocks of microcode. By a *straight-line block* is meant a collection of sequential *lines* of microinstructions. At the end of each block, the processor must be capable of executing an arbitrary m-way branch (conditional branch, goto, call or return) based on arbitrary combinations of boolean flags.
- Subroutine calls are allowed if the architecture provides a mechanism for storing and retrieving return addresses, but subroutines are not reentrant because the compiler does not assume the existence of a stack for storing local variables. Hence, nested subroutine calls are allowed as long as the same routine is not called more than once. In particular, recursive calls are not allowed.

Figure 3-14 illustrates the execution model. This particular example arises from the RL code

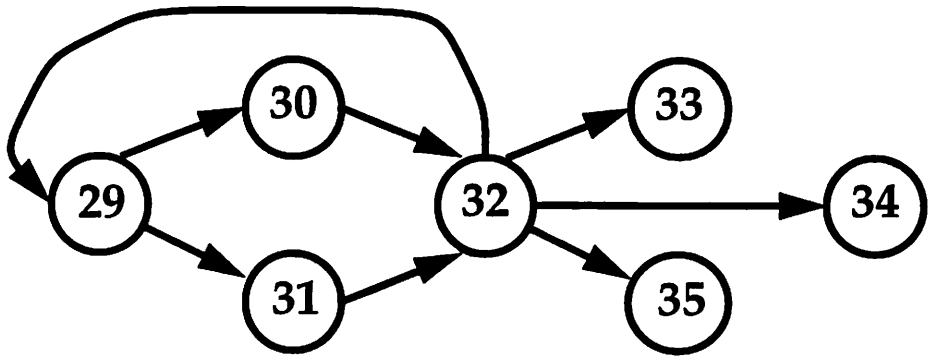
```
for (k=1; k<=NUMIT; k++) {
     /* Block 29 is here */
     if (y>0) {
          /* Block 30 is here */
     } else {
          /* Block 31 is here */
     }
     /* Block 32 is here */
}
if (smallflag) {
     /* Block 33 */
} else if (bigflag){
     /* Block 34 */
} else {
     /* Block 35 */
}
```

Figure 3-15 Code fragment corresponding to Figure 3-14

fragment shown in Figure 3-15. The code fragment is a **for**-loop followed by a 3-way **if**-statement. Moreover, there is also a 2-way **if**-statement inside the **for** loop. The RL compiler knows that multiway branching is available and creates a 4-way branch involving both the loop test and the 3 different cases of the **if** statement that follows the loop.

The contents of the various blocks are not completely shown in Figure 3-15, but roughly it is as follows: Block 29 computes the branching condition for the **if** (y>0) statement and branches accordingly to Block 30/31. Block 30/31 themselves contain some simple computations that are not shown. Block 32 increments the counter (k++), and then evaluates the loop condition (k<=NUMIT) and the 3 **if**-conditions in parallel, causing either a jump back to block 29 or one of the **if**-blocks 33,34,35.

The compiler generates BRANCH instructions for the processor as shown in Figure 3-16. It is up to the microcode assembler to translate these instructions into controller parameters. The boolean variable _BC is computed in an earlier statement and has the value _BC:=(NUMIT-k>=0).




```
BRANCH {
      _BC                                => GOTO 29;
      AND(!_BC,smallflag)                => GOTO 33;
      AND(!_BC,!smallflag,bigflag)       => GOTO 34;
      AND(!_BC,!smallflag,!bigflag)      => GOTO 35;
}
```

Figure 3-16 Branch instruction generated by the compiler at the end of Block 32

3.9 Controller structure

We have already mentioned the basic functionality required from the controller: It must contain a block sequencer which orchestrates the jumps between blocks, and it must contain a line sequencer (program counter) for linear sequencing inside each block. The block sequencer must support arbitrary multiway branches. There are many possible ways of implementing such a controller. Two examples will be described here.

Cathedral-II controller

This controller architecture was developed as part of the Cathedral-II [rabaey88] project. A simplified diagram of the controller is shown in Figure 3-17. This controller architecture is a possible candidate for the C-to-Silicon system. The multiway branches are implemented through a jump address table, and the "line" sequencing is performed with a simple increment-by-1 program counter. When a jump instruction occurs, the status logic is instructed to look up the new program

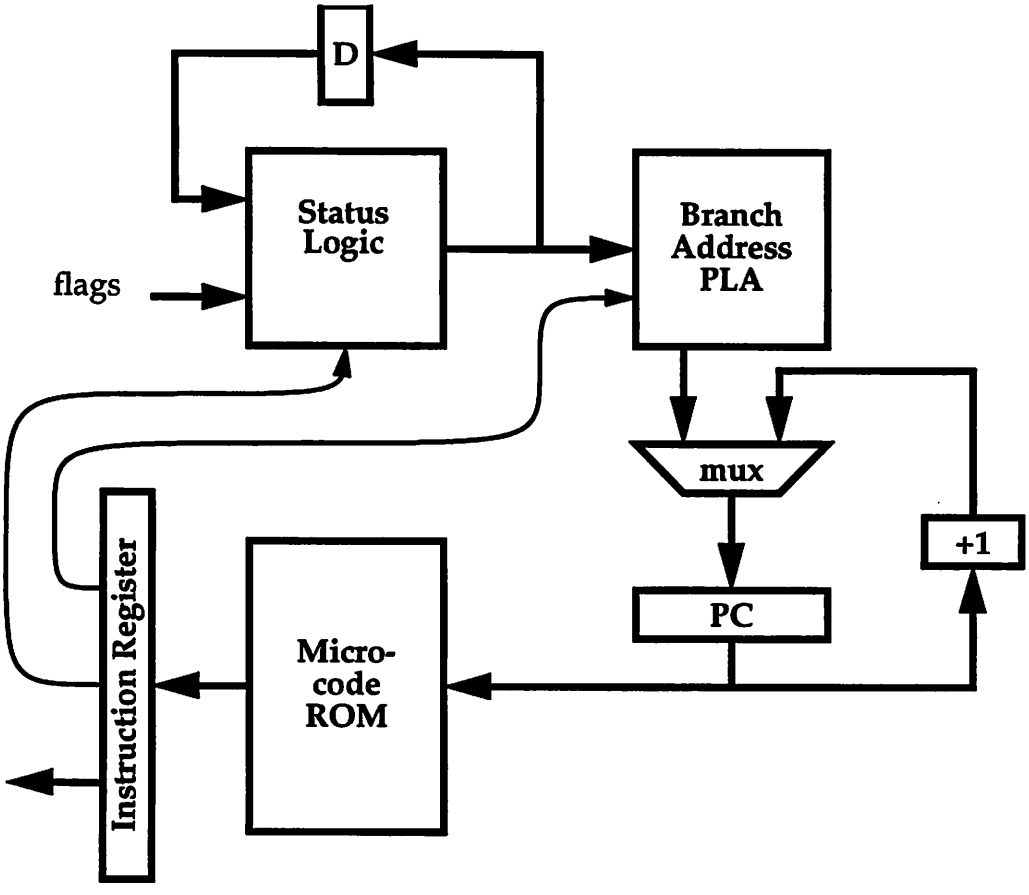


Figure 3-17 Controller architecture used in the Cathedral II system

address in the Branch Address PLA (which functions as a branch address memory), and the mux is used to load the new value into the Program Counter (PC).

Kappa controller

Another possible controller structure [azim88] is shown in Figure 3-18. The main parts of the controller is the block sequencer, the line sequencer and the microcode ROM. The block sequencer is a finite-state machine built around a PLA. Each state of the FSM corresponds to a basic block and outputs the appropriate basic block address. The value of the current state is fed back into the FSM as inputs along with the external inputs that constitute flag or status variables. Branching to a new block is triggered by the EOB (end-of-block) bit which comes from the microinstruction word (control word). The EOB bit is set only in the very last instruction (line) of a block. For each block,

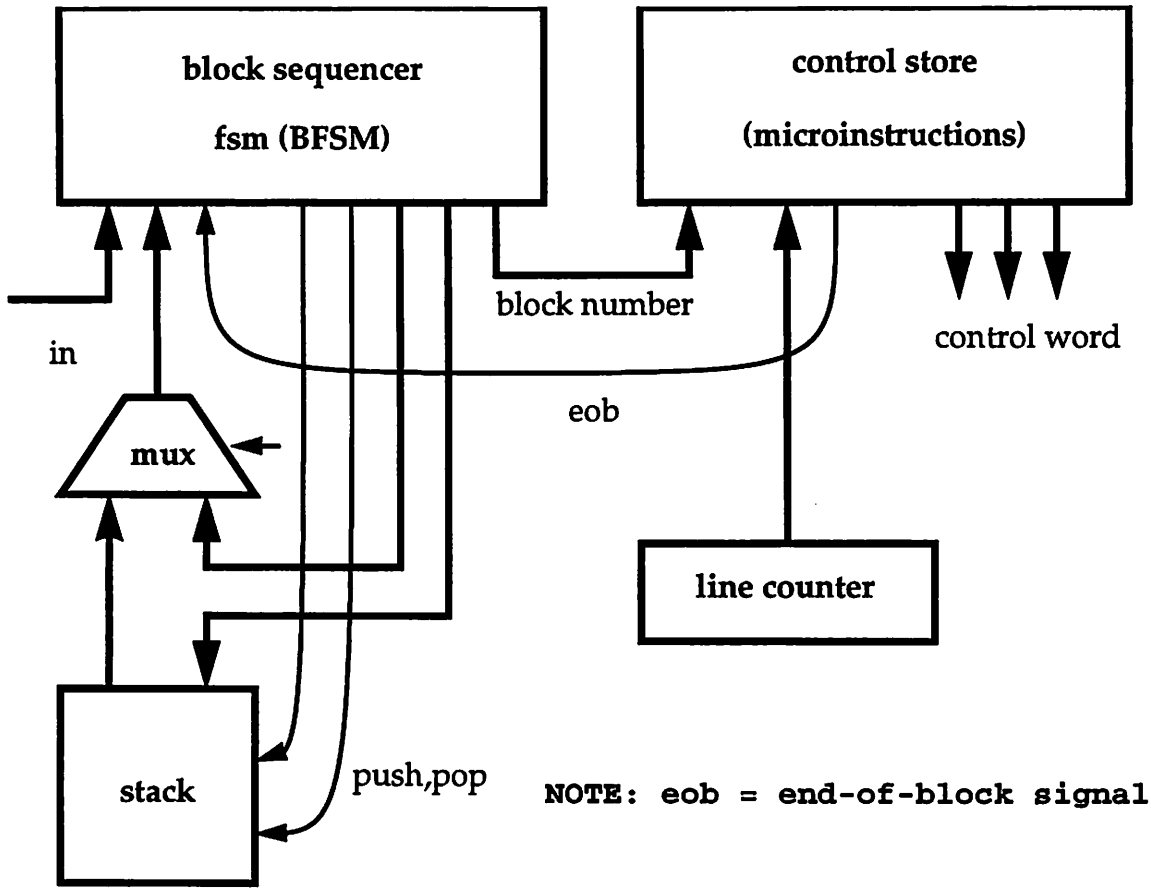


Figure 3-18 Kappa controller architecture

the block sequencer PLA contains one minterm that is sensitive to EOB=0 and keeps the FSM in the current state, so that the blockaddress remains the same. There are also one or more minterms that are sensitive to EOB=1, and hence get activated at the end of each block. Each such minterm corresponds to a particular jump target (next block or next state). The minterm is sensitized to the appropriate flag and status variables so that only one minterm[1] is activated by a given combination of current state and input. An m-way branch is implemented by having m minterms with different input conditions and different next state outputs. The controller also supports subroutine calls. The stack (Figure 3-18) stores arbitrary return addresses which can be popped on return from the

1. That is, before PLA optimization. After processing by Espresso, the minterms may no longer be mutually exclusive, but the overall functionality of course remains the same.

subroutine. The Kappa controller architecture was used, with some modifications, in the PUMA chip described in Chapter 4.

The original version of the controller from Kappa also contained a *timer* and a *loop counter*. The timer is used to cause a periodic restart of the program, as opposed to restarting the program using an external reset or handshake. The timer is useful when the program processes external data arriving at fixed sample intervals.

The loop counter was introduced to provide 0-overhead single-instruction loops, which are useful in some DSP applications. To use this feature it is necessary to program directly in machine language, as a high-level compilers generally are unable to determine whether or not such a special feature can be used on any given loop. This was not a problem before, as the Kappa architecture was programmed directly in machine language. The loop counter is not used in the C-to-Silicon system. The timer can be used if desired.

Comparison

The functional difference between the Cathedral-II and Kappa controllers is mainly that Kappa has a subroutine call/return capability. If the application requires subroutines, the Kappa controller is a natural choice. The Kappa controller also has the local advantage that it can be easily assembled from standard components in the LAGER cell library. It is therefore the controller of choice in the C-to-Silicon system. The Cathedral-II controller has a slightly more flexible branching and addressing scheme, in that it does not distinguish between block and line addresses. In Kappa, the complete instruction address is the *concatenation* of the block/line addresses. This means that there may be "holes" in the address space where there are no instructions, since not all blocks will have the same length. However, this is not really a problem because the ROM in LAGER uses a PLA-style address decoder, meaning that unused address locations can simply be left out from the layout. One idiosyncracy of the Kappa controller is that it actually uses two different output fields for the *next state* and the *block address*. The natural solution would be to let the state and the block address be one and the same.

Program name	Function
DMoct	LAGER/OCT Design Manager
dpp	Datapath module generator
TimLager	Tiled macrocell module generator
Stdcell	Standard cell place and route
Flint	Macrocell place and route
Padroute	Pad to core routing

Table 3-1 Main programs and design styles of the LAGER system

Svensson [svensson90] suggested a modification to the controller architecture, where an additional ROM (or PLA) is inserted between the stack and the BFSM. The idea is that only a subset of the blocks will be targets of a subroutine return. Hence, the addresses of the return targets can be encoded with fewer bits than a complete address (saving width in the BFSM minterms) and then later looked up in the PLA or ROM, either before or after being stored on the stack. Translation after being popped from the stack might cost a cycle and result in a performance penalty.

As used in Kappa, some of the status flags that are inputs to the BFSM come from a Logical Unit (LGU) which is used to compute (and store) boolean flags for use in branching decisions. Having a Boolean Memory is necessary if a branch variable is computed early and needs to be stored until reaching the branch instruction. Having a boolean computation capability is also useful, since it can be used to combine multiple status signals into a single flag, hence reducing the number of wires that must be fed into the BFSM.

3.10 Silicon Assembly with LAGER

The logic and layout level design in the C-to-Silicon system is performed within the framework of the LAGER Silicon Assembly System [rb92][shung91]. A detailed description of LAGER can be found in [rb92]. This section only aims to describe enough of LAGER to enable the reader to

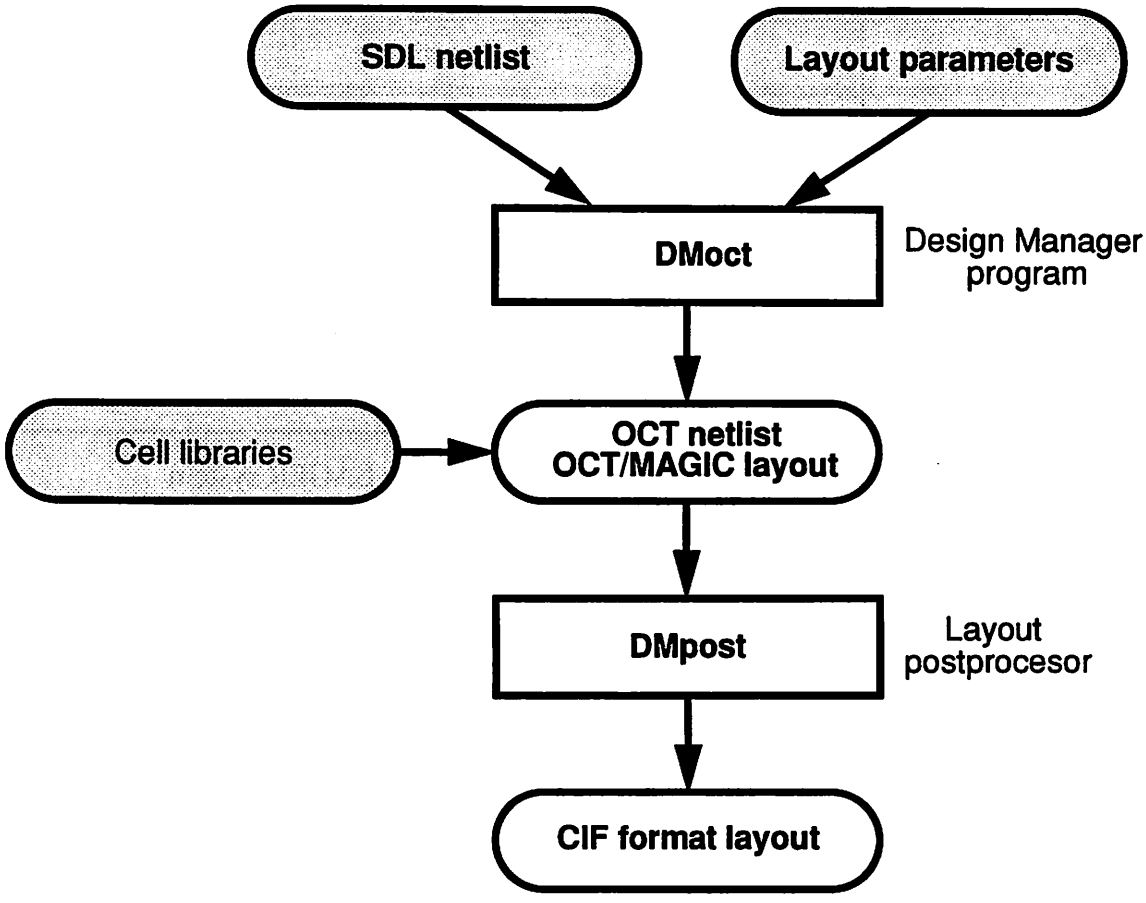


Figure 3-19 The chip design process in LAGER

understand the basics of the system and to understand how it is used as part of the C-to-Silicon system.

Overview of LAGER

LAGER is most easily understood by looking at the design process it implements. The basic function of LAGER is to create layout from a netlist (or schematic) provided by the user, making use of pre-designed leafcells and module generators. Several design styles are supported: Standard cells, Tiled Macrocells, Bit-Slice Datapaths and Macrocell Place-and-Route, as indicated in Table 3-1. These design styles can be mixed and matched freely in any particular design. The design process of LAGER is shown in Figure 3-19. The user provides an SDL (Structure Description

Language) netlist, which may be (and most often is) a hierarchical description that refers to other SDL files. Each SDL file may refer to parameters which are used to personalize the subcell instance generated through the SDL file. One could say that an SDL file is a parameterized cell template that is *instantiated* by its use with a given set of parameters. Since SDL descriptions are hierarchical, the design manager DMoct traverses the design tree and constructs the database from the bottom up, starting with the cells at the bottom level.

A more detailed picture of the design process is shown in Figure 3-20. The design manager program DMoct translates SDL into a parameterized OCT netlist format which is known as the *structure_master*. The name *master* is used because the netlist is a template (or master) that can be used to create different *instances* (copies) personalized by the specified parameter values. The next

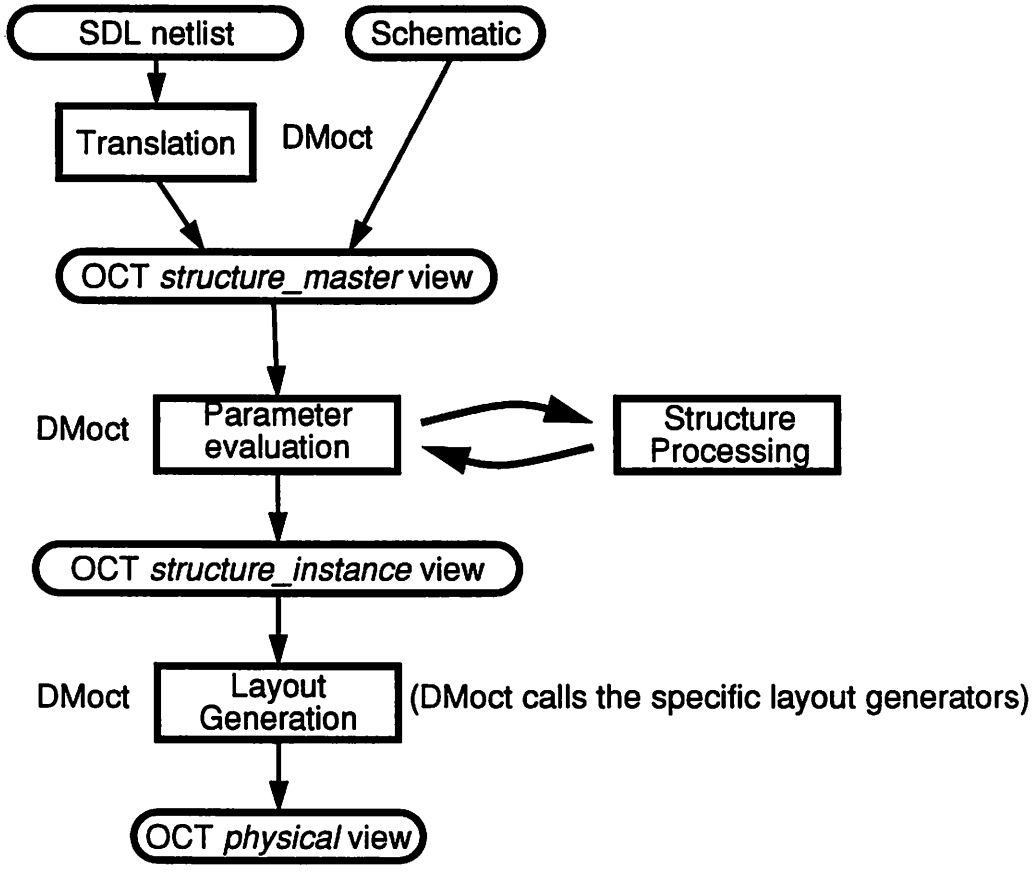


Figure 3-20 More detailed view of LAGER and OCT interaction during the design process

step is known as instantiation: The master is combined with a set of parameters to create a *structure_instance* of the cell. Typical parameters used to personalize a cell are entities such as wordlengths or specifications of the content of a PLA or ROM. For example, a wordlength parameter can be used to specify the number of bits in a certain bus or terminal of the cell. Once instantiated, all nets, terminals and other parameterized structures of a cell become fixed by their parameter values. The final step is layout generation. In this step, the design is created bottom-up as DMoct calls the various layout generators to create the subcells and physically lay out the connections specified by the hierarchical netlist.

The OCT database

The LAGER system is built around the OCT database [spickelmier90][rb92]. OCT is an object-oriented database format for electronic CAD applications. It provides a simple interface for storing and retrieving information about all relevant aspects of an evolving chip or system design. The database can store design descriptions at various levels of detail, ranging from Printed Circuit Board packages and interconnections down to individual geometries of a chip layout. An OCT database is created and accessed through a set of C language functions that are provided with the Octtools software release.

The primitive data type of the OCT database is the **octObject**. An octObject is a *variant* structure that can contain a number of different types of actual data. The common content of all octObjects is shown in Table 3-2. The variant (*union*) part of the object can contain data of any of the primitive types shown in Table 3-3. These object types have been carefully selected for their relevance to electronic design and have been found sufficient for a variety of chip and board level CAD applications. The top-level object of an OCT database is always an **octFacet** object, which is used as a container to store other (non-facet) objects. Figure 3-21 shows how an octFacet can contain various design products or design specifications.

Element type	Element name
int (integer)	type
octId	objectId
union	contents

Table 3-2 Fixed contents of an octObject

Data type	Data name	Data type	Data name
octBag	bag	*octLabel*	label
octBox	box	*octLayer*	layer
octChangeList	changeList	*octNet*	net
octChangeRecord	changeRecord	*octPath*	path
octCircle	circle	*octPoint*	point
octEdge	edge	*octProp*	prop
octFacet	facet	*octTerm*	term
octInstance	instance		

Table 3-3 Variable contents of an octObject

Use of OCT in LAGER

The OCT database is used to represent several abstraction levels in the LAGER system. Parameterized cell templates are stored as *structure_master* views. Instances of parameterized cell templates are stored as *structure_instance* views. Actual layout of cells and interconnections are stored as *physical* views.

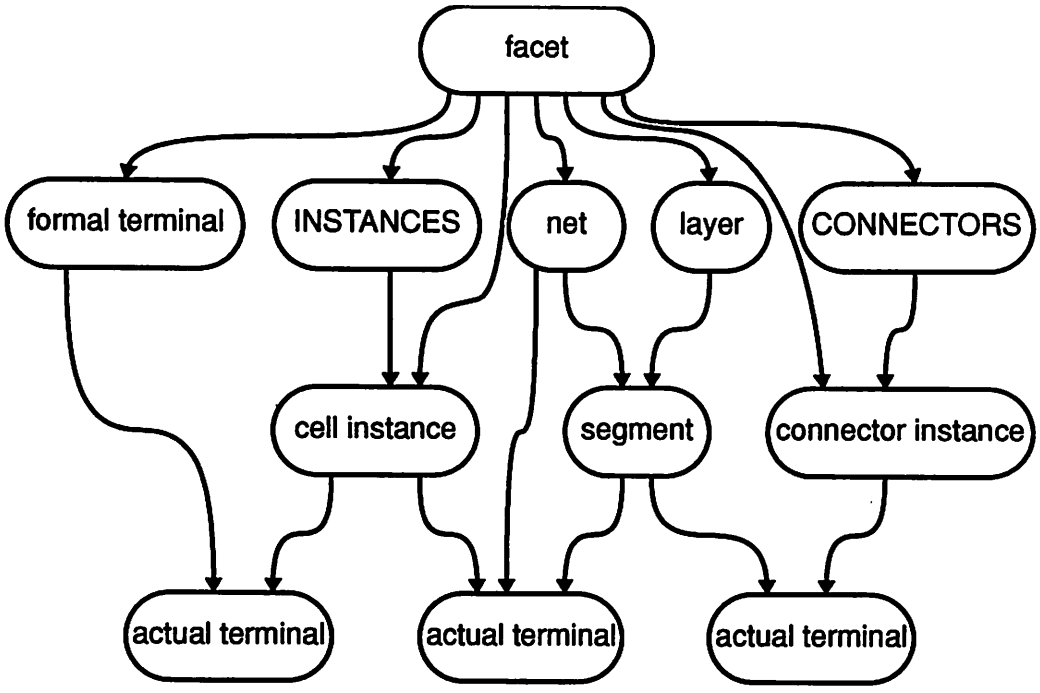


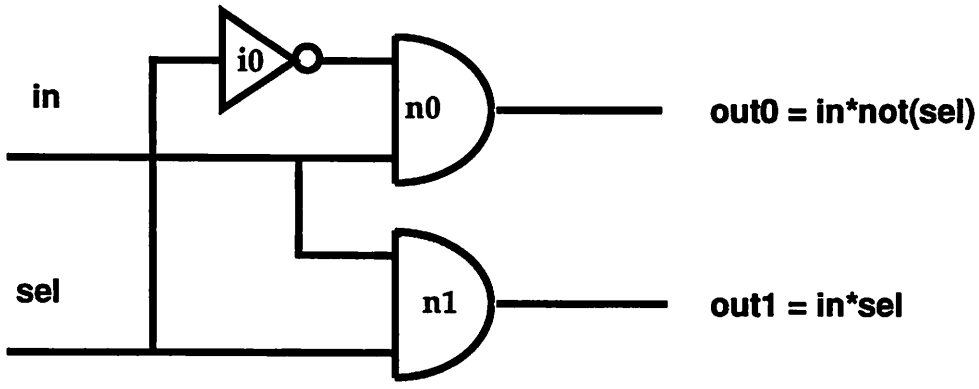
Figure 3-21 Example of OCT facet containing design specification or information (from [rb92])

3.11 Design styles in LAGER

As shown in Table 3-1, LAGER supports several design styles which can be mixed and matched in any given chip design. This section provides a quick overview of the design styles and how they are typically applied in the C-to-Silicon system.

Standard cell (Stdcell)

The Standard Cell design style is based on gate-level primitives (leafcells) that can be interconnected in an arbitrary fashion. The LAGER layout generator for Standard Cell designs is called **Stdcell**. An example of a Stdcell SDL file and the corresponding logic-level schematic is shown in Figure 3-22. The example is a simple decoder function. In C-to-Silicon, the Stdcell is most commonly used to implement random logic blocks for local control and buffering purposes.




```
(parent-cell decode)
(layout-generator Stdcell (r 1)) ;Use r=1 rows
(subcells
   (nanf211 (n0 n1))(invf101 i0)) ;nanf211 is a nand/and

(instance i0 ((a1 sel)(o selb)))
(instance n0 ((a1 in)(b1 selb)(o2 out0) ))
(instance n1 ((a1 in)(b1 sel) (o2 out1) ))
(instance parent
   (in in (TERM_EDGE TOP))
   (sel sel (TERM_EDGE TOP))
   (out0 out0 (TERM_EDGE BOTTOM)
   (out1 out1 (TERM_EDGE BOTTOM))
   (Vdd Vdd) (GND GND))
(end-sdl)
```

Figure 3-22 Example of a Stdcell design specification

A Stdcell block is generated by placing the individual gate primitives into rows of cells and routing the interconnections between the cells in the channels between the rows. Connections from one channel to another is handled by inserting feedthrough cells unless there are already sufficient feedthroughs built into the cells themselves. An example of a 4-row Stdcell layout is shown in Figure 3-23. The layout process for Stdcell designs consists of two main parts: First, the placement and global routing of the cells is determined by the TimberWolfSC program, using a simulated annealing algorithm. Then the detailed channel routing is created by YACR (Yet Another Channel Router). The Wolfepost program is responsible for creating the necessary Vdd and GND terminals on each row, and also producing MAGIC format layout in addition to the OCT layout.

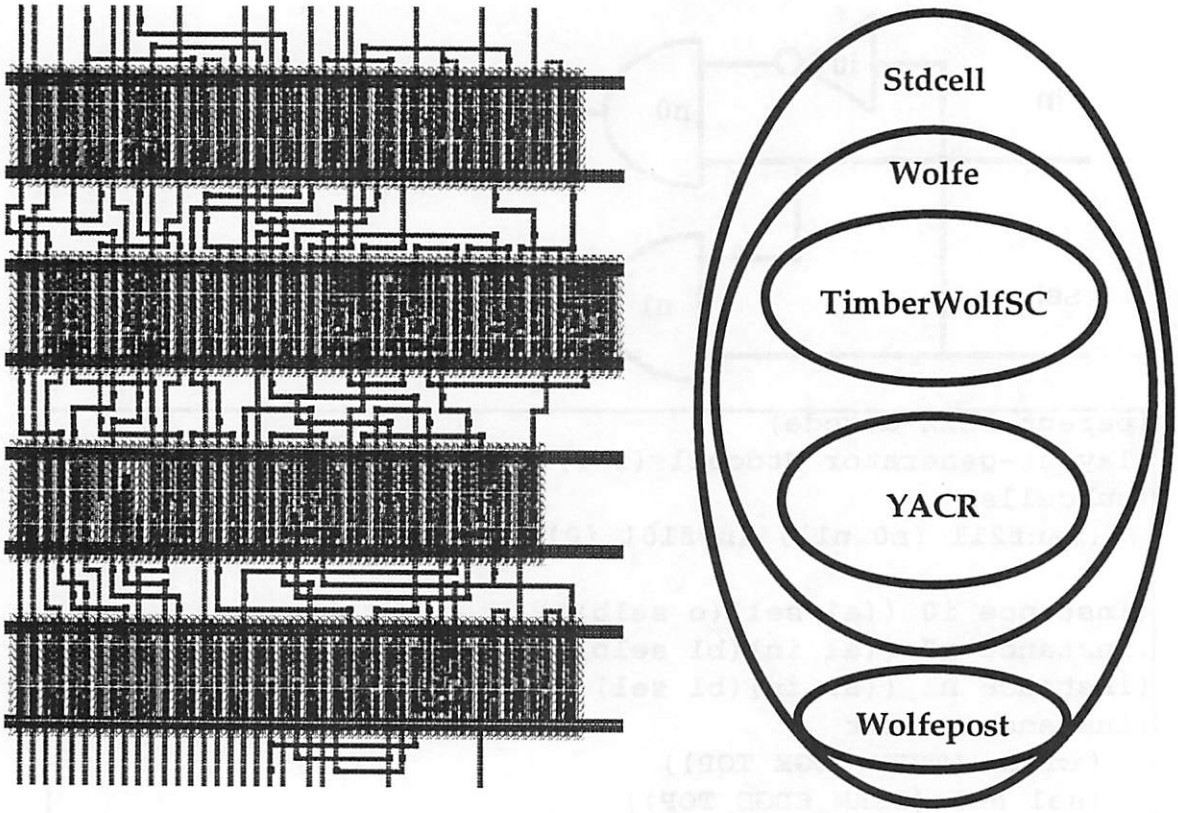


Figure 3-23 Example of 4-row Stdcell layout. The Stdcell program is a shell script which invokes a set of specialized placement and routing programs to create the layout

Tiled macrocells (TimLager)

The tiled macrocell designed style is used to create regular 1- or 2-dimensional layout structures from handmade "tiles" (leafcells) that are stacked next to each other according to a tiling specification. Tiling is typically used for regular and repetitive structures such as PLAs, ROMs, RAMs, array multipliers and datapath stages. Parameterization is a key feature of most tiled macrocells. Parameters are used to specify specific properties of the layout, such as the width of the inplane and outplane of a PLA. Table 3-4 shows the parameters that control the layout of some Tiled Macrocells from the LAGER library. A simple example of a 1-dimensional tiling is shown in Figure 3-24. This is an example of a latch macrocell controlled by only one parameter (*width*). The tiling procedure stacks a number *width* latch leafcells on top of each other and renames the

Block name	Function	Parameters
pla.sdl	PLA	inwidth, outwidth, minterm, input-plane, output-plane
ram3t.sdl	RAM	width, words, ram-address-plane, ram-bit-plane
cs3.sdl	Carry-select adder	n, g, csindex
pmult.sdl	Pipelined mult	xwidth, ywidth
pmultvma.sdl	Pipelined mult	xwidth, ywidth, n, g, csindex
latch	Latch	width

Table 3-4 Example of parameters for some tiled macrocells

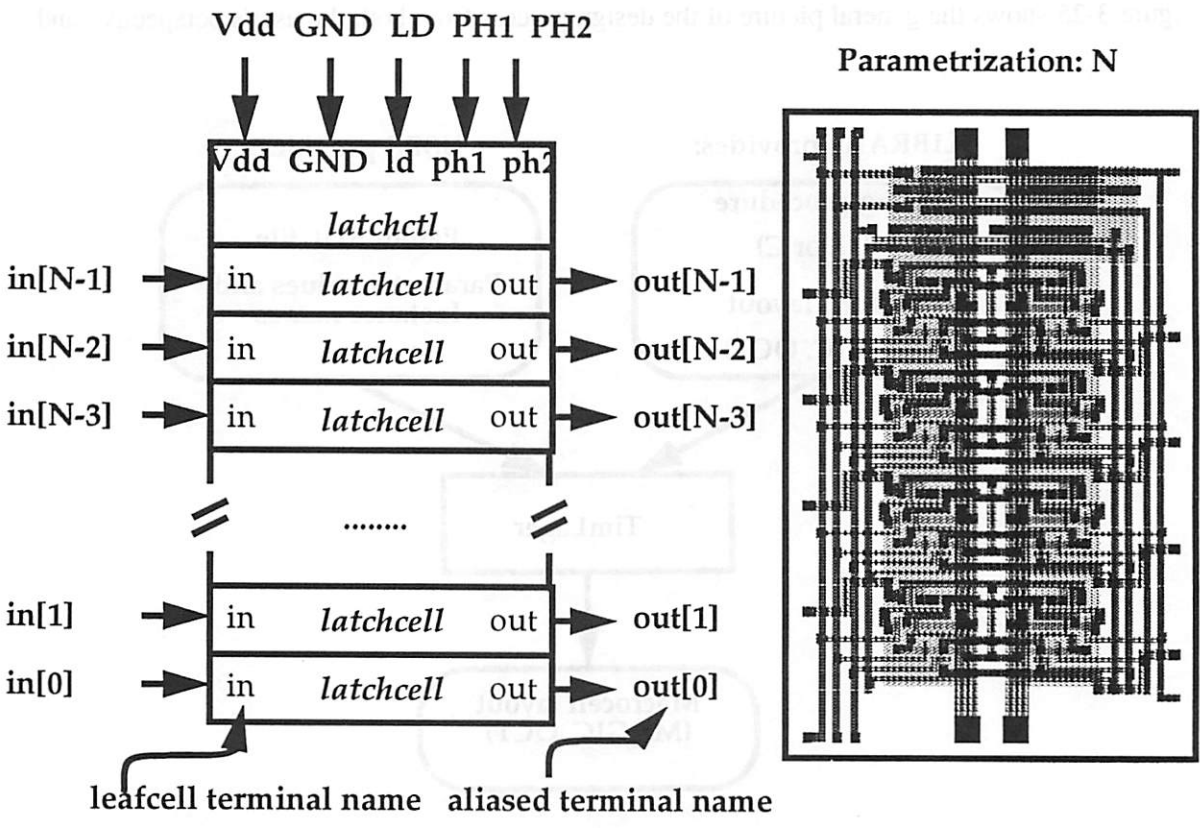


Figure 3-24 1-dimensional tiling example (adapted from [rb92])

terminals with indexed names so that each bitslice has in/out terminals numbered according to position. Finally, a *control slice* is stacked on top to buffer some of the control signals that pass vertically through the latch slices.

C-to-Silicon users will most likely use pre-designed tiled macrocells that already exist in the library. However, if the library does not provide the desired function, the user has the option of designing a new library cell. There are two main tasks in the design process for a tiled macrocell:

- Leafcell design and test tiling. The various leafcells needed must be designed by hand using the MAGIC layout editor. Typically, the designer also creates manually an example of a tiling to check that the leafcells fit together as intended.
- Writing a tiling procedure. The tiling procedure is a C-language function which calls special library functions that place the leafcells in the desired fashion.

Figure 3-25 shows the general picture of the design process from both the user's perspective and

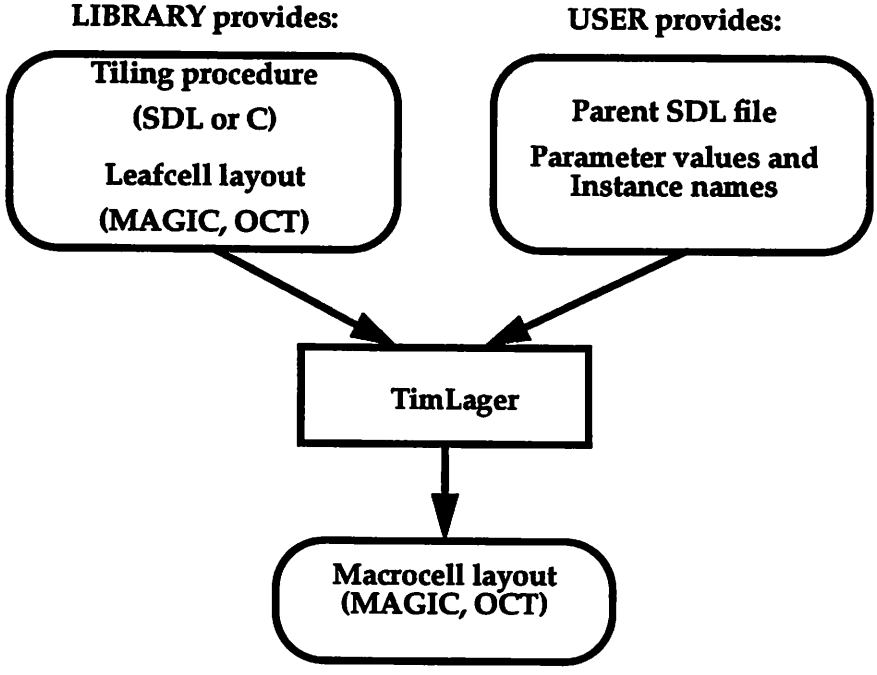


Figure 3-25 User perspective and Library Designer perspective of a Tiled Macrocell (TimLager cell)

the library designer's perspective.

The tiling specification can be made either directly in the SDL file or more generally as a C tiling function. An example of a tiling function (somewhat simplified) is shown in Figure 3-26. The example is a structure consisting of N rows, where each row consists of two leafcells (leafA and leafB) tiled next to each other. The layout is generated by a loop that iterates over the rows. Each call to **Addup(leafcell)** places a leafcell so that the LL (lower left) corner of the leafcell is abutted with the UL (upper left) corner of whatever has been tiled already. **Addright(leafcell)** places the LL corner of the leafcell so that it abuts the LR (lower right) corner of the *most recently* placed leafcell. It should be mentioned that much more general mechanisms are available for placing a new cell relative to previously placed cells, but for row-oriented layout. Addup and Addright are usually sufficient.

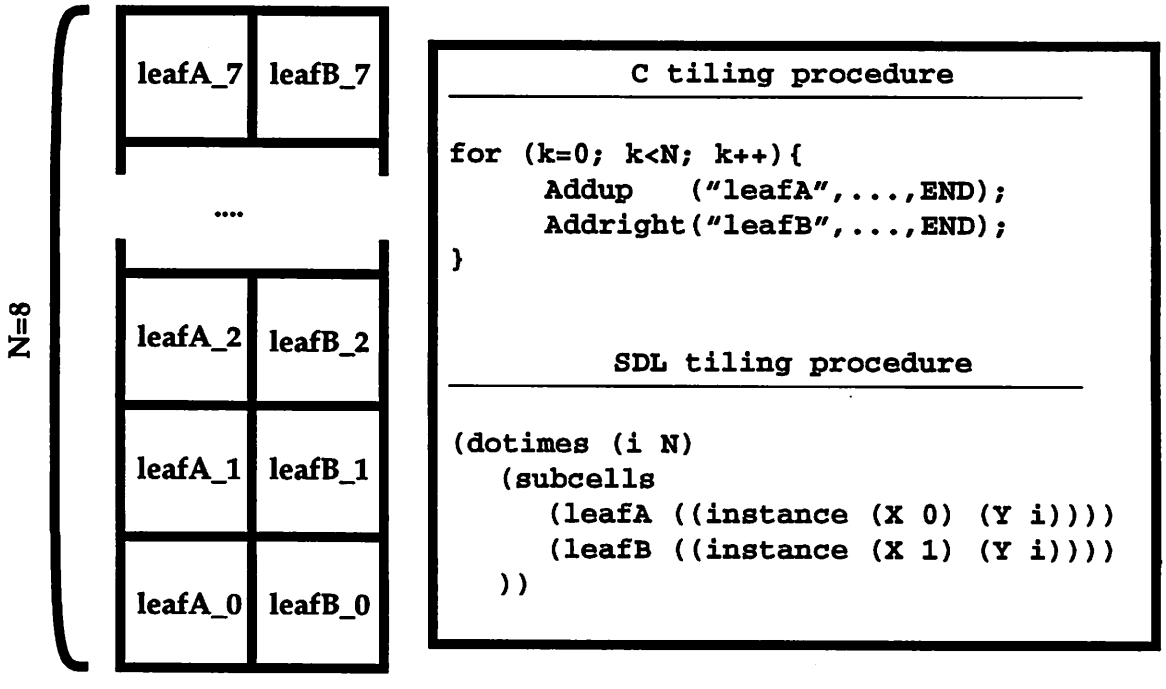


Figure 3-26 A simple 2-dimensional tiling example. Both C tiling procedure and SDL tiling procedure shown

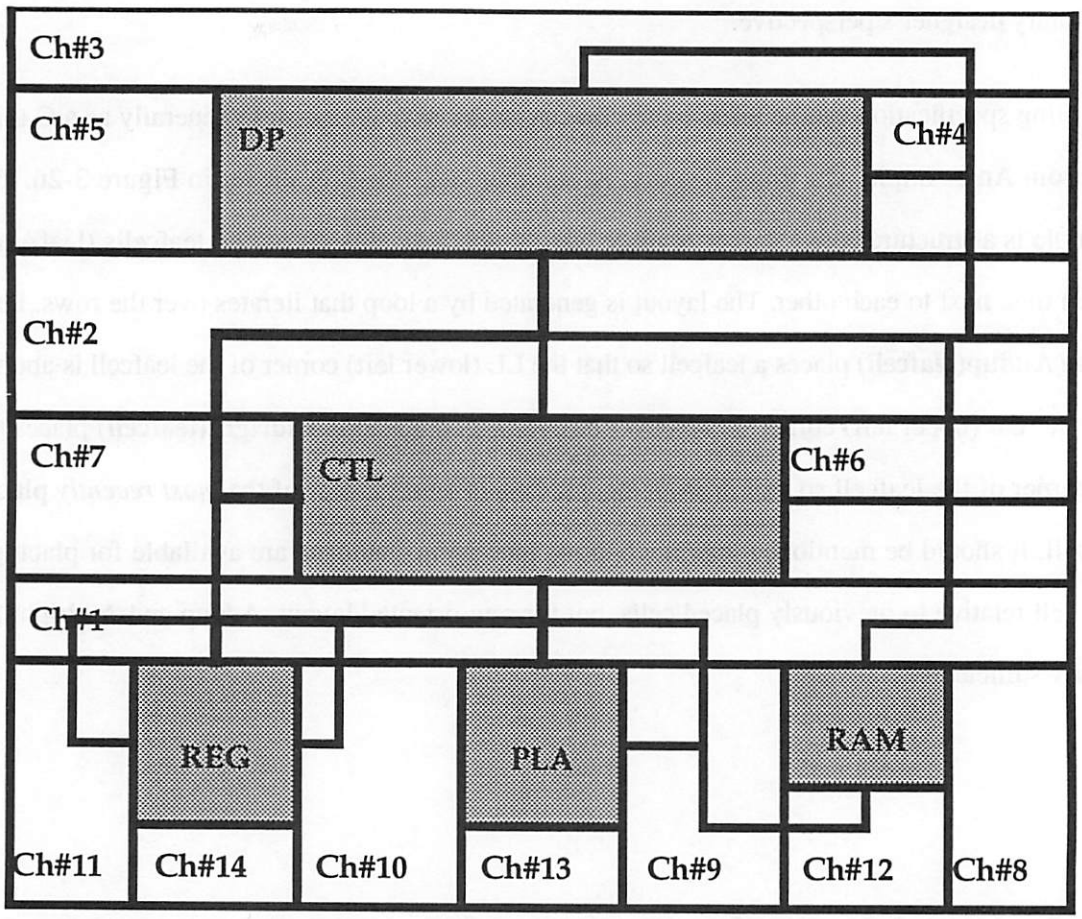


Figure 3-27 Example of Flint floorplan and global routing

For simple tiling procedures it is also possible to specify the tiling directly in the SDL file, as shown in Figure 3-26. The (x,y) relative position of each cell is specified by attaching X and Y *properties* to the subcells as they are declared. This is simpler than writing the corresponding C function.

Macrocell place-and-route (Flint)

After designing macrocells using layout generators such as Stdcell and TimLager, other tools are needed to place the macrocells and route the connections between them. Flint, the Macrocell Place-and-Route tools is the most general layout generator in the LAGER suite. It can place and route macrocells in arbitrary numbers and sizes. The place-and-route process consist of 5 distinct

steps: *placement, channel definition, global routing, absolute placement and detailed routing.* The first three steps are often referred to as *floorplanning* and can be performed in an interactive or automated fashion. Placement simply means a relative placement of the macrocell blocks. Channel definition means defining the routing channels in between the blocks. Since Flint is based exclusively on channel routing, it is necessary that the placement and the resulting channels form a *slicing structure* [otten82]. This means that it is often easier to let Flint create the channel structure for you than to do it manually. Global routing means the assignment of net groups (*cables*) that have the same source and destination to a sequence of routing channels to form a path from the source to the destination. Each side of each macrocell is considered a distinct source or destination.

Once the floorplan is ready, the channel sizes can be estimated and the absolute placement of the cells can be made. Finally, the detailed router creates the actual geometries for the nets in each channel. Figure 3-27 shows an example of a Flint floorplan and global routing.

Flint is often used multiple times in the chip design process, and at several levels of the hierarchy. This means that a cell that was made by Flint at one level can be used again as a Flint subcell at a higher level. Usually this will result in suboptimal routing, as it is always better to optimize the placement and routing of *all* cells at the same time as opposed to dividing the problem into several pieces. By using the FLATTEN feature of DMoct, it is possible to do all of the macrocell place-and-route at the same time, even if the designer has chosen to specify the design in a hierarchical fashion. Hierarchical specifications are often easier to create and debug, since there may be less nets and terminals to consider at each level.

Datapath compiler (dpp)

The datapath compiler (**dpp**) is a specialized tool for creating bit-sliced datapaths. The idea is that a datapath consists of functional blocks (adders, registers, shifters, multiplexers, ...) that all have the same wordlength and that are interconnected in some arbitrary fashion. Figure 3-28 shows an example of a simple datapath consisting of an adder, a multiplexer that allows for saturation of the addition result, and an accumulator which can feed its content back to one of the adder inputs.

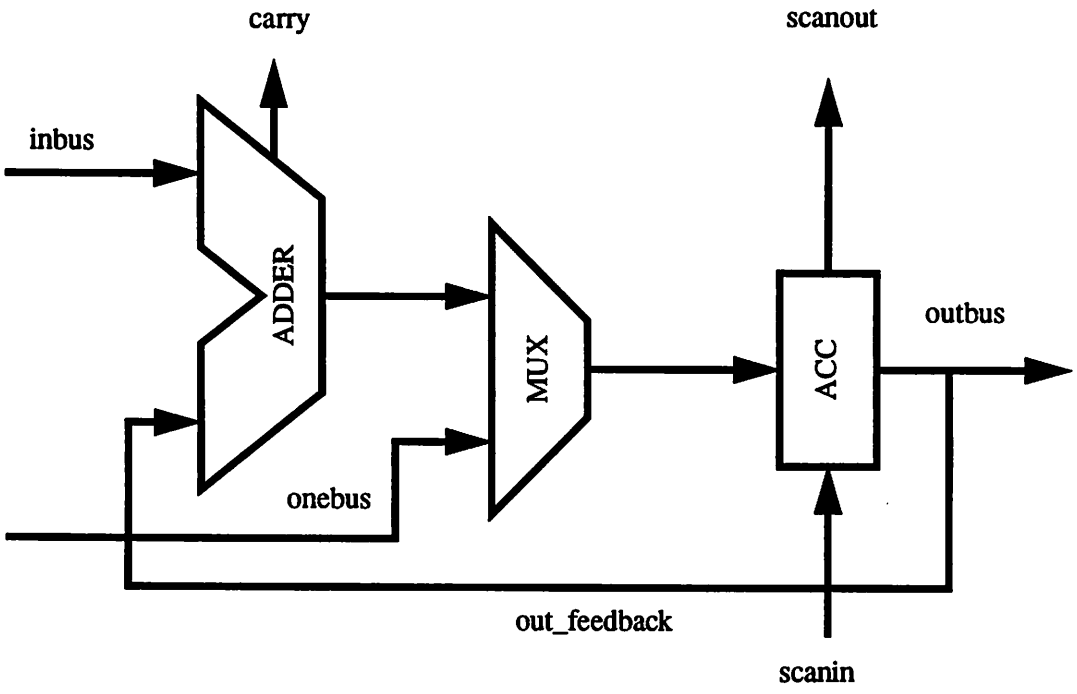


Figure 3-28 A simple datapath (adapted from [rb92])

The overall strategy of the datapath compiler is that the datapath functional blocks (or *stages*) are ordered linearly along the horizontal direction, with the stage-to-stage nets routed in the horizontal direction and the control signals flowing vertically in each stage. Nets that need to be routed over (through) a stage to reach another stage are handled by built-in feedthrough lines in the bit-slices, or by adding extra feedthrough cells as necessary. The datapath compiler generates the stages of the datapath, decides the ordering (unless specified manually), and creates all the routing for the data and control nets. The bitslices of all stages are also equalized in height by inserting special *stretch* cells in between the bitslice leafcells.

Dpp in fact uses TimLager and Flint to perform all layout generation. TimLager is used to tile the individual stages, including feed, stretch and control slice cells. Flint is then used to create the routing between the stages and to the periphery of the datapath. The main function of dpp is to estimate the height of the bitslices in each stage, including necessary feedthroughs, so that the proper amount of feedthroughs and stretching can be inserted. Dpp also generates a complete floorplan for Flint. An example is shown in Figure 3-29.

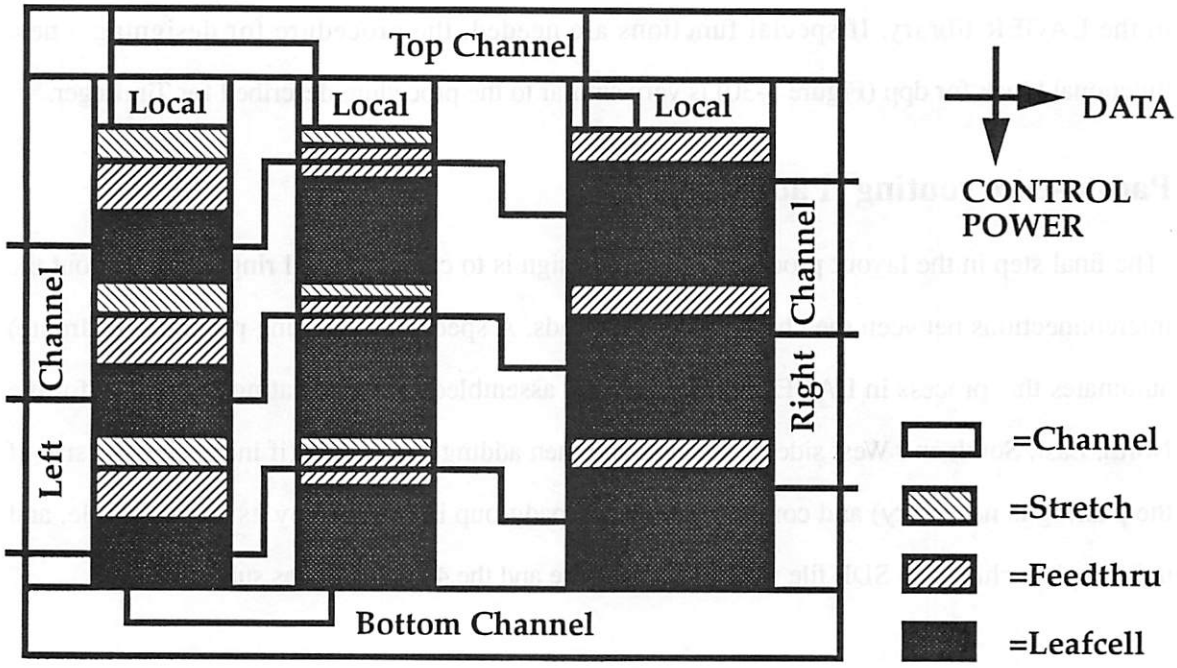


Figure 3-29 Generic floorplan for a datapath

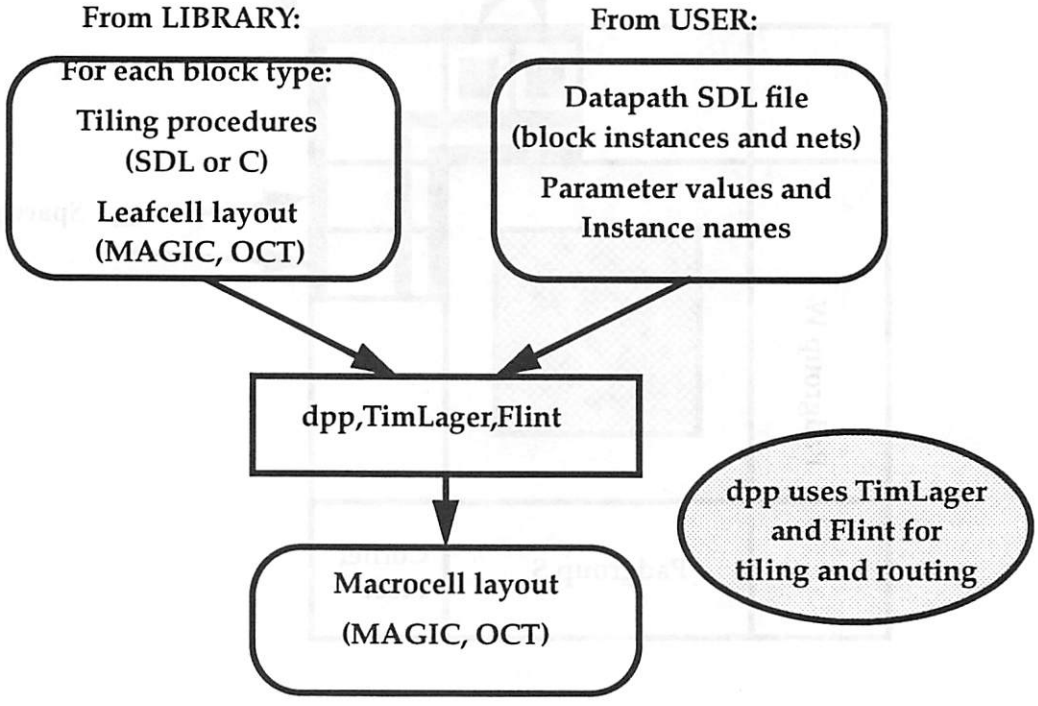


Figure 3-30 User's and Library Designer's perspective of the datapath compiler

As for tiled macrocells, the C-to-Silicon user will most often use pre-designed dpp stages that exist in the LAGER library. If special functions are needed, the procedure for designing a new functional block for dpp (Figure 3-30) is very similar to the procedure described for TimLager.

Pad-to-core routing (Padroute)

The final step in the layout process for a chip design is to create the pad ring and to lay out the interconnections between the chip core and the pads. A special pad routing program (Padroute) automates this process in LAGER. The padring is assembled by first creating padgroups for the North, East, South and West side of the chip and then adding space pads (if increasing the size of the padring is necessary) and corner pieces. Each padgroup is specified by its own SDL file, and the complete chip is an SDL file which uses the core and the 4 padgroups as subcells.

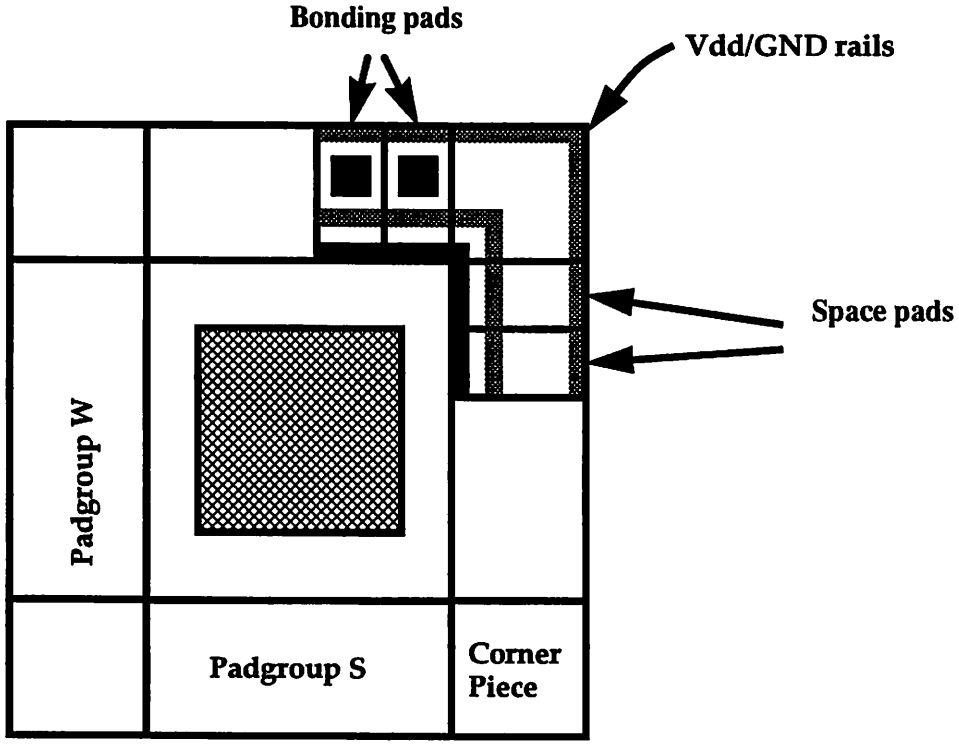


Figure 3-31 Padring generation and pad-to-core routing (adapted from [rb92])

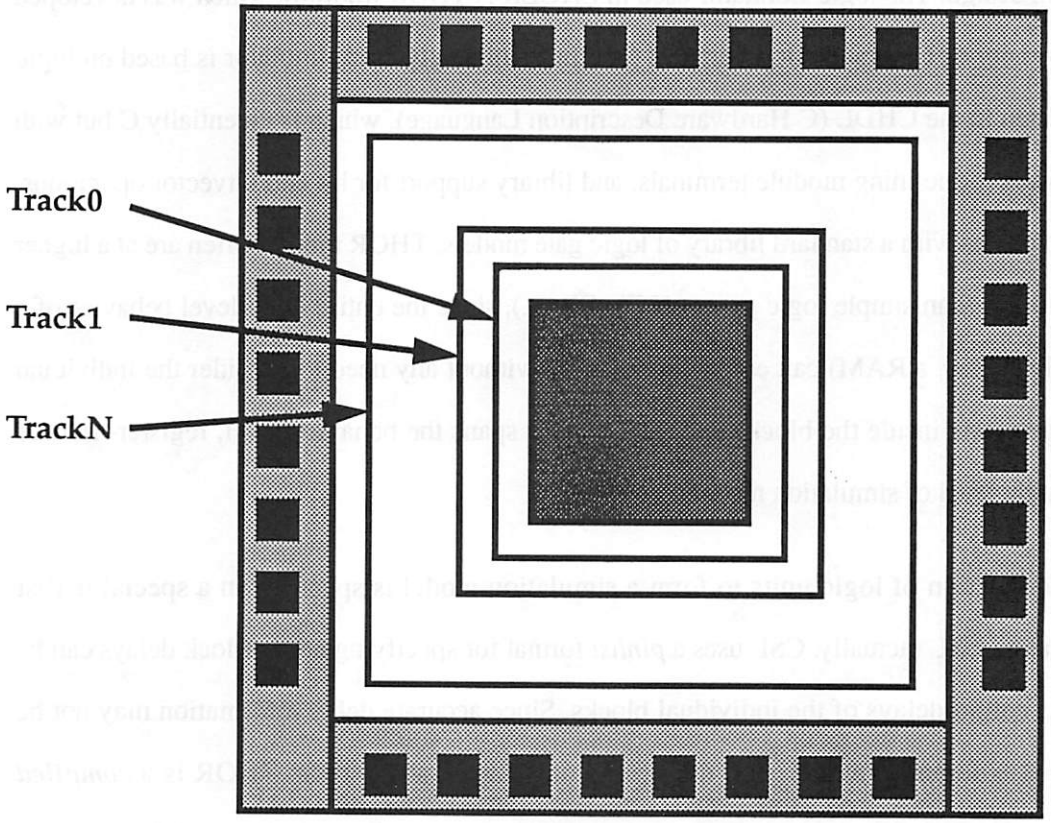


Figure 3-32 Padroute uses a special channel router for ring-shaped channels

The pad-to-core routing problem is somewhat unusual in that a circular routing area is involved. Padroute treats all of the routing area as one channel (Figure 3-32), using a special radial channel router [lettang89][rb92].

3.12 Logic-level simulation

While not an integral part of the C-to-Silicon system, logic and switch level simulation tools play an important role in the C-to-Silicon design process. This and the following section is an overview of the simulations tools and strategies that are employed in the system.

Logic simulation is used to verify the basic topology (interconnectivity) and logic-level implementation of an architecture, and hence is an important part of the verification process for a

C-to-Silicon design. The logic simulator used in LAGER is THOR [thor88], which was developed at the University of Colorado (Boulder) and Stanford University. The simulator is based on logic models written in the CHDL (C Hardware Description Language), which is essentially C but with special macros for defining module terminals, and library support for bit and bitvector operations. THOR also comes with a standard library of logic gate models. THOR models often are at a higher abstraction level than simple logic gates (AND, OR, ...), since the entire logic-level behavior of a complex block (say, a RAM) can easily be modelled without any need to consider the individual gates or transistors inside the block. In fact, THOR is spans the behavior level, register-transfer level and logic level of simulation models.

The interconnection of logic units to form a simulation model is specified in a special netlist language called CSL (actually, CSL uses a *pinlist* format for specifying nets). Block delays can be specified as output delays of the individual blocks. Since accurate delay information may not be available, it is common to use zero delay or unit delay in all models. THOR is a *compiled* simulator, meaning that for each design, a unique binary program is created to model the design. Both CHDL files and CSL files are preprocessed and converted into standard C files that are compiled using any standard UNIX C compiler. THOR is equipped with an *analyzer* program, which is a graphical presentation interface which can show selected logic-level waveforms from inside the circuit.

The LAGER cell library contains THOR models for all cells. The problem of logic simulation hence becomes a question of generating the netlist for the design (in CSL format), and to compile the simulator. Since CSL uses a completely different syntax and a pinlist format, there may be considerable difference between the original SDL files and the corresponding CSL files. SDL supports both pinlist and netlist format concurrently, so it is necessary to parse the SDL files completely (by hand or with a program) in order to generate the corresponding CSL files.

The obvious solution for an automated approach is to use the already parsed information as it appears in the OCT database. A translator named MakeThorSim was created by Svensson and

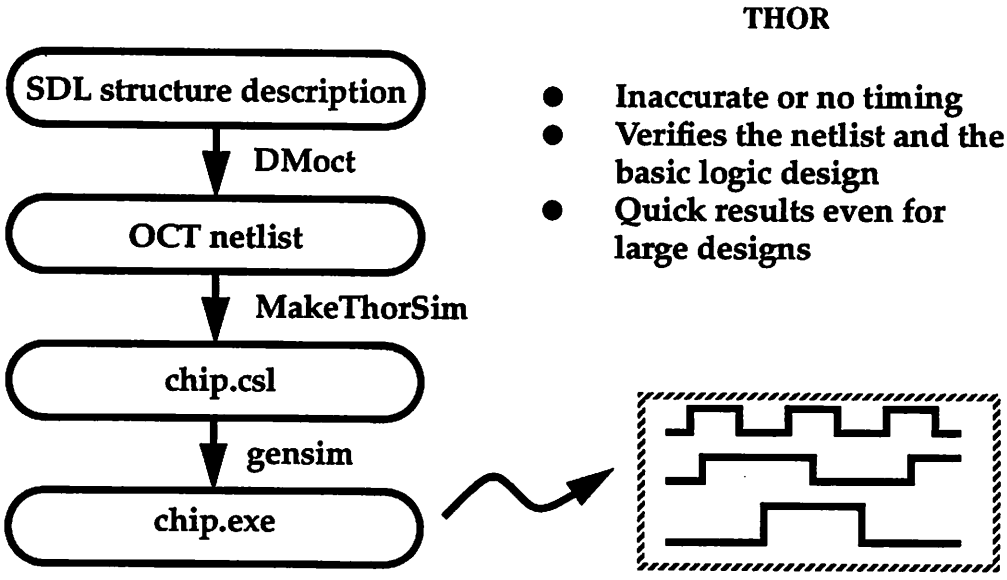


Figure 3-33 Generating a THOR simulator from SDL

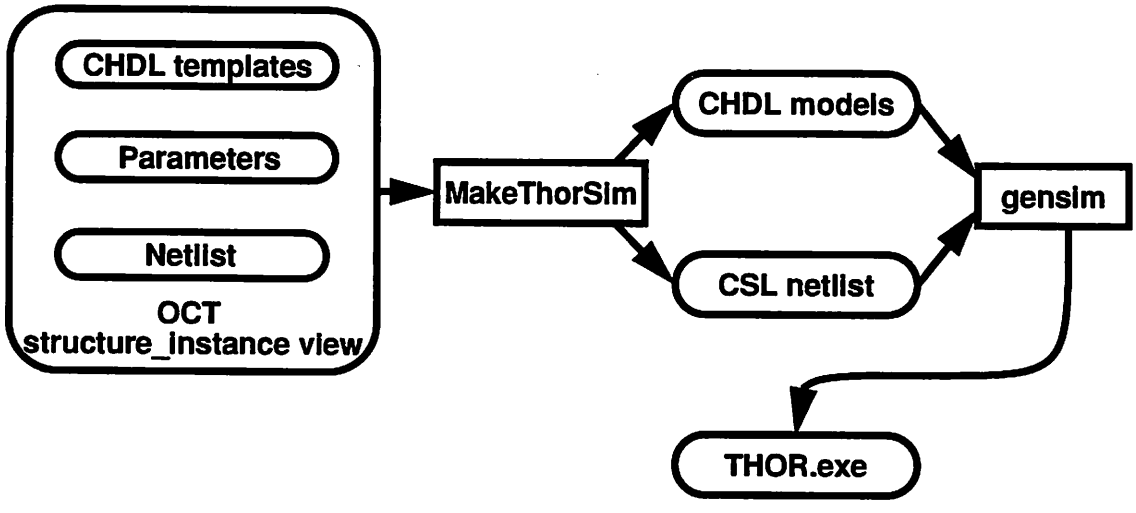


Figure 3-34 CHDL templates are stored inside the OCT views and instantiated and interconnected using MakeThorSim

Sheng [rb92] for this purpose. MakeThorSim generates the CSL netlist directly from the OCT database, so that consistency between the original SDL files and the THOR netlist is ensured. Figure 3-33 shows the process of generating a simulation model of the design chip.sdl. MakeThorSim flattens the OCT netlist down to the level where each "leaf" has a THOR model. The CSL is then created. If a net has different names at different levels of the SDL/OCT hierarchy, the netname at the top level is used. Since SDL files have parameters, the THOR models stored in the cell library are in fact parameterized templates that are instantiated by MakeThorSim (the actual parameter values are inserted). The C compilation of the THOR simulator is performed by the *gensim* program (Figure 3-34). MakeThorSim does not provide net delay estimation.

The advantage of the THOR simulator is that it is very quick, and provides verification of the basic logic design. However, it does not provide accurate timing information when used as described here. It is possible to generate timing information, but this would require parameterized delay models for each CHDL block model, and a procedure for back-annotating the CSL file with additional delay values estimated from actual wiring capacitances in the layout. This also requires that the layout is generated before the THOR simulation takes place, which is not always convenient.

3.13 Switch-level simulation

Switch level simulation is used to verify the functionality and timing of finished layout. The switch level (N)MOS transistor device model [terman83][horowitz84][chu88][salz90] is shown in Figure 3-35. A switch level model of a circuit is made from the original transistor network by replacing the transistors by their switch level models and adding in all the wire and diffusion capacitances as lumped capacitors at the appropriate nodes. IRSIM [salz90] is a circuit simulator based on this model. It uses 4 possible states for each node: low(0), high(1), unknown(x) and forced undefined(u). Waveforms are modelled as step functions. The transistor parameters C_{gate} and R_{eq} are known or can be derived from the device sizes and the fabrication process parameters.

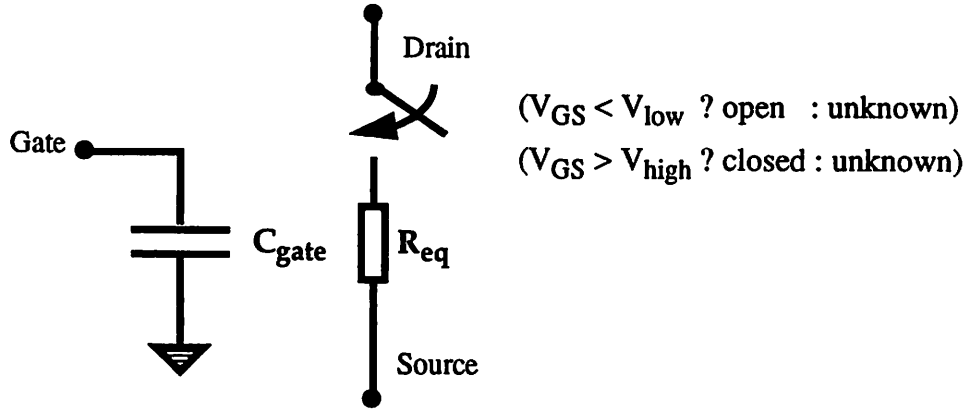


Figure 3-35 Switch level NMOS transistor device model used in IRSIM

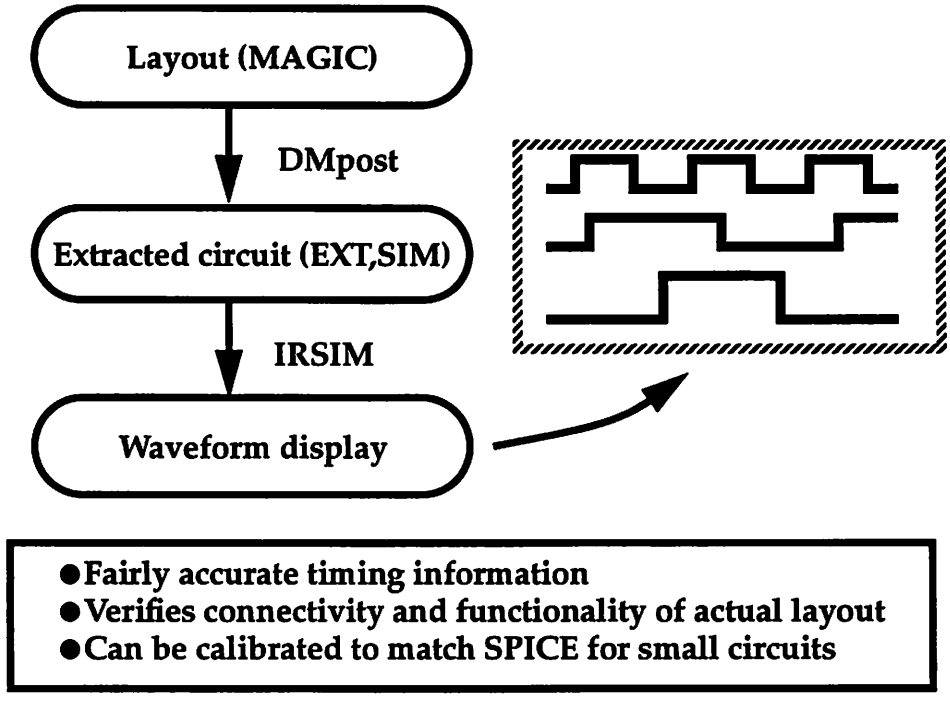


Figure 3-36 LAGER support for IRSIM simulation from layout

Parameter name	Value	Comment	
capga	.00153	gate capacitance pF/μm^2	
lambda	0.60	microns/lambda	
lowthresh	0.4	logic low threshold (normalized)	
highthresh	0.6	logic high threshold (normalized)	
Channel resistances	*width*(μm)	*length*(μm)	*resistance* (Ω)
n-chan dynamic-high	10.0	1.2	1771.0
n-chan dynamic-low	10.0	1.2	908.0
n-chan static	10.0	1.2	944.0
p-chan dynamic-high	20.0	1.2	1182.0
p-chan dynamic-low	20.0	1.2	2435.0
p-chan static	20.0	1.2	1122.0

Table 3-5 The primary IRSIM model parameters for a MOSIS 1.2μm SCMOS process. Equivalent resistance values have been computed from SPICE simulations

An IRSIM simulation is most often based directly on extracted layout to make the circuit model reflect the layout as accurately as possible.

IRSIM input data

IRSIM requires two input files:

- *The parameter file* contains electrical parameters for the circuit technology, mainly properties of the transistors, such as threshold voltages, gate area capacitances and dynamic channel resistances (for the piecewise linear model). There are also area capacitances for various layers, but this information is not used by IRSIM presently. IRSIM provides a calibration facility where a SPICE model (e.g. as provided by MOSIS) can be used to tune the parameter file so that there is close correspondence between IRSIM and SPICE delay values. The difference between SPICE and IRSIM is often less than 10% when using tuned models on a moderately sized circuit (say, a full adder with 50 transistors). An example of a parameter file is shown in Table 3-5.

- *The circuit file* is a flat circuit description in the MAGIC sim file format. This file is typically generated directly from the circuit layout by using the MAGIC extract facility and then running the `ext2sim` program to flatten the hierarchical *ext* description into a flat *sim* file.
- Additional command files and/or interactive entry can be used to specify input patterns, timing and simulation commands. The simulation results are displayed through a graphical interface called the *analyzer.*

Using IRSIM

IRSIM is the basic tool to check a layout for connectivity, functionality and timing. Any missing connections or shorts in the layout will be revealed if properly exercised by the test patterns. Functionality can likewise be established by applying appropriate input sequences. The timing accuracy is also quite reasonable as long as the wiring resistance in the circuit can be ignored. Since IRSIM does not model interconnection resistance, it will not produce accurate results if, for example, a layout contains long polysilicon lines. LAGER provides an a post-processing tool (DMpost) which takes care of circuit extraction and generating the sim file (Figure 3-36). An example of IRSIM simulation is shown in Chapter 4.

3.14 The RL language

The RL language [rimey89] is an approximate subset of C. RL includes only those features of C that correspond closely to the capabilities of DSP architectures—recursion, for example, is not supported. RL includes two major extensions: fixed point types and register type modifiers. It is therefore not strictly compatible with C.

Fixed point types are a convenience for the programmer. The underlying integer arithmetic is inconvenient to write by hand, partly because simple fixed point constants correspond to huge integers, and partly because the natural multiplication for fixed point numbers is not the same as integer multiplication. In adding a new numerical type to a programming language, finding an elegant notation for the new constants can be difficult. In RL, all constants are typeless real

numbers that take on appropriate types from context. In declarations and type casts, the fixed point type of range $-2^n \le x < 2^n$ is denoted by `fix:`*n*; or if $n = 0$ then just by `fix`.

Register type modifiers, which are generalized C register declarations, let the programmer suggest storage locations for critical variables. For example,

```
register "r" fix y;
```

declares the variable `y` to be a fixed point number to be stored in the register bank `r`. A reasonable default is chosen if the name of the register bank is omitted. Register type modifiers are also helpful with multiple memories, and they can be applied to pointers. For example,

```
"mem" fix * "mem2" p;
```

declares `p` to reside in mem2 and point into mem.

Limitations

Many parts of C have been left out of RL for the sake of simplicity:

- There is no separate compilation.
- There are no explicit or implicit function declarations; functions must be defined before they are used.
- Initial values may only be specified in declarations of variables that are to be stored in read-only memory.
- There are no `struct`, `union`, or `enum` types; no `char`, `float`, or `double` types; and no `short`, `long`, or `unsigned` modifiers. This leaves only `void`, `int`, pointer types, array types, and the RL-specific types, `bool` and `fix`.
- There are no `goto`, `switch`, `continue`, or `break` statements.
- There is no `typedef`, no `sizeof`, and there are no octal or hexadecimal constants.

Because the target processors do not provide a stack for local variables, it is also necessary to prohibit recursive function calls. For the same reason, the programmer has to be aware that doing a function call within the scope of a register declaration will force the compiler to produce rather

poor code.

Type modifiers

In RL, the `const` type modifier is used mainly in declaring variables that are to be stored in read-only memory. The `volatile` type modifier is used mainly to identify boolean variables that represent signals on external pins. A `volatile bool` variable represents an output pin which is set by the processor. A `const volatile bool` variable represents an input pin which is sensed by the processor.

Pragmas

In RL, pragmas have the same form as the `#define` preprocessor command, but start with `#pragma` instead. Pragmas define flags and parameters that control the RL compiler and other software, as in these examples:

- `arch_file` gives the name of the machine description file to use
- `word_length` determines the number of bits in a processor word
- *x*`_capacity` sets a limit on the number of registers that the compiler may assume for the register bank *x*.

Register declarations and register type modifiers

The RL compiler assigns a variable to a specific memory or register bank depending on

- whether or not it is a `register` variable,
- its base type, and
- if the base type is a pointer type, the bank that it points into.

The defaults for a given architecture are specified by pragmas in the machine description, but can be overridden by pragmas in the RL program. For example, to override the usual defaults for Kappa and store non-`register` integer variables, and pointers into bank `mem`, in bank `x` instead of in bank `mem`, the programmer would put the following pragmas into the RL program:

```
#pragma int_memory "x"
#pragma mem_pointer_memory "x"
```

Assigning all variables to default memory and register banks is sometimes too crude. For such cases, RL has register type modifiers. A register type modifier is written as the name of a memory or register bank in double quotes. It is a type modifier, like `const` and `volatile`, that can appear wherever `const` and `volatile` can appear. For example, an integer variable `x` stored in the bank `foo` would be declared like this:

```
"foo" int x;
```

A more complex example is a pointer to `int`, residing in the bank `bar` and pointing into the bank `foo`:

```
"foo" int * "bar" p;
```

The boolean type

In C, boolean values (true and false) are represented by integers, which is convenient for typical general-purpose computers. In contrast, our application specific target processors perform boolean operations on (and store) individual bits. This is the reason for having a distinct boolean type, `bool`, in RL.

In RL, there are no implicit conversions to or from `bool`, except in certain cases involving literal numbers. True can be written as `(bool)1`; false, as `(bool)0`; and in most cases, the casts can be omitted.

The operations that return booleans as results are the relationals (`<`, `>`, `<=`, `>=`, `==`, `!=`) and the boolean operations (`&&`, `||`, `!`). The operations that require boolean operators are the three boolean operations, and the conditional expression (*condition* `?` *then-part* `:` *else-part*). The tests in `if`, `while`, `do-while`, and `for` statements are also required to be boolean.

Fixed point numbers

RL has a set of fixed point types. Arithmetic on fixed point numbers is saturating, except in shift operations. This is in contrast to integer arithmetic, which is always non-saturating.

The fixed point types have names of the form `fix:`*n*, where *n* is a possibly negative integer. The form `fix` is a shorthand for `fix:0`. Values of type `fix:`*n* have a machine-dependent precision (controlled by the pragma `word_length`) and lie in the range $-2^n \le x < 2^n$. Casts may be used to convert between the different fixed point types, but conversions between fixed point and integer types are not allowed. A cast of a fixed point datum to another fixed point type is typically implemented with an arithmetic shift operation.

All of C's floating-point arithmetic operators are available in RL for fixed point arithmetic. With the exception of multiplication and division, the arguments of a binary fixed point operator must have the same type, as must the second and third arguments in a conditional expression. Casts are commonly used to accomplish this. Fixed point values may be explicitly shifted with the arithmetic shift operators << and >>.

Predefined functions

RL has three predefined functions: `abs()`, `in()`, and `out()`. These functions are overloaded to take arguments of type `int` as well as type `fix:`*n*. The value returned by `in()` may be considered to be of type `number`, that is, the resulting type (after implicit conversion) depends on a limited amount of context. In ambiguous cases, casts must be used.

User-defined operations

Hardware-supported operations that are not predefined in RL can be specified in the machine description file. An operation is defined and given a name, and one or several implementations of the operation are specified in the same way as for the predefined operations. An operation defined in this way is available in RL in the form of a "function call", where the function has the same name as the operation. This is useful for hardware lookup tables and in general for handwritten, idiomatic instruction sequences. For example, a multiplication step with some particular behavior on overflow might be implemented as a user-defined operation because it would not be compiled into efficient code if written in pure RL.

Preprocessor commands

There are four new preprocessor commands in addition to those of standard C. They are useful for unrolling and partially unrolling loops: #repeat, #endrepeat, #rrepeat, and #endrrepeat. The form

```
#repeat id N
...text...
#endrepeat
```

is roughly equivalent to

```
#define id 0
...text...
#undef id
#define id 1
...text...
#undef id
...
#define id N-1
...text...
#undef id
```

`#rrepeat` and `#endrrepeat` are similar, except that they count backwards.

Program structure

The last difference between RL and C is that the RL programmer may (and often will) leave `main()` undefined. In its place, the code should define `loop()` and optionally `init()`. The compiler then supplies an implicit `main()`, where `init()` is called once (if it has been defined), and then `loop()` is called indefinitely. This is an appropriate form for a program which reads a indefinite input stream.

3.15 Summary

The C-to-Silicon system is a powerful design tool for Application Specific Processors for Numerical Processing and DSP. The system supports easy architecture exploration and performance evaluation at a the architecture level, without having to perform detailed logic and

layout level design. High-level algorithm simulation is also supported. C-to-Silicon uses the LAGER Silicon Assembly System to perform layout and simulation tasks, resulting in a very powerful and general system. It has been shown that C-to-Silicon is flexible with respect to the range of architectures and algorithms that can be implemented.

CHAPTER 4

THE PUMA PROCESSOR

The PUMA processor has served as a first test case for the C-to-Silicon system. The design of the PUMA chip demonstrates and exercises all the central features of the system, including architecture exploration and system simulation. One of the important results that will be presented is that small and inexpensive changes to a generic architecture can have a dramatic impact on cost and performance.

This chapter is arranged as follows: The first section presents the computational task to be implemented on the chip. The task is examined in detail to identify its primary computational characteristics. In the second section, this knowledge is applied to select and develop algorithms that will allow to an efficient integrated circuit implementation. Following the program code development, the computation is simulated both in floating point and fixed point to verify the numerical soundness of the formulation. The next section describes the architecture exploration process and discusses the merits of the various architectural alternatives. The final sections describe the finished chip developed in this process and its physical characteristics, including the chip layout design and the chip-level verification results.

4.1 Characteristics of the computation

The PUMA chip performs the Inverse Position-Orientation computation for the PUMA 560 industrial robot (Figure 2-3). The background for this problem, as well as the specifics of the computation were described earlier in section 2.3 (p27). At this point we are concerned with translating the expressions (2-11) to (2-27) into RL code and at the same time assess the computational needs in evaluating the expressions.

The RL code started out as a C program that was used for an initial investigation of the PUMA IPO problem. The C code had been debugged and verified before the idea of a chip design was conceived, and it consisted of a pretty straightforward translation of the IPO equations, using the C math library to evaluate the necessary elementary functions. Some important facts about the IPO computation were found by studying the C program, in particular it was found that the computation is intensive in multiplications and trigonometric function evaluations.

Table 4-1 contains a summary of the operations involved (for computing all eight solutions). The table shows clearly that there is strong need for efficient algorithms for the sin/cos/atan2/sqrt operations. The standard method used in general purpose computer systems is *rational approximation.* That is, using an approximation which is the ratio of two polynomials [cody80]. Rational approximations usually involve polynomials of degree 3-4 each. This adds up to 4-6 multiplications and 1 division [which would cost about 8 multiplications using Newton-Rhapson's

Operation	Count	Operation	Count
Mult(var)	208	atan2	22
Mult(const)	24	cos	14
Add	108	sin	14
Sub	104	sqrt	2
Divide (var)	1	Divide (const)	1

Table 4-1 The IPO algorithm is intensive in multiplication and trigonometric functions

function	shift and add	add	mult
atan2	34	17	0
cos and sin	36	19	0
sqrt	32	18	1

Table 4-2 Cordic functions consist mostly of shift/add operations

method], resulting in 12-14 multiplications for each atan2 evaluation.

To minimize area requirements it is always desirable to employ an architecture without an array multiplier. Not having an array multiplier makes the rational approximation approach very time consuming. In fact, just evaluating atan2 22 times would involve more multiplications and additions than all the remaining parts of the algorithm. This fact provided a strong motivation to investigate the use alternative algorithms for sin/cos/atan2/sqrt, based on CORDIC operations.

4.2 Algorithm selection

CORDIC [blahut85][walther71][volder59] is a family of algorithms that meets our requirements: It can compute all the functions needed, and in the absence of an array multiplier, it is also much faster (fewer cycles) than rational approximation. For a 20 bit wordlength (and full accuracy), the operation count is shown in Table 4-2.

Note that cos and sin are computed at the same time at no extra expense. This is quite handy in our case, because both sin and cos of each angle are always needed. As will be seen later, CORDIC can be efficiently implemented on a datapath which has an adder with a variable preshifter for one of the adder inputs. A description of a typical CORDIC algorithm follows.

4.2.1 CORDIC algorithm for atan2

The function atan2 (y,x) is defined as the angle between the x-axis and the vector (x,y). It is an extension of the regular atan(y/x) to cover all 4 quadrants. The atan2 CORDIC algorithm is based

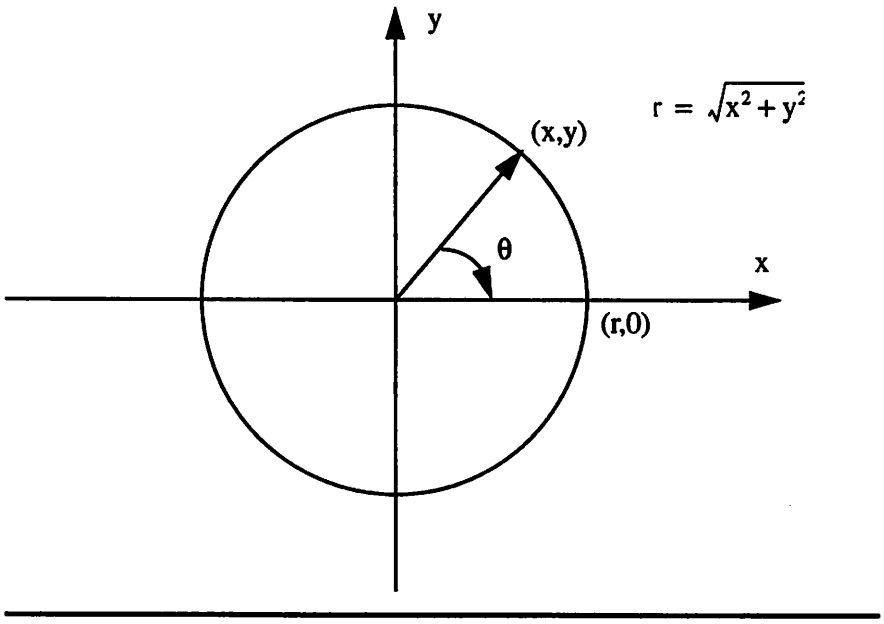


Figure 4-1 The CORDIC algorithms use vector rotations to compute elementary functions

on rotating the vector (x,y) through a sequence of rotation angles until it becomes (r,0), as indicated in Figure 4-1. The arctangent can then be found by adding up all the angles that (x,y) were rotated by. The key to the efficiency of the CORDIC method is that there are certain angles by which rotation is very simple, and that any angle can be approximated by a (signed) sum of such angles. CORDIC uses the angles $\varphi_k = \text{atan}(2^{-k})$, meaning that we have

$$\cos\varphi_k = \frac{1}{\sqrt{1+2^{-2k}}}, \qquad \sin\varphi_k = \frac{2^{-k}}{\sqrt{1+2^{-2k}}} \tag{4-1}$$

Rotating (x,y) by the angle φ_k results in a new vector (x',y') that is related to (x,y) by the equation

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\varphi_k & \sin\varphi_k \\ -\sin\varphi_k & \cos\varphi_k \end{bmatrix} \times \begin{bmatrix} x \\ y \end{bmatrix} = \frac{1}{\sqrt{1+2^{-2k}}} \begin{bmatrix} 1 & 2^{-k} \\ -2^{-k} & 1 \end{bmatrix} \times \begin{bmatrix} x \\ y \end{bmatrix} = K \begin{bmatrix} 1 & 2^{-k} \\ -2^{-k} & 1 \end{bmatrix} \times \begin{bmatrix} x \\ y \end{bmatrix} = K \begin{bmatrix} x + 2^{-k}y \\ -2^{-k}x + y \end{bmatrix} \tag{4-2}$$

Ignoring the factor K, note that the rotation only involves addition and multiplication by negative powers of 2. The latter can be performed by a downshifter, so that no actual multiplications are necessary. Ignoring K (which is <1) means that x' and y' will both be too large by the same factor.

k	φ_k	k	φ_k	k	φ_k
0	45.000000	7	0.447614	14	0.003497
1	26.565051	8	0.223810	15	0.001748
2	14.036243	9	0.111905	16	0.000874
3	7.125016	10	0.055952	17	0.000437
4	3.576334	11	0.027976	18	0.000218
5	1.789910	12	0.013988	19	0.000109
6	0.895173	13	0.006994	20	0.000054

Table 4-3 The set of angles used in the CORDIC iterations

However, to get the correct arctangent, only the *ratio* y'/x' is relevant, and it remains correct when both the numerator and the denominator are off by the same factor. In other words, we simply ignore the K-factor, and atan2 can therefore be computed solely with shifts and adds. The actual algorithm consists of a loop where at each iteration we examine the sign of y and apply a rotation by either $-\varphi_k$ or $+\varphi_k$, whichever angle rotates the vector towards the origin.

There remains the question of whether this process converges, and whether there is a bound on the approximation error. Indeed, it can be shown that

$$\forall\theta \in (-\frac{\pi}{2}, \frac{\pi}{2}) \qquad \exists(\psi_k) \in \{-1, 1\} \qquad \text{such that} \qquad \theta = \sum_{k=0}^{\infty} \psi_k \varphi_k \tag{4-3}$$

This means that for all angles θ in quadrants 1 and 4, there exists a sequence of rotation *directions* ψ_k=+/-1that will make the accumulated rotation angles converge to θ. The actual values of the angles φ_k are shown in Table 4-3. The error in the approximation at step k is always less than the next angle φ_{k+1}.

4.2.2 RL program for atan2

Figure 4-2 shows the complete program text of the CORDIC atan2 function (catan2), as programmed in the RL language, using the fixed point data type.

```
/************************ catan2.k ************************/1
/*            Copyright (c) Lars E. Thon 1988            */2
fix      catan2 (sin, cos)                                 3
fix      sin, cos;                                         4
{                                                          5
    register int    k;                                     6
    register fix    x, y;                                  7
    fix             theta;                                 8
                                                           9
    /* Start Cordic. The first step takes care of quadrants10
      2 and 3 */                                           11
    if (cos < 0) {                                         12
        if (sin >= 0) {                                    13
            theta = FIXPIHALF; x = sin; y = -cos;          14
        } else {                                           15
            theta = -FIXPIHALF; x = -sin; y = cos;         16
        }                                                  17
    } else {                                               18
        theta = 0; x = cos; y = sin;                       19
    }                                                      20
                                                           21
    /* Scale x,y so they don't overflow when amplified */  22
    x= (x>>1); y= (y>>1);                                  23
                                                           24
    /* The Cordic iterations work in quadrants 1 and 4 */  25
    for (k = 0; k <= NUMIT; k++) {                         26
        fix     xnew, ynew;                                27
        if (y > 0) {                                       28
            theta += ctable[k];                            29
            xnew = x + (y >> k); ynew = y - (x >> k);      30
            x = xnew; y = ynew;                            31
        } else {                                           32
            theta -= ctable[k];                            33
            xnew = x - (y >> k); ynew = y + (x >> k);      34
            x = xnew; y = ynew;                            35
        }                                                  36
       }                                                   37
                                                           38
    /* The accumulated angle is returned as result */      39
    return theta;                                          40
}                                                          41
```

Figure 4-2 RL code for the atan2 function computed using the CORDIC method

Variable	Requirement	Representation (wordlength w)	
x	$abs(x) < 2^S$	$rep(x,w) = integer(x*2^{w-1}/2^S)$	
Operation	**Scales**	**Requirements**	**Result scale**
$x_1 \pm x_2$	s_1, s_2	$s_1 = s_2$	$s = s_1 = s_2$
$x_1 * x_2$	s_1, s_2	none	$s = s_1 + s_2$
x_1 / x_2	s_1, s_2	none	$s = s_1 - s_2$
Class	**Range**	**Actual scale**	**Binary point pos.**
lengths (p_x, a_i, d_i)	± 2048	2048	12.8
lengths^2	$\pm 2048^2$	2048^2	23.(-3)
angles (θ_i)	$\pm\pi$	π	3.17 (approx.)
units (s_i, c_i, r_{ik})	± 1	2	2.18

Table 4-4 (a) Fixed point representation (b) Rules for fixed point computation (c) Scaling classes for the variables of the IPO computation

One of the reasons for this fairly detailed exposition on atan2 CORDIC algorithm is to highlight a small detail of the algorithm which turns out to be of special significance to the architecture design. The detail in question is the occurrence of the operations of the type **(x>>k)** inside the for-loop, where **k** is the loop index. This will be referred to later as the *variable shift* operation.

The RL program for the PUMA IPO algorithm consists of 5 functions and a main program. Total code size is 658 lines of text, out of which 263 lines contain one or more actual RL statements (after removing comments, blank lines, etc.). It is clear that the IPO algorithm is nontrivial both in size and complexity, and therefore constitutes a good test case for the C-to-Silicon system. The complete RL code (puma.k, etc.) can be found in Appendix A.

4.3 Fixed point computation

Since our target processor does not support the floating point data type, it is important to perform a careful analysis of how to implement the algorithm efficiently in fixed point arithmetic. The goal

is to minimize the wordlength w (sometimes denoted N). The parameters determining the wordlength are the *precision* and *range* requirements of the variables in the program.

The basic concepts of fixed point computation are reviewed in Table 4-4. The first sub-table explains how fixed point numbers are represented, that is, how to compute an integer which will contain the appropriate bit pattern when converted to 2C (two's complement) binary form. For a given variable x, if abs(x)<2^s then a scale $S=2^s$ can be used. The basic tradeoff is to choose 2^s large enough to give sufficient range and small enough to give sufficient accuracy. Scaling by powers of 2 is convenient, because the processor can easily convert between numbers of different scale by performing shift operations. Converting scales will, however, lead to loss of significant bits as the bits are shifted out on the right-hand side. It is sometimes handy to use other scale values than powers of two, for example π as the scale value for angles, as will be explained below.

Since RL only allows power-of-two scales, other scales must be simulated by doing the appropriate scaling operations outside the chip and declare the variables to be of type *fix:0*. In fact, it was easier in the program **puma.k** to declare ALL variables to be of the type *fix:0*. Note that constants and input data must be prescaled according to the third section of Table 4-4.

The second section of Table 4-4 lists the rules for computing the resulting scale when applying an arithmetic operator to a pair of fixed point numbers, and the requirements for the operands. Adding two numbers only makes sense if they have the same scale, and multiplying to numbers yields a third number with a different scale.

The third section of the table lists the scales used for the variables in the **puma.k** RL program. The scale 2048 for *lengths* is chosen because the maximum reach of the robot is about 900mm. We cover this with a safety factor of two. Products of lengths get the scale 2048^2 for consistency. The reason for scaling angles to π is the following: The formulas for θ_1 and θ_3 both involve the subtraction of two angles. Since each of the two angles may be in $[-\pi,+\pi]$, the result can in principle be anywhere in the range $[-2\pi,+2\pi]$. Hence there will be a need to reduce the value

modulo π so that it falls inside $[-\pi,+\pi]$. If we use π as the scale of the angles, the modulo reduction comes for free during the subtraction (due to the modulo arithmetic of the processor when operating in non-saturating mode).

It would also seem reasonable to use scale 1 for the c_i and the s_i: We know that a sine/cosine will always be between -1 and 1, so a scale of 1 should be sufficient. This is tempting, but consider the effect of inaccuracy: If cos=0.999 becomes cos=1.001, the value will wrap around and become cos=–0.999. These two values are not at all "close", because they correspond to very different angles. (This is *not* analogous to the situation with angle values, where $+179.99^\circ$ and -179.99° describes essentially the same angle.) Hence we decided to use a scale of 2.

The wordlength chosen was w=20. It was derived as follows: The target is to compute $\theta_1,...,\theta_6$ with an error of less than 0.05°, or 4.5 decimal digits. To achieve this, about 5.5 decimal digits of precision is needed in the intermediate calculations. This corresponds to about 19 bits. Adding one bit to account for the negative numbers we end up with w=20.

4.4 High-level simulation

Using the above scaling scheme, the IPO computation was programmed in RL, using CORDIC subroutines for the elementary functions. To make sure that the program and the scaling were sound, we used the KT tools to perform first floating point and then fixed point simulation.The simulations showed that the program works well unless the specified goal frame is close to a singularity ([craig86] p146). It should be noted that a floating point program will also produce inaccurate results in this case. Moreover, the loss of accuracy is often accompanied by the property that the position/orientation is only weakly dependent on the value of the particular inaccurate angle. It is also possible to detect during the computation that we are close to a singularity, and issue an error signal. The typical case had an angle error of less than 0.02° for each one of the 48 angles when simulated using target positions/orientations generated with a random number generator Some results of the high-level simulation are shown in Table 4-9.

4.5 Architecture design and exploration

The Kappa architecture was used as the starting point for the architecture exploration. Kappa originated in audio (speech processing) applications [pope84], and was developed further for use in a PID robot joint controller [azim88]. See Figure 3-11 for a block diagram of the Kappa datapath. Starting with this datapath, I went through several changes to the architecture, each time trying to make inexpensive modifications that would improve the efficiency in executing the algorithm. It should be stressed that most of these changes only had to be done on paper or in the machine description file, as explained in Chapter 3. Hence it was possible quickly to evaluate a number of alternatives without expensive investment in logic or layout design.

For each variation of the architecture, the C-to-Silicon system was used to collect several cost and performance metrics: Number of basic blocks, Total code size (static instruction count) and Total execution time in cycles (dynamic instruction count). From these results were also derived area measures. Area costs of any additional hardware blocks were also considered.

Architectural variations

The design alternatives that were considered are illustrated graphically in Figure 4-3 and also listed in Table 4-5. Each variation was created by making a set of choices between the pairs of alternative architectural features shown in the figure. The details of the alternatives will be explained below. Note that some of the alternatives are not independent. For example, the question of whether or not to use a subroutine for multiplication is relevant only if the array multiplier is not used.

The first pair of alternatives from Figure 4-3 is the use of a shift-add multiplication strategy versus a full array hardware multiplier. The second pair of alternatives is relevant if the shift-add strategy is used. The alternatives are between expanding (in-place) all multiplications into N (the wordlength) shift-add operations, or to provide the architecture with a subroutine call/return capability so that one block of code can be used for all the multiplications.

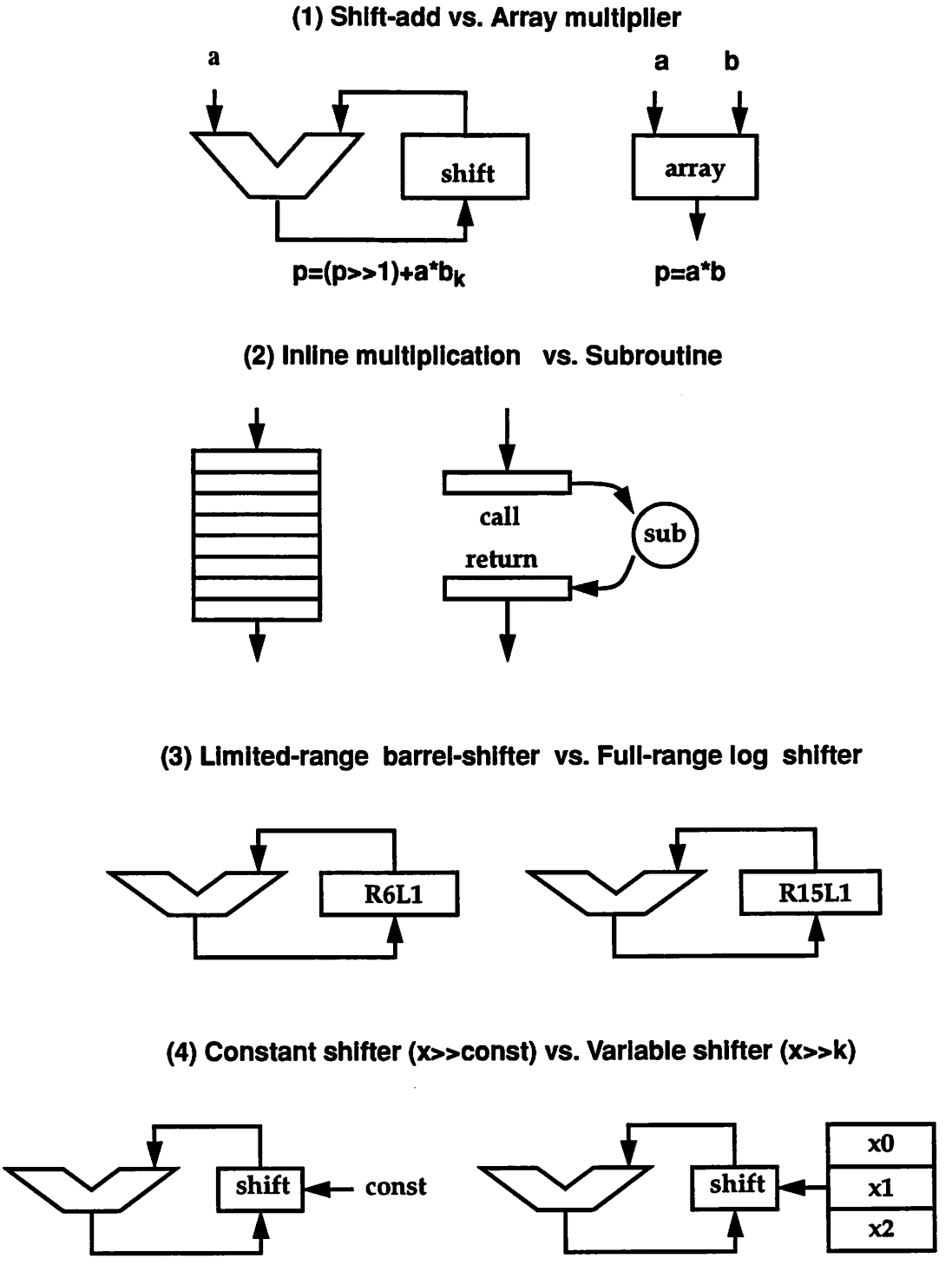


Figure 4-3 Small architecture variations had significant impact on the PUMA chip performance and cost (area)

The third pair of alternatives is concerned with shifter types. The notation R<n>L<m> is used to denote a shifter that can shift (up to) n places to the right or m places to the right in one cycle. The alternatives are an R6L1 limited range barrel shifter and a R15L1 full range logarithmic (1-2-4-8) shifter.

Finally, the tradeoff between having a constant shifter and a variable shifter was considered. A constant shifter is a shifter which can only shift by an amount which is fixed in the instruction at compile time (also known as an *immediate* constant). In contrast, the variable shifter also can shift by an amount which comes from a register, in this case any one of the index registers of the address unit.

Evaluation of alternatives

Table 4-6 shows the results of evaluating the architectural alternatives. The first 4 entries (0-3) assume that the architecture has no array multiplier, i.e. that shift-add multiplication is used. Entries 0-1 in the table contrast the use of inline code versus a subroutine call for multiplication. Using the subroutine means an increase in the number of basic program blocks, but a large decrease (34%) in code size, since one piece of code is shared by all the multiplication operations. Since the architecture has a low-overhead subroutine call, there was essentially no difference in execution time. Additional hardware cost (a small stack circuit) is minimal.

Now consider entry 2. Compared to entry 0, the difference is that the constant shifter has been replaced by a variable shifter. This has an even more dramatic effect than introducing subroutine capability. The code size is down by 41% compared to case 0. The explanation is simple: The catan2 RL program (Figure 4-2) contains a **for**-loop where the variable **k** is the loop index. The variable **k** is also used inside the loop to specify the amount of shift, as in the expression (**x>>k**). What happens if the architecture does not support variable shifts? Then the loop cannot be compiled as written. The loop has to be *unrolled*, meaning that its contents must be duplicated 17 times (NUMIT=16), each time with a different value of **k** inserted as a constant. Similar unrolling is necessary in the other CORDIC routines. This becomes very expensive in terms of static

Alternative 1	Alternative 2
array multiplier (possibly pipelined)	iterative shift/add multiplier
inline multiplication code	subroutine call
R6L1 shifter	R15L1 logarithmic shifter
constant shifter (r>>I)	variable shifter (r>>x[I])

Table 4-5 Design tradeoffs affect layout area, static instruction count and dynamic instruction count

Case	Shifter	Mult type	Num blocks	Code size	ΔA (mm2)	Cycles
0	constant	inline code	201	2924	0.00	18156
1	constant	subroutine	255	1920	-13.63	18156
2	variable	inline code	66	1720	-18.17	18156
3	variable	subroutine	120	717	-31.78	18156
4	variable	array (delay 1)	66	683	-27.78	9192
5	variable	array (delay 1*)	66	642	-28.37	9028
6	variable	array (delay 3*)	66	723	-27.20	9352

Table 4-6 Effect of design decisions on code size (static instruction count) and code execution time (dynamic instruction count)

instruction count, as evident from Table 4-6. A solution is to introduce an extra instruction bit which selects between the immediate constant and the lower 4 bits of an index registers X0-X2 as source for the shift amount. This was a very inexpensive addition to the hardware, but it paid off greatly by reducing program (and hence ROM) size, without changing the execution speed.

Entry 3 shows the combined effect of a subroutine stack and a variable shifter. The code size is down by 75%. Note that the cycle count remains the same, as we are still executing the same sequence of datapath operations.

Entries 4-6 show the results of introducing an array multiplier unit as part of the datapath (see

Figure 4-5). First of all, the number of blocks is reduced because the multiplication subroutine calls go away. More impressive is that the execution time is cut in half. A reduction was expected, considered that the program has a large amount of multiplications. The code size, however, shows little improvement. The small reduction consists mostly of the space taken up by the former multiplication subroutine.

The three different cases 4-6 were done as an experiment to see whether the introduction of pipeline delay and/or extra input multiplexers would make a large difference in the performance. Case 4 assumes that each multiplier input can only come from one particular source (e.g. the left input from mbus and the right input from the ram). Cases 5-6 assumes that either input can come from either source (marked with a * in the table), which could be important for example when squaring a number. We observe that neither the pipeline delay nor the input routing had much of an impact on either static or dynamic instruction count. This is positive evidence that the compiler is doing a good job at both scheduling and of data routing.

Discussion

The table shows that alternative 3 is a clear winner area-wise, and with the same speed as alternatives 0-2. The alternatives 4-6 provide higher speed at the cost of additional area. Considering the cost and design effort for a 20x20 array multiplier, we decided against using one. Previous layout indicated that a 20x20 array would be at least 2.54x2.60mm in 2μm technology, plus a substantial overhead in hooking up the busses between the multiplier and the datapath. The original R6L1 shifter was also rejected. The R6L1 shifter does not provide adequate shifts for the CORDIC operations at the given wordlength (N=20), because shifts up to (NUMIT–1)=15 are required. Repeated shifts would then be necessary. More serious is that variable shifts would not be possible, as there would be no easy way to break a variable shift into 3 repeated shift operations. Finally, the logarithmic shifter does not require any decoding of the shift amount, making the datapath implementation easier.

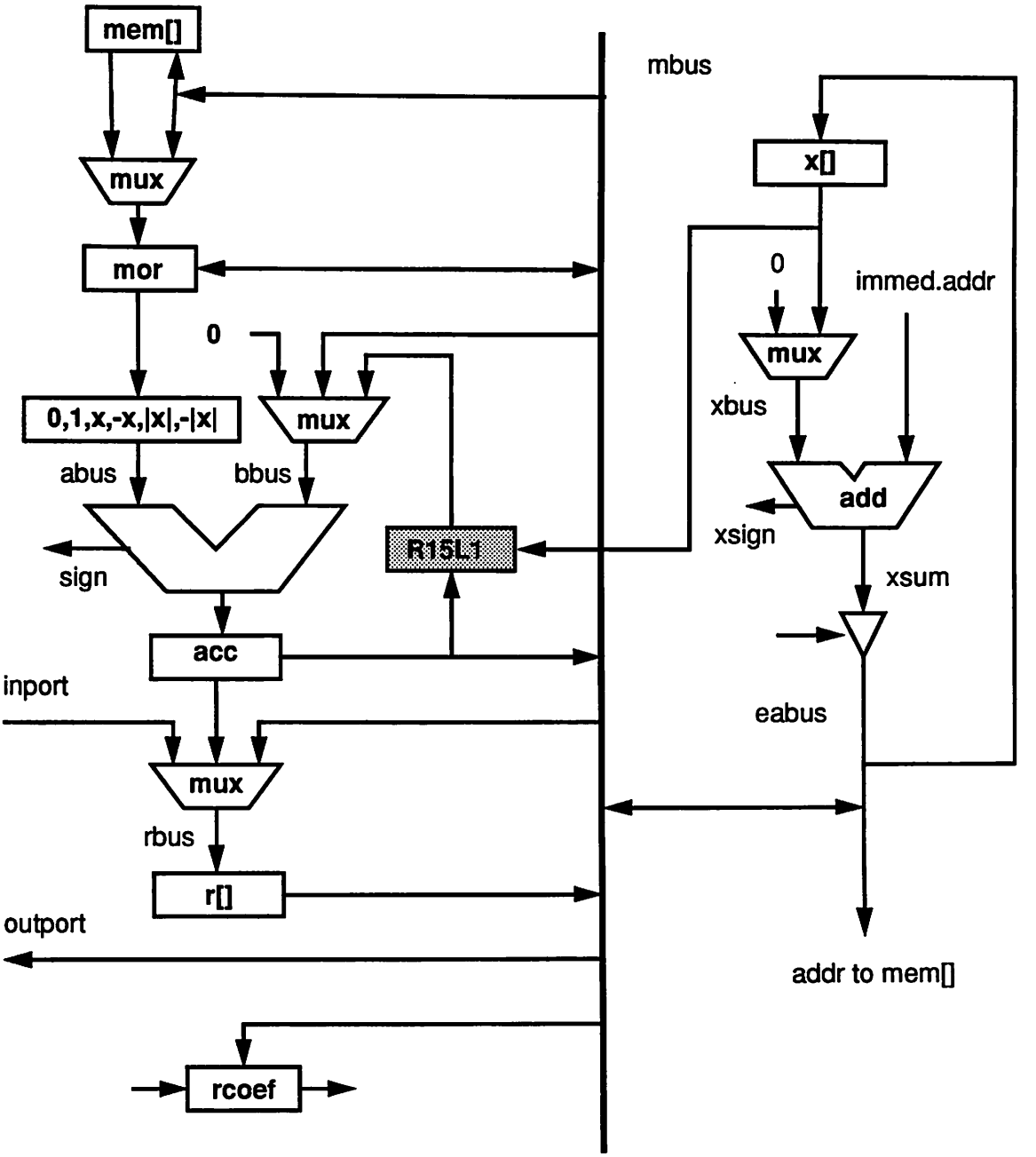


Figure 4-4 The PUMA datapaths

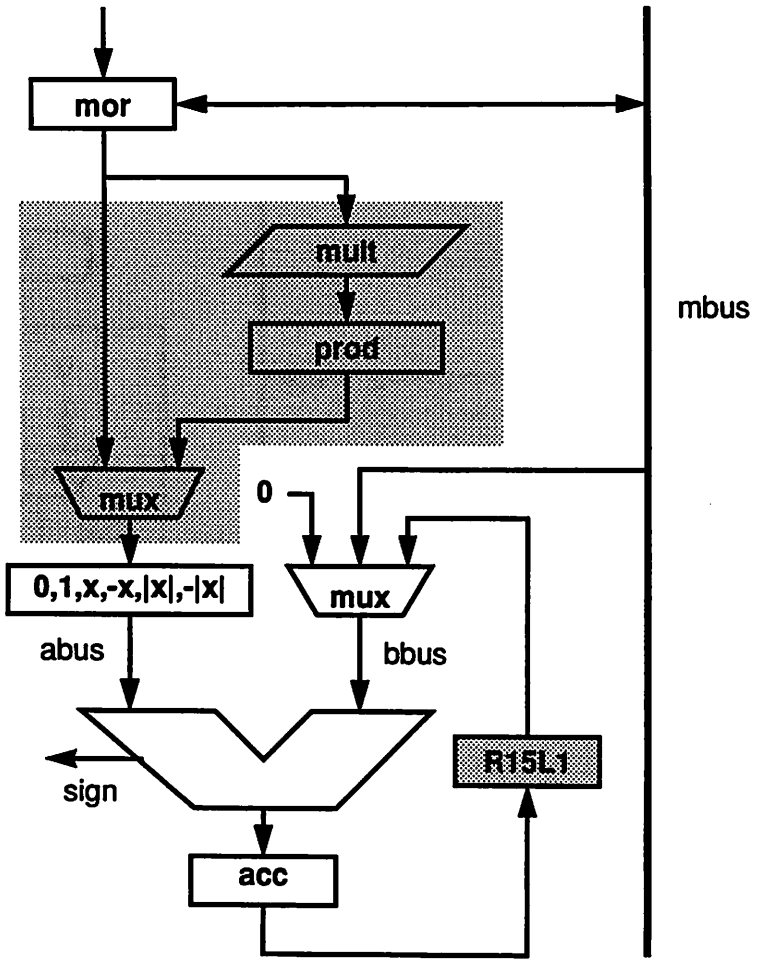


Figure 4-5 Datapath with array multiplier (only relevant part of datapath is shown)

Conclusion

In summary, we decided to use the architecture with the R15L1 logarithmic shifter, variable shift capability, a subroutine stack and no array multiplier. The final PUMA datapath is shown in Figure 4-4. Assuming that the chip can run at 10MHz clock rate, the IPO equations can be solved at a rate of $10^7/18156=551$ times per second. This is sufficient for most purposes (most robots have a control loop that runs at less than 100Hz, and the IPO loop typically is run at a slower rate than the control loop).

4.6 Chip verification and layout design

The logic and layout design of PUMA was carried out in the LAGER environment, following the methodology described in section 3.10 (p71). The PUMA chip core consists of 221 macrocells, with 6 macrocells at the top level and 7 levels of hierarchy.

Most of the design was created by translating or redesigning old Kappa SDL files, which were used in the previous LAGER-III system [shung89][shung91]. LAGER-III was based on Franz Lisp and the Flavors database and the current LAGER is a thorough modernization based on the C language and the OCT database. Creating PUMA from Kappa was a complicated task, as the old design had never been simulated and there were numerous bugs that had to be discovered and fixed both in the logic design and in the various library modules and leafcells. The architectural modifications turned out to be fairly easy to accomplish.

Because of the size of the microcode (670 lines) it became necessary to make the microcode ROM double-wide and insert an output multiplexer to select the appropriate half of the bits. This made the chip floorplanning manageable at the cost of some delay and extra design time.

4.6.1 Logic-level simulation

The logic-level simulation was carried out using THOR and MakeThorSim (Chapter 3). Models for the library cells had to be developed and installed in some cases. MakeThorSim turned out to be a great help as soon as the program reached a stable condition. The main problem with THOR was the lack of delay modelling, which meant that there would often be multiple transitions of the same signal at the same time. This made it crucial that the models keep track of the time the signals last changed, to avoid infinite evaluation loops.

Debugging the logic design with THOR was a somewhat complex task, but was aided by some special THOR utility models that were especially developed for the PUMA simulation. For example, a *breakpoint* model was developed especially to watch the value of the **(blocknumber,**

Model name	Model function
breakpoint	stop simulation when (block, line) reaches certain value
file_binmon	print binary value to file every time input changes
file_hexmon	print hex value to file every time input changes
rename	dummy block to rename a signal to something more meaningful
watchdog_strobe	watch a strobe signal and print bus value when strobe is high
watchdog_break	watch a bus an stop simulation whenever a certain value appears
CPU type	**Simulation time**
SUN 4/60 (Sparc 1)	31:07 min (12:04 for the graphic analyzer) (100ms per clock cycle)
SUN 4/75 (Sparc 2)	17:04 min (6:13 for the graphic analyzer) (57ms per clock cycle)

Table 4-7 Special THOR utility models for PUMA debugging. Simulation execution times.

linenumber) pair and stop the simulation temporarily at any given location so that the user could check the state of the simulation and decide what to do next. Similarly, the *file_binmon* model was developed to watch buses and print the time and value of every change into a file for further analysis. A complete list of the utilities are shown in Table 4-7.

THOR is a fairly speedy simulator, allowing a complete simulation (~18000 cycles) of the RL program on the PUMA chip in less than 1 hour. Combined with MakeThorSim and the breakpoint and watchdog utilities, it formed a reasonably powerful environment where it was possible to diagnose and fix several bugs per day, as long as the symptoms were not too obscure. This is not to say that the debugging environment would work well for the casual designer, but it was sufficient for an experienced designer. Examples of the execution speed of THOR are shown in Table 4-7.

4.6.2 Switch-level simulation

The PUMA chip layout was simulated extensively using the IRSIM switch-level simulator (section 3.13 on page 90). Circuit descriptions extracted from the PUMA layout were used as input for IRSIM, leading to more accurate delay and timing results than what is possible in the

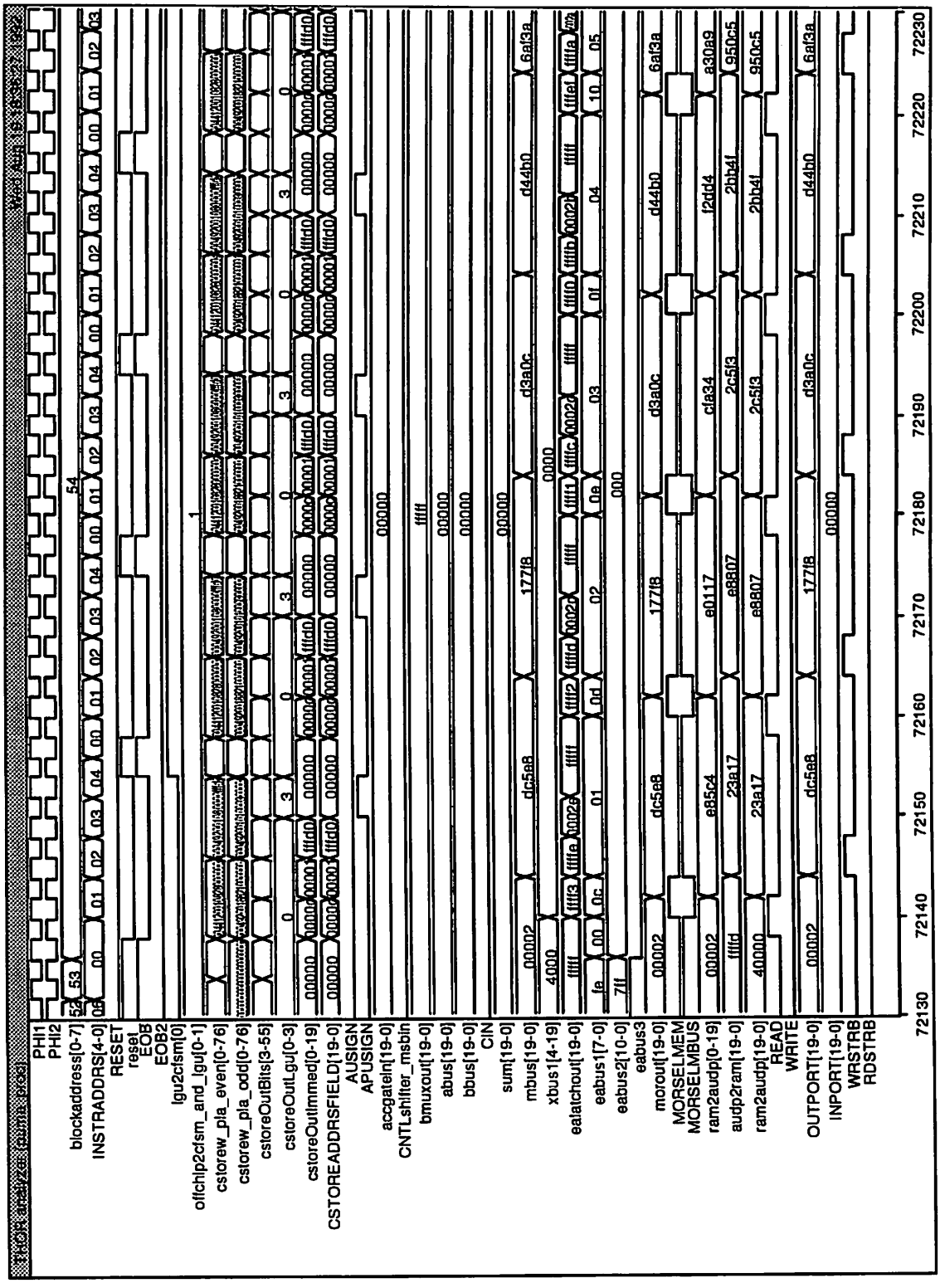


Figure 4-6 THOR simulation of PUMA. The analyzer shows the output phase of the RL program (starting at OUTPORT=dc5e8[hex])

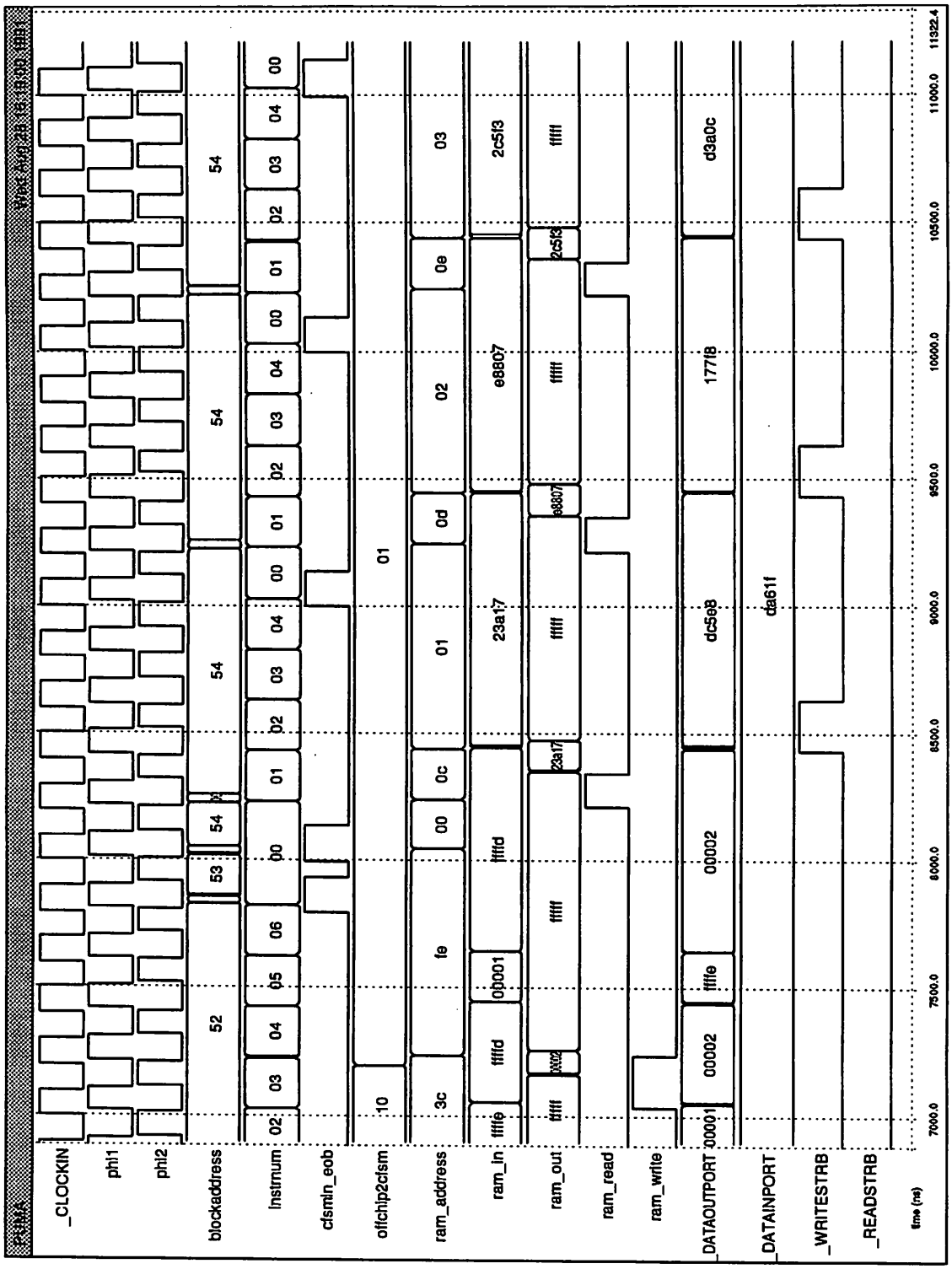


Figure 4-7 IRSIM simulation of PUMA. The bus _DATAOUTPORT is the same as the bus denoted _OUTPORT in the THOR simulation

logic level simulation. The execution of the microcode, including data input and output, was simulated in its entirety, and the results have been verified against those computed in the high-level and logic-level simulations.

4.6.3 Electrical rule checking

Electrical design rules are design rules that apply to the electrical interconnection of layout geometries, as opposed to topological properties of the geometries. Examples of electrical design rule violations are listed in Table 4-8. The MAGIC design rule checker is only a topological checker and does not know about electrical design rules, partly because such rules tend to vary (especially between analog and digital designs).

Because of the complexity of PUMA and the many levels of hierarchy, it became difficult to check electrical design rules manually. A program named **erc** (electrical rule checker) was developed which automated this task. **erc** checks for all the errors listed in Table 4-8.

4.6.4 Chip testing

The PUMA chip was tested using a Tektronix DAS 9100 pattern generator/data acquisition unit. There was complete functional agreement between the measurements of the chip and the THOR/ IRSIM simulations. Table 4-9 shows all the simulation and chip measurement results for a random test case. Note that the biggest error between the floating point and fixed point simulation is 0.01

Geometries	Electrical rule violation
pwell/nwell	floating (not connected to anything)
pwell/nwell	tied to Vdd/GND instead of GND/Vdd
pwell/nwell	tied to something other than GND/Vdd (e.g. a signal)
Vdd/GND	unconnected
Vdd/GND	shorted together

Table 4-8 The electrical design rule checked by the **erc** program

degrees, indicating that the fixed point formulation is sound. Also note that THOR, IRSIM and the chip itself were 100% in agreement, and that the difference between the high-level fixed point simulation and the chip is at most 0.01 degrees.

IRSIM is usually a conservative predictor of chip speed. For the PUMA chip, the simulation worked up to 6.5 MHz (using untuned, conservative transistor parameters for a 2μm process). Measurements on the actual chip showed that it was fully functional only up to 4.6MHz. The first block to fail was the RAM. The datapaths, the program ROM and the block sequencer were all functional up to 8.2MHz at 5V. The discrepancy between IRSIM and the measurements is due to the fact that the circuit extraction does not include wire resistances (IRSIM does not handle wire resistance). Resistance extraction is important when there are long polysilicon lines in the layout. This was the case in the RAM and PLA modules. These modules had been optimized with respect to area by using polysilicon lines instead of metal lines in certain key circuits.

Source of results	θ1	θ2	θ3	θ4	θ5	θ6
Floating point (kt -float)	-50.11	33.05	-62.40	-61.46	150.40	-167.04
	153.67	122.03	-62.40	39.41	167.04	92.74
Fixed point (kt -fix)	-50.11	33.05	-62.40	-61.47	150.40	-167.05
	153.67	122.03	-62.40	39.40	167.04	92.74
Error (Fix - Float)	0.00	0.00	0.00	0.01	0.00	0.01
	0.00	0.00	0.00	0.01	0.00	0.00
THOR, IRSIM, and measured on the chip (hex)	dc5e8	177f8	d3a0c	d44b0	6af3a	89368
	6d460	56c6e	d3a0c	1c05c	76c84	41f34
Converted to degrees	-50.11	33.04	-62.40	-61.46	150.40	-167.04
	153.67	122.03	-62.40	39.41	167.04	92.74

Table 4-9 Simulation and test chip measurement results. Two out of eight solutions are shown for a randomly generated test case

It is clear that by spending more area and using metal lines, the speed can be increased substantially. In fact, a subsequent C-to-Silicon chip design [mmar92] simulates at 80MHz. The chip has not returned from fabrication at the time of this writing. The main reasons for the speed improvement is that the RAM has been reworked to avoid the polysilicon lines, the overall smaller RAM size (26 words), the use of newer technology (1.2 µm versus 2.0 µm) and that the program running on the machine is about 1/7 the size of PUMA's program. Nevertheless, the results indicate that the C-to-Silicon system can also be used for high-performance designs.

Characteristic	Value	Comment
wordlength	20	size of all the datapaths
cstore PLA	13x649x77	49973 bits (microcode ROM)
lgu PLA	16x32x8	*inputs*× *minterms*× *outputs*
cfsm PLA	21x171x26	*inputs*× *minterms*× *outputs*
data RAM	172x20	3440 bits
technology	2µ	scalable CMOS (nwell)
width x height	9864 x 9608	λ^2
transistors	99384	
pads	126	
package	208 pin PGA	

Table 4-10 Physical design characteristics of the PUMA chip

Block	Speed	Comment
chip (IRSIM)	6.2 MHz	without resistance modelling
chip	4.6 MHz	limited by RAM speed
RAM	4.6 MHz	long poly lines (area optimized)
datapath	8.2 MHz	
control store	8.4 MHz	
control fsm	8.4 MHz	
program counter	>10 MHz	

Table 4-11 Measurements on the PUMA chip

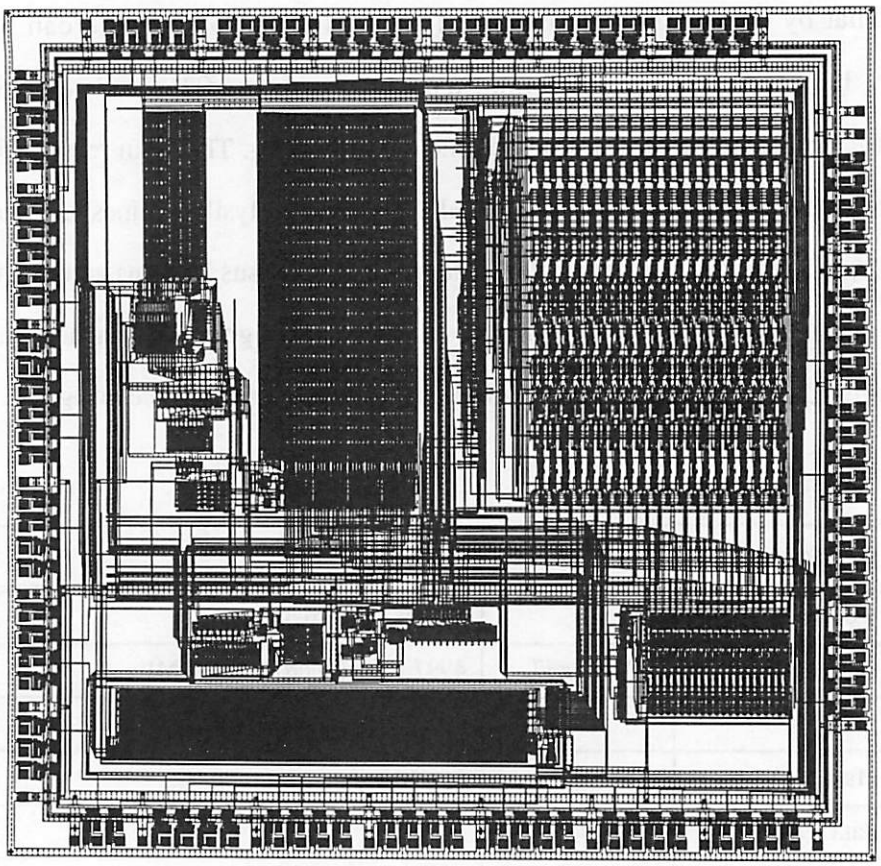


Figure 4-8 CIF plot of the PUMA chip

4.6.5 Physical design results

The IPO algorithm is quite complex compared to the algorithms employed in many DSP applications. Therefore, the resulting microprogram is large (about 670 lines after compression), yielding a chip of 9.8 x 9.6 mm^2 in 2μ technology. Table 4-10 sums up the key results of the physical chip design. The completed chip is shown in Figure 4-8. It was fabricated through the MOSIS service.

4.7 Summary

The design of the PUMA chip demonstrated the feasibility of the C-to-Silicon system, and served as a driving force during the development of the system. All the main features of the system were exercised and tested during the design of PUMA. Easy architecture exploration proved to be an indispensable feature for performance and tradeoff evaluation. High-level simulation (floating and fixed point) were used to develop a numerically sound algorithmic formulation of the IPO computation. Automatic layout generation was successfully employed to create the complete chip. Logic and switch level simulation support tools were used to debug the design and verify the layout and timing of the processor. PUMA has been successfully fabricated and tested, and C-to-Silicon is now currently being applied to other design projects.

CHAPTER 5

SOLVING nxn POLYNOMIAL SYSTEMS

The *general* IPO problem for 6R robots will now be considered. It is markedly more complex than the PUMA case, and constitutes a suitable test case for investigating architectures and implementations of Numerical Processing systems. Since the general IPO problem can be cast as solving a system of n polynomial equations in n unknowns, I will start out by describing the development of a family of C software programs (named ConsolC) for solving such systems, using the homotopy continuation method described in Chapter 2. There were several reasons for developing the ConsolC software:

- Portability. Existing software was only available in the Fortran 77 language, which is not widely used in the UNIX workstation environment. We also wanted to take advantage of the UNIX programming environment, since frequent software modifications were expected.
- DSP chip compilers. We wanted to investigate how the IPO computation would fare on commercial DSP chips. C compilers are available for most current DSP architectures, whereas Fortran is much less common. By developing a C package, the migration to DSP chips is much simpler.

- Algorithm insight. It was assumed that developing and running the C code would provide additional knowledge about the algorithms, both structurally and numerically. Such knowledge can be applied to identify the time-critical parts of the computation and to determine how these could benefit from an application specific architecture.

The original Fortran Consol software was developed by Morgan [morgan87a]. This chapter describes the development of the C version, starting with a description of the algorithm and how various software modules were designed to implement the computation. In particular, the software features especially required for the IPO computation are described. An important task was to test the software on some realistic inputs, so as to assess the robustness of both the method and the implementation.

Following the development of ConsolC, the package was used to investigate the general IPO computation. The purpose of the investigation was to learn about the numerical properties and the complexity of the algorithm, since this knowledge is important when considering Application Specific Processor implementations. In particular, convergence, path lengths, variable ranges (max values) and execution profiles were scrutinized. Some experiments with fixed point arithmetic for selected parts of the algorithm is also included. The fixed-point version of the ConsolC package was created by translating all the C code into C++ and developing a special **Fix** package in C++ to perform the fixed-point arithmetic. This was much simpler in C++ than it would have been in C due to the *operator overloading* mechanism available in C++.

5.1 Software architecture of ConsolC

The purpose of ConsolC is to track the paths of the homotopy continuation

$$\mathbf{h}(\mathbf{x}, t) = (1-t)\cdot\mathbf{g}(\mathbf{x}) + t\cdot\mathbf{f}(\mathbf{x}) \qquad t \in [0, 1] \tag{5-1}$$

where $\mathbf{h(x,0)=g(x)}$ is the starting system and $\mathbf{h(x,1)=f(x)}$ is the goal system. ConsolC uses Newton's method to move along each path as explained on page 36. At each step, the program solves the equation

$$0 \approx \mathbf{h}(\mathbf{x}_0, \Delta t) + \mathbf{D}\mathbf{h}_x(\mathbf{x}_0, \Delta t)\,\Delta \mathbf{x} \qquad (5\text{-}2)$$

and uses the value of Δx to move to the next point. A generic flowchart for the programs in the ConsolC family is shown in Figure 5-1, and a listing of the main subroutines or modules are given in Table 5-1. The steps of the algorithm should be familiar from Chapter 2: The first task is to read

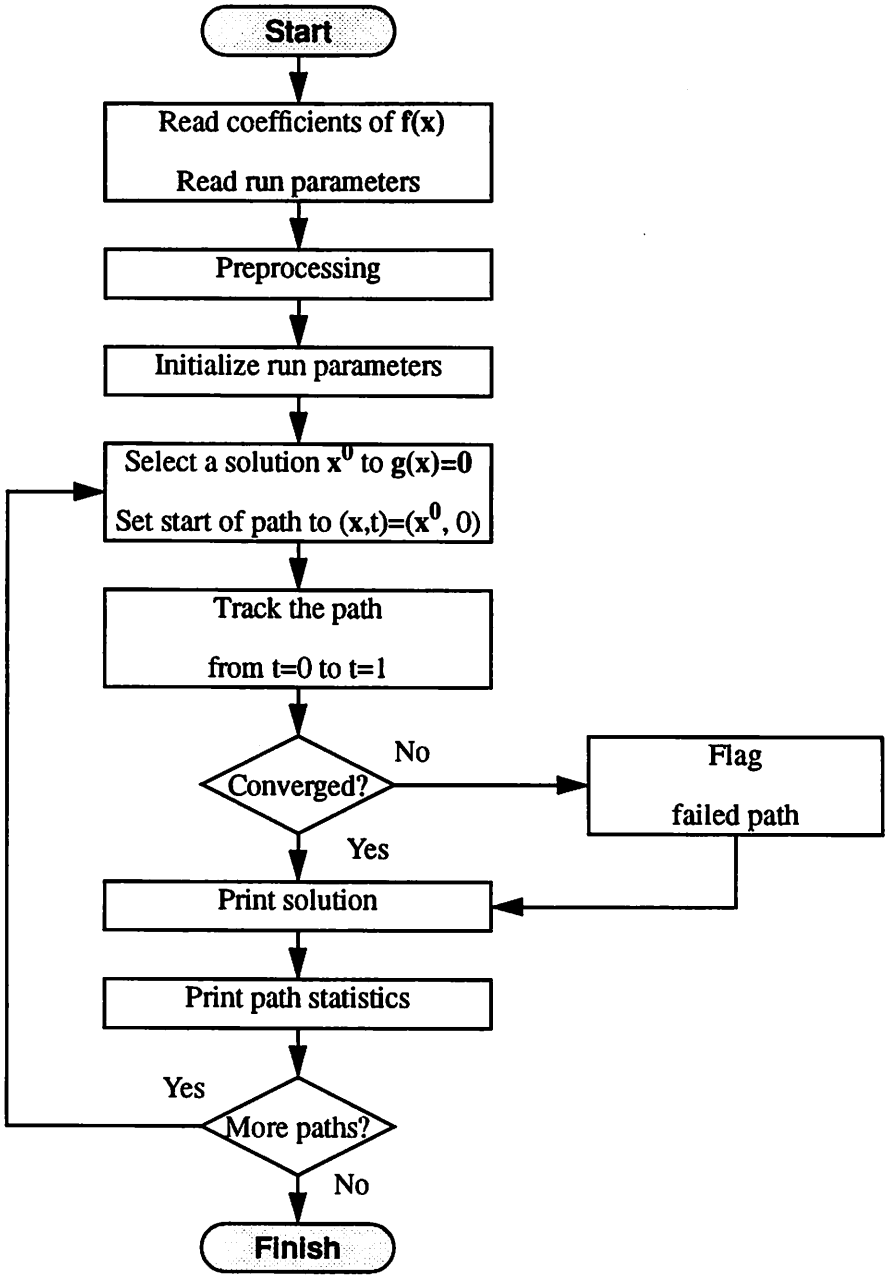


Figure 5-1 Generic flowchart for ConsolC programs

Function	Software module	Function	Software module
Start	consol8.c	Set start of path to $(x,t)=(x^0, 0)$	consol8.c
Read coefficients of f(x)	inputa.c	Track the path from t=0 to t=1	consol8.c, predict.c, correct.c
Read run parameters	inputb.c	Check convergence	stepcheck.c
Initialize run parameters	consol8.c	Print solution	postproc.c
Select solution to g(x)=0	startpoint.c	Print path statistics	postproc.c

Table 5-1 Software modules of the ConsolC family

Homotopy	Formulation	Source	System size
256 path	0- or 1-homogenous	[morgan84]	8x8 complex, 16x16 real
96 paths	2-homogenous	[morgan86]	8x8 complex, 16x16 real
64 path	2-homogenous	[morgan87b]	8x8 complex, 16x16 real
16 path	2-homogenous	[wampler89]	11x11 complex, 22x22 real

Table 5-2 Different homotopies used to formulate and solve IPO equations

the coefficients of the equation we want to solve, and to set runtime parameters such as desired path tolerances and the number of steps that should be allowed before a path is considered divergent. Preprocessing can mean a number of tasks. In the case of the IPO equations it is used to convert a goal point and a set of Denavit-Hartenberg parameters into the coefficients of the goal system. The main loop of the program consists of selecting a new starting point, tracking the path from t=0 to t=1 and recording the solution (endpoint) and the path statistics.

5.2 ConsolC variants

As mentioned in Chapter 2, there are a variety of formulations of the IPO polynomial system, with a varying number of paths to track. Table 5-2 shows the most useful systems currently known The first 3 systems (256,96,64) were known at the time ConsolC was developed, whereas the 16-path method is a more recent development.

The ConsolC package contains programs suitable for each of the 256/96/64 formulations, as well as generic variants that can handle any type of polynomial system.Table 5-3 contains a listing of the various programs in the ConsolC package. A description of the different versions follows in

Program name	Properties of program
consol6r	f,g are both 2 quadratic equations in 2 unknowns
consol8qp	f,g=generic quadratic, 1-h projective transform
consol8tp	f,g=tableau, 1-h projective transform
robot8tp	f=tableau[robot input], 1-h projective transform, 256 paths
robot8p1	f=hard-coded robot, g=generic quadratic, 1-h projective transform, 256 paths
robot8p2	f=hard-coded robot, g=generic quadratic, 2-h projective transform, 256 paths
start96p2	f=hard-coded robot 2-h, g=Sommese 96 path start system with 2-h projective transform, h=gamma factor
robot96p2	f=hard-coded robot, g=generic robot, h=gamma factor, 2-h transform, 96 paths
robot64p2	f=hard-coded robot, g=generic robot, h=gamma factor, 2-h transform, 64 paths
robot64p2g	f=hard-coded robot, g=generic robot, h=gamma factor, 2-h transform, 64paths, LU=gauss with full pivot
robot64p2gp	f=hard-coded robot, g=generic robot, h=gamma factor, 2-h transform, 64 paths, LU=gauss with row pivot
p1tran	Transform solutions from Euclidian space to 1-h projective space
p2tran	Transform solutions from Euclidian space to 2-h projective space

Table 5-3 Programs in the ConsolC package

the next few sections.

5.2.1 General polynomial solvers

The most general form of ConsolC is known as consol8t2p. This version allows both **f** and **g** to be specified in "tableau" (or tabular) form. This means that each equation can have the completely general polynomial form

$$f_i(\mathbf{x}) = \sum_{t=1}^{t_i} a_{it} x_1^{m_{it1}} x_2^{m_{it2}} \ldots x_n^{m_{itn}} \qquad (5\text{-}3)$$

For each equation, the program reads first how many terms t_{in} the equation contains. Then for each term, it reads the coefficient a_{it} and then the exponent m_{itk} (possibly 0) for each of the k=1:n variables in the system (or n+p variables in the case of a p-homogenous transformed system). The tableau form is practical for general investigations but the evaluation of the functions **f,g,h** and their Jacobians are slow. Since the IPO problem involves quadratic equations, `consol8t1p` was instead designed to solve only such systems but in a faster manner by hardcoding the quadratic form into the function evaluation code. The general polynomial solvers typically uses a starting system of the form

$$g_i(\mathbf{x}) = p_i x_i^{d_i} + q_i \qquad d_i = \deg(f_i) \qquad i=1:n \qquad (5\text{-}4)$$

but as we shall see later, it is sometimes beneficial to use more specialized starting systems.

A simple version of ConsolC that can only do systems of 2 equations in 2 unknowns and of 2nd degree is known as `consol6r`. This program was initially used to explore the continuation method and to generate examples of continuation paths. An example of the 4 continuation paths (each with 2 complex variables) generated by `consol6r` is shown in Figure 5-2. The particular system used in this example was $\mathbf{f}: \mathbf{C}^2 \rightarrow \mathbf{C}^2$ given by

$$f_1(\mathbf{x}) = a_{11}x_1^2 + a_{12}x_1x_2 + a_{13}x_2^2 + a_{14}x_1 + a_{15}x_2 + a_{16} \qquad (5\text{-}5)$$

$$f_2(\mathbf{x}) = a_{21}x_1^2 + a_{22}x_1x_2 + a_{23}x_2^2 + a_{24}x_1 + a_{25}x_2 + a_{26} \qquad (5\text{-}6)$$

As starting system, consol6r always uses the system

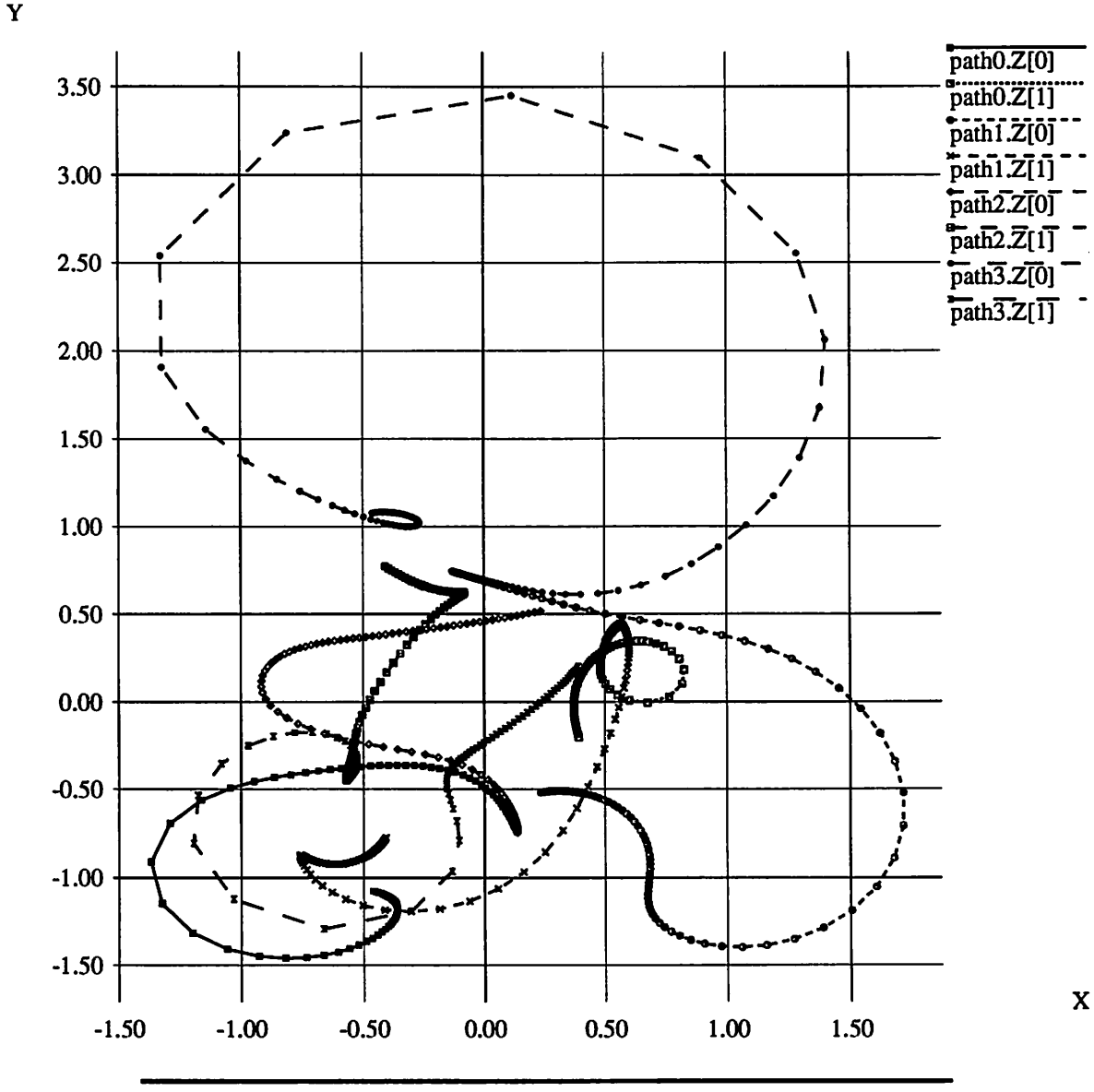


Figure 5-2 Example of continuation paths in the complex plane. These particular paths are for 2 equations of 2nd degree in 2 unknowns. The complex variables are x1= Z[0] and x2=Z[1] in the figure. There a 4 paths (8 trajectories) corresponding to the 4 solutions of the system

Coef	Real part	Imaginary part	Coef	Real part	Imaginary part
p1	0.12324754231	0.76253746298	q1	0.58720452864	0.01321964722
p2	0.93857838950	-0.99375892810	q2	0.97884134700	-0.14433009712

Table 5-4 Coefficients used for the random starting system in consol6r

Coefficient	Value	Coefficient	Value
a11	0.706286	a21	0.655203
a12	0.977589	a22	0.668641
a13	0.150897	a23	0.823866
a14	0.400489	a24	0.209538
a15	0.312564	a25	0.230056
a16	0.679323	a26	0.879568

Table 5-5 Coefficients used in the system solved in Figure 5-2. All coefficients are real, so that solutions generally exist in complex-conjugate pairs

Startpoint	Real	Imag	Endpoint	Real	Imag
x1	0.138191	-0.747731	x1	-0.461010	-1.075700
x2	0.568454	0.448099	x2	0.392620	-0.203800
x1	-0.138191	0.747731	x1	0.234650	-0.516840
x2	0.568454	0.448099	x2	-0.407340	-0.772730
x1	0.138191	-0.747731	x1	0.234650	0.516840
x2	-0.568454	-0.448099	x2	-0.407340	0.772730
x1	-0.138191	0.747731	x1	-0.461010	1.075700
x2	-0.568454	-0.448099	x2	0.392620	0.203800

Table 5-6 Starting points an end points for the 4 continuation paths of Figure 5-2. Note that there is considerable symmetry in the starting points due to the simple form of the starting system (5-7). This fact is reflected in the figure as some trajectories starting at the same point (By trajectory is meant the trace of *one* variable of the N=2 variables that make up a continuation path)

$$g_1(\mathbf{x}) = p_1 x_1^2 + q_1 \qquad g_2(\mathbf{x}) = p_2 x_2^2 + q_2 \tag{5-7}$$

where p_i, q_i are "random" complex numbers. In `console`, the actual numbers used for **g** and the coefficients of the system **f** solved in Figure 5-2 are shown in Table 5-4. One notable property of

the paths in Figure 5-2 is the widely varying spacing between the points along the path. This is partly due to the fact that consol6r is a simple program which does not use stepsize control but rather a fixed increase of the continuation parameter t at each step.

5.2.2 Robot polynomial solvers

The remaining programs of ConsolC are all specialized IPO solver programs that were independently developed and do not have any direct counterparts in the Fortran Consol package. Robot8t1p is a tableau-type program, but in this case the tableau is generated from robot parameter inputs (goal point, Denavit-Hartenberg) instead of outside the program (by the user). Hence it is easier to use for the IPO application but still slow.

The remaining IPO solver programs are specialized versions that have been optimized for speed at the cost of lesser generality. The goal system for all the IPO programs is

$$\begin{aligned} f_i(z) = {} & a_{i,1}z_1z_3 + a_{i,2}z_1z_4 + a_{i,3}z_2z_3 + a_{i,4}z_2z_4 + a_{i,5}z_5z_7 + a_{i,6}z_5z_8 + a_{i,7}z_6z_7 + a_{i,8}z_6z_8 \\ & + a_{i,9}z_1 + a_{i,10}z_2 + a_{i,11}z_3 + a_{i,12}z_4 + a_{i,13}z_5 + a_{i,14}z_6 + a_{i,15}z_7 + a_{i,16}z_8 \\ & + a_{i,17} \qquad i = 1 \rightarrow 4 \\ f_i(z) = {} & z_{2i-9}^2 + z_{2i-8}^2 - 1 \qquad i = 5 \rightarrow 8 \end{aligned} \tag{5-8}$$

These equations are the expansion of the shorthand equations (2-34). The expressions for the coefficients $a_{i,k}$ are generally very complicated functions of the goal point and the Denavit-Hartenberg parameters. The equations for $f_i(z)$ are available in Morgan's book [morgan87a] but in such a complicated form that it is unlikely one could copy them without making errors. I therefore decided to use the symbolic algebra program Macsyma [mac83] to derive the necessary equations from scratch, and then compare the result to Morgan's book. The results are shown in Table 5-12 (at the end of the chapter). They agree with Morgan's derivations. The parameters involved in the coefficient expressions in the table are the Denavit-Hartenberg parameters a_i, d_i, $\lambda_i = \cos\alpha_i$, $\mu_i = \sin\alpha_i$ and the position/orientation parameters (Chapter 2), reproduced here as (5-9).

$$T_i = \begin{bmatrix} R & p \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} c_i & -s_i\lambda_i & s_i\mu_i & a_ic_i \\ s_i & c_i\lambda_i & -c_i\mu_i & a_is_i \\ 0 & \mu_i & \lambda_i & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad T = \begin{bmatrix} R & p \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & p_x \\ r_{21} & r_{22} & r_{23} & p_y \\ r_{31} & r_{32} & r_{33} & p_z \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} l_x & m_x & n_x & p_x \\ l_y & m_y & n_y & p_y \\ l_z & m_z & n_z & p_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (5\text{-}9)$$

256-path versions

The programs robot8p1 and robot8p2 both are 256-path homotopy trackers for the IPO system. Their only difference is that they use a 1-homogenous and 2-homogenous formulation (Definition 2.3 on page 42) of the problem, respectively. The 1-homogenous form of (5-12) is

$$\begin{aligned} f_i(z) = {} & a_{i,1}z_1z_3 + a_{i,2}z_1z_4 + a_{i,3}z_2z_3 + a_{i,4}z_2z_4 + a_{i,5}z_5z_7 + a_{i,6}z_5z_8 + a_{i,7}z_6z_7 + a_{i,8}z_6z_8 \\ & + a_{i,9}z_1z_9 + a_{i,10}z_2z_9 + a_{i,11}z_3z_9 + a_{i,12}z_4z_9 \\ & + a_{i,13}z_5z_9 + a_{i,14}z_6z_9 + a_{i,15}z_7z_9 + a_{i,16}z_8z_9 + a_{i,17}z_9^2 \qquad i = 1 \rightarrow 4 \\ f_i(z) = {} & z_{2i-9}^2 + z_{2i-8}^2 - z_9^2 \qquad i = 5 \rightarrow 8 \end{aligned} \quad (5\text{-}10)$$

A polynomial system does not necessarily have a 2-homogenous form, but the IPO system has a 2-homogenous form based on the variable groupings (1,2,5,6) and (3,4,7,8). This results in the following 2-homogenous formulation:

$$\begin{aligned} f_i(z) = {} & a_{i,1}z_1z_3 + a_{i,2}z_1z_4 + a_{i,3}z_2z_3 + a_{i,4}z_2z_4 + a_{i,5}z_5z_7 + a_{i,6}z_5z_8 + a_{i,7}z_6z_7 + a_{i,8}z_6z_8 \\ & + a_{i,9}z_1z_9 + a_{i,10}z_2z_9 + a_{i,11}z_3z_{10} + a_{i,12}z_4z_{10} \\ & + a_{i,13}z_5z_9 + a_{i,14}z_6z_9 + a_{i,15}z_7z_{10} + a_{i,16}z_8z_{10} + a_{i,17}z_9z_{10} \qquad i = 1 \rightarrow 4 \\ f_i(z) = {} & z_{2i-9}^2 + z_{2i-8}^2 - z_9z_{10} \qquad i = 5 \rightarrow 8 \end{aligned} \quad (5\text{-}11)$$

The purpose of the n-homogenous forms is to avoid the problems with solutions at infinity. The system described by (5-12) has degree 256, so it has at most 256 solutions unless it has an infinite number of solutions (in the case of a singularity). Morgan proved [morgan87b] that (5-12) has at most 64 finite solutions. This means that there will be at least 192 paths leading to solutions at infinity. In the same paper, Morgan proved that in the special case when (5-12) represents an IPO system, there are at most 64 finite solutions. Hence there will be at least 224 divergent paths to

track, whereas only 64 paths may lead to finite solutions. Tracking divergent paths is always a problem because the program must use a rule to decide when to abandon a path and declare it to be divergent. There is no universal rule that always works. The n-homogenous formulation plus the use of the projective transform generates a system with no solutions at infinity. This shortens the computation time because there are no diverging paths that need to be followed for any length of time. Also, the solutions of the n-homogenous system can be transformed back to solutions of the original system and it is now easy to separate the finite solutions from the solutions at infinity.

96 path version

The main problem with the 256 path version is that there are many paths to track that will not lead to an interesting solution. We would rather not compute all the solutions at infinity since there is no real use for them. Morgan and Sommese [morgan86] also made headway with this problem, showing that the number of paths could be reduced to 96 by using a starting system **g** of the same form as **f**, but with different ("random") parameters. That is, **g** has the form

$$\begin{aligned} g_i(z) = {} & b_{i,1}z_1z_3 + b_{i,2}z_1z_4 + b_{i,3}z_2z_3 + b_{i,4}z_2z_4 + b_{i,5}z_5z_7 + b_{i,6}z_5z_8 + b_{i,7}z_6z_7 + b_{i,8}z_6z_8 \\ & + b_{i,9}z_1 + b_{i,10}z_2 + b_{i,11}z_3 + b_{i,12}z_4 + b_{i,13}z_5 + b_{i,14}z_6 + b_{i,15}z_7 + b_{i,16}z_8 \\ & + b_{i,17} \qquad i = 1 \rightarrow 4 \\ g_i(z) = {} & z_{2i-9}^2 + z_{2i-8}^2 - 1 \qquad i = 5 \rightarrow 8 \end{aligned} \tag{5-12}$$

Both **f** and **g** are cast in their 2-homogenous form before the computation is performed. The 2-homogenous Bezout number of **f** and **g** are 96, meaning that there are at most 96 distinct solutions. Hence there are only 96 paths to track. However, there is one problem with the method: We do not know the solutions to **g(z)=0**, so we have no points to start the tracking process from. Sommese solved this problem by inventing a simple system **G(z)** which can be solved by hand, and which can be used (once) as a starting system for finding the solutions of **g(z)**. Once we have the solutions of **g(z)**, the system can be used over and over again. The artificial system **G(z)** has the following form:

$$
\begin{aligned}
G_1(z) &= z_1(z_3 - 3z_7 + 11) \\
G_2(z) &= z_5(z_4 - 7z_8 + 5) \\
G_3(z) &= z_3(z_1 - 3z_5 + 11) \\
G_4(z) &= z_7(z_2 - 7z_6 + 5) \\
G_i(z) &= g_i(z) \;\; i = 5 \rightarrow 8
\end{aligned}
\tag{5-13}
$$

The solutions of this system has to be computed by hand and in one case by a reduction that produces a system that can in turn be solved in a ConsolC run. Because of the complexity of these hand calculations, they are not included here, but Table 5-6 contains some hints about how to compute them. There is definitely some work involved, but the hints are a good start.

After the hand calculations, the ConsolC program was used to compute all 96 solutions of **g(z)=0.** The solutions were then successfully used as starting points in the program robot96p2.

64 path version

As an additional result, Morgan and Sommese discovered and proved that (5-10) always has an identical set of 8 multiplicity 4 solutions at infinity. Moreover, when using the generic starting system (5-12), the same set of starting points will always lead to these 32 uninteresting solutions, no matter the particular robot coefficients in the goal system. This naturally leads to a 64 path

Case	Assumptions	Useful implications
1	z1=z3=z5=z7=0	z2,z4,z6,z8=+−1
2	z1=z3=0	z2,z4=+−1
3	z1=z7=0	z2,z8=+−1
4	z5=z7=0	z6,z8=+−1
5	z3=z5=0	z4,z6=+−1
6	z1,z3,z5,z7!=0	Create 2 eqns in 2 unknowns and solve numerically

Table 5-7 Hints for solving (5-13) by hand

version (robot64p2) were we simply skip the paths that lead to the 8x4 solutions. The programs robot64p2, robot64p2g and robot64p2gp are all variants of the 64-path version.

Further path number reductions

Since the time of this work, some further reductions in the number of paths have been made. It has been known since 1980 [duffy80] that there could be at most 32 solutions to the general IPO problem, meaning that the 64-path version does at least twice as much work as should be necessary. Moreover, Primrose [primrose86] proved that there are in fact 16 or less different solutions to the IPO problem. Since systems exist that do in fact have 16 solutions, this means that no further reduction is possible in the general case.

Wampler and Morgan [wampler89] developed a 11x11 polynomial system for which only 16 paths have to be tracked. Again, they used the method of the generic case to skip 304 of the 320 paths that would otherwise have to be tracked for this system, ending up with only 16 paths to track. The formulation is general, but has to be broken down in two cases depending on whether the robot has any joints with zero twist angles.

5.3 ConsolC and the IPO problem: Numerical properties

This section contains information about the numerical behavior of ConsolC (actually robot64p2gp) when applied to various instances of the general 6R IPO problem. Robot64p2gp was first run on the 3 examples used by Morgan in his book [morgan87a]. A large collection of software programs was developed along with ConsolC to postprocess the ConsolC outputs into various useful formats. For example, one postprocessor searches the output file for the joint angles and sort them so that they can be compared to Morgan's results. This verified that ConsolC was in working order.

Continuation path plots

Another interesting use of the output data is to plot the continuation paths in the complex plane.

robot64p2gp [–dhpsS] [–x xgraphfile] < infile > outfile	
Option	Function
–d	Divtest. Print when dividend>divisor
–h	Print values of f,Df,g,Dg,h,Dh at every evaluation
–p	Print running maximum of pivots
–s	Collect maximum statistics on x,f,g,h,df,dg,dh (per run)
–S	Collect maximum statistics on x,f,g,h,df,dg,dh (per path)
–x	Output xgraph plot data to named file

Table 5-8 Options for the robot64p2gp program

The ConsolC programs have an option (-x) which allows logging (to a file) the value of x at each step along the continuation path. The file can then be postprocessed into a format suitable for the xgraph program and displayed on the workstation. Figure 5-3 shows two continuation paths produced by robot64p2gp. The paths are from the same family of 64 paths generated by one execution of robot64p2gp. The starting system is g=Example3 (as always in robot64p2gp) and the goal system is f=Example1. Path 6 is a "nice" path where all variables stay close to the origin. Path 81 is the worst path from this run and has 4 variables straying quite far from the origin whereas the other 4 variables stay close, in fact so close that their paths are not discernible on this plot. This path takes longer to track and is more of a challenge to the accuracy of the computer arithmetic.

Path lengths

The length of a continuation path is defined as the sum of the Euclidean distances between the points along the path. The distance is measured in $\mathbf{C}^8$, since the path is in an 8-dimensional complex space. ConsolC provides path length information on demand. For example, path 81 has length 714.7 and path 6 has length 17.8, approximately. By sorting and examining all the path lengths for each run it became clear that there is a wide variation in the path lengths both inside a

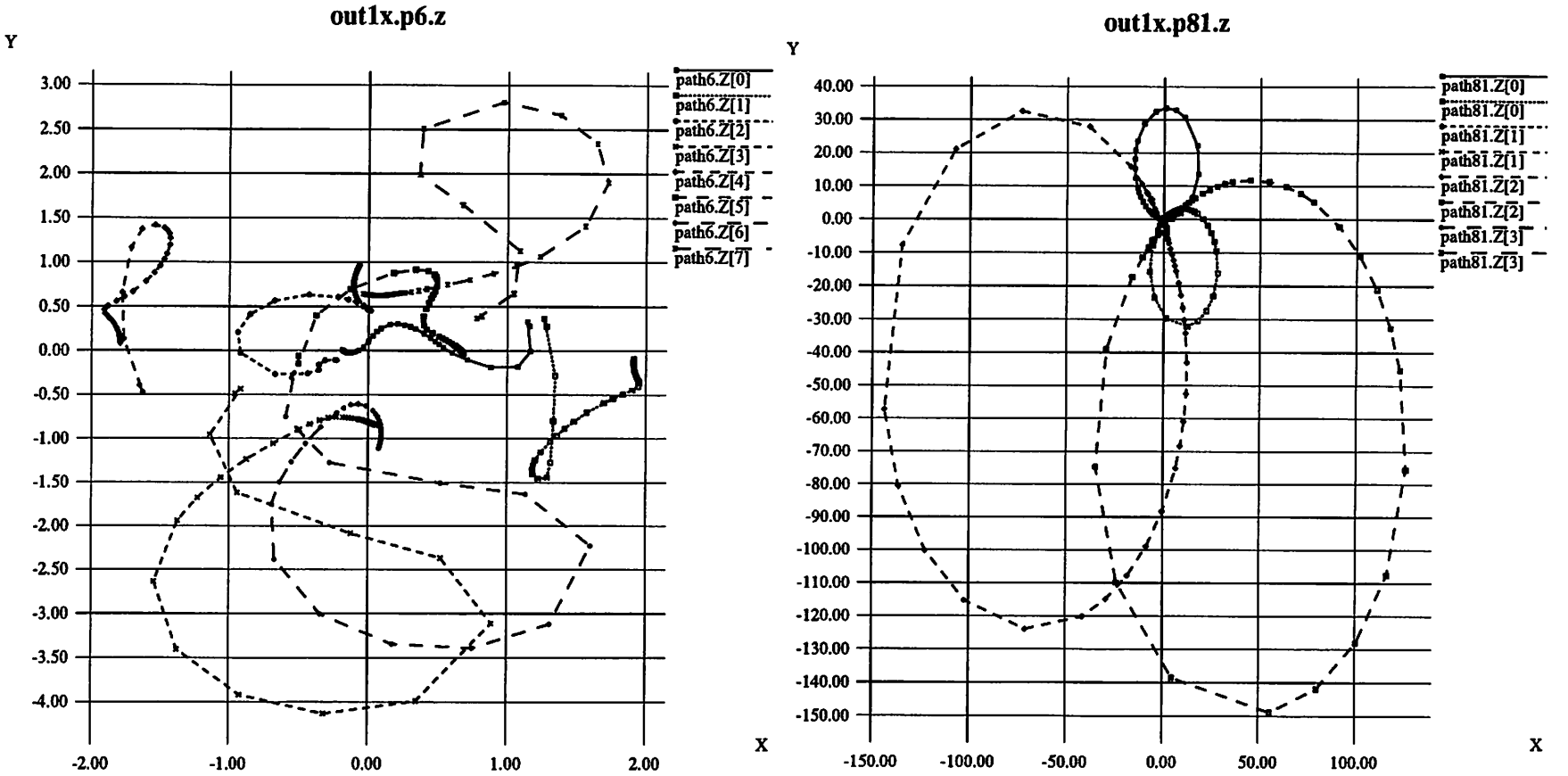


Figure 5-3 Example of continuation paths from the robot64p2gp program. Path 6 is a "nice" path which does not stray far from the origin. Path 81 is the worst of all the 64 paths in this run, which used g=Example3 as the start system and f=Example1 as the goal system. Each curve represents 1 out of 8 complex variables that make up a continuation path.

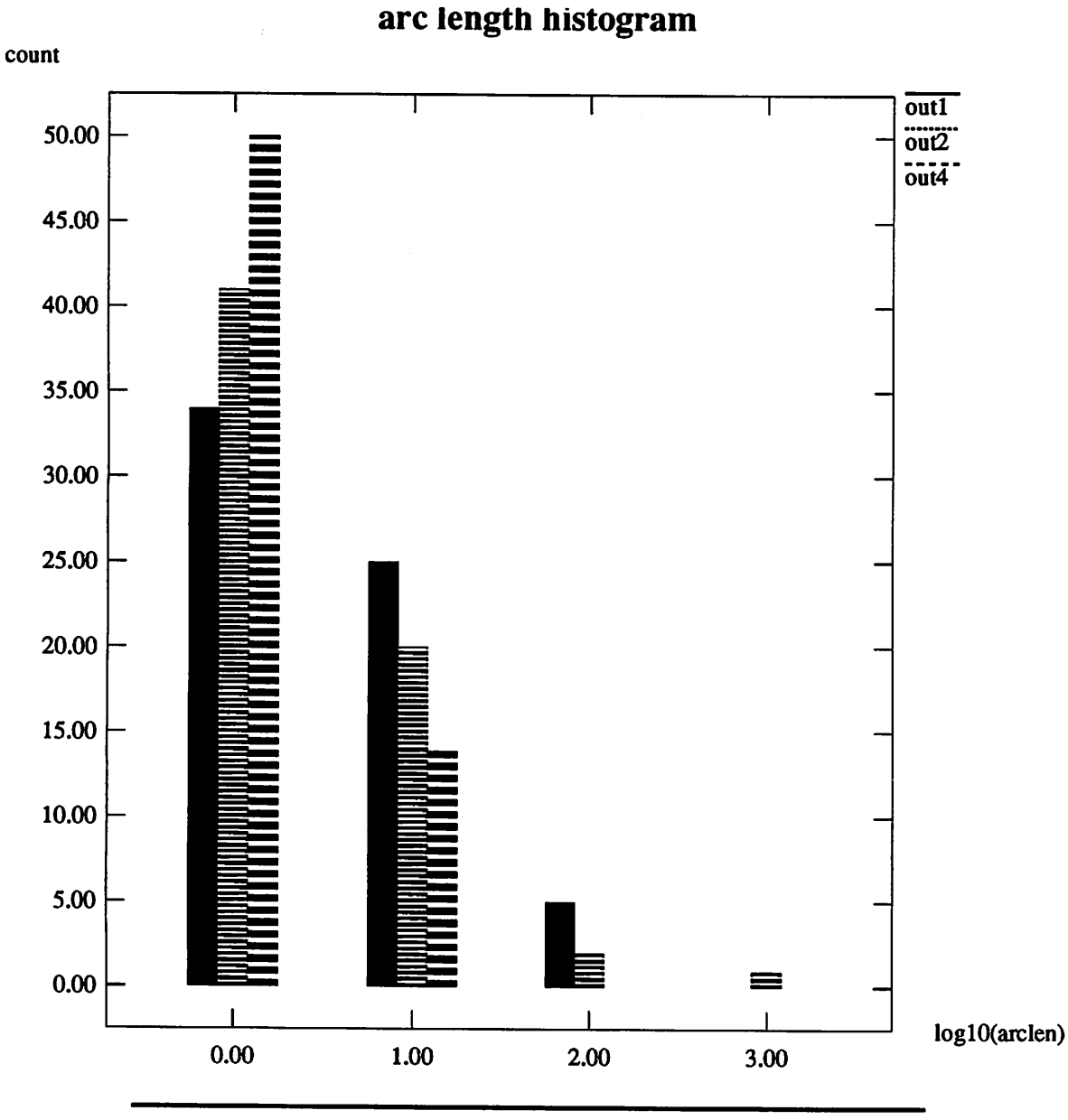


Figure 5-4 Individual histograms showing the frequency of various arc (path) lengths among the 64 paths generated by each one of 3 different runs

given run and between different runs. For example, the shortest path was path 29 which had length 1.79. Figure 5-4 shows the frequency of path lengths among the 64 paths of 3 different runs (note that the x-axis is logarithmic). The 3 runs are using f=Example1, f=Example2 and f=Example4. Example 4 is similar to the starting system (Example 3) but with a different goal point. Not surprisingly, Example 4 generates the most short paths and the lowest maximum path length as well. This is because the system is fairly close to g=Example3 (the starting system). The main

observation to make from the histograms is that most paths are fairly short (10^0 < path length < 10^1), but that in most of the test cases there are a few paths that are unusually long. Note that having a long path does not imply that the variables take on large values, even though it may look as if this is the case from path 81. The counter-example is a path which crisscrosses the area around the origin without any variable straying very far. The opposite implication is, however, true: Short paths imply small values of the variables.

Path maximum statistics

One of the most important implementation issues for Numerical Processing is the range and precision needed for the data types representing the real numbers, as well as the format (fixed point, floating point). While path length statistics can give some idea of what range and precision is needed, we need more complete information on the range of the variable and function values that can be expected during a continuation run.

ConsolC programs can provide the *maximum* absolute values of all components of **x, g(x), f(x), h(x), Dg(x), Df(x), Dh(x)**, either on a per-run or per-path basis. This feature was used to collect a large number of statistics from different runs using robot64p2g. 500 runs each were made with the goal robot being the Panasonic NM-5740, the PUMA 560 and Morgan's Example 3, using random goal position/orientation points. Each of these 3 cases generated 500x64=32000 max values for each of the variables/functions mentioned. This large data collection was then histogrammed in two different ways. The first 3 histograms (Figure 5-5 and Figure 5-6) are for the 3 different robots (Pana, Puma, Example 3) and contain max values for all the components individually. The main observations to make is that it is typically **Dh** which takes on the largest values and whose peak in the distribution is the farthest to the right. Values as high as 10^9 have been observed for some paths.

I also have made 7 plots (one for each of **x, g(x), f(x), h(x), Dg(x), Df(x), Dh(x)**) that each contain 3 histograms (one for each of Pana, Puma, Example). These plots (Figure 5-6 - Figure 5-9) make it easier to compare between the different target robots. The observation to make here is that

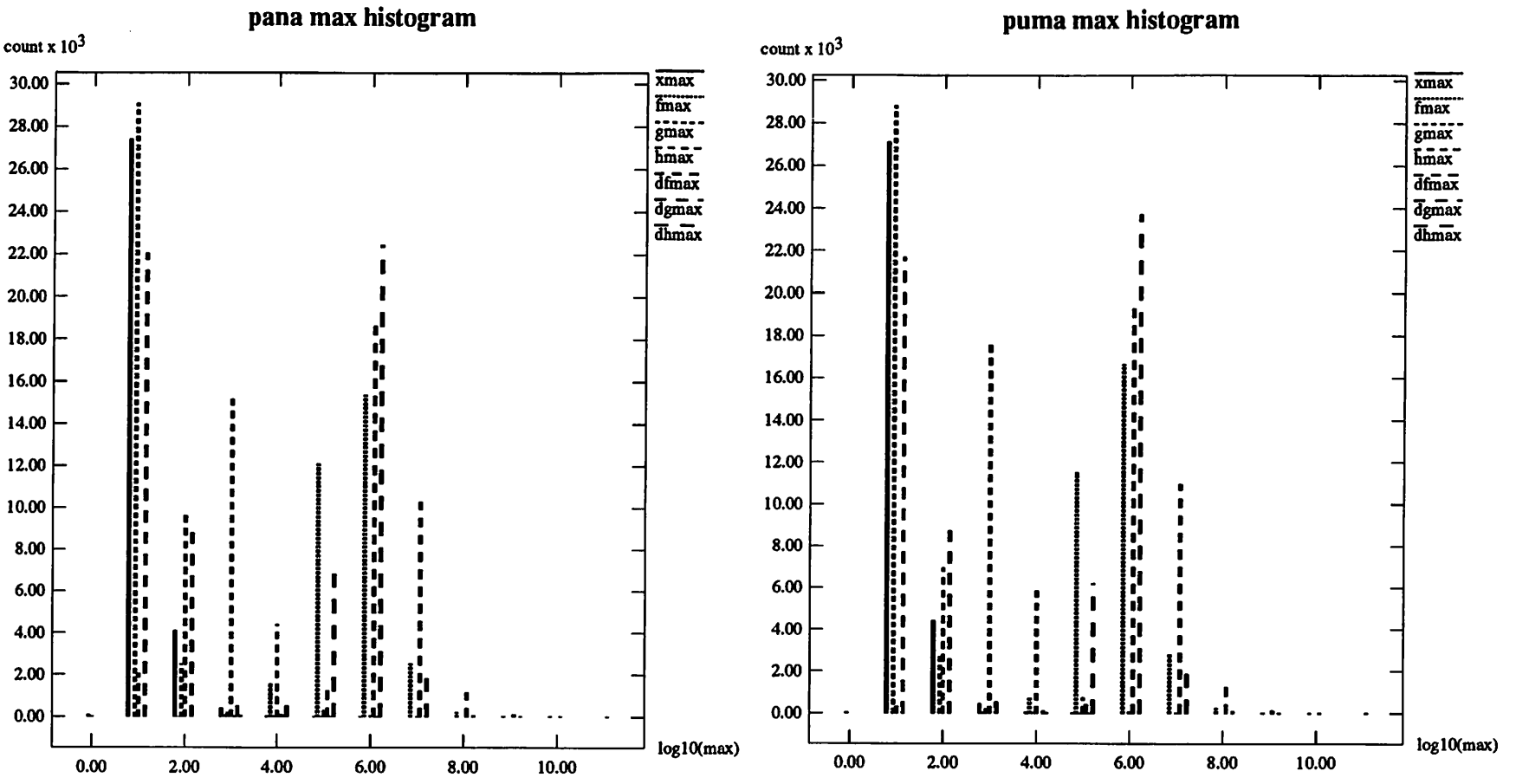


Figure 5-5 Histograms of max absolute values of variable and function components on a per-path basis over 500x64 paths. **Df(x)** has the largest values in this sample

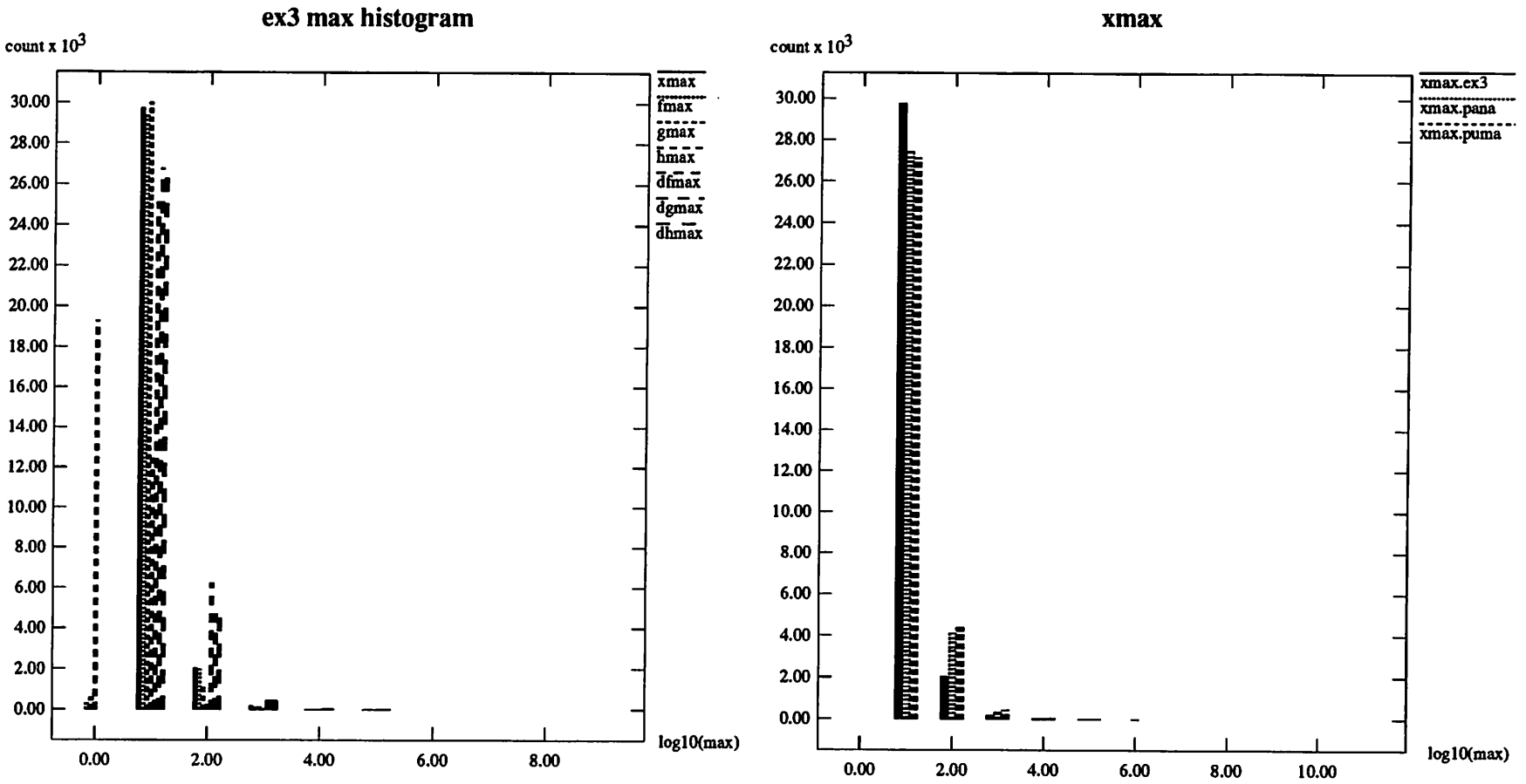


Figure 5-6 Left: Max absolute value histogram for Example 3 (500x64 paths). Right: Max absolute value of components of x (3x500x64 paths)

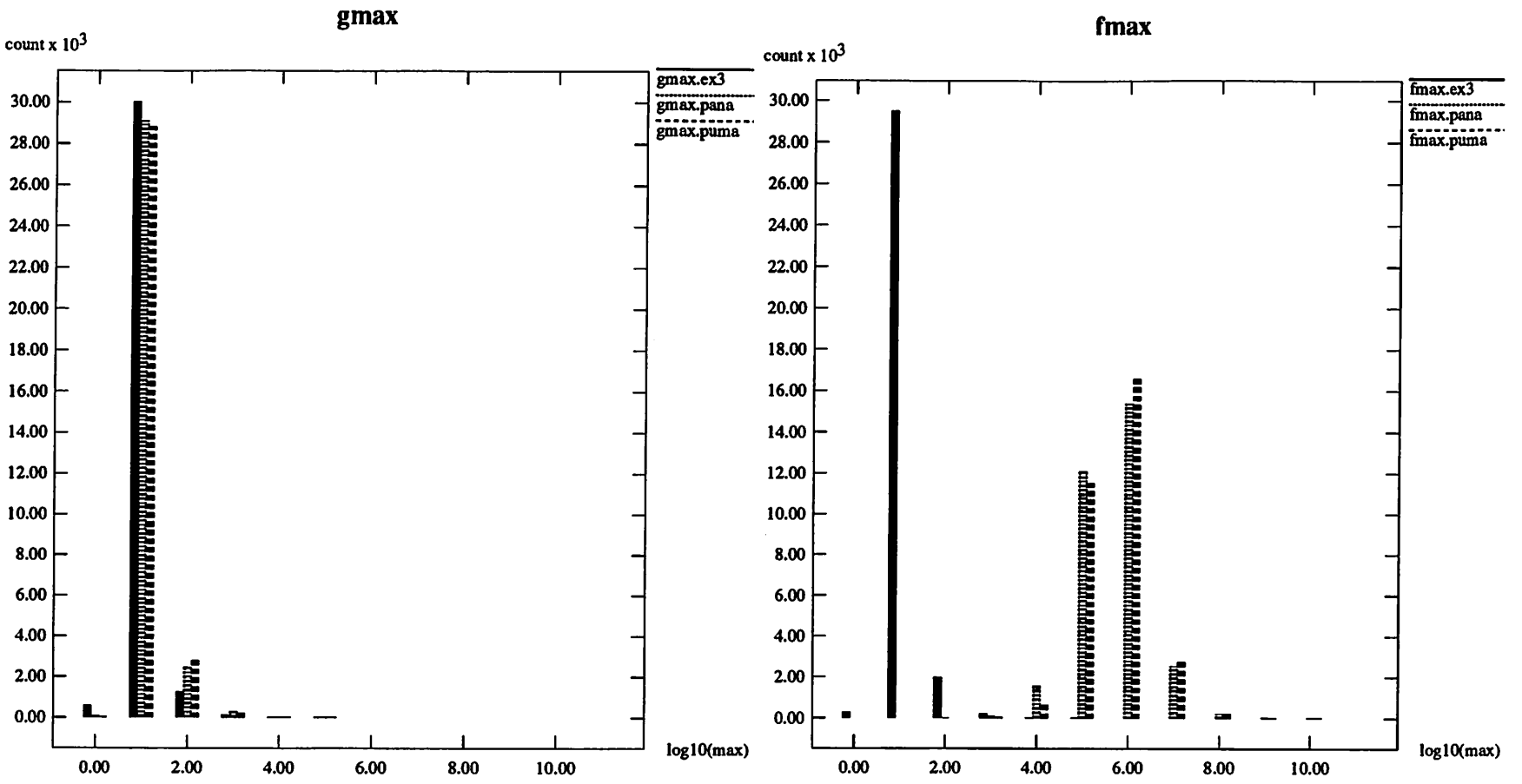


Figure 5-7 gmax and fmax histograms for f=(pana, puma, Example 3) goal systems and 500 random goal points.There are 3x500x64 paths

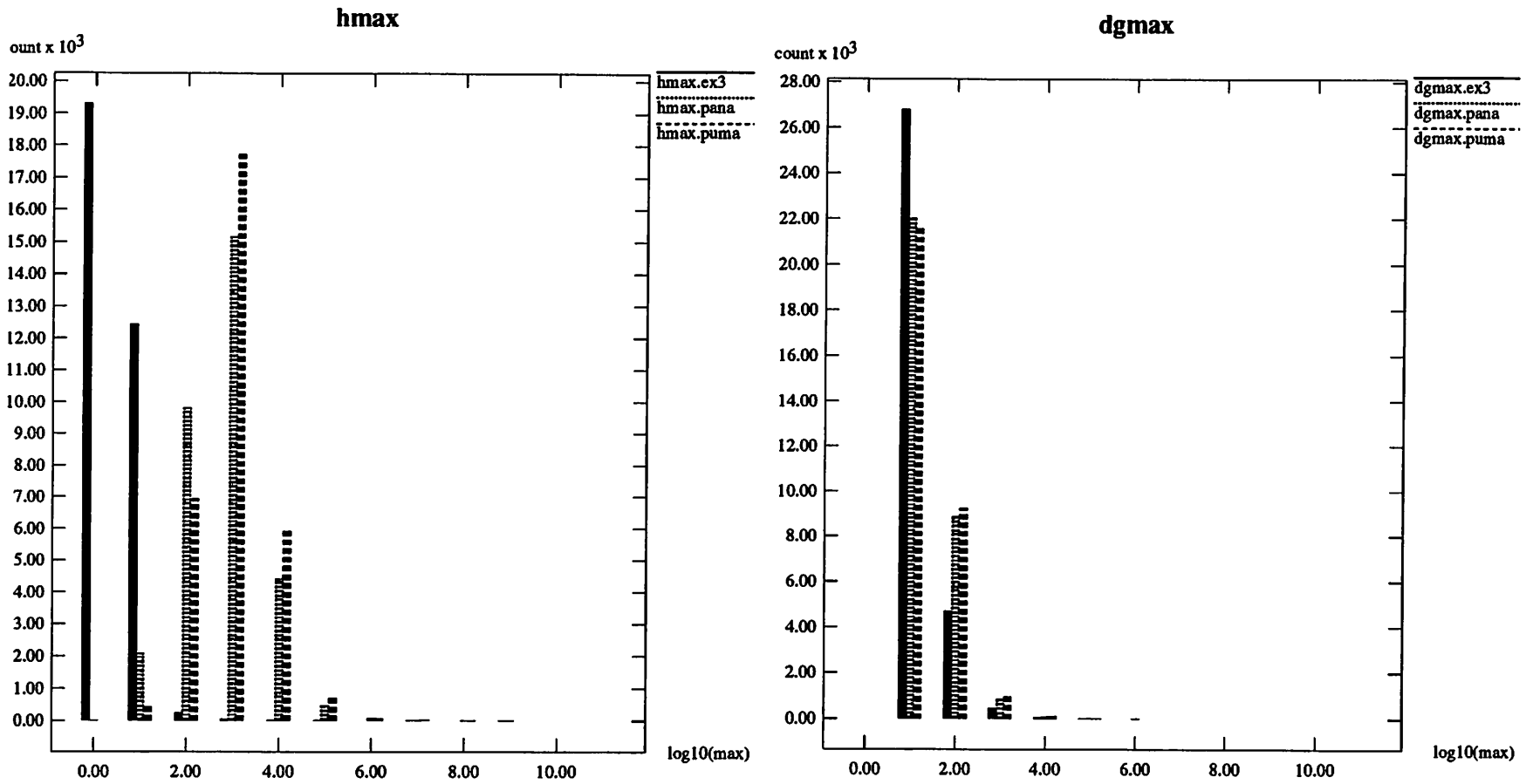


Figure 5-8 hmax and Dgmax histograms for f=(pana, puma, Example 3) goal systems and 500 random goal points. There are 3x500x64 paths

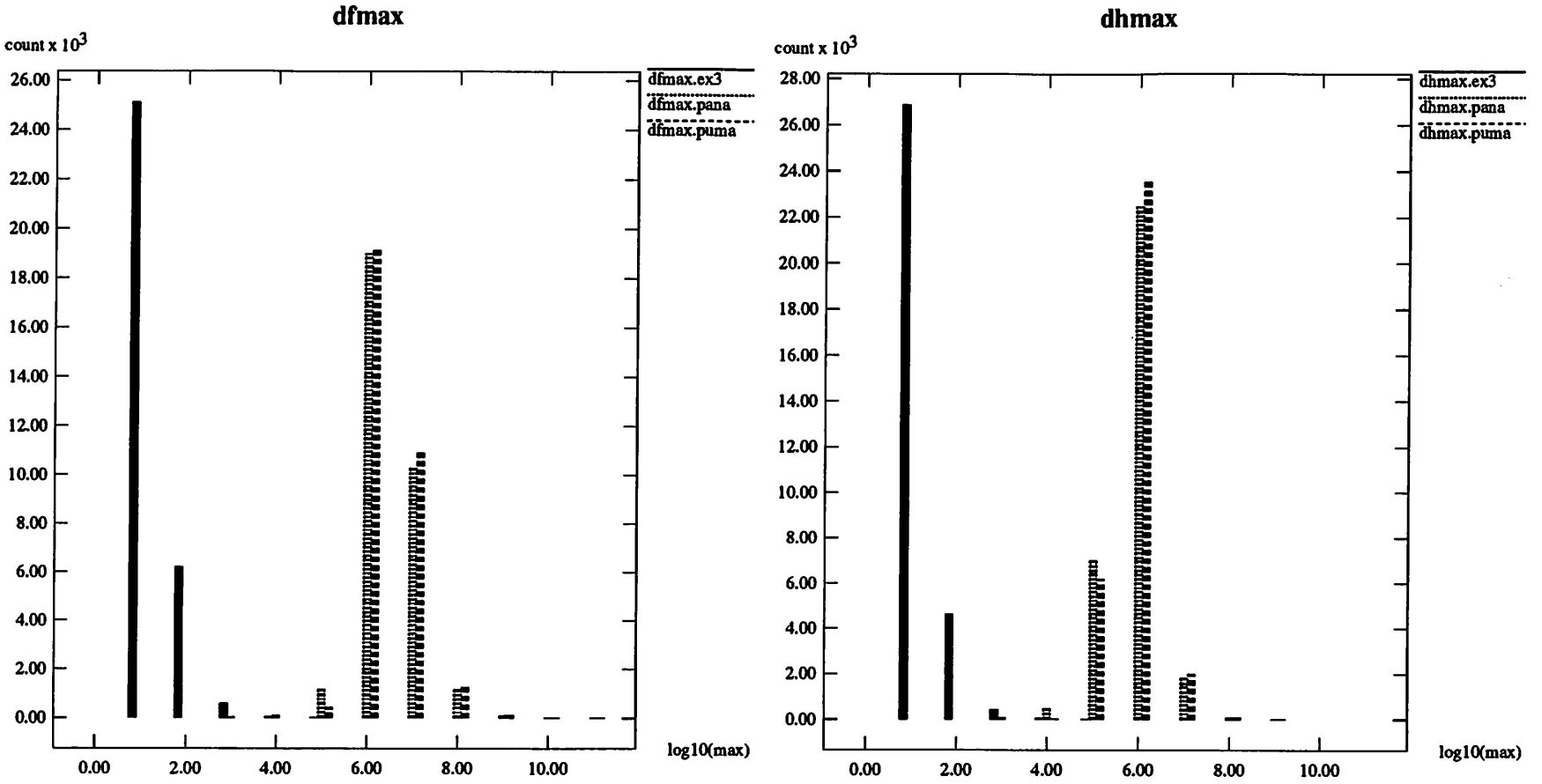


Figure 5-9 hmax and Dgmax histograms for f=(pana, puma, Example 3) goal systems and 500 random goal points. There are 3x500x64 paths

Example 3 predictably has the lowest maximums, whereas Pana and Puma are much higher but about the same (Puma generally slightly higher than Pana).

5.4 Profiling

Execution profiling refers to the practise of determining how much time a given program spends executing the various subroutines of the code. For ConsolC, the purpose of profiling is to find out what parts of the code are the most time intensive, so as to identify where an ASP can have the most impact on the execution speed. The UNIX profiling tool (gprof) was used for this purpose, and the main results are shown in Table 5-8. The main work being performed by the program is to evaluate functions and to solve the linear equations for the Newton iterations. The profile shows that the function evaluations, that is, **g(x)**, **f(x)**, **h(x)**, **Dg(x)**, **Df(x)** and **Dh(x)**, take 57.2% of the total running time (which was 62.01 seconds), whereas solving the linear equations takes 36.6%.

One should not jump to the conclusion that this means that function evaluation is the bottleneck. In fact it is not. The reason is that the function evaluation can easily be parallelized: Each component of **g(x)**, **f(x)**, **Dg(x)** and **Df(x)** can be computed independently and in parallel. Combining the results into **h(x)** and **Dh(x)** is a small matter. Indeed, if all the components of the functions were equally complex to compute, we could get a speedup of 8x8+8x8+8+8=144 by computing all the components in parallel on different processors. An additional factor of 2 can be

Function	Percent time	Subfunction	Percent time
hfunt	57.2%	gfunt	21.2%
		ffunt	19.0%
linnr	36.6%	lnfngap	26.8%
		lnsngap	9.6%

Table 5-9 Profiling results for robot64p2gp. The *hfunt* function evaluates **h(x)** and **Dh(x)**. The *linnr* function is the linear equation solver

Program part	Before parallelization		After parallelization (estimated)	
	seconds	percent	seconds	percent
h(x), Dh(x)	35.47	57.2%	0.25	0.9%
Linear equations	22.70	36.6%	22.70	84.7%
Other	3.84	6.2%	3.84	24.4%

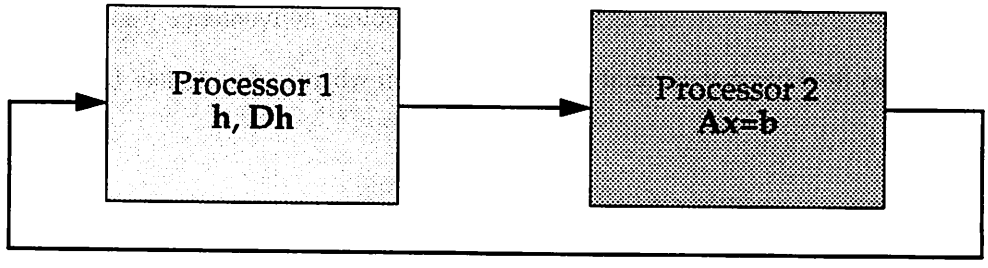
Table 5-10 Impact of parallelization on relative runtime of function evaluations versus linear system solving

realized by computing the real and imaginary parts in parallel. The assumption that all function components are equally complex does not hold *exactly* for the IPO system, but it is accurate enough to make our point, which is that the function evaluation part of ConsolC is easily parallelizable and hence not the true barrier to high performance. Table 5-9 shows the impact of parallelizing the function evaluations 144 times. The result is that the linear system solving now takes up 84.7% of the time and the function evaluations only 0.9%. (The rest of the time is various i/o and pre/postprocessing overhead that may or may not be present if the computation is done in an Application Specific Processor.)

5.5 Pipeline interleaving

In addition to the concurrency available in the function evaluation, there is also the possibility of having several continuation paths being computed at the same time. This is not necessarily a good idea if one has to spend more hardware. On the other hand, it is likely that there are two main processors in the system: One for the function evaluation and one for solving the linear equations. Once Processor 1 has computed **h** and **Dh**, Processor 2 takes over and solves the linear system. This means that both processors will be idle half the time.

Since there are multiple paths to be tracked, the solution to this efficiency problem is to use *pipeline interleaving* [lee86], meaning that we process 2 paths at the same time, with one path



Schedule time →

P1	h_a	h_b	h_a	h_b	h_a	h_b	.....
P2	.....	LU_a	LU_b	LU_a	LU_b	LU_a	LU_b

Figure 5-10 Pipeline interleaving with 2 processors, and 2 paths being computed concurrently

being in the function evaluation stage while the other one is in the linear equation stage. The method is illustrated in Figure 5-10.

5.6 Arithmetic experiments

One of the most important considerations for an ASP implementation of IPO/ConsolC (or parts thereof) is the arithmetic requirement of the computation. This section describes how this aspect of the design was explored using the ConsolC environment. The first subsection presents the results floating point experiments whereas the other subsection presents the results of a partial fixed point computation.

Single precision floating point computation

This experiment was performed simply by using the C datatype **float** throughout all the programs and apply the appropriate compiler options to turn off the common practice that **double** is used in intermediate results. The results were promising, but using float required that some of the ConsolC

convergence parameters be adjusted to less strict values before convergence was possible. This is to be expected, as float has 6.9 decimal digits of precision whereas double has 16.9 or more. The

epsbig	epssmall	t=1	t>=0.99999	t<0.99999	t<0.99
MIPS	cc -f -Dfptype float				
0.01	epsbig/1000	28	24	12	1
0.01	epsbig/100	32	31	1	1
0.01	epsbig/10	38	25	1	1
0.01	epsbig	45	18	1	1
SUN4	/usr/bin/cc -fsingle -Dfptype=float				
0.01	epsbig/1000	28	24	12	1
0.01	epsbig/100	32	31	1	1
0.01	epsbig/10	38	25	1	1
0.01	epsbig	45	18	1	1
SUN4	/usr/lang/cc -fsingle -Dfptype=float -O				
0.01	epsbig/1000	28	24	12	1
0.01	epsbig/100	32	31	1	1
0.01	epsbig/10	38	25	1	1
0.01	epsbig	45	18	1	1
SUN4	gcc -ffloat-store -Dfptype=float -O				
0.01	epsbig/1000	28	26	10	5
0.01	epsbig/100	32	31	1	1
0.01	epsbig/10	38	25	1	1
0.01	epsbig	44	19	1	1

Table 5-11 Convergence of ConsolC/robot64p2gp in single-precision arithmetic. Variations over different machine architectures and compilers

main convergence results are shown in Table 5-10, which shows for each test how many of the paths reached t=1 or got reasonably close to t=1 before ConsolC gave up. The table refers to some of the convergence parameters of ConsolC, known as *epsbig* and *epssmall*. These variables are used to set the convergence criterion for the Newton steps as follows:

- `eps = (t<0.95 ? epsbig : epssmall);`
- `converged = (norm2(Δx)<eps);`

In other words, the epsbig is used for the crude stepping during the path tracking and the epssmall is used while zeroing in on the final solution towards the end of each path. The convergence criterion is that the euclidean norm of the last Δx should be less than eps. The value typically used in ConsolC (double-precision) is epsbig=0.01 and epssmall=epsbig/1000. As can be seen from the table, convergence improves if epssmall is reduced. The paths that do not converge well even at epssmall=epsbig are ones leading to the singular solutions at infinity, so they are not really a practical problem. One path (number 16) never gets closer than t=0.97 in any of the tests, and it was found (by checking the corresponding double-precision path) that this is due to a somewhat nasty path shape towards the end. The conclusion is that ConsolC is viable in single precision floating-point arithmetic, but it is unlikely that much less precision than this is practical.

The table shows some small variations between various compilers and architectures, but the variations are small.

Fixed point computation

Considering the above results, there is little hope that all of ConsolC can be implemented in fixed point arithmetic and produce accurate solutions at any reasonable wordlength. However, it is likely that certain sub-parts of ConsolC can implemented in fixed point. As explained in the previous section, the main tasks of ConsolC are function evaluation and solving linear systems. The arithmetic requirements for these two processes are quite different. It is well known [dahlquist74], [golub83][golub89] that one can easily construct examples of linear systems that are quite

impractical to solve in fixed-point arithmetic. In fact, any system which is even close to singular will cause great problems. Since we know that 32 of the 64 IPO paths always lead to singular solutions, it is clear that fixed point Gaussian elimination is not viable.

The other main computational part of ConsolC is function evaluation. As shown in section 5.3, the values of the variable **x** and the function values **g(x), f(x), h(x), Dg(x), Df(x), Dh(x)** can span over a wide range of values, from 0 to about 10^9. This means that fixed point implementation is difficult. but not impossible. One favorable property of the functions is that they all are polynomials, and hence have a simple sum-of-products form. This makes them easier to compute since, for example, one never has to worry about dividing one number with a potentially very small other number.

It was decided to investigate fixed point implementation further by trying to perform the computation of **f(x), Df(x)** in fixed point, convert the results to floating point and then perform all other computations in (double-precision) floating point. This approach was taken partly because it enables us to isolate the inaccuracies. The actual implementation of the fixed point arithmetic is presented in the next section.

Wordlengths and scaling is even more critical in ConsolC/robot64p2gp than in PUMA (Chapter 4). The range and precision required is also much larger. Values of x_i up to 10^4 have been observed (cf. earlier histograms), meaning that about 14 bits are required for the integer portion of the variable. Another 14 bits are required to get a resolution of 0.0001, and with some extra bits as a

variable	range, precision (no paths converged)	range, precision (nice paths converged)
x, Df	9, -21	5, -25
x^2, f	18, -12	10, -20
Fcoef	1, -29	1, -29

Table 5-12 Scaling of the fixed point variables used in computing **f(x)** and **Df(x)**

safeguard we get close to 32 bits as a minimum for x. 31 bits were used (see next section) in the fixed point computation because of hardware/software limitations. The experiments revealed that the precision was not at all sufficient, and that all the paths diverged in the middle (0<t<1). Since there was no easy way to extend the wordlength, I tried to increase the precision at the expense of the range just to see if I could get convergence at least for those paths that were nice enough not to cause any overflows. The two variations are shown in Table 5-11. The parameters *range* and *precision* refer to how many of the 31 bits were allocated before and after the binary point. It was found that by reducing the range for x to 5 it was possible to get most of the paths to converge. The results were accurate to 4-5 decimal digits in the case of nonsingular solutions.

The conclusion is that fixed point computation of the function parts of ConsolC is viable in certain application, but it is questionable whether it is economical at the wordlengths (64 bits?) that would be required to ensure both accuracy and margin against overflow.

5.7 The `Fix.cc` fixed point arithmetic package

The fixed point version of ConsolC/robot64p2gp was implemented in the C++ programming language. C++ was chosen because it is a superset of ANSI C and provides user-defined data types (Classes) and operator overloading. Operator overloading allows the programmer to provide special functions to extend the standard arithmetic operators (such as +,-,*,/) to apply to an arbitrary datatype. This is very practical since it is cumbersome to use functions and write e.g. `sum(a,sum(b,c))` when the natural form of the expression is `(a+b+c)`.

For ConsolC, a new datatype (Class Fix) was developed. This meant that the C code could be used almost unaltered except for changing the datatype of the affected variables from **double** to **Fix** and specifying the appropriate scale factors. Some available fixed point packages for C++ that are publicly available were evaluated [gnu90], but found to be inappropriate for the purpose since they did not support variable (or mixed) scale factors. The declaration of the Fix class is shown in Figure 5-11. All the standard arithmetic operations have been implemented.

```
class Fix
{
   int    bits;
   int    scale;
public:
   member int       fbits();
   member int       fscale();
   member           Fix (Fix&);
   member           Fix (int=0);
   member           Fix (double&, int);
   member           ~Fix();
   member           setFix(double&, int);
   member           setScale(int);
   friend Fix       int2Fix  (int, int=0);
   friend Fix       double2Fix(double, int=0);
   member           operator double();
   friend Fix       rescale(Fix&, int=0);
   friend int       operator== (Fix&, Fix&);
   friend int       operator!= (Fix&, Fix&);
   friend int       operator<  (Fix&, Fix&);
   friend int       operator<= (Fix&, Fix&);
   friend int       operator>  (Fix&, Fix&);
   friend int       operator>= (Fix&, Fix&);
   friend int       operator== (double, Fix&);
   friend int       operator!= (double, Fix&);
   friend int       operator<  (double, Fix&);
   friend int       operator<= (double, Fix&);
   friend int       operator>  (double, Fix&);
   friend int       operator>= (double, Fix&);
   friend int       operator== (Fix&, double);
   friend int       operator!= (Fix&, double);
   friend int       operator<  (Fix&, double);
   friend int       operator<= (Fix&, double);
   friend int       operator>  (Fix&, double);
   friend int       operator>= (Fix&, double);
   member Fix&      operator=  (Fix&);
   member Fix&      operator=  (double&);
   member Fix&      operator=  (int&);
   member Fix&      operator+= (Fix&);
   member Fix&      operator-= (Fix&);
   member Fix&      operator*= (Fix&);
   member Fix&      operator/= (Fix&);
   member Fix&      operator*= (int);
   member Fix&      operator/= (int);
   member Fix&      operator>>=(int);
   member Fix&      operator<<=(int);
   friend Fix       operator-  (Fix& x);
   friend Fix       operator+  (Fix&, Fix&);
   friend Fix       operator-  (Fix&, Fix&);
   friend Fix       operator*  (Fix&, Fix&);
   friend Fix       operator/  (Fix&, Fix&);
   friend Fix       operator*  (Fix&, int);
   friend Fix       operator*  (int , Fix&);
   friend Fix       operator/  (Fix&, int);
   friend Fix       operator<< (Fix&, int);
   friend Fix       operator>> (Fix&, int);
   friend char*     Ftoa (Fix&, int=1, int=12, int=4);
   friend Fix       atoF (const char*, int scale=0);
   friend istream&  operator >> (istream&, Fix&);
   friend ostream&  operator << (ostream&, Fix&);
   friend void      error(char* msg);
   friend void      overflow (Fix&, char *msg="");
   friend void      scaleerr (char*);
   friend void      rangeerr (char*);
};
```

Figure 5-11 The declaration of the **Fix** class used for fixed point computation

5.8 Theoretical Bounds on variable and function values

The problem of finding a tight bound on the continuation variables is difficult. To see why, recall that the continuation variables are

$$x = (c_1, s_1, c_2, s_2, c_4, s_4, c_5, s_5) \in C^8 \qquad c_i = \cos\theta_i \qquad s_i = \sin\theta_i \tag{5-14}$$

The key observation is that for the *complex* variables c_i and s_i,

$$c_i^2 + s_i^2 = 1 \not\Rightarrow \|c_i\|^2 + \|s_i\|^2 = 1 \tag{5-15}$$

For example, if

$$a^2 = -1 + 10i,\ b^2 = 2 + 10i \qquad \text{then} \qquad a^2 + b^2 = 1 \qquad \text{but} \qquad \|a\|^2 + \|b\|^2 = 205 \tag{5-16}$$

Hence, the sine/cosine constraint does not in reality provide a useful bound. For the 16-path continuation, the intermediate systems (0<t<1) actually correspond to physical robots. If one could guarantee that all the robots corresponding to 0<t<1 have 16 solutions, it could be argued that the variables will be pure real and hence the paths will be bounded by the sine/cosine constraint. However, no condition has been proven under which such a guarantee would hold, and it is easy to find a case where some 0<t<1 corresponds to a robot with fewer than 16 solutions.The same problem occurs (at least for the system used here) for the bounds on function values.

Some bounds on the solutions of polynomial systems can be found in the literature, but they are generally so slack that they are not useful here. For example, [canny88] refers to the "Gap Theorem", which states that

$$f(x) = 0 \Rightarrow \forall i = 1 \rightarrow n \qquad \|x_i\| < w(R(f)) < (3dc)^{nd^n} \tag{5-17}$$

where w(R(f)) is the sum of the absolute values of the coefficients of R(f), the *resultant* of f. The coefficients of R(f) are a function of the coefficients of f, leading to the explicit bound shown in the equation, where

- n is the number of equations
- d is the degree of each equation

- c is a bound on the coefficient size

For the IPO equations, we have n=8, d=2 and (say) c=1. This results in the bound

$$\|x_i\| < (3 \cdot 2 \cdot 1)^{8 \cdot 2^8} = 6^{2048} \cong 10^{1594} \quad \text{(5-18)}$$

This bound does not provide any useful information for our purposes.

5.9 Summary

This chapter has presented the development and use of the ConsolC software package. ConsolC was used as the basis of a 64-path homotopy continuation solver (robot64p2gp) for the Inverse Position-Orientation computation for a general 6R robot. The qualitative behavior of the algorithm was examined by creating plots of the continuation paths. The program has been tested on more than 1500 different sets of input data and a number of statistics have been collected. The path length statistics show that most paths are "nice", but that almost every run will have some "nasty" paths that take a long time to track. The maximum value statistics show that most paths have low maximum values for **x** (say, $< 10^2$) whereas **h,Dh** and the other function values can be as high as 10^6 for the nice path and as high as 10^9 in the worst cases.

Experiments have shown that robot64p2gp is viable also in single precision floating point, and that the function evaluation part of the algorithm can even be computed in fixed point in some cases. However, it looks like using floating point is more economical because of the wordlength required in fixed point. A special fixed point computation package was developed in the C++ language to perform the fixed point experiments.

Profiling of the algorithm shows that in the straightforward implementation, the function evaluation part of the algorithm is more time-consuming than solving the linear equations. However, parallelization of the function evaluation is straightforward and leaves solving linear systems as the foremost computational bottleneck.

Coef	Value
a[1,1]	la[1]*mu[2]*la[6]*ny+la[1]*mu[2]*mu[6]*my
a[1,2]	-mu[2]*la[6]*nx-mu[2]*mu[6]*mx
a[1,3]	-la[1]*mu[2]*la[6]*nx-la[1]*mu[2]*mu[6]*mx
a[1,4]	-mu[2]*la[6]*ny-mu[2]*mu[6]*my
a[1,5]	-mu[3]*la[4]*mu[5]
a[1,8]	mu[3]*mu[5]
a[1,9]	mu[1]*la[2]*la[6]*ny+mu[1]*la[2]*mu[6]*my
a[1,10]	-mu[1]*la[2]*la[6]*nx-mu[1]*la[2]*mu[6]*mx
a[1,11]	mu[1]*mu[2]*la[6]*nz+mu[1]*mu[2]*mu[6]*mz
a[1,13]	-mu[3]*mu[4]*la[5]
a[1,15]	-la[3]*mu[4]*mu[5]
a[1,17]	-la[1]*la[2]*la[6]*nz-la[1]*la[2]*mu[6]*mz+la[3]*la[4]*la[5]
a[2,1]	la[1]*mu[2]*py-la[1]*mu[2]*d[6]*la[6]*ny-la[1]*mu[2]*d[6]*mu[6]*my- la[1]*mu[2]*a[6]*ly
a[2,2]	-mu[2]*px+mu[2]*d[6]*la[6]*nx+mu[2]*d[6]*mu[6]*mx+mu[2]*a[6]*lx
a[2,3]	-la[1]*mu[2]*px+la[1]*mu[2]*d[6]*la[6]*nx+la[1]*mu[2]*d[6]*mu[6]*mx +la[1]*mu[2]*a[6]*lx
a[2,4]	-mu[2]*py+mu[2]*d[6]*la[6]*ny+mu[2]*d[6]*mu[6]*my+mu[2]*a[6]*ly
a[2,6]	mu[3]*la[4]*a[5]
a[2,7]	mu[3]*a[5]
a[2,9]	mu[1]*la[2]*py-mu[1]*la[2]*d[6]*la[6]*ny-mu[1]*la[2]*d[6]*mu[6]*my -mu[1]*la[2]*a[6]*ly
a[2,10]	-mu[1]*la[2]*px+mu[1]*la[2]*d[6]*la[6]*nx+mu[1]*la[2]*d[6]*mu[6]*mx +mu[1]*la[2]*a[6]*lx
a[2,11]	mu[1]*mu[2]*pz-mu[1]*mu[2]*d[6]*la[6]*nz-mu[1]*mu[2]*d[6]*mu[6]*mz -mu[1]*mu[2]*a[6]*lz-d[1]*mu[1]*mu[2]

Table 5-13 The coefficients of (5-12), in terms of the goal point position, orientations and the robot Denavit-Hartenberg parameters. la[i]=λ_i, mu=μ_i

Coef	Value
a[2,12]	a[1]*mu[2]
a[2,13]	-mu[3]*mu[4]*d[5]
a[2,14]	mu[3]*a[4]
a[2,16]	la[3]*mu[4]*a[5]
a[2,17]	-la[1]*la[2]*pz+la[1]*la[2]*d[6]*la[6]*nz+la[1]*la[2]*d[6]*mu[6]*mz +la[1]*la[2]*a[6]*lz+la[3]*la[4]*d[5]+la[3]*d[4]+d[3]+d[2]*la[2]+d[1]*la[1]*la[2]
a[3,1]	a[2]*la[6]*nx+a[2]*mu[6]*mx
a[3,2]	la[1]*a[2]*la[6]*ny+la[1]*a[2]*mu[6]*my
a[3,3]	a[2]*la[6]*ny+a[2]*mu[6]*my
a[3,4]	-la[1]*a[2]*la[6]*nx-la[1]*a[2]*mu[6]*mx
a[3,5]	-d[3]*mu[3]*la[4]*mu[5]
a[3,6]	a[3]*mu[5]
a[3,7]	a[3]*la[4]*mu[5]
a[3,8]	d[3]*mu[3]*mu[5]
a[3,9]	-mu[1]*d[2]*la[6]*ny+a[1]*la[6]*nx-mu[1]*d[2]*mu[6]*my+a[1]*mu[6]*mx
a[3,10]	a[1]*la[6]*ny+mu[1]*d[2]*la[6]*nx+a[1]*mu[6]*my+mu[1]*d[2]*mu[6]*mx
a[3,12]	mu[1]*a[2]*la[6]*nz+mu[1]*a[2]*mu[6]*mz
a[3,13]	-d[3]*mu[3]*mu[4]*la[5]
a[3,14]	a[3]*mu[4]*la[5]
a[3,15]	-d[4]*mu[4]*mu[5]-d[3]*la[3]*mu[4]*mu[5]
a[3,16]	a[4]*mu[5]
a[3,17]	-la[6]*nz*pz-mu[6]*mz*pz-la[6]*ny*py-mu[6]*my*py-la[6]*nx*px-mu[6]*mx*px +la[1]*d[2]*la[6]*nz+d[1]*la[6]*nz+la[1]*d[2]*mu[6]*mz+d[1]*mu[6]*mz+d[6] +d[5]*la[5]+d[4]*la[4]*la[5]+d[3]*la[3]*la[4]*la[5]
a[4,1]	2*a[2]*px-2*a[2]*d[6]*la[6]*nx-2*a[2]*d[6]*mu[6]*mx-2*a[2]*a[6]*lx

Table 5-13 The coefficients of (5-12), in terms of the goal point position, orientations and the robot Denavit-Hartenberg parameters. la[i]=λ_i, mu=μ_i

Coef	Value
a[4,2]	2*la[1]*a[2]*py-2*la[1]*a[2]*d[6]*la[6]*ny-2*la[1]*a[2]*d[6]*mu[6]*my -2*la[1]*a[2]*a[6]*ly
a[4,3]	2*a[2]*py-2*a[2]*d[6]*la[6]*ny-2*a[2]*d[6]*mu[6]*my-2*a[2]*a[6]*ly
a[4,4]	-2*la[1]*a[2]*px+2*la[1]*a[2]*d[6]*la[6]*nx +2*la[1]*a[2]*d[6]*mu[6]*mx+2*la[1]*a[2]*a[6]*lx
a[4,5]	2*a[3]*a[5]
a[4,6]	2*d[3]*mu[3]*la[4]*a[5]
a[4,7]	2*d[3]*mu[3]*a[5]
a[4,8]	-2*a[3]*la[4]*a[5]
a[4,9]	-2*mu[1]*d[2]*py+2*a[1]*px+2*mu[1]*d[2]*d[6]*la[6]*ny-2*a[1]*d[6]*la[6]*nx +2*mu[1]*d[2]*d[6]*mu[6]*my-2*a[1]*d[6]*mu[6]*mx+2*mu[1]*d[2]*a[6]*ly -2*a[1]*a[6]*lx
a[4,10]	2*a[1]*py+2*mu[1]*d[2]*px-2*a[1]*d[6]*la[6]*ny-2*mu[1]*d[2]*d[6]*la[6]*nx -2*a[1]*d[6]*mu[6]*my-2*mu[1]*d[2]*d[6]*mu[6]*mx-2*a[1]*a[6]*ly -2*mu[1]*d[2]*a[6]*lx
a[4,11]	-2*a[1]*a[2]
a[4,12]	2*mu[1]*a[2]*pz-2*mu[1]*a[2]*d[6]*la[6]*nz-2*mu[1]*a[2]*d[6]*mu[6]*mz -2*mu[1]*a[2]*a[6]*lz-2*d[1]*mu[1]*a[2]
a[4,13]	2*a[3]*a[4]-2*d[3]*mu[3]*mu[4]*d[5]
a[4,14]	2*a[3]*mu[4]*d[5]+2*d[3]*mu[3]*a[4]
a[4,15]	2*a[4]*a[5]
a[4,16]	2*d[4]*mu[4]*a[5]+2*d[3]*la[3]*mu[4]*a[5]
a[4,17]	-pz*pz+2*d[6]*la[6]*nz*pz+2*d[6]*mu[6]*mz*pz+2*a[6]*lz*pz+2*la[1]*d[2]*pz +2*d[1]*pz-py*py+2*d[6]*la[6]*ny*py+2*d[6]*mu[6]*my*py+2*a[6]*ly*py -px*px+2*d[6]*la[6]*nx*px+2*d[6]*mu[6]*mx*px+2*a[6]*lx*px -2*la[1]*d[2]*d[6]*la[6]*nz-2*d[1]*d[6]*la[6]*nz-2*la[1]*d[2]*d[6]*mu[6]*mz -2*d[1]*d[6]*mu[6]*mz-2*la[1]*d[2]*a[6]*lz-2*d[1]*a[6]*lz-d[6]*d[6] -a[6]*a[6]+d[5]*d[5]+2*d[4]*la[4]*d[5]+2*d[3]*la[3]*la[4]*d[5]+a[5]*a[5] +d[4]*d[4]+2*d[3]*la[3]*d[4]+a[4]*a[4]+d[3]*d[3]+a[3]*a[3]-d[2]*d[2] -2*d[1]*la[1]*d[2]-a[2]*a[2]-d[1]*d[1]-a[1]*a[1]

Table 5-13 The coefficients of (5-12), in terms of the goal point position, orientations and the robot Denavit-Hartenberg parameters. la[i]=λ_i, mu=μ_i

CHAPTER 6

ALGORITHMS FOR LINEAR EQUATIONS

In Chapter 5 it was shown that solving linear equations is the bottleneck to high-speed execution of programs in the ConsolC family, in particular the IPO solver known as robot64p2gp. As linear equations is a task common to many Numerical Processing applications, an in depth study will be made. The purpose is to identify the critical parts of the most widely used algorithms (e.g. Gauss, LU, Crout, Doolittle), and study the properties that are critical to the speed performance of the algorithms. This knowledge is used as basis for Chapter 7, whose purpose is to evaluate how well the algorithms can be implemented on architectures that are either commercially available or that have been proposed in the research literature. The platforms that will be considered are

- Commercial DSP chips and RISC processors
- Systolic Arrays
- Vector Processors and Massively Parallel Architectures

Suitable *custom* architectures for an Application Specific Processor will then be considered in Chapter 8.

6.1 "Realification" of complex equations

Before starting to look at algorithms, there is one small detail that needs to be discussed. The equations we would like to solve have the form of a system with variables and coefficients that are complex numbers:

$$\mathbf{Ax} = \mathbf{b} \qquad \mathbf{A} \in \mathbf{C}^{NxN} \qquad \mathbf{x} \in \mathbf{C}^{N} \qquad \mathbf{b} \in \mathbf{C}^{N} \tag{6-1}$$

This poses additional problems for computers that do not compute directly with complex numbers (most do not), so in the rest of this dissertation we shall assume that the system has been "realified" as described next. By splitting up the matrix and the vectors into their real and imaginary parts, the system can be rearranged in the following way:

$$(A_1 + iA_2) \cdot (x_1 + ix_2) = b_1 + ib_2 \tag{6-2}$$

$$\begin{aligned} A_1x_1 - A_2x_2 &= b_1 \\ A_2x_1 + A_1x_2 &= b_2 \end{aligned} \tag{6-3}$$

$$\begin{bmatrix} A_1 & -A_2 \\ A_2 & A_1 \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} \tag{6-4}$$

This means that the original NxN complex system has become a 2Nx2N real system. From now on, when referring to an nxn system of equations, we typically mean the realified system with n=2N.

6.2 Algorithms for solving linear equations

There exists a number of different algorithms for solving nxn systems of linear equations. Some of the most common ones are plain Gaussian elimination (Gauss), LU decomposition (LU), Crout's method and Doolittle's method [dahlquist74]. These methods have many properties in common, but still are sufficiently dissimilar that their efficiency can be markedly different on different architectures, or even on one and the same architecture.

6.3 The Gauss/LU algorithm

The Gauss algorithm and the LU decomposition are so closely related that they will be treated together. Figure 6-2 contains the program text for a function which solves **Ax=b** using the Gauss/LU method. The basic purpose of the Gauss/LU algorithm is to factor **A** into a product **LU** where **L** is an nxn lower triangular matrix and **U** is an nxn upper triangular matrix. The only difference between plain Gauss and LU is that in Gauss, the **L** matrix is discarded whereas in LU, both **L** and **U** are stored, typically replacing the contents of the original **A** matrix. We assume that the reader is familiar with the basic idea of the Gauss/LU algorithms, which can be found in [dahlquist74], [strang80] or almost any other textbook on linear algebra or matrix computations. What we want to concentrate on here is the data storage, data addressing and computational aspects of the algorithm. For simplicity, we first consider Gauss/LU without *pivoting*. Pivoting means to rearrange (either literally or via an extra level of indirect addressing) the remaining rows of **A** (which now contains **L,U** under construction) so that the row with the largest (absolute value) first

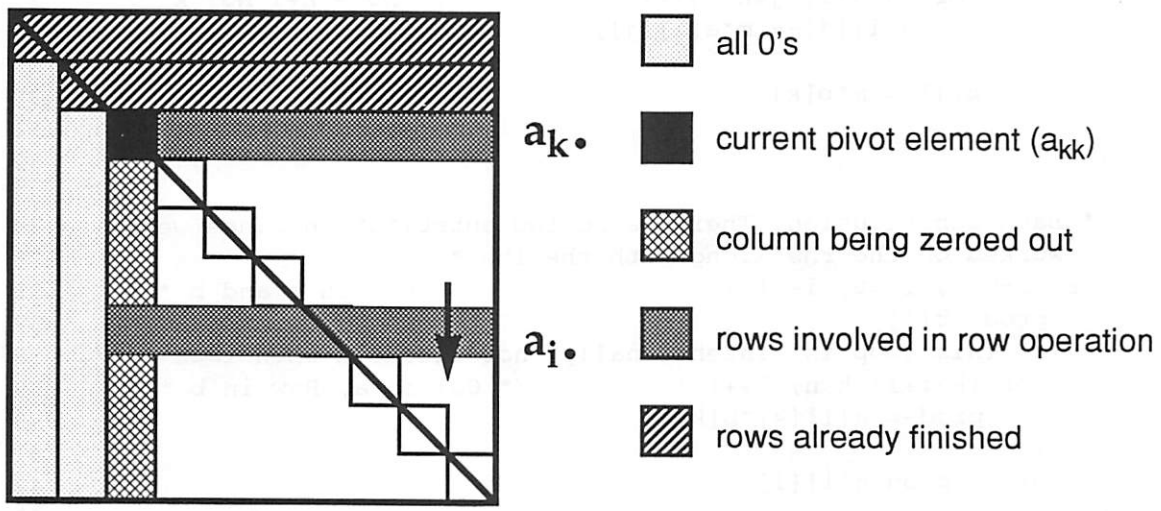


Figure 6-1 Gauss/LU step number **k**. All the numbers underneath the pivot element are about to be zeroed out by primitive row operations of the form

$$\mathbf{a}_{i\cdot} = \mathbf{a}_{i\cdot} - \left(\frac{a_{ik}}{a_{kk}}\right)\mathbf{a}_{k\cdot} \qquad i = k+1 \rightarrow n \qquad \text{(6-5)}$$

```
#include "linreal.h"                                                    1
#define fptype double                                                   2
                                                                        3
int linsol (n, a, b)                                                    4
     int        n;                                                      5
     fptype     a[N][N], b[N];                                          6
{                                                                       7
    fptype  temp, test, m, prod;                                        8
    int     i, j, k;                                                    9
    fptype  eps0= 1.0e-22;                                             10
                                                                       11
    /* Forward elimination */ /* k is the diagonal index */           12
    for (k=0; k<n-1; k++) {                                            13
        /* Check for singularity */                                    14
        temp= a[k][k];                                                 15
        if (temp<0.0) temp= -temp;                                     16
        if (temp<=eps0) {                                              17
            printf("linsol: singular matrix encountered (k=%d)\n",k); 18
            printf("temp= %le\n", temp);                               19
            return(-1);                                                20
        }                                                              21
                                                                       22
        /* Perform elimination step */                                 23
        for (i=k+1; i<n; i++) {              /* Note k+1 NOT k */      24
            /* Store m-factor where the 0's in U would go */           25
            m = (a[i][k]*=(1/a[k][k]));                                26
            for (j=k+1; j<n; j++) {          /* Note k+1 NOT k */      27
                a[i][j]-= m*a[k][j];                                   28
            }                                                          29
            b[i]-= m*b[k];                                             30
        }                                                              31
    }                                                                  32
                                                                       33
    /* Back substitution. There is no fwd substitution since we       34
       worked on the rhs along with the lhs */                         35
    for (i=n-1; i>=0; i--) {                /* Row in a and b */       36
        prod= b[i];                                                    37
        /* This loop is *intentionally* not executed when i==n-1 */   38
        for (k=i+1; k<n; k++) {          /* Col in a, Row in b */      39
            prod-= a[i][k]*b[k];                                       40
        }                                                              41
        b[i]= prod/a[i][i];                                            42
    }                                                                  43
                                                                       44
    return;                                                            45
}                                                                      46
```

Figure 6-2 Gauss/LU algorithm without pivoting. A common refinement is to store (1/a[k][k]) on top of a[k][k], since it is needed 2•(n-k) times

```
int linsol (n, a, b)                                                    1
     int        n;                                                      2
     fptype     a[N][N], b[N];                                          3
{                                                                       4
    fptype      temp, test, m, prod, eps0= 1.0e-22, y[N];               5
    int         i, j, k, imax, itemp, ir, kr, row[N];                   6
    /* Start with the identity permutation */                           7
    for (i=0; i<n; i++) row[i]= i;                                      8
    /* Forward elimination, k is the diagonal index */                  9
    for (k=0; k<n-1; k++) {                                            10
        /* Find largest row leader in remaining (n-k)x(n-k)matrix.*/    11
        temp=0; imax=k;                                                12
        for (i=k; i<n; i++) {                                          13
            ir= row[i];test= a[ir][k];                                 14
            if (test<0) test= -test;                                   15
            if (test>temp) {imax=i; temp= test;}                       16
        }                                                              17
        if (temp<=eps0) {                                              18
            printf("linsol: singular matrix encountered (k=%d)\n",k); 19
            printf("temp= %le\n", temp); exit(-1);                     20
        }                                                              21
        /* Change permutation arrays to make a[imax][k] the pivot */   22
        itemp=row[k]; row[k]=row[imax]; row[imax]=itemp;               23
        /* Perform elimination step */                                 24
        kr= row[k];                      /* Location of pivot */       25
        for (i=k+1; i<n; i++) {          /* Note k+1 NOT k */          26
            ir= row[i];                                                27
            m = (a[ir][k]*=(1/a[kr][k]));                              28
            for (j=k+1; j<n; j++) {      /* Note k+1 NOT k */          29
                a[ir][j]-= m*a[kr][j];                                 30
            }                                                          31
            b[ir]-= m*b[kr];             /* Right hand side */         32
        }                                                              33
    }                                                                  34
    /* Back substitution */                                            35
    for (i=n-1; i>=0; i--) {                                           36
        ir  = row[i];                    /* Row in a and b */          37
        prod= b[ir];                                                   38
        for (k=i+1; k<n; k++) {          /* Col in a, Row in b */      39
            kr= row[k];                                                40
            prod-= a[ir][k]*b[kr];                                     41
        }                                                              42
        b[ir]= prod/a[ir][i];                                          43
    }                                                                  44
    /* Unscramble b into y and then copy back into b */                45
    for (i=0; i<n; i++) { ir =row[i]; y[i]= b[ir]; }                   46
    for (i=0; i<n; i++) { b[i]= y[i]; }                                47
    return;                                                            48
}                                                                      49
```

Figure 6-3 Gauss/LU algorithm with partial (row) pivoting

element becomes the pivot row. This is done because it improves the numerical stability of the algorithm by avoiding divisions by small numbers.

6.3.1 Architectural implications

Without pivoting, the C code for LU/Gauss has the form shown in Figure 6-2. Figure 6-1 shows in pictorial form what happens at step k of the algorithm. Think of k as the counter that picks the next element on the diagonal as the pivot element. The critical statements from Figure 6-2 have been extracted and analyzed in Table 6-1. *This table is a key to understanding the architectural*

Line	Operation	Mult	Div	Add	Read	Write
Without pivoting						
26	`m=(a[i][k]/=a[k][k]);`	0	1	0	2	1
28	`a[i][j]-= m*a[k][j];`	1	0	1	2	1
30	`b[i]-= m*b[k];`	1	0	1	2	1
40	`prod-= a[i][k]*b[k];`	1	0	1	2	0
42	`b[i]= prod/a[i][i];`	0	1	1	1	1
With partial (row) pivoting						
25	`kr= row[k];`	0	0	1	1	0
27	`ir= row[i];`	0	0	1	1	0
28	`m=(a[ir][k]/=a[kr][k]);`	0	1	0	2	1
30	`a[ir][j]-= m*a[kr][j];`	0	1	0	2	1
32	`b[ir]-= m*b[kr];`	0	1	0	2	1
37	`ir  = row[i];`	0	0	1	1	0
41	`prod-= a[ir][k]*b[kr];`	1	0	1	2	0
43	`b[ir]= prod/a[ir][i];`	0	1	1	1	1

Table 6-1 The key arithmetic instructions of the Gauss/LU algorithm

requirements of the Gauss/LU algorithm. The main result to notice is that most of the important operations (lines 26,28,30) in Gauss/LU require 3 memory accesses (2 reads and 1 write) each time they are executed. Any architecture which aims to execute the inner loop (line 28) of the algorithm at the rate of one statement per cycle must support at least this amount of memory traffic. If the datapath can finish line 28 at the rate of one result per cycle, it will do no good unless the memory is a 3-port which can provide 2 reads and 1 write to supply the operands and store the results. Note here that we assume that items such as the variable **m** is stored in a local register while the computation is taking place to avoid further demands on the memory bandwidth.

Memory bandwidth

The main consequence of the previous paragraph is that a memory system that only supports 1 (or 2) operations per cycle will slow down the execution of the algorithm at least by a factor of 3 (or 2), even if the datapath itself is able to keep up with the computation.

In addition to analyzing memory accesses, it is also useful to count how many times the various

Line	Operation	Repetition count	For n=16
	Without pivoting		
13	`for (k=0; k<n-1; k++) {`	n	16
24	`for (i=k+1; i<n; i++) {`	(1/2)n(n-1)-1	119
26	`m=(a[i][k]/=a[k][k]);`	(1/2)n(n-1)-1	119
28	`a[i][j]-= m*a[k][j];`	(1/6)(n-1)n(2n-1)	1240
30	`b[i]-= m*b[k];`	(1/2)n(n-1)-1	119
37	`prod= b[i];`	n	16
40	`prod-= a[i][k]*b[k];`	(1/2)n(n-1)-1	119
42	`b[i]= prod/a[i][i];`	n	16

Table 6-2 Statement profile for Gauss/LU algorithm without pivoting

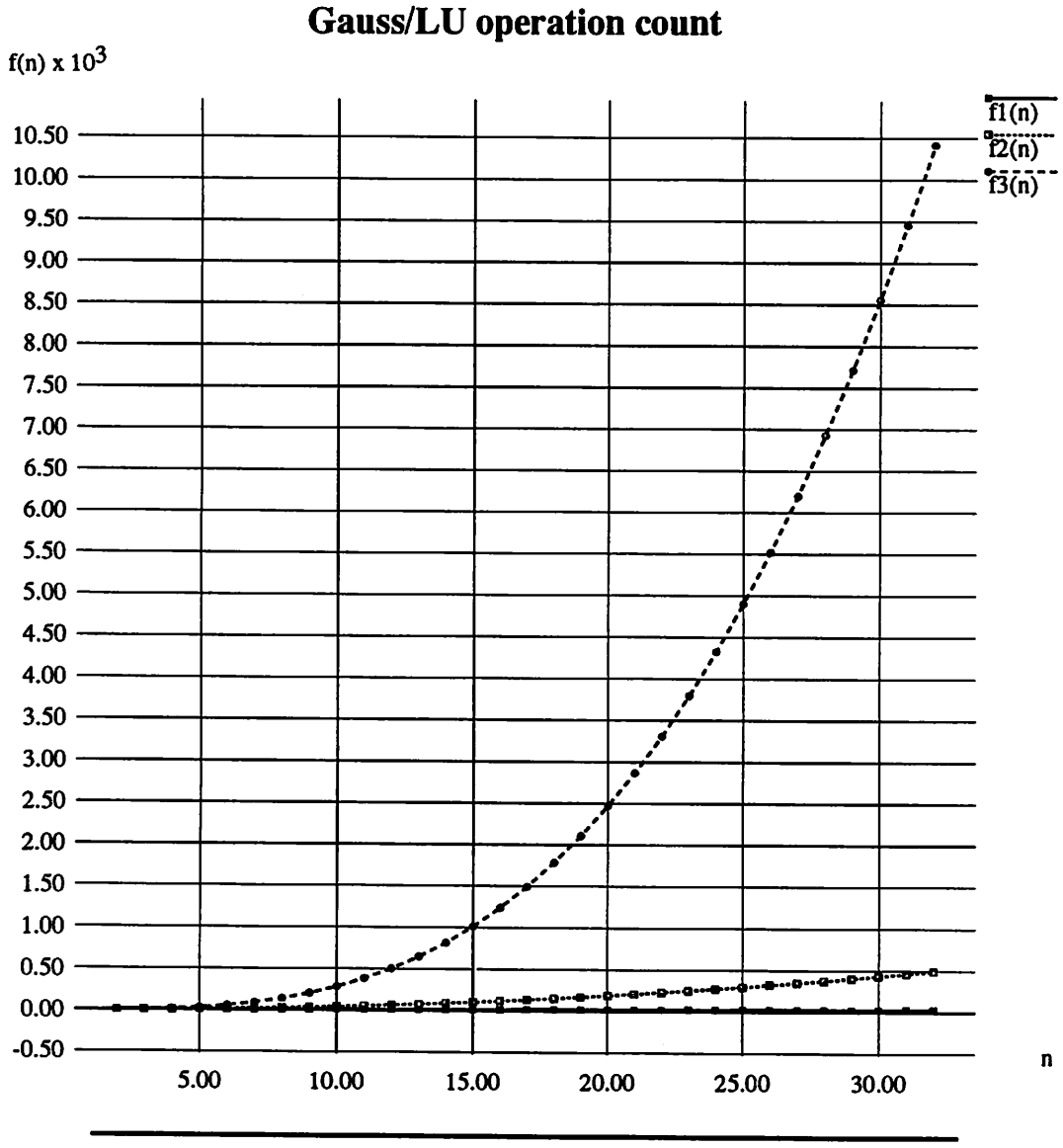


Figure 6-4 Repetition count for selected lines of Gauss/LU algorithm, as function of n

statements of the algorithm are executed. Table 6-1 shows how many times each of the important statements of Gauss/LU is executed. The numbers are found by looking at the loops and applying the formulas

$$\sum_{k=1}^{n} k = \frac{1}{2} n (n-1) \qquad \text{and} \qquad \sum_{k=1}^{n} k^2 = \frac{1}{6} n (n+1) (2n+1) \tag{6-6}$$

to derive the repetition count functions

$$f_1(n) = n \qquad f_2(n) = \frac{1}{2}n(n-1) - 1 \qquad f_3(n) = \frac{1}{6}(n-1)n(2n-1) \tag{6-7}$$

Pipelining and pipelining margins

The datapath operations of the Gauss/LU algorithm are multiplications, divisions (or reciprocals), additions, subtractions, or combinations thereof, such as multiply-accumulate (MAC) and multiply-add (MADD). For example, the innermost loop of Gauss/LU (line 28) is a MADD instruction.

By pipelining we mean (in this context) to partition a datapath operation and its corresponding hardware on a sub-functional level so that e.g an add is broken down into several clocked stages, each of which can be clocked at a higher speed than would be possible if the entire add operation was to be performed in one stage.The main property of an algorithm which allows us to use pipelining is that the operands being fed into the pipeline must not depend on the results that are currently being computed inside the pipeline. If a dependency exists, the pipeline must idle until the results are available to be used as input operands again. The MAC operation is the simplest example of an operation that is not easily pipelineable: For example, the MAC sequence

$$p = b_i - \sum_{k=i+1}^{n-1} a_{ik}b_k \tag{6-8}$$

which would typically be implemented as

```
p= b[i]; for (k=i+1; k<n; k++) p-= a[i][k]*b[k];
```

cannot have a pipelined subtraction because we need the result (the running sum) to feed back as one of the arguments to the next addition.

During the elimination process, the Gauss/LU algorithm has the very favorable property that whenever a matrix element is read out to be updated (line 28), it will not be needed again until $(n-k)^2$ operations later, where n is the size of the matrix and k is the step number (k=0:n-2) as in Figure 6-2. Of course, as k becomes larger, this *pipelining margin* becomes smaller and smaller (but never smaller than 4, for k=n-2). Even for the next-to-last step (k=n-3), the margin is 9

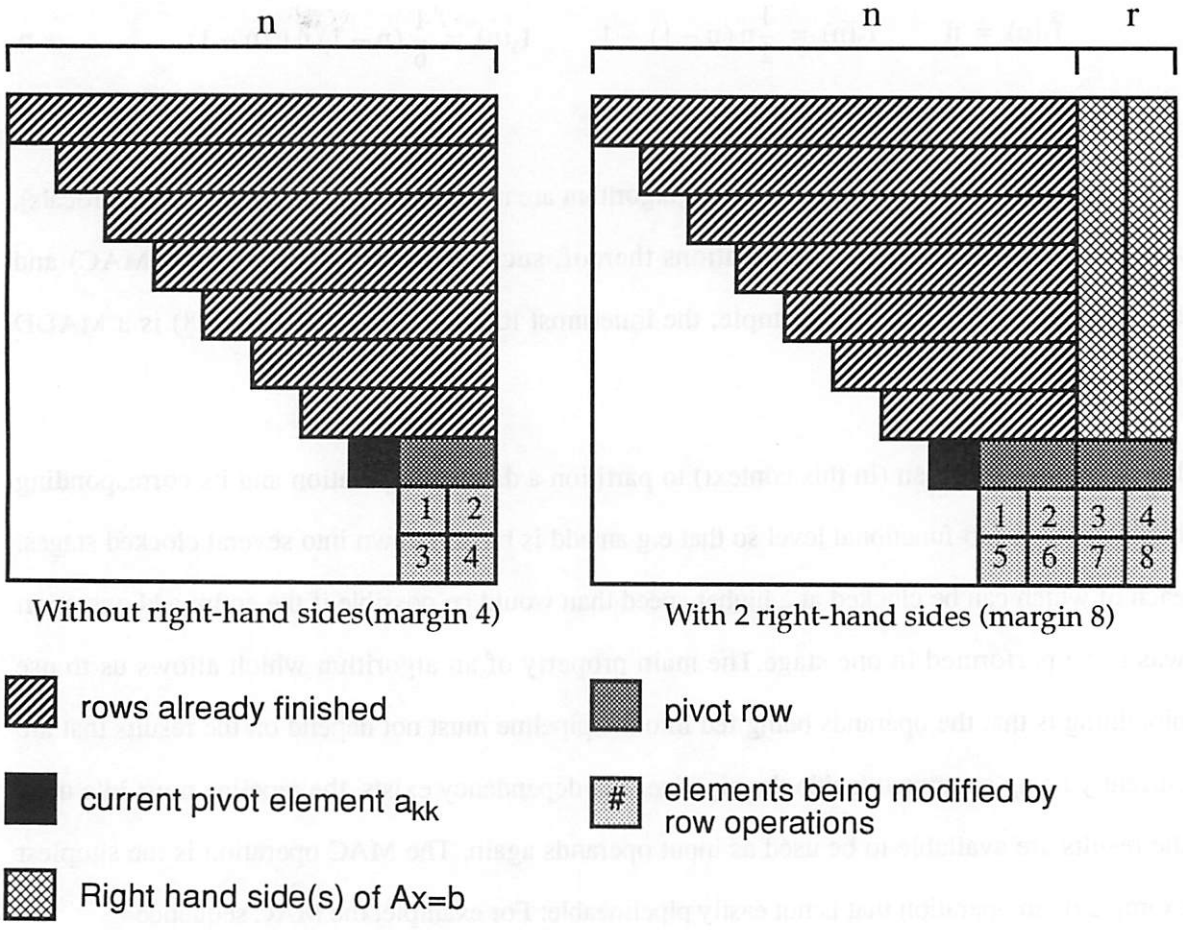


Figure 6-5 Pipelining margin (PM)

PM at step k
= the number of steps before an update target is needed as input again
= the number of update steps in step k
= the size of the (n-k)x(n-k) submatrix that is being updated
= $(n-k)^2$ steps, k=1:n-2, if there are no right-hand sides
= (n+r-k)(n-k) steps, k=1:n-2, if there are r right-hand sides

The minimum pipelining margin over all k=1:n-2
= 2*2 = 4; if there are no right-hand sides
= (r+2)*2; if there are r right-hand sides

General formula

$$PM(n, k, r) = (n + r - k) \cdot (n - k) \qquad (6\text{-}9)$$

operations, meaning that we could potentially have a datapath with 9 pipeline stages (for a MADD), and be able to keep it 100% occupied all up until the very last iteration of Gauss/LU. This is an important observation that can be used during architecture design. If the system that we are solving has one or more right-hand sides (Ax=b, Ax=b', ...), the pipelining margin increases to (n+r-k)(n-k), with a minimum value of (2+r)*2 for k=n-2. See Figure 6-5.

The concept of pipelining margins appears to have general applicability. The following is an attempt at defining the term more precisely.

Definition 7.1 [pipelining margin] The pipelining margin of an expression, in the context of the remaining algorithm, is the minimum number of operations which take place between the initiation of the evaluation of the expression and the time at which the result is needed as an input to another expression of the algorithm.

Pipeline interleaving

The back-substitution part of the algorithm (line 40) is a problem spot because it contains the dreaded MAC operation, and hence cannot easily be pipelined. However, there is another trick that can be applied here. The computation of the loop can be rearranged (Figure 6-6) so that the we compute each running sum b[i] one piece at a time, instead of finishing each one of them completely before moving on to the next one. Each time b[i] is updated, it will not be needed again until n-k operations later. This is not quite as favorable as the $(n-k)^2$ pipelining margin seen in the elimination loop, but it helps, especially for large matrices. The term *pipeline interleaving* [lee86] was coined to describe the general concept of interleaving independent operations on a pipelined datapath.

Using pipeline interleaving, the biggest source of pipeline bubbles in the overall algorithm is the computation of 1/a[k][k] in line 12. A common refinement is to store a[k][k]-1 on top of a[k][k], since a[k][k] is no longer needed and the inverse will be needed 2•(n-k) additional times. This will minimize the problem.

Pivoting

Pivoting is a numerical safeguard made necessary by the limited range and precision of computer arithmetic, and is sometimes also a *fundamental* requirement reflecting a need to rearrange the equations of a linear system. [dahlquist74] (section 5.3.3 p150) contains examples that show the importance of pivoting. In the program of Figure 6-3, pivoting is implemented (lines 11-17) by a search among the remaining rows (i=k:n) for the row with the largest leading element, and then swapping this row (called the *pivot row*) with row k. The swapping of the rows is not literally performed. Instead, we use a permutation table (the array `row[]`) which translates any given row index into the physical index for where that row is stored. This saves the time otherwise spent on copying rows from one location to another.

Pivoting slows down the Gauss/LU algorithm, both because of the search and because of the extra level of indirection caused by the table lookup. The lookup overhead can be reduced somewhat by precomputing variables such as `ir=row[i]` (line 27) whenever multiple elements in the same row will be accessed (line 30). The search overhead is not easily reducible on a general purpose computer.

```
/* Regular back substitution */                                    1
for (i=n-1; i>=0; i--) {          /* Row in a, Row in b */         2
    prod= b[i];                                                    3
    for (k=i+1; k<n; k++) {       /* Col in a, Row in b */         4
        prod-= a[i][k]*b[k];                                       5
    }                                                              6
    b[i]= prod/a[i][i];                                            7
}                                                                  8
                                                                   9
/* Interleaved back substitution */                               10
for (k=n-1; k>=0; k--) {          /* Row in a, Row in b */         11
    b[k]= b[k]/a[k][k];                                            12
    for (i=k-1; i>=0; i--) {      /* Col in a, Row in b */         13
        b[i]-= a[i][k]*b[k];                                       14
    }                                                              15
}                                                                  16
```

Figure 6-6 Regular versus Interleaved back substitution

Summary of Gauss/LU characteristics

This section has presented several important characteristics of the Gauss/LU algorithm, including memory bandwidth requirements, the MADD (not MAC) character of the algorithm, the concept of pipelining margin and how a positive margin can be created by interleaving, and finally pivot searching and permutation lookups.

6.4 The Crout algorithm

The Crout algorithm [dahlquist74] is an alternative formulation of the LU decomposition computation. It is different from the Gauss/LU method in that it does not involve repetitive *updates* of the same matrix element. Instead, the elements of **L,U** (also called **M,U** in Dahlquist's terminology) are computed completely one at a time using a an accumulation (MAC) sequence. As in Gaussian elimination, in step *k* the *k*th column of **L** and the *k*th row of **U** are determined, but in Crout the elements a_{ij} with i,j>k are not touched until later steps. Crout's algorithm can be derived

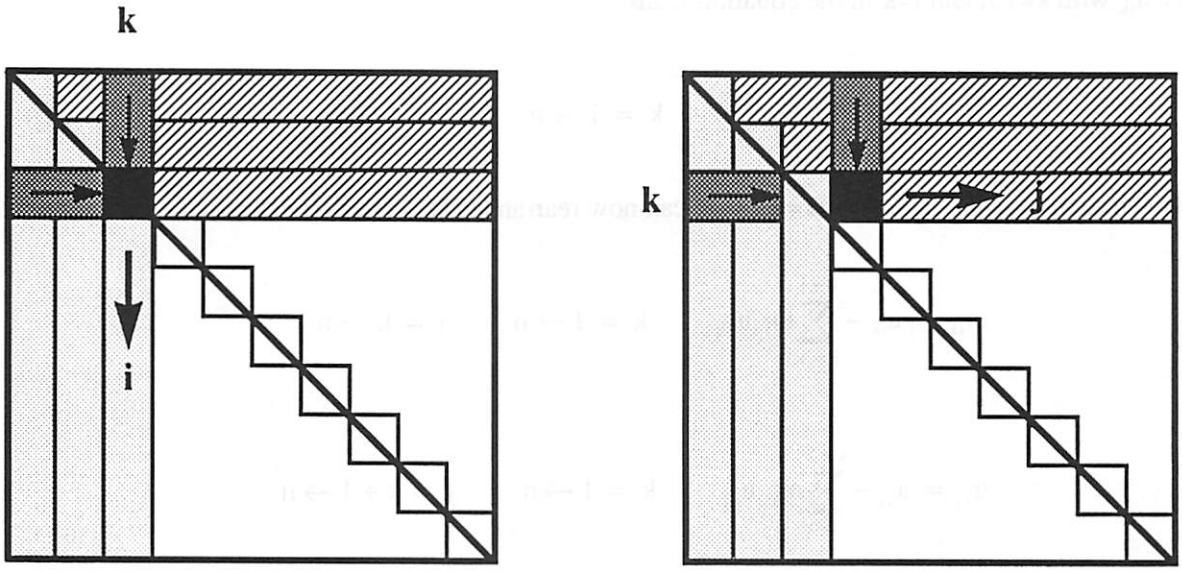


Figure 6-7 The memory access patterns for the Crout algorithm. Left side shows the computation of m_{ik} and right side shows the computation of u_{kj}

$$m_{ik} = a_{ik} - \sum_{p=1}^{k-1} m_{ip} u_{pk} \qquad i = k \rightarrow n \qquad u_{kj} = a_{kj} - \sum_{p=1}^{k-1} m_{kp} u_{pj} \qquad j = k+1 \rightarrow n$$

from the matrix equation A=LU as follows: The element form of A=LU is

$$a_{ij} = \sum_{p=1}^{r} m_{ip} u_{pj} \qquad r = \min(i, j) \tag{6-10}$$

where the use of r=min(i,j) is a convenient way to exclude the 0-elements of the triangular matrices from the sum. Equation (6-10) with i=1:n and j=1:n produces n^2 equations for the unknowns of L,U. Note that L,U each contains (1/2)n(n+1) unknown elements, for a total of n(n+1) unknowns. This is a reflection of the fact that the LU decomposition of a matrix is not unique. In Crout's method, one chooses u_{kk}=1 (k=1:n) to get n^2 equations in n^2 unknowns out of (6-10). Now consider two cases of (6-10). If we look at the upper triangle of the matrix $A=(a_{ij})$, that is, a_{kj} with k=1:n and j=k+1:n, the equation reads

$$a_{kj} = \sum_{p=1}^{k} m_{kp} u_{pj} \qquad k = 1 \rightarrow n \qquad j = k+1 \rightarrow n \tag{6-11}$$

Note that r=min(k,j)=k in this case. Similarly, if we look at the lower triangle and the diagonal, that is, a_{ik} with k=1:n and i=k:n, the equation reads

$$a_{ik} = \sum_{p=1}^{k} m_{ip} u_{pk} \qquad k = 1 \rightarrow n \qquad i = k \rightarrow n \tag{6-12}$$

By virtue of the choice u_{kk}=1 (k=1:n) we can now rearrange (6-12) and (6-11) as

$$m_{ik} = a_{ik} - \sum_{p=1}^{k-1} m_{ip} u_{pk} \qquad k = 1 \rightarrow n \qquad i = k \rightarrow n$$

$$u_{kj} = a_{kj} - \sum_{p=1}^{k-1} m_{kp} u_{pj} \qquad k = 1 \rightarrow n \qquad j = k+1 \rightarrow n \tag{6-13}$$

Starting with k=1 and progressing towards k=n, these equations can then be evaluated in order so that at each k we find $m_{kk}, \ldots, m_{nk}$ (one column of L) and then $u_{k+1,k}, \ldots, u_{kn}$ (one row of U) at a time. Figure 6-7 is a picture of the way Crout's method progresses through the matrix, and how L,U can be stored on top of A. C code for Crout's algorithm with and without pivoting is included

Line	Operation	Mult	Div	Add	Read	Write
Without pivoting						
26	`prod-= a[k][p]*a[p][j];`	1	0	1	2	0
33	`prod-= a[k][p]*b[p];`	1	0	1	2	0
44	`prod-= a[i][k]*b[k];`	1	0	1	2	0

Table 6-3 The key arithmetic instructions of the Crout algorithm

in Figure 6-8 and Figure 6-9, respectively.

Crout's method has different properties than the Gauss/LU algorithm. The most important property of Crout is that the inner loops (lines 26,33,44 of Figure 6-8) are of the multiply-accumulate (MAC) variety as opposed to the multiply-add (MADD) type found in the Gauss/LU algorithm. The consequence is that Crout's method is not as amenable to pipelining as the Gauss/LU algorithm. The *advantage* of Crout is that none of the inner loops have more than 2 memory operations (both are read operations), whereas Gauss/LU has 3. This fact is of course related to the MAC or MADD character of the algorithms: If the algorithm performs accumulations, the running sum is kept in a datapath accumulator register. If the algorithm is of the multiply-add-update type, the result must be written to memory every cycle.

Summary of Crout characteristics

The Crout algorithm is more attractive than Gauss/LU when it comes to memory bandwidth, but less attractive when it comes to pipelining margin (there is none). As for pivoting (see Figure 6-9), Crout is *worse* than Gauss/LU, because the algorithm strides through rows in its inner loop (line 25), meaning that a row address must be permuted (translated) at every step of the inner loop. This was not the case in the Gauss/LU algorithm.

```
#include "linreal.h"                                                    1
#define fptype double                                                   2
                                                                        3
int linsol (n, a, b)                                                    4
     int        n;                                                      5
     fptype     a[N][N], b[N];                                          6
{                                                                       7
    fptype      prod, bprod, eps0= 1.0e-22;                             8
    int         i, j, k, p;                                             9
                                                                       10
    /* k is the step index */                                          11
    for (k=0; k<n; k++) {                                              12
        /* Compute Dahlquist's m[i,k] and store in a[i,k] */           13
        for (i=k; i<n; i++) {                                          14
            prod= a[i][k];                                             15
            for (p=0; p<k; p++) {                                      16
                prod-= a[i][p]*a[p][k];                                17
            }                                                          18
            a[i][k]=prod;                                              19
        }                                                              20
                                                                       21
        /* Compute Dahlquist u[k,j] and store in a[k,j] */:            22
        for (j=k+1; j<n; j++) {                                        23
            prod= a[k][j];                                             24
            for (p=0; p<k; p++) {                                      25
                prod-= a[k][p]*a[p][j];                                26
            }                                                          27
            a[k][j]= prod/a[k][k];                                     28
        }                                                              29
        /* Forward elimination of L*y = b */                          30
        prod= b[k];                                                    31
        for (p=0; p<k; p++) {                                          32
            prod-= a[k][p]*b[p];                                       33
        }                                                              34
        b[k]= prod/a[k][k];                                            35
    }                                                                  36
                                                                       37
    /* Back substitution. There is no fwd substitution since we       38
       worked on the rhs along with the lhs. Assume u(i,i)=1 (Crout) */39
    for (i=n-1; i>=0; i--) {                                           40
        prod= b[i];                                                    41
        /* This loop is *intentionally* not executed when i==n-1 */   42
        for (k=i+1; k<n; k++) {          /* Col in a, Row in b */      43
            prod-= a[i][k]*b[k];                                       44
        }                                                              45
        b[i]= prod;                                                    46
    }                                                                  47
}                                                                      48
```

Figure 6-8 Crout algorithm without pivoting (lincrsolnr.1.c)

```
int linsol (n, a, b)                                                    1
     int        n; fptype     a[N][N], b[N];                           2
{                                                                       3
    fptype      temp, test, prod, eps0= 1.0e-22, y[N];                  4
    int         i, j, k, p, imax, itemp, ir, kr, pr, row[N];            5
                                                                        6
    for (i=0; i<n; i++) row[i]=i; /* Identity permutation */            7
    /* k is the step index */                                           8
    for (k=0; k<n; k++) {                                               9
        /* Find next pivot row: imax= argmax{i=k:n) |a[irow[i],k]|*/    10
        temp=0; imax=k;                                                 11
        for (i=k; i<n; i++) {                                           12
            ir= row[i]; test= a[ir][k]; if (test<0) test= -test;        13
            if (test>temp) {imax=i; temp= test;}                        14
        }                                                               15
        if (temp<=eps0) {                                               16
            printf("linsol: singular matrix encountered (k=%d)\n", k);17
            printf("temp= %le\n", temp); exit(-1);                      18
        }                                                               19
        /* Change permutation arrays to make a[imax][k] the pivot */    20
        itemp=row[k]; row[k]=row[imax]; row[imax]=itemp;                21
        /* Compute Dahlquist's m[i,k] and store in a[i,k] */            22
        for (i=k; i<n; i++) {                                           23
            ir=row[i]; prod= a[ir][k];                                  24
            for (p=0; p<k; p++) {pr=row[p]; prod-= a[ir][p]*a[pr][k];}25
            a[ir][k]=prod;                                              26
        }                                                               27
        /* Compute Dahlquist's u[k,j] and store in a[k,j] */            28
        kr=row[k];                                                      29
        for (j=k+1; j<n; j++) {                                         30
            prod= a[kr][j];                                             31
            for (p=0; p<k; p++) {pr=row[p]; prod-= a[kr][p]*a[pr][j];}32
            a[kr][j]= prod/a[kr][k];                                    33
        }                                                               34
        /* Forward elimination of L*y = b */                           35
        kr= row[k]; prod= b[kr];                                        36
        for (p=0; p<k; p++) {pr= row[p]; prod-= a[kr][p]*b[pr];}        37
        b[kr]= prod/a[kr][k];                                           38
    }                                                                   39
    /* Back substitution */                                             40
    for (i=n-1; i>=0; i--) {                                            41
        ir  = row[i]; prod= b[ir];                                      42
        for (k=i+1; k<n; k++) {kr= row[k]; prod-= a[ir][k]*b[kr];}      43
        b[ir]= prod;                                                    44
    }                                                                   45
    /* Unscramble b into y and then copy back into b */                 46
    for (i=0; i<n; i++) {ir =row[i]; y[i]= b[ir];}                      47
    for (i=0; i<n; i++) {b[i]= y[i];}                                   48
}                                                                       49
```

Figure 6-9 Crout algorithm with pivoting (lincrsolpr.1.c)

6.5 The Doolittle algorithm

The Doolittle algorithm is similar to the Crout algorithm in that it can be derived from (6-10). The basic idea is the same, but now $m_{kk}=1$ (k=1:n) is chosen instead $u_{kk}=1$ (k=1:n). The derivation again starts by splitting Equation (6-10) into two cases, this time first for the lower triangle and then for the upper triangle and the diagonal. For the lower triangle we have

$$a_{kj} = \sum_{p=1}^{k} m_{kp}u_{pj} \qquad k = 1 \rightarrow n \qquad j = k+1 \rightarrow n \tag{6-14}$$

and for the upper triangle/diagonal we have

$$a_{ik} = \sum_{p=1}^{k} m_{ip}u_{pk} \qquad k = 1 \leftarrow n \qquad i = k+1 \rightarrow n \tag{6-15}$$

By virtue of the choice $m_{kk}=1$ (k=1:n) we can now rearrange (6-14) and (6-15) as

$$u_{kj} = a_{kj} - \sum_{p=1}^{k-1} m_{kp}u_{pj} \qquad k = 1 \rightarrow n \qquad j = k \rightarrow n$$

$$m_{ik} = a_{ik} - \sum_{p=1}^{k-1} m_{ip}u_{pk} \qquad k = 1 \rightarrow n \qquad i = k+1 \rightarrow n \tag{6-16}$$

Starting with k=1 and progressing towards k=n, these equations can then be solved for $u_{kk}, \ldots, u_{kn}$ (one row of **U**) and then $m_{k+1,k}, \ldots, m_{nk}$ (one column of **L**) at a time.

Properties of Doolittle's algorithm

The difference between Crout and Doolittle from an implementation point of view is marginal. They have exactly the same properties when it comes to memory bandwidth, pipelining margin and permutation lookups.

6.6 Summary

The preceding material is an attempt to provide some insight into the nature of the various algorithms that can be used to solve dense systems of linear equations. Which algorithm is best depends greatly on what hardware architecture is available, so it is useful to summarize the key properties for future reference. Table 6-4 contains a relative ranking of Gauss/LU, Crout and Doolittle with respect the properties *memory bandwidth*, *pipelining margin* and *permutation lookup frequency.*

Property	Gauss/LU	Crout	Doolittle
memory bandwidth	–	+	+
pipelining margin	+	–	–
permutation lookup frequency	0	–	–

Table 6-4 Simplified view of some key properties of linear equation algorithms

CHAPTER 7

ConsolC IMPLEMENTATION ALTERNATIVES

The algorithms described in Chapter 6 can be implemented, with varying degrees of efficiency and difficulty, on a wide range of commercially available processors and also on a number of experimental architectures that have been described in the research literature. Examples of relevant processors and architectures are

- Commercial DSP chips
- Systolic arrays and Massively Parallel Architectures
- Standard microprocessors (RISC chips)
- Vector Processors and Supercomputers

This chapter is an investigation of the efficiency of a selection of architectures and processors. The purpose is to identify the main bottlenecks to *efficient* execution. For the memory subsystem, efficiency is measured as the memory's ability to retrieve operands and store results without idling the datapath. For the datapath, efficiency is generally measured in term of latency and throughput. The control unit (program sequencer and address generation unit) is judged on its ability to generate the addresses and control signals necessary to keep both the memory and the datapath

fully occupied with useful operations. One concrete metric for efficiency is the utilization rate of the datapaths, where utilization is defined as

$$\text{Utilization} = \frac{\text{ActiveCycles}}{\text{TotalCycles}} \qquad \text{or} \qquad U = \frac{A}{T} \tag{7-1}$$

The active cycles are the cycles where new datapath instructions are issued, as opposed to the idle cycles, where the datapath is waiting for operands to arrive or a status value to become available. Note that the utilization rate can depend both on the datapath itself (if a missing operand is currently being computed but not yet finished) and the memory (if the operand is coming from memory). This serves to illustrate that the overall performance of the system will depend both on individual properties of the major units (memory, datapath, control) and on how they work together.

7.1 Commercial DSP chips

The commercial DSP chips that will be considered in this study are the AT&T DSP32C, the Texas Instruments TMS 320C30 and the Motorola MC96002. It turns out that many of the relevant architectural features of these processors are the same. For this reason, only the DSP32C will be studied in great detail. Afterwards, the analogies between the DSP32C and the other processors can be drawn fairly easily and many of the conclusions will be the same. All the processors mentioned are high-end units that have built-in floating point hardware. This means that they are (at least arithmetically) suited for numerical processing tasks.

7.1.1 The AT&T DSP32C digital signal processor

A simplified block diagram of the DSP32C is shown in Figure 7-1 [att88]. The main architectural features of this processor is a highly multiplexed program/data bus (for simplicity referred to as just the data bus hereafter) which operates at 4 times the instruction rate, a multiplexed set of RAM/ROM banks, and a modestly pipelined (2 stages) floating point datapath, while at the same time having a highly pipelined approach to the fetching of operands, instructions and the writing of

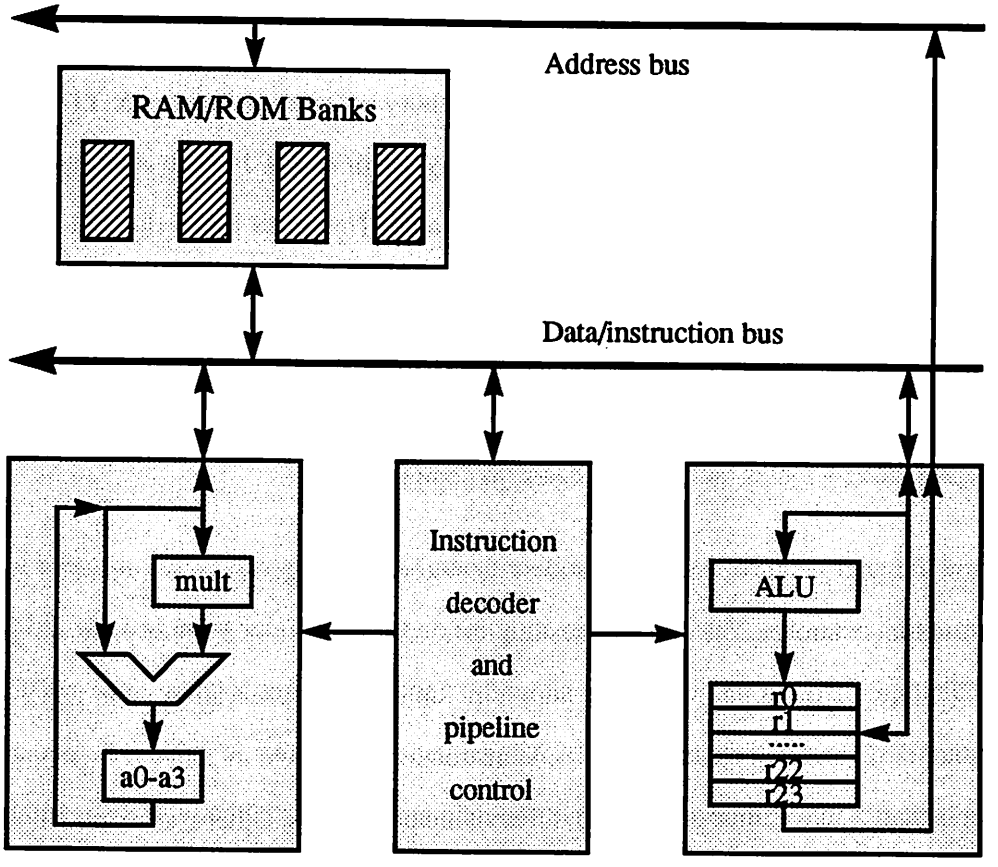


Figure 7-1 Block diagram of the AT&T DSP32C signal processor

State	0	1	2	3
Clock cycle	0	1	2	3
Clock time	20ns	20ns	20ns	20ns
Instruction	number k			
Data bus	Z_{k-4}	X_{k-3}	Y_{k-3}	I_k
Function	Write result	Read X operand	Read Y operand	Read Instruction

Table 7-1 Reservation table for data bus during one machine cycle. The clock frequency is 50MHz

results, where a given instruction may span as many as 4 instruction cycles.

The most general DSP32C instructions involve 4 memory accesses (denoted Z,X,Y,I) as shown in Table 7-1. An instruction cycle consists of 4 clock cycles or *states*. In each state, the databus is occupied by a different data item, as shown in the table. The index k is a running count of the machine cycles. What the table says is that the memory write (Z) corresponding to the instruction issued 4 cycles ago occurs in the same cycle as the operand reads (X,Y) for instruction k-3 and the reading of the opcode (I) for instruction k itself. This means that a complete accumulate with a memory store takes 4 cycles (actually 17 states) to complete.

To see how the architectural features manifest themselves during execution, we need to look at some instruction examples. The most interesting instructions from our point of view are the MAC and MADD type instructions, which are shown in Table 7-2. A reservation table which applies both for the MAC instructions and the MADD instructions is shown in Table 7-3. One important restriction is that X and Y must come from different memory banks. Otherwise there will be 1 *wait state* between the accesses, and a corresponding delay of the pipeline. In other words, each RAM needs 2 states per read operation. Fortunately, a new MAC/MADD instruction can be started every cycle as long as the accumulated value stored to Z is not needed as an operand until 4 cycles later at the earliest. Otherwise, one would have to wait the full 17 states. If X,Y are from the same memory bank, there will be 6 states per MAC/MADD instead of 4. As mentioned earlier, the floating point datapath is not pipelined except one stage between the multiplier and the adder. The delays of the most important hardware blocks are shown in Table 7-4. *Interleaving* is the term used to describe the practise of accessing X and Y from different memories. RAM[0-2] refer to the on-chip RAM banks whereas RAM[A-B] refer to additional off-chip memory banks. The number *n* denotes extra wait states.

7.1.2 Solving linear equations on the DSP32C

The purpose of this section is twofold. First it aims to establish how efficiently the DPS32C can

Instruction type	Generic format	Example
MAC	`[Z=]aN=[-]aM{+,-}Y*X`	`*r2++r18= a1= a0+ *r2 * *r2++r19`
MADD	`[Z=]aN=[-]Y{+,-}aM*X`	`*r2++r18= a1= *r2 + a0* *r2++r19`

Table 7-2 Generic form of MAC and MADD instructions. []=optional { }=alternative. aN, aM denote accumulator registers and X,Y,Z are memory locations. rN=address register

State	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3
RAM0	X	rd														Z wr	
RAM1		Y rd															
Data bus			X	Y										Z			
Multiplier																	
Adder																	

Table 7-3 Reservation table for relevant hardware units during multiply-accumulate or multiply-add

Block	Delay	Interleave with	Block	Delay
RAM[0-2]	2	Any other RAM	adder	4
RAM[A-B]	2+n	Only RAM[0-2]	multiplier	5

Table 7-4 Delays of DSP32C hardware blocks (number of states)

be programmed to solve linear equations. Secondly, we want to see how the architecture can be improved to increase the speed of execution.

As mentioned in the introduction to this chapter, one efficiency measure for a given processor is the utilization rate of its datapaths. This makes sense situations in which the datapath is a given and unchangeable entity.

Gauss/LU on the DSP32C

To get an exact, provably minimal cycle count for an algorithm on a given architecture is usually not feasible. However, one can usually get bounds on the performance by studying the inner loops of the algorithm and see how fast they can be executed. The inner loop of the Gauss/LU algorithm is the elementary row operation

```
m= (a[ir][k]*=(1/a[k][k]));
for (j=k+1; j<n; j++) a[ir][j]-= m*a[kr][j];
```

The basic elimination operation is a[]= a[] - m*b[]. It can be performed as

```
do 0, r12
*r1++ = a1 = *r1 + a0 * *r2++
```

where r1 is a pointer to row a[], r2 is a pointer to b[] and m is stored in a0. If a[] and b[] come from the same memory bank (which is natural, since they are rows of the same matrix), it is possible to execute this loop at a rate of 6 (not 4) states/iteration. The 2 additional states are wait states incurred by having to access the same memory bank in succession, as opposed to interleaving two different banks. This means that there is a basic inefficiency factor of 1.5 between what the processor can actually do and what the datapath would be able to produce *if* the operands were delivered on time and the result could be written on time.

Additional slowdowns will occur in the outer loops of the algorithm due to such tasks as pivot searching, permutation lookups and loop administration. As noted above, deriving the exact numbers is difficult. One alternative is to use sample code written by an expert programmer and compare the execution time (cycles) to the theoretical minimum which can be found by simple counting of operations in the algorithm. The DSP32C comes with a number of hand-coded subroutines for various purposes, and one of them (matinv.lib.s) is a routine that finds the inverse of an nxn matrix **A**. The inverse of **A** is the matrix **X** which satisfies **AX=I**. This equation can be viewed as n systems of linear equations with the same coefficient matrix **A** but n different right-hand sides, namely the n columns of **I**. These systems can all be solved together by treating the different right-hand sides at the same time during elimination. The work (MADD operations)

involved is

$$\text{work}(AX=I) = 2*\text{forward\_elim}(Ax=b) + N*\text{back\_subst}(Ax=b) \quad (7\text{-}2)$$

which in turn is

$$\text{work}(AX=I) = 2*(1/3)(N^3-N) + N*(1/2)N^2 = (7/6)N^3 - (2/3)N \quad (7\text{-}3)$$

This amounts to 4768 operations for N=16, or 4*4768=19072 DSP32C states if the processor datapaths works at 100% efficiency. This number was compared to the state execution count for the hand-coded routine matinv.lib.s, including additional loop optimizations performed by myself in order to make the code as efficient as at all possible. The most efficient hand-coded version was a factor of 3.4 less efficient than a fully exploited datapath would be (Table 7-5). It should be noted that matinv.lib.s uses pointer arithmetic for all array addressing, so that all the overhead which is seen here essentially comes from pivot searching, permutation (which is performed by swapping lines), loop administration and memory conflicts.

Potential speedup

What are the weaknesses of the DSP32C chip with respect to the Gauss/LU algorithm? A factor of 3.4 datapath inefficiency is actually not bad for a general purpose chip, but it could probably be reduced to almost 1 by a special purpose memory architecture and controller. Assuming the datapath (in its current incarnation) could be kept 100% busy, the next step to achieve additional

name	states (factor)	wait states	loops optimized
matinv.lib.s	83957(4.4)	7865	none
matinv.lib1.s	71701(3.7)	-	2 (D,E)
matinv.lib2a.s	65107(3.4)	-	3 (D,E,J)
matinv.lib2b.s	64687(3.4)	-	4 (D,E,J,P)
matinv.lib2c.s	63847(3.4)	7419	5 (D,E,J,P,Q)

Table 7-5 Optimized hand-coded versions of matinv.lib.s routine

performance must be to improve the speed of the datapath itself.

In the DSP32C, there is no pipelining inside the functional blocks. What if the datapath was pipelined at the clock cycle level instead of at the instruction cycle level? That is, if the adder had 4 pipeline stages and the multiplier 5 stages, both would be able to accept new operands every clock cycle. As noted on page 171, the forward elimination process has a pipelining margin of at least 9 (which happens to equal 4+5) except in the very last iteration of the main loop. Assuming that we can make full use of this pipelining margin, it would enable an increase in performance by a factor of 4 (the datapath throughput increases by a factor of 4 because the clock is 4 times faster than the original instruction cycle).

Of course, such a scheme would in turn increase the burden on the memory system and the data bus by a factor of 4. The original system of a time-multiplexed data bus would have to be replaced by a multiport/multibus memory system that works at the same speed as the clock and is able to support 3 reads and 1 write every clock cycle. As an alternative, one could change the Von-Neumann architecture of the DSP32C (common data and instruction storage) to a Harvard architecture (separate data and instruction storage) and use a 2-read/1-write *data* memory system.

The total effect of these changes would be about an order of magnitude performance improvement for the total system.

Crout or Doolittle on the DSP32C

As mentioned in Chapter 6, the distinguishing feature of the Crout (or Doolittle) algorithm is that it relies on MAC (multiply-accumulate) operations in the inner loop. These can be executed at the rate of 5 states per iteration, 1 state of which is a wait state for accessing X and Y from the same memory. This is slightly better than Gauss/LU. However, even without wait states the inefficiency of Gauss/LU is 2.96 (versus 3.4), so the expected gain is less than 15%. This is not to say that Crout does not make sense on the DPS32C as is (it certainly does), but if we try to pipeline the datapath as suggested for Gauss/LU in the previous section, there will be no gain available due to

the lack of a pipelining margin in the Crout algorithm.

7.1.3 The Motorola MC96002 digital signal processor

A simplified but realistic block diagram of the Motorola MC96002 (MC96k hereafter) is shown in Figure 7-2 [mot89]. The MC96k architecture is a quite different from the DSP32C. Most noticeable is the large number of independent data and address bussesthat connect the internal and external RAM banks to the main datapaths. The internal RAM consists of 3 independently operated banks known as P, X and Y. The P(rogram) bank is dedicated to storing instructions, and the X,Y banks are dedicated to storing data. This is different from the DSP32C, where data and program can be mixed freely in the RAM banks. There are *two* external memory busses (portA, portB) which can connect to physically separate external RAM banks or other MC96k chips. The instruction set of the MC96k is register oriented, but with parallel data moves.

An instruction cycle on the MC96k consists of 2 clock cycles. Table 7-7 shows how the various RAM banks and buses are allocated during instruction cycles. The most striking feature of the MC96k architecture is that 50% of all the memory cycles for blocks P,X,Y are *statically* allocated for DMA access. This means that even though P,X,Y are capable of 2 accesses per instruction, only 1 such access is available to the executing program.There is of course a good reason for this scheme. First of all, the processor is intended for DSP *multiprocessing* applications, and the DMA channels are intended for transferring data between multiple processors and/or between each processor and a shared memory. The static allocation scheme makes the DMA control simpler, and also ensures predictable performance during interprocess communication. The actual DMA instructions are programmed by storing control codes, starting addresses and transfer counts into special DMA registers (which are in fact mapped into the X memory address space).

The datapath (called Data Unit in Figure 7-2) is also quite different from the one found in DSP32C. The programming model of the MC96k is that all datapath operations are register-to-register operations, but with the possibility of specifying 0-2 *parallel moves* which transfer values

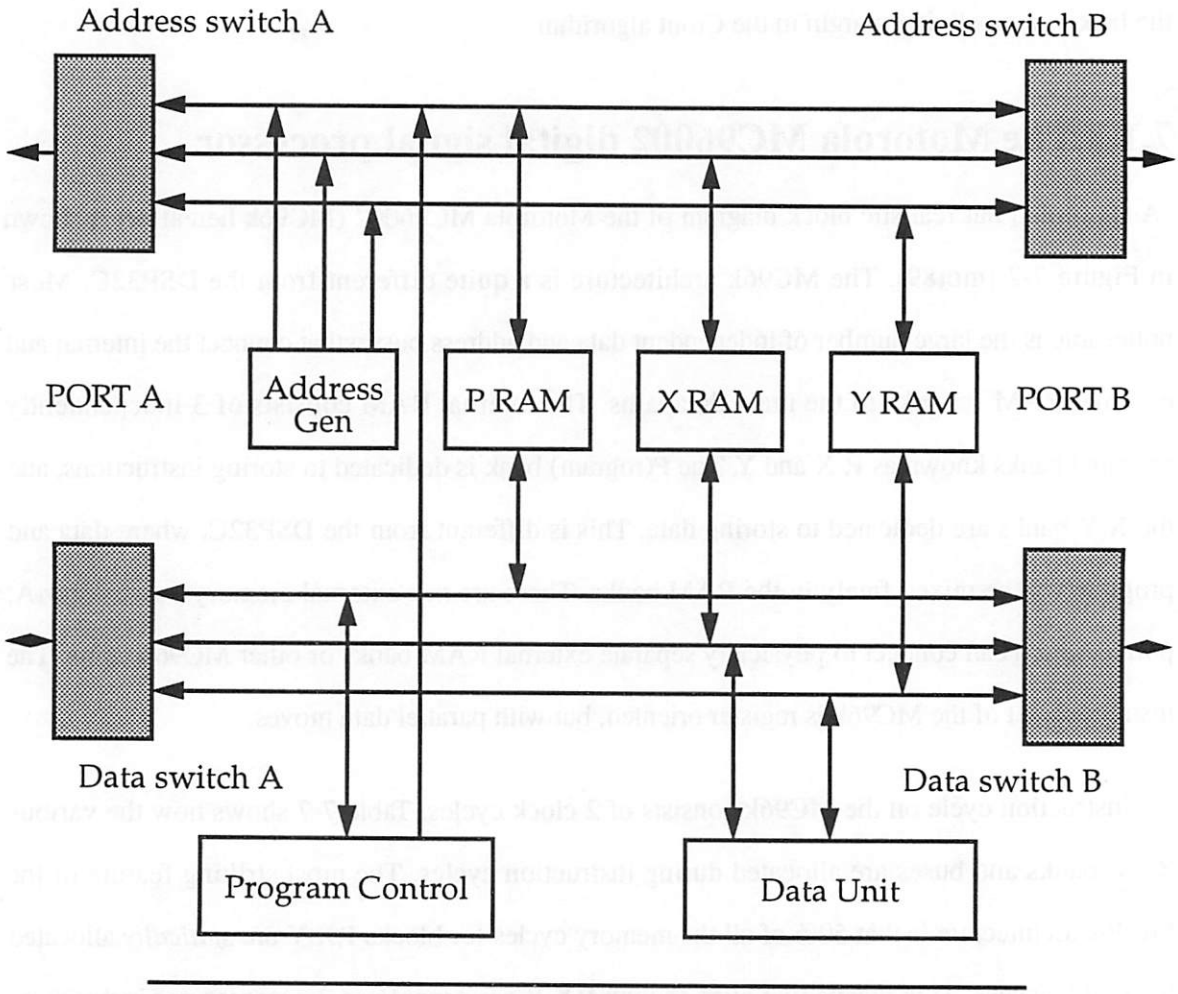


Figure 7-2 Simplified block diagram of the MC96002 chip. The data and address switches are the main connections between the chip and the outside world

Icycle	0	1	2	3	4
Fetch	F0	F1	F2	F3	F4
Decode	-	D0	D1	D2	D3
Execute	-	-	E0	E1	E2

Table 7-6 Instruction pipeline of the MC96k

Clock cycle	0	1	2	3	4	5
Clock time	25ns	25ns	25ns	25ns	25ns	25ns
Clock phase	t0, t1	t2, t3	t0, t1	t2, t3	t0,t1	t2,t3
Instruction	0		1		2	
X RAM	read/write	dma	read/write	dma	read/write	dma
Y RAM	read/write	dma	read/write	dma	read/write	dma
P RAM	read/write	dma	read/write	dma	read/write	dma
PortA and B	read or write		read or write		read or write	

Table 7-7 Memory and bus allocation during MC96k instruction cycles. Assumes f_C=40Mhz (the fastest part available)

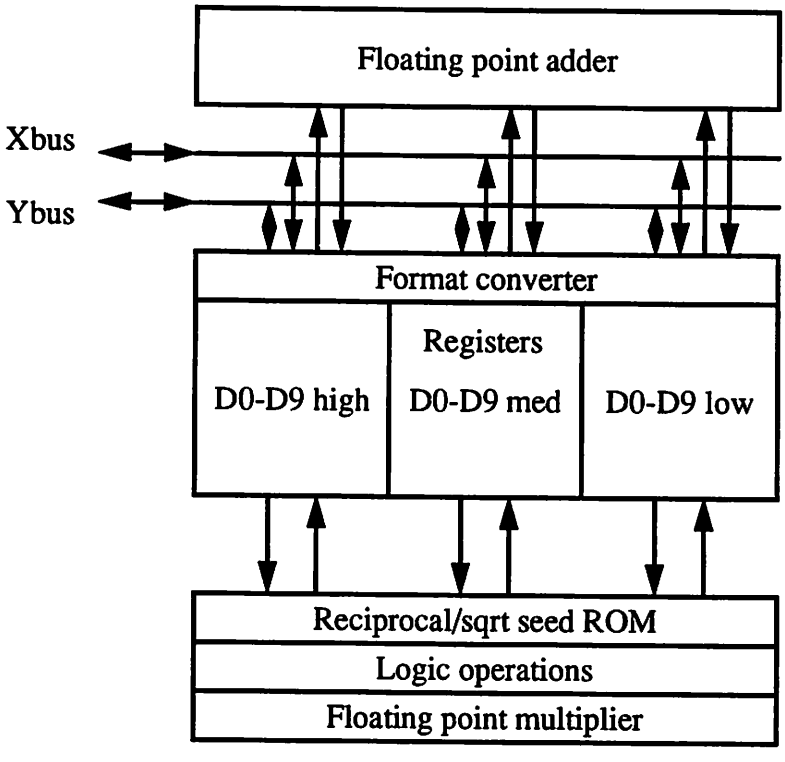


Figure 7-3 The MC96k datapath. The register file is 3x32=96bits wide and accommodates IEEE extended precision operands, but the unit itself computes only single-extended (44 bit) results

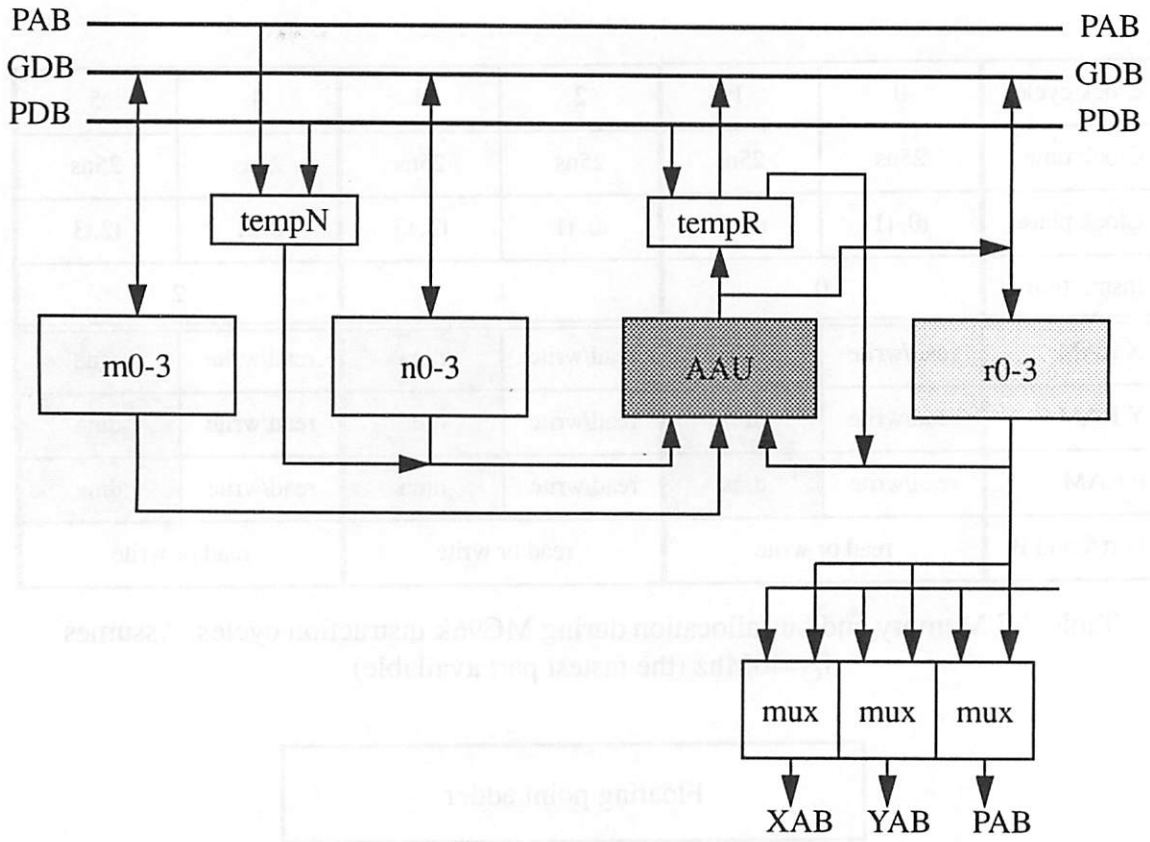


Figure 7-4 The Address Generation Unit (AGU) of the MC96k. Only 1/2 of the unit is shown. Other independent half contains the registers m4-7, n4-7, r4-7

between the registers and the RAM banks (in either direction). The main datapath (Figure 7-3) contains 10 registers named D0-D9. The registers are 96 bits wide and always store the operands using the 96-bit IEEE double-extended precision format. This is mostly for the convenience of communicating with the outside world, as all the arithmetic instructions of the MC96k are carried out in 44-bit IEEE single-extended precision. Higher precision IEEE standard arithmetic can be performed in software at the cost of lower speed.

The datapath contains an adder/subtracter unit and a multiplier unit for floating point operations. Both units have a latency of 1 instruction cycle (there is no datapath pipelining) and can operate in parallel. The results of one unit can be used as operands of either unit in the next instruction cycle.

7.1.4 Solving linear equations on the MC96k

As mentioned earlier, the X and Y banks are allowed 1 access each per instruction. This means that even with its multitude of busses and address generators, MC96k is still not capable of executing a 2-read, 1-write instruction that can efficiently perform the inner loop of the Gauss/LU algorithm. In fact, the MC96k is even less capable than the DSP32C in this respect, because it can only perform 2 data accesses (from different banks!) per instruction, whereas DSP32C can perform 3 (but with penalty when X,Y,Z come from the same RAM bank).

Because the MC96k is a register-register machine with parallel moves (the DSP32C is a memory-memory machine), there are more alternative code sequences for computing the Gauss/LU inner loop than on the DSP32C. Figure 7-5 and Figure 7-6 show 4 different assembly code fragments for the Gauss/LU inner loop. As before, the basic elimination operation is a[]= a[] - m*b[].

Case 1 is the "natural" formulation with a[] and b[] residing in the same memory bank. While the assembly code is somewhat complicated, the conclusion is easy: Because the X bank has to be accessed 3 times per iteration, and the architecture allows only 1 access per instruction, the inner loop is 3 instruction cycles long, which is not very good. The datapath is capable of doing 6 flops in this period, but only 2 useful operations are performed, meaning the efficiency is 33%.

Case 2 was designed to alleviate the memory bottleneck by assuming a[] is in the X bank and b[] is in the Y bank (the cost of copying b[] from X to Y will most likely negate any savings, but we would like to check anyway). Surprisingly enough, the loop still takes 3 instruction cycles. This time, the problem is that MC96k only has *2-operand* subtraction (and addition). "2-operand" means that the result of a subtraction must be written back on top of one of the operands, as in dest=dest–src. This causes a problem on line 9 of the code, where we would have liked to say something like d2.s=d0–d1 instead of being forced to say d0.s=d0–d1. If this were possible, the code could be arranged as shown in Case 4, with 2 instructions in the inner loop.

Case 3 unrolls the loop by a factor of 2 to avoid the problem caused by the 2-operand subtract

Common definitions for Case 1-3

```
#define a[k]     (r0)                                          1
#define a[k++]   (r0)+                                         2
#define b[k]     (r4)                                          3
#define b[k++]   (r4)+                                         4
#define m        (r1)                                          5
```

Case 1 : Both a[] and b[] in the x:ram

```
;label  ;falu operation        ;x-move          ;y-move       ;cycles  1
init    move                   r0= #aaddr                     1        2
        move                   r4= #baddr                     1        3
        move                   r1= #maddr                     1        4
        move                   d4.s= x:m                      1        5
        move                   d5.s= x:b[k++]                 1        6
loop    do #n, endloop                                        3        7
        d1.s = d4*d5           d0.s= x:a[k]                   N        8
        d0.s = d0-d1           d5.s= x:b[k++]                 N        9
        move                   x:a[k++]= d0.s                 N       10
endloop                                                       ;Total   11
                                                              3N+8     12
```

Case 2 : a[] in x:ram and b[] in y:ram.
No improvement due to lack of 3-operand subtraction

```
;label  ;falu operation        ;x-move          ;y-move       ;cycles  1
init    move                   r0= #aaddr                     1        2
        move                   r4= #baddr                     1        3
        move                   r1= #maddr                     1        4
        move                   d4.s= x:m        d5.s= y:b[k++]1        5
loop    do #n, endloop                                        3        6
        d1.s = d4*d5           d0.s= x:a[k]                   N        7
        d0.s = d0-d1                            d5.s= y:b[k++]N        8
        move                   x:a[k++]= d0.s                 N        9
endloop                                                       ;Total   10
                                                              3N+7     11
```

Figure 7-5 Assembly code for Gauss/LU inner loop a[]=a[]-m*b[] on the Motorola MC96k processor. Some liberties have been taken with the assembly language syntax to make the code more readable: (1) The `#define` statements define textual substitutions that allow us to use the mnemonic names such as a[k] in the program text instead of the actual register names such as (r0). (2) Arithmetic operations have been written in the natural form (`d1.s=d4*d5`) instead of the standard syntax (`fmpy.s d4,d4,d1`). (3) The parallel moves have been written as assignments (`x:a[k++]= d0.s`) instead of the *src,dest* syntax (`d0.s, x:a[k++]`)

Case 3 : a[] in x:ram and b[] in y:ram
Workaround that involves unrolling the loop by a factor of two

```
;label  ;falu operation          ;x-move          ;y-move          ;cycles 1
init    move                     r0= #aaddr                        1       2
        move                     r4= #baddr                        1       3
        move                     r1= #maddr                        1       4
        move                     d4.s= x:m        d5.s= y:b[k++]   1       5
        d1.s= d4*d5              d0.s= x:a[k]                      1       6
loop    do #n/2, endloop                                           3       7
        d0.s = d0-d1             d2.s= x:a[k+1]   d5.s= y:b[k++]   N/2     8
        d1.s = d4*d5             x:a[k++]= d0.s                    N/2     9
        d2.s = d2-d1             d0.s= x:a[k+1]   d5.s= y:b[k++]   N/2    10
        d1.s = d4*d5             x:a[k++]= d2.s                    N/2    11
endloop                                                            ;Total  12
                                                                   2N+8    13
```

Case 4 : a[] in x:ram and b[] in y:ram
If MC96k actually had a 3-operand subtraction

Definitions

```
#define a[k]     (r0)                                                  1
#define a[k+1]   (r0+n0)                                               2
#define a[k++]   (r0)+                                                 3
#define b[k++]   (r5)+                                                 4
#define m        (r1)                                                  5
                                                                       6
;label  ;falu operation          ;x-move          ;y-move          ;cycles 7
init    move                     r0= #aaddr                        1       8
        move                     r4= #baddr                        1       9
        move                     r1= #maddr                        1      10
        move                     n0= 1                             1      11
        move                     d4.s= x:m        d5.s= y:b[k++]   1      12
        d1.s= d4*d5              d0.s= x:a[k]                      1      13
loop    do #n, endloop                                             3      14
        d2.s= d0-d1              d0.s= x:a[k+1]   d5.s= y:b[k++]   N      15
        d1.s= d4*d5              x:a[k++]= d2                      N      16
endloop                                                            ;Total 17
                                                                   2N+9    18
```

Figure 7-6 Assembly code for Gauss/LU inner loop on Motorola MC96k (continued)

limitation. In this version, d0 and d2 are used as *alternating* destinations for the result of the subtraction. The inside of the loop is now 4 instructions long, but is executed only N/2 times. This means that in effect we have 2 instructions per iteration. The key instructions in Case 3 are the ones numbered 8 and 10. These lines contain the code where d0 (d2) is computed at the same time as d2 (d0) is filled with a new a[]-value from the X memory. Case 4 shows the more elegant code which would result if the MC96k had a 3-operand subtraction.

The above programming exercise allows us to draw some conclusions about the basic efficiency of the MC96k with respect to the Gauss/LU algorithm. If b[] is not copied to the Y bank, the efficiency is 1/3 in the datapath. If b[] could be copied to Y at no cost (which is not actually possible, even using DMA), the efficiency could be made 1/2 by unrolling the loop by a factor of 2.

Crout or Doolittle on the MC96k

Crout and Doolittle require 2 accesses to the same bank, which means 2 instructions in the inner loop and a 50% datapath utilization. Another possibility is to copy pieces of the matrix to the other bank, which will allow the use of a 1-instruction multiply-accumulate (MAC) loop. This is the same basic loop that is used in an FIR filter, which is one of the traditional uses of a chip such as the MC96k. Figure 7-7 shows an assembly code fragment for the inner loop under the assumption that one of the vectors has been copied to bank Y. The copying overhead will to some extent negate the savings, even though one can get away with copying only on piece of data and use it several times before having to copy again.

Potential speedup

The MC96k chip is less well-suited for Gauss/LU than the DSP32C. For Crout, they are about the same, as the DSP32C needs wait states whereas the MC96k needs copying between memory banks. I have no complete, handwritten and optimized code for the MC96k corresponding to the `matinv.lib.s` code for the DSP32C, but it can be expected that they will fare about the same,

Case 5 : Crout algorithm inner loop (dot product) s=sum(i=1:n, a[i]*b[i]),

assuming that a[] is in bank X and b[] has been copied to bank Y

```
#define a[k]     (r4)                                                    1
#define a[k++]   (r4)+                                                   2
#define b[k]     (r0)                                                    3
#define b[k++]   (r0)+                                                   4
#define B[k]     (r5)                                                    5
#define B[k++]   (r5)+                                                   6
                                                                         7
;Label   ;falu operation          ;x-move          ;y-move          ;cycles 8
init     move                     r0= #aaddr                        1       9
         move                     r4= #baddr                        1      10
         move                     r5= #btmp                         1      11
         move                     d0.s=b[k++]                       1      12
loop     do #N, endcopy                                             1      13
         move                     d0.s=x:b[k++]    y:B[k++]=d0.s    N      14
endcopy                                                                  15
         move                     d4.s=x:a[k++]    d5.s=y:B[k++]    1      16
loop     do #N, endloop                                             1      17
         d1.s=d4*d5, d2.s=d2+d1   d4.s=x:a[k++]    d5.s=y:B[k++]    N      18
endloop                                                           ;Total 19
                                                                  2N+7   20
```

Figure 7-7 Assembly code for Crout algorithm inner loop on the MC96k

meaning that the MC96k will have a utilization rate of about 1/3. Again, another factor of 4 can be gained by pipelining the datapath. As before, this also requires a memory system update to accommodate the increased bandwidth requirement.

7.1.5 Texas Instruments TMS320C30 digital signal processor

The TMS320C30 (C30 for short) is somewhat similar to the MC96k in that it has numerous internal buses and two external bus interfaces. Figure 7-8 shows a simplified block diagram of the chip [ti88]. The C30 is somewhat more complex than the MC96k in that almost all memories (the exception is the cache) are connected to all possible buses. Surprisingly, there is only one bus connection between the memory system and the ALU. This bus is time-multiplexed so that two memory accesses can take place in one instruction cycle. The instruction pipeline timing is shown in Table 7-8. Each instruction spans 4 instruction cycles (Fetch, Decode, Read, Execute). The

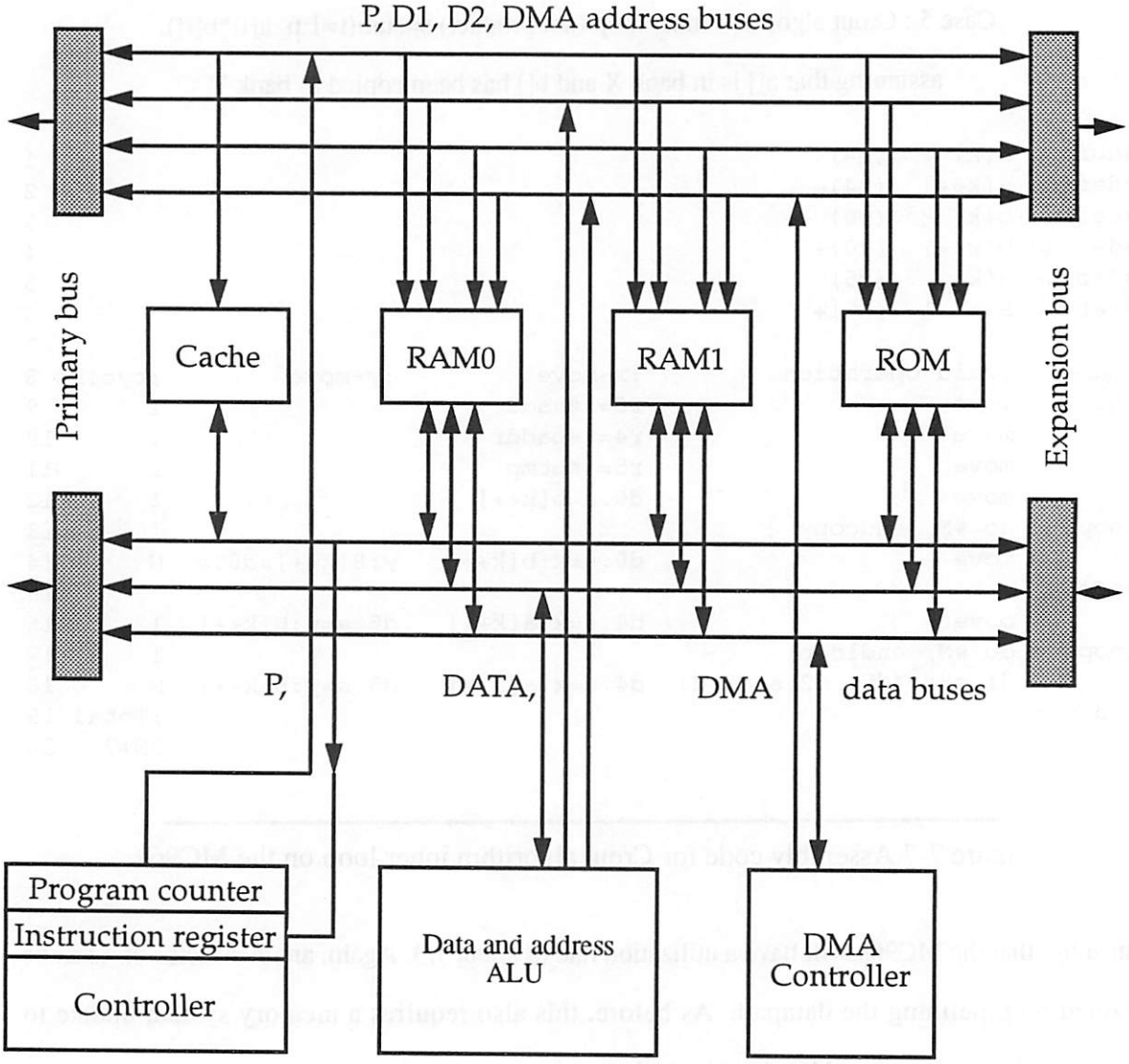


Figure 7-8 Block diagram of the TMS320C30 chip

Icycle	0	1	2	3	4	5	6
Fetch	F0	F1	F2	F3	F4	F5	F6
Decode	-	D0	D1	D2	D3	D4	D5
Read	-	-	R0	R1	R2	R3	R4
Execute	-	-	-	E0	E1	E2	E3

Table 7-8 Instruction pipeline of the TMS320C30

Clock cycle	0	1	2	3	4	5
Clock time	30ns	30ns	30ns	30ns	30ns	30ns
Instruction	0		1		2	
RAM0-1	read/write	read/write	read/write	read/write	read/write	read/write
ROM, Cache	read/write	read/write	read/write	read/write	read/write	read/write
Primary bus	read	read	write			read
Expand bus	read or write		read or write		read or write	

Table 7-9 Memory and bus allocation during C30 instruction cycles. Assumes f_C=33.4MHz. The latency of the Primary Bus is variable, depending on the order of operations (read, write) and the speed of the external RAM. Example shown is for 0-wait RAM. The expansion bus always takes 2 clock cycles per read or write

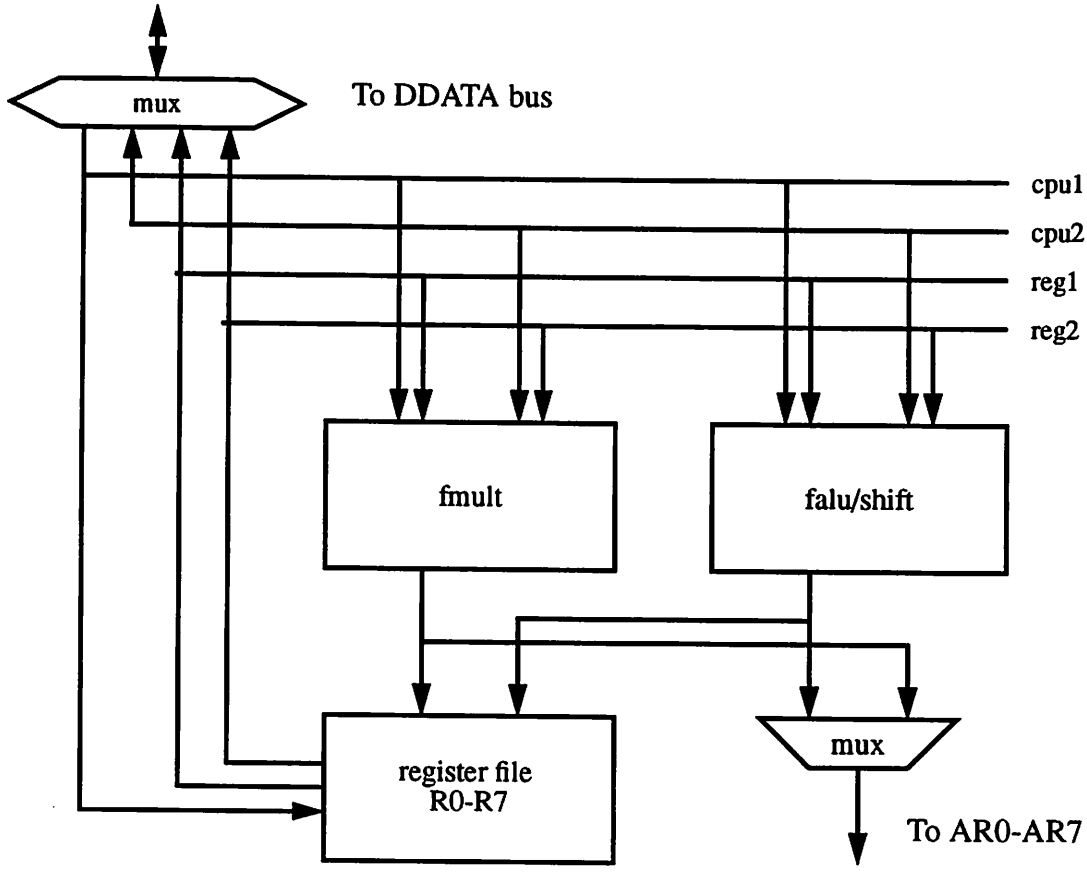


Figure 7-9 Main datapath of the TMS320C30

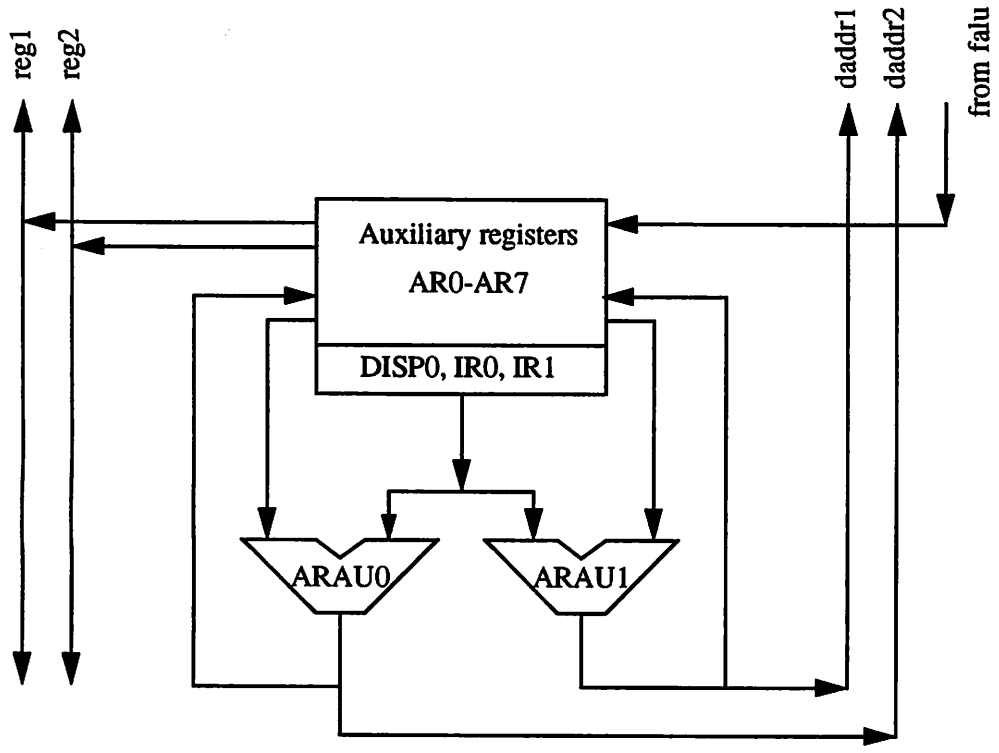


Figure 7-10 TMS320C30 auxiliary register file and address arithmetic unit

memory and bus allocation during an instruction cycle is shown in Table 7-7. The instruction cycle consist of two clock cycles. Figure 7-9 shows the main datapath of the C30, and Figure 7-10 shows the auxiliary registers (AR0-AR7) and the address arithmetic unit. The AR registers are mainly used for address computations.

The main datapath is fairly standard, with a floating point multiplier and a floating point adder that can operate in parallel. The programming model is a mix of the register-register and the memory-memory model: Up to two of the operands can be *read* from memory whereas the remaining ones must be registers. If there are no reads, *one* store can be performed per instruction. For parallel multiply/add operations, the destinations must be registers. The address unit has two independent adders which can be used to increment, offset or displace a base address coming from a register AR0-AR7. This corresponds to the ability of having two memory accesses per cycle.

The main difference between the C30 and the DSP32C/MC96k is that C30 is the only processor

Case 1 : Gauss/LU inner loop a[]= a[]-m*b[]

a[] and b[] in same RAM

```
                                                                    1
;Label  ;Operations and moves    ; Comment                      ;Cycles2
        ldi     @a,ar0           ; points to a[]                1      3
        ldi     @b,ar1           ; points to b[]                1      4
        ldi     @m,ar2                                          1      5
        ldf     *ar2,r0          ; m->r0                        1      6
        ldi     N,rc             ; N->rc (number of elements)   1      7
        ;                                                               8
        mpyf3   r0,*ar1++,r1     ; m*b[]->r0                    1      9
        subi    1,rc             ; rc=N-1                       1     10
        ;                                                              11
loop    rptb    elim             ; setup the repeat block       4     12
        mpyf3   r1,*ar1++,r0     ; m*b[k+1]->r0                 N     13
||      subf3   *ar0,r1,r2       ; a[k]-m*b[k]->r2              -     14
elim:   stf     r2,*ar0++       ; a[k]-m*b[k]->a[k]           N      15
                                                              ;Total16
                                                              2N+11 17
```

Case 2 : Crout inner loop s=sum(a[],b[],i=1:n)

a[] and b[] in same RAM

```
#define a[k++]  *ar0++                                                 1
#define b[k++]  *ar1++(ir0)                                            2
                                                                       3
;Label  ;Operations and moves    ; Comment                      ;Cycles4
        ldi     @a,ar0           ; points to a[]                1      5
        ldi     @b,ar1           ; points to b[]                1      6
        ldi     N,rc             ; N->rc (number of elements)   1      7
        ldi     N,ir0            ; N->rc (number of elements)   1      8
        ;                                                              9
        subi    1,rc             ; rc=N-1                       1     10
        mpyf3   a[k++],b[k++],r0; r0= a[]*b[]                   1     11
eloop   rptb    elim             ; set up the repeat block      4     12
        mpyf3   a[k++],b[k++],r0; r0= a[]*b[]                   N     13
||      addf3   r2,r2,r0         ; r2= r2+a[]*b[]               -     14
elim:                                                                  15
       stf     r2,*ar0++        ; a[k]-m*b[k]->a[k]           1       16
                                                              ;Total17
                                                              N+11  18
```

Figure 7-11 Assembly code for Gauss/LU and Crout on the TMS320C30

capable of 2 reads from the same RAM bank in one clock cycle.

7.1.6 Solving linear equations on the C30

The C30 is not ideally suited for the Gauss/LU algorithm, since it can perform at most 2 memory accesses per instruction. Figure 7-11 contains assembly code for the inner loop of Gauss/LU. The need to insert an *stf* (store float) instruction in line 15 is the reason why there will be 2 cycles per iteration.

The C30 fares much better with the Crout algorithm than with Gauss/LU. As mentioned earlier, C30 is the only processor (so far) which can read two operands out of the same memory without any speed penalty. This makes C30 the best candidate for the Crout algorithm: The inner loop only needs to contain one instruction (see Figure 7-11). There is 100% datapath utilization when running the Crout algorithm on the C30.

Speedup potential

As it stands, the C30 is a very good processor for the Crout algorithm. The overhead in looping, addressing, pivoting and row swapping has not been accurately estimated, but it can be expected that for the complete algorithm one can get about 50% datapath utilization (a factor of 2 overhead).

As is the case for the other processors, pipelining the datapath will not help unless the memory system is upgraded so that one can use a Gauss/LU algorithm with sufficient pipelining margin.

7.2 Vector processors

Vector Processors (VP) are computer architectures that are especially designed to process arrays (vectors) of floating-point numbers at high speed by using heavily pipelined functional units that rely on the programmer and/or the program compiler to exploit the pipelining margins that can be found in scientific numerical computation algorithms.

Supercomputers

The most well-known vector processors belong to the class of "Supercomputers", meaning the highest speed, multi-million dollar machines such as the Cray-1, Cray-2, Cray X-MP, Cray Y-MP, Fujitsu VP100/200, Hitachi S810/820, IBM3090/VF, NEC SX/2 and the Convex C-1 [hepa90]. There are also smaller, board level array processors such as the ones formerly made by Floating Point Systems corporation.

Common to all these systems is that they are aimed at medium-to-large scale scientific computation problems, and not small real-time problems such as repeatedly solving 16x16 systems of linear equations. Hennessy and Patterson [hepa90] make it clear in their exposition that due to the heavy startup penalty of the vector operations on a commercial supercomputer, they are not very efficient for small systems. In fact, of the machines mentioned, all but one have vector registers of size 64 or larger, implying that smaller vector lengths are not efficient.

Supercomputers are impractical in embedded applications due to their prohibitive size and cost. For the purpose of this dissertation, the main utility of Supercomputers is that they use some implementation techniques and architectural techniques that can also be applied on a smaller scale. This fact will be explored further in Chapter 8.

Vector processing chips

The NEC corporation Vector Pipelined Processor (VPP) is a single-chip implementation of some of the central functions of a supercomputer [okamoto91]. Figure 7-12 shows the main datapath of the VPP. VPP has 2 functional units (adder and multiplier), both containing 5 pipeline stages. The processor contains 8 separate 2-port SRAM blocks which are used as vector registers to feed the functional units, store results and as buffers between the vector registers and main memory (offchip). VPP is made in BiCMOS 0.8 μm technology and runs at f=100MHz, which means that the peak throughput is 200 Mflops.

The paper [okamoto91] shows clearly that the chip is not intended for stand-alone use. It appears

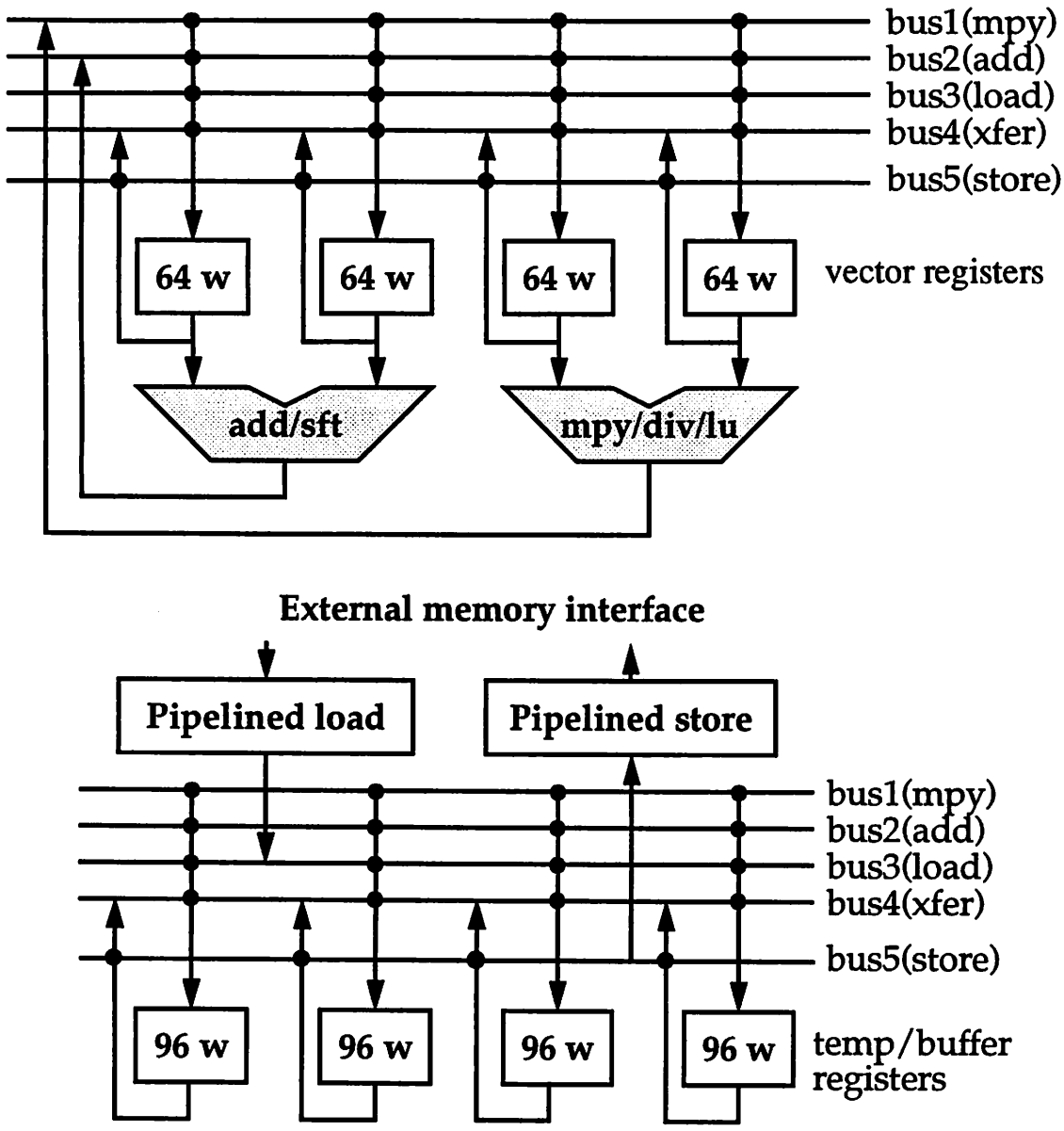


Figure 7-12 The central parts of the NEC Vector Pipelined Processor (VPP). The various vector registers have either 64 word or 96 word capacities

that the main function of the chip is to implement some of the most central and time-critical operations of a Supercomputer on one chip, which is vastly more efficient than traditional approaches such as using ECL (Emitter Coupled Logic) gate arrays. ECL implementations typically require many chips and fluid cooling, making the system larger and more expensive. The

speed of VPP is not high enough to rival that of ECL-based Supercomputers, though.

As for applicability to small-scale linear systems, the VPP uses 64-word vectors, which is too long for our purposes, and requires a sophisticated interleaved-bank external memory system. It appears that the VPP can be a very good solution for medium-scale computational problems where a relatively inexpensive and compact solution is required.

Massively Parallel Architectures

Massively Parallel (MP) architectures such as the Connection Machine CM-5 [hwang92] are similar to the Vector Processors (Supercomputers) mentioned above in that they are large and expensive systems intended for large-scale scientific problems. MP architectures involve many (from 16 to 65636 or more) processors that communicate over a special interconnection networks. (the hypercube type network is one popular example). The most pronounced problem in MP architectures is exactly the communication between the processing nodes, and how to exchange data efficiently. As a result, the hardware utilization is lower and the cost of communication circuits is much higher in MP architectures than in VP architectures [spectrum92].

7.3 Systolic Arrays

There is a multitude of architectures and design techniques that can be grouped under the general heading Systolic Arrays (SAs for short). The term was originally coined [htkung78] to describe a 1- 2- or n-dimensional structure of (relatively) simple processing elements with only nearest-neighbor communication and one central memory which is connected to the array only at (some of) the boundaries of the array.

Example of a 1-d systolic array (the WARP processor [anna86]) is shown in Figure 7-13. It is commonly agreed upon (but not universally so) that Systolic Arrays belong to the class of SIMD (Single Instruction, Multiple Data) architectures, meaning that all the processing elements perform the same task at the same time, but on different pieces of data.

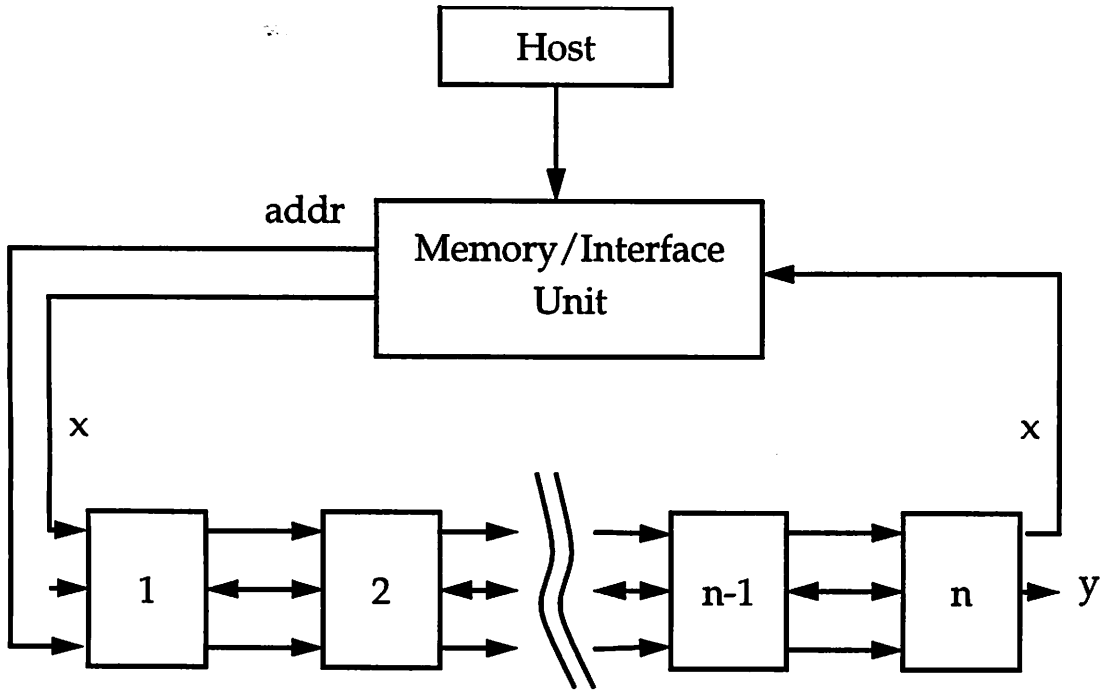


Figure 7-13 Simplified block diagram of the WARP systolic processor

A strict definition [roy88] of a Systolic Array demands regularity (mostly identical processors), spatial locality (only local interconnections), temporal locality (all combinational elements are latched, no zero delay operations) and pipelined operation (throughput independent of the order/ size of the system). Many of the early systolic architectures were special purpose and could only perform one task efficiently. After the initial plethora of architectures and implementations, researchers discovered that one of the most difficult problems with systolic architectures is to program them to execute a variety of algorithms in an efficient manner. In fact, [roy89] has shown that based on the strict definition, it can be formally proven that matrix algorithms that use pivoting, including Gauss/LU, are *not* systolic. That is, they *cannot* be executed on such an architecture without breaking one of the requirements of the definition.

These experiences and the observed limitations have led to developments in the field of Regular Iterative Arrays (RIA) [jag85][jag87][rao85][rao88], Regular Processor Arrays [roy88][roy89] and Mesh Array Computational Graphs [le92][moreno90], which are less restricted relatives of

systolic architectures. For example, it is allowed to insert FIFO and LIFO buffers in between the processors. Most of the reported work has concentrated on synthesizing an architecture for a given algorithm and/or mapping an algorithm onto a given array. Some interesting results have been achieved, but all the methods appear to have in common that they do not (so far) address what is the main concern in this work, which is efficient resource usage and high datapath utilization. For example, [roy89] shows that Gauss/LU can be performed in $O(N^2)$ time with N processors, but no efficiency measures are provided. [le92] demonstrates a technique that can map the Gauss/LU algorithm (without pivoting) into the Mesh Array Computational Graph format, which can then be mapped onto a regular array. Again, the datapath efficiency is not known.

Regular Array architectures are in-between Vector Processors and Massively Parallel architectures in complexity. Programming a Regular Array is a difficult task even if it is not critical to have the highest possible resource utilization. SA architectures also have in common with Vector Processors and Massively Parallel architectures that the implementations are typically too large and too expensive for embedded applications. SA architectures are usually implemented as board-level or wafer-scale designs, meaning that they are large and costly. For small linear systems it appears that a single-chip implementation is necessary for economical reasons.

Architecture	Implementations	Architectures	Implementations
SUN Sparc	CY700, Viking, Tsunami	HP PA-RISC	PA 7100 series
Motorola 88k	MC88100	IBM POWER	RS6000/500 series
MIPS R-series	2000, 3000, 4000, 6000	Intel RISC	i860, i960
DEC Alpha	21064		

Table 7-10 Some commercial RISC families and chips

7.4 Standard microprocessors

Standard microprocessors such as the families and implementations listed in Table 7-10 are commonly used to execute numerical algorithms in a Unix workstation environment. For example, CAD programs such as SPICE and CAZM [erdman90] involve large (and sparse) linear systems which are solved by C subroutines that are compiled and run on one of the RISC (Reduced Instruction Set Computer) chips listed in the table.

Most users never know how efficiently their workstations solve this (or any other) problem, and are satisfied with the speedup that comes along with the periodical technological advances. This makes perfect sense since a workstation is a general purpose computing environment and number-crunching typically makes up a small part of the overall workload.

It is therefore not very surprising to find that many of the RISC chips are not particularly efficient for solving systems of linear equations, be they the large and sparse variety or smaller dense systems of the sort we are interested in.

7.4.1 The SPARC family

The most advanced member of Sparc architecture family to date is the Viking (a.k.a. SuperSparc) chip [sparc92]. It is called a "superscalar" implementation of the Sparc architecture because it can issue and execute up to 3 different instructions every clock cycle. (Recall that many DSP and RISC chips have a pipelined execution with perhaps as many as 4-5 instructions being in different stages of execution at the same time. However, only *one* new instruction is issued (started up) each clock cycle. In a superscalar machine there are multiple instruction pipelines as well.)

The Viking chip has certain restrictions that apply to any group of 3 instructions that are candidates for being issued in parallel:

- Maximum 2 integer results
- Maximum of one data memory reference

- Maximum of one floating point arithmetic instruction
- A group cannot include a control transfer (branch)

These restriction tells us right away that the Viking cannot be very efficient at executing either the Gauss/LU or the Crout algorithm, as the memory bandwidth simply is not available. For the Gauss/LU algorithm, there will be at least 3 instruction cycles, corresponding to the three necessary memory accesses. For the Crout algorithm there will be at least 2 instruction cycles. The restriction of one floating-point operation per cycle also means that even without the memory access restriction, one could still not get below 2 instructions per iteration.

One important feature of the Sparc family (and other RISC chips) is that they do not have any *explicitly* parallel instructions, such as the **fmult||fsub** instruction found in the MC96k. One reason behind this fact is that the RISC families are designed to be scalable architectures that can have both low-end and high-end implementations. In the low end, it is necessary that the architecture does not specify parallel instructions that may be costly to implement, and in the high end, there may be so much parallelism that it is unreasonable to try to specify all of it explicitly in instructions. Instead, the RISC families use the above mentioned SuperScalar approach, where extra hardware is expended to search the instruction stream of groups of instructions that can be issued in parallel. This search becomes exponentially expensive with the number of parallel instructions. In fact some of the new RISC chips could humorously be called "Complex Instruction *Sequencing* Computers", as a pun on the old "Complex Instruction Set Computer" acronym (CISC).

7.4.2 The Motorola 88k family

The MC88100 (MC88k for short) is a heavily pipelined load/store register-register machine with a Harvard architecture (separate external program and data busses), and a built-in floating point unit [mot90]. One instruction is issued every clock cycle (machine cycle=clock cycle) and the instruction pipeline is 3 cycles long. Offchip memory access latency is 3 cycles plus wait states (if

any) but the loads/stores are pipelined so that up to 3 loads/stores can be in progress at the same time with an effective access time of 1 cycle each.

Since each cycle can have only one data access, Gauss/LU takes at least 3 cycles and Crout at least 2 cycles per iteration. Because of the heavy pipeline delay (fadd=5, fmult=6 cycles), Crout will in reality take at least 11 cycles per iteration. There is no provision in the instruction set for doing parallel load/stores or multiple floating point instructions (say, multiply-accumulate) in the same cycle.

7.4.3 The MIPS R-series

The fastest member of the MIPS family is the R4000 chip [kane92], which is also the only implementation that includes the floating-point unit on-chip. R4000 is heavily pipelined with an 8-stage instruction execution unit and 1 instruction issue per clock cycle. The latencies are fadd=4 and fmult=7 cycles, which precludes efficient execution of Crout's algorithm. The Gauss/LU algorithm is hampered by the restriction of 1 load/store per instruction.

7.4.4 The DEC Alpha 21064

The 21064 is the first implementation of the Alpha architecture [dobber92][dec92]. It has the highest clock speed of any microprocessor to date (f=150MHz) and a true 64-bit architecture, as well as 2-issue superscalar operation. From the documentation that is currently available it is not possible to determine what kinds of instructions are allowed to be executed in parallel, but it is clear that since the Alpha is a register-register and load/store architecture, it cannot provide more than 2 memory accesses per instruction cycle. This implies that Gauss/LU will take at least 2 cycles per iteration, and most likely more. The pipeline delay of the floating point unit is 10 cycles, meaning that Crout will not be very efficient.

7.4.5 The Intel i860 XP

The i860 XP is the most advanced member of the Intel RISC family [intel92]. It has a load/store

register-register architecture which can execute one integer operation and one floating point operation per cycle by explicitly programming the parallel instructions. The floating point instruction set include some dual-operation instructions that execute but a multiply and an add/subtract at the same time. This means that a maximum of 3 instructions can be issued each cycle. However, there can be at most 1 load/store (counts as an integer operation even if it load is to a floating point register) per machine cycle. Hence Gauss/LU will take at least 3 cycles per iteration and Crout will take at least 2. Also, the pipeline delay of 2 for the floating point add will have a negative effect on Crout.

7.4.6 Other RISC μP families

A clear pattern can be established from the above 5 examples of current RISC architectures. All the common architectures are lacking one or more of the features required for Gauss/LU and Crout, such as

- Insufficient memory bandwidth per instruction cycle (Gauss/LU, Crout)
- Insufficient parallel load/store/fp instructions (Gauss/LU, Crout)
- Too large pipeline latencies (Crout)

The other 2 architectures listed in Table 7-10 (HP, IBM) are difficult to evaluate because the companies generally are not willing to provide architecture information beyond the instruction set level. The detailed timing of the instructions are not available. It can still be expected that they have one or several of the limitations listed above.

In general, it is clear that the RISC chips are less suited to solving linear equations than are the current crop of DSP chips. This is quite reasonable taking into account that the RISC architectures are intended for general purpose computing.

7.5 Summary

This chapter quantifies and compares the efficiency of a variety of architectures and chip/processor implementations with respect to the Gauss/LU and Crout algorithm, as applied to small-scale linear systems.

The main conclusions are that the commercial DSP chips are probably the most efficient means commercially available for solving linear equations. The DSPs are small, quite fast and relatively inexpensive, but it is also clear that their DSP-tailored architectures are not exactly what is needed for the given problem. The overhead (and corresponding speed reduction) is usually a factor of 2-4, and it appears that a completely tailored pipelined architecture can increase the speed by an order of magnitude.

Commercial RISC chips are slower than the DSP chips because their main application areas do not warrant the more specialized architectural techniques used in DSP chips. Other architectures such as Vector Processors and Massively Parallel Architectures have features that can be applied in custom architectures for matrix operations, but the commercial solutions are too large and too costly for embedded applications.

The next chapter describes how application specific architectures and circuits can be applied to create Numerical Processors that are more efficient than the commercially available solutions.

CHAPTER 8

THE *SMAC* (SMall Matrix Computer) ARCHITECTURE

All the commercial hardware platforms considered in the previous chapter have some less-than-optimal properties when applied to solving small systems of linear equations (Table 8-1). Some of the platforms are very fast but too large and costly. Others are fairly inexpensive but not as computationally efficient as they could be.

This chapter describes a new architecture called SMAC (SMall Matrix Computer) which is aimed specifically at solving small systems of linear equations in an efficient manner. SMAC was designed partly using features borrowed from known computer architectures, and partly by developing new methods specifically aimed at efficient execution of the Gauss/LU algorithm. It will be shown how the various architectural features support the Gauss/LU algorithm.

The architecture is based on a collection of building blocks (memory, control functions and datapaths), the most critical of which have been designed as CMOS circuits all the way through layout, fabrication and testing. Design details, test results and performance evaluations of the building blocks are presented.

8.1 SMAC requirements

The requirements that have been identified in the preceding chapters are the following:

- Memory architecture:

  The memory must be able to sustain 2 reads and 1 write per cycle.

- Datapath architecture and pipelining:

  The hardware should exploit the available pipelining margin (PM) as much as possible by having an appropriately pipelined MADD (multiply-add) unit.

- Pivoting:

  The pivot search must be fast, preferably in parallel with other useful other operations.

- Permutation:

  The permutation operation (address translation or row swapping) should be transparent.

- Address generation:

  SMAC must have an efficient set of address generators and a control sequencer that will make efficient use of the datapath.

The hardware should be programmable to some extent (especially matrix sizes), and it should be possible to exploit pipeline interleaving during the back substitution phase.

Processor	Cost	Size	Raw speed	Actual speed	Efficiency
DSP	+	+	0	0	0
RISC	0	0	0	–	–
Supercomputer	– –	– –	++	0	–
VPP	–	–	++	+	0
MPA	– –	– –	++	0	– –
Systolic Arrays	– –	– –	++	0	– –

Table 8-1 Relative merits of commercial hardware platforms

8.2 Datapath and memory architecture

There are 3 main tasks to be performed by the datapath and the memory:

- Forward elimination.
- Pivoting.
- Back substitution.

These tasks create different demands for the datapath and memory. The challenge is to create a datapath/memory architecture that can handle all 3 tasks without either one of them requiring a large hardware overhead that is not useful for the other 2 tasks. To develop the architecture, one possible approach is to create 3 different datapath architectures and then try to consolidate them into one efficient datapath for all 3 tasks.

Forward elimination

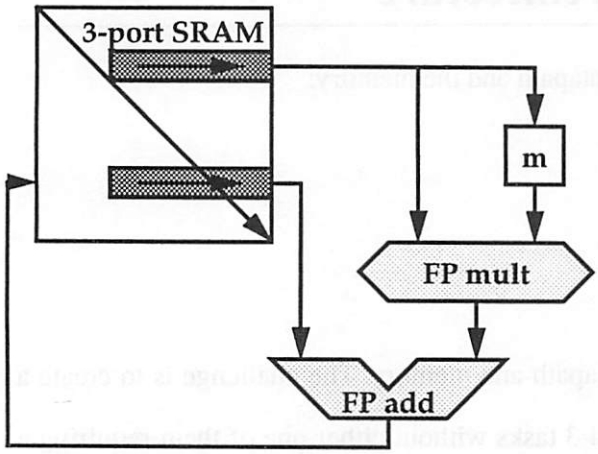
A datapath suitable for the elimination operations is shown in Figure 8-1. The main features are the 3-port memory, which provides 2 operands and stores 1 result each cycle, and a pipelined multiply-add datapath. The register **m** is used to hold the scale factor a[i][k]•(1/a[k][k]).

Back substitution

The back substitution datapath (Figure 8-2) is quite similar to the elimination datapath but requires an accumulator for the multiply-accumulate operations.

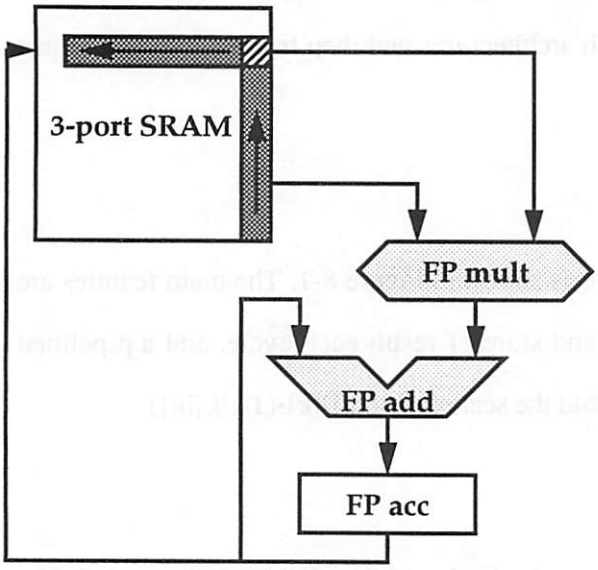
Pivot search

Pivot searching can be done in a number of ways. The datapath shown in Figure 8-3 takes a naive approach were the pivot candidates are searched serially after all of them have been computed. The accumulator is used to hold the currently largest pivot, and its value is compared to the next candidate by using the adder circuitry. The adder outputs a status signal which is used to select which if the two values to store back into the accumulator.



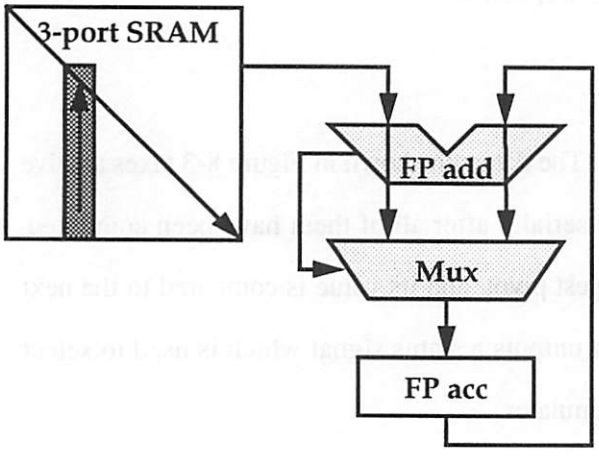

```
m= (a[i][k]*= (1/a[k][k]));   1
for (j=k+1; j<n; j++) {       2
   a[i][j]-= m*a[k][j];       3
}                             4
b[i]-= m*b[k];                5
```

Figure 8-1 Elimination datapath




```
prod= b[i];                   1
for (k=i+1; k<n; k++) {       2
   prod-= a[i][k]*b[k];       3
}                             4
b[i]= prod*(1/a[i][i]);       5
```

Figure 8-2 Back substitution datapath



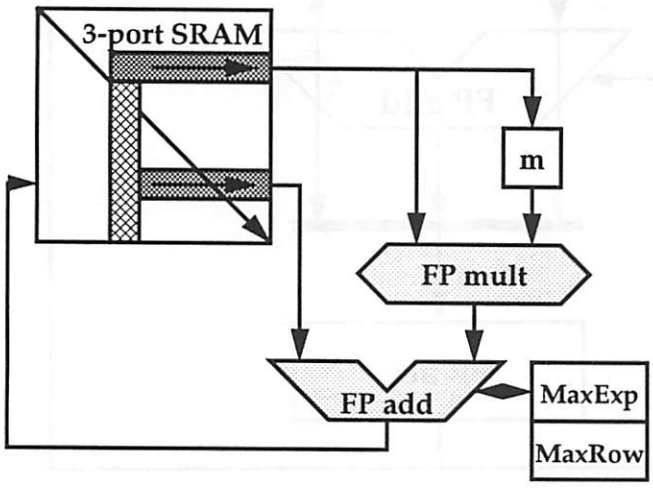

```
temp=0; imax=k;               1
for (i=k; i<n; i++) {         2
   ir= row[i]; test= a[ir][k];3
   if (test<0) test= -test;   4
   if (test>temp) {           5
      imax=i; temp= test;     6
   }                          7
}                             8
```

Figure 8-3 Pivot search datapath

Parallel pivot search

Another approach to pivot searching is to look at the values of the row leaders a[i][k+1] as they are computed (instead of waiting until all of them are finished). Parallel searching requires some extra hardware since the regular adder will be busy computing the updated values. Because of its size, it would be prohibitive and wasteful to replicate the entire adder just to use it for pivot searching, as it would be idle at all other times. One low-cost alternative solution is to compare only the *exponents* of the candidates with each other. This means that only the mantissa part of the floating point adder needs to be replicated, and that we need only store the (8-bit) exponent in-between comparisons. This is a very attractive option.

Comparing only the exponents does not guarantee finding the true maximum value among the pivot element candidates, but this is not really a problem since two floating point numbers with the same exponent can at most be a factor of two different from each other. Such a small factor will not adversely affect the numerical accuracy of the Gauss/LU algorithm. Figure 8-4 shows how the current MaxExp and MaxRow can be stored and fed into an additional exponent comparator in the adder.



- **Only search/compare among the row leaders**
- **Update MaxExp each time it is surpassed**
- **Update MaxRow accordingly**
- **Before starting next round (k++) of elimination, perform the row permutation (several methods possible)**

Figure 8-4 Parallel pivot search based on comparing just the exponent part of the candidates

Consolidated data path

The datapaths in Figure 8-1 to Figure 8-4 are reasonably similar, and consolidating them all into one datapath (Figure 8-5) is mostly an exercise in multiplexing the inputs and outputs so that the final datapath can take on either one of the required personalities by applying the appropriate set of control signals. Figure 8-6 shows how the datapath is used for elimination, parallel pivot searching

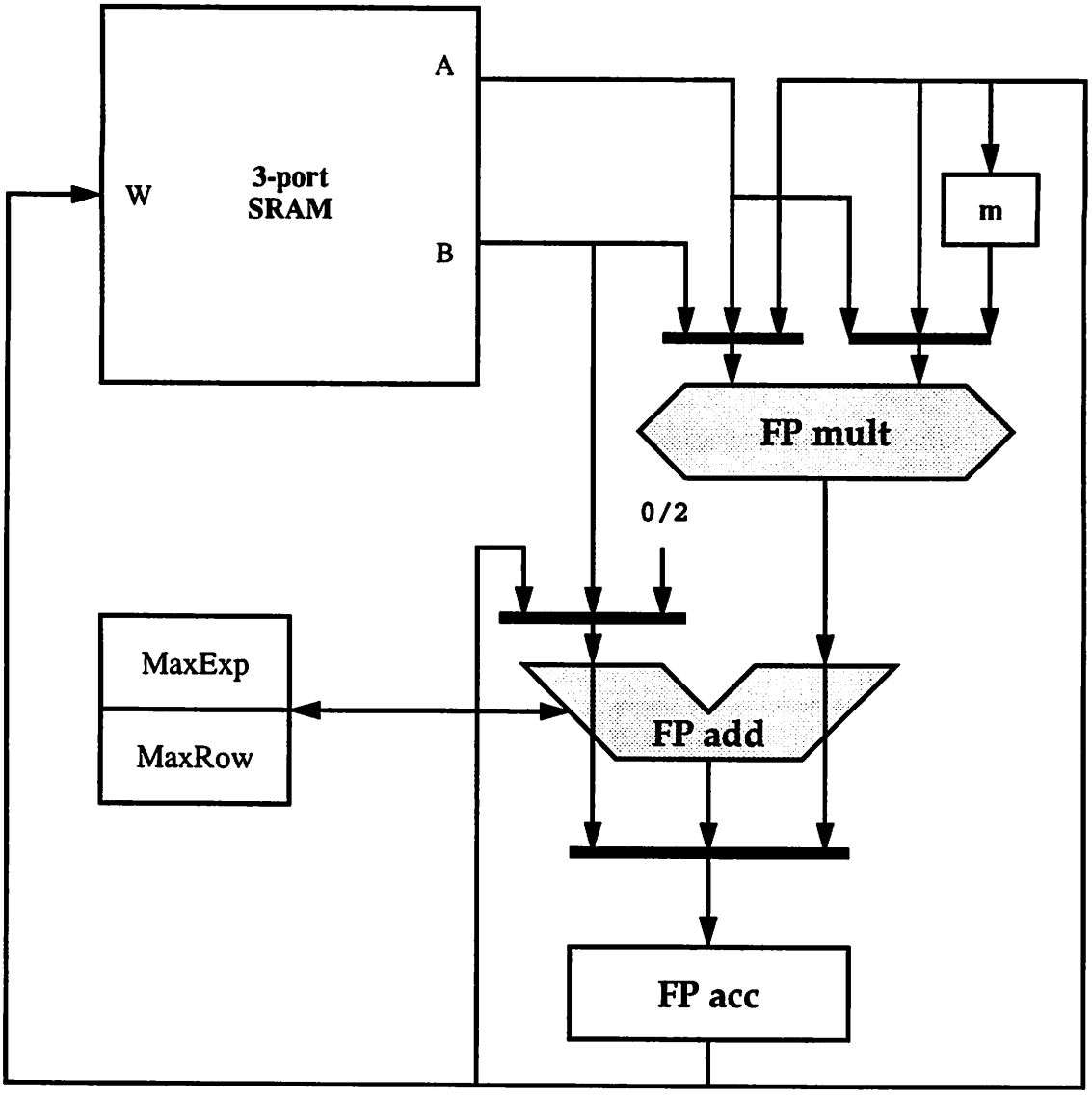
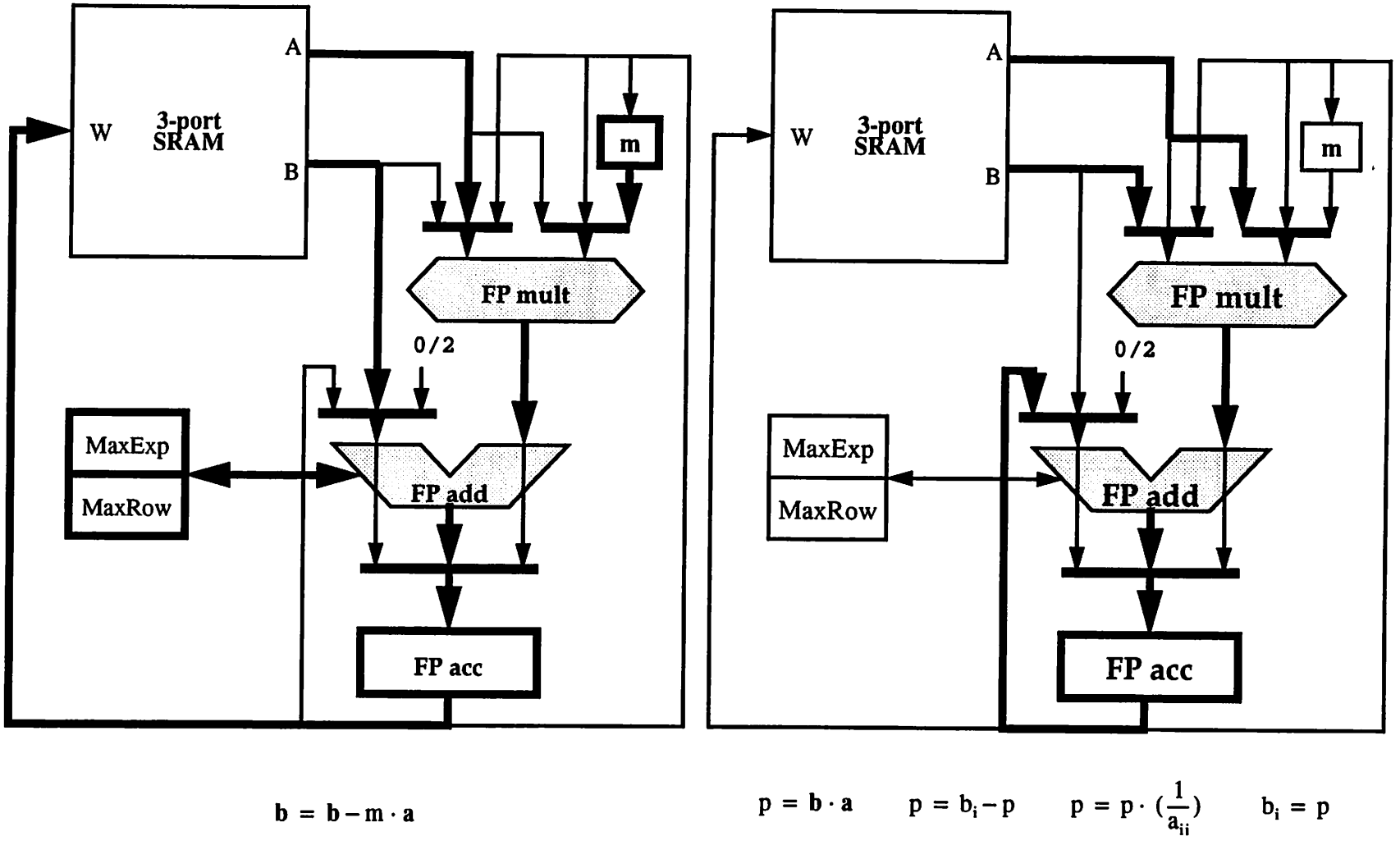


Figure 8-5 Consolidated datapath which can perform all three basic tasks



$b = b - m \cdot a$

$p = b \cdot a$ $p = b_i - p$ $p = p \cdot (\frac{1}{a_{ii}})$ $b_i = p$

Figure 8-6 (a) Datapath in elimination and parallel pivoting mode (b) Datapath in back-substitution mode

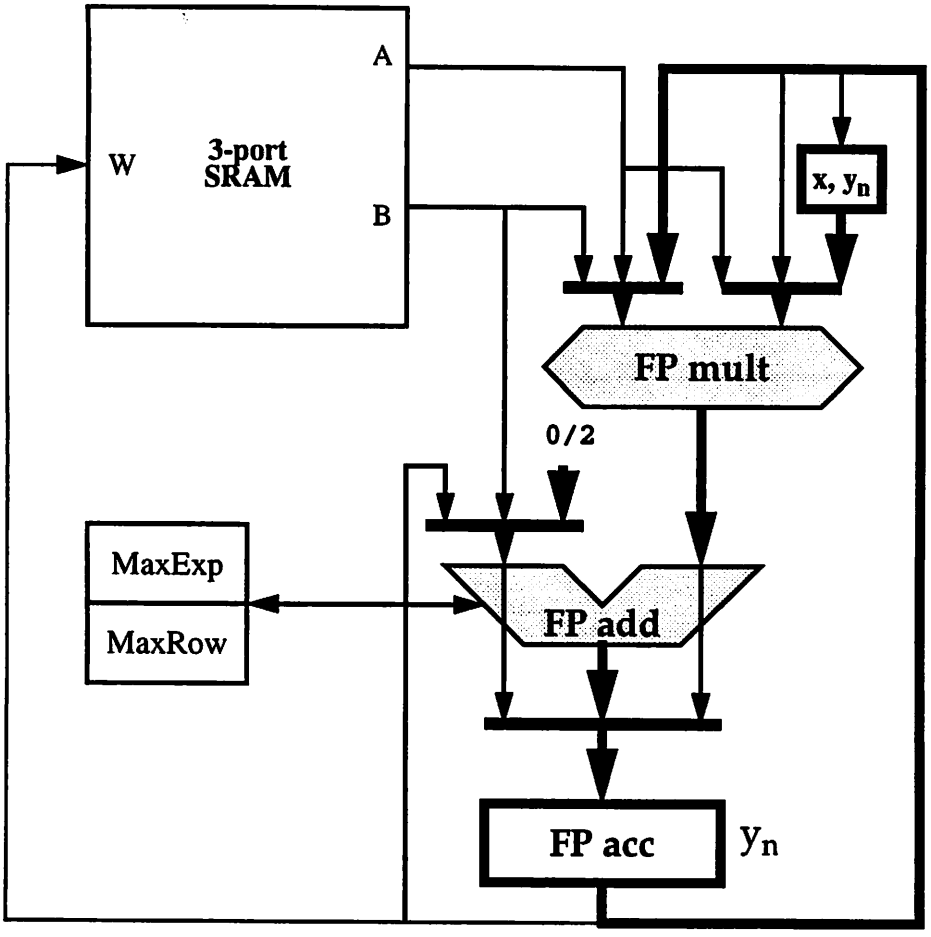


Figure 8-7 Datapath in reciprocal computation mode

$$2 - x \cdot y_n(acc) \rightarrow acc \qquad acc \cdot y_n(tmp) \rightarrow y_{n+1}(acc)$$

and back substitution. The signal flow for each case is drawn in bold type. Figure 8-7 shows how the datapath can be used to compute reciprocals (for division) according to Newton's method. One possibility not shown in the figure is to use a *seed* ROM lookup table to speed up the reciprocal computation.

8.3 Pivot row permutations

The row exchange that follows a pivot search can either be performed literally by swapping rows (via a temporary storage space) or by leaving the rows in place and instead apply a permutation

function to every row index before using it to address the memory. Figure 8-8 shows the basic principle of using a permutation table. Instead of swapping the entire rows, the table allows us just to swap the indices of the table. SMAC therefore uses a table instead of row swapping. The lookup in the permutation table (PTAB from now on) can be pipelined so that it does not slow down the memory access.

8.4 Addressing and address generation

The loop structure of the Gauss/LU algorithm produces a complicated but at the same time regular memory access pattern for each of the 3 memory ports. SMAC must contain an address unit which can generate these patterns with the appropriate timing. In particular, the address unit must pass the addresses through the permutation table to effect the row index translation.

To develop the proper structure of the address unit, it is easiest to start out with the memory structure itself and see what the requirements are. The address variables can generally be called **i,j,k** since these are the names customarily used in the C program code. The main idea of the addressing scheme is that the RAM address for a certain matrix element are created by

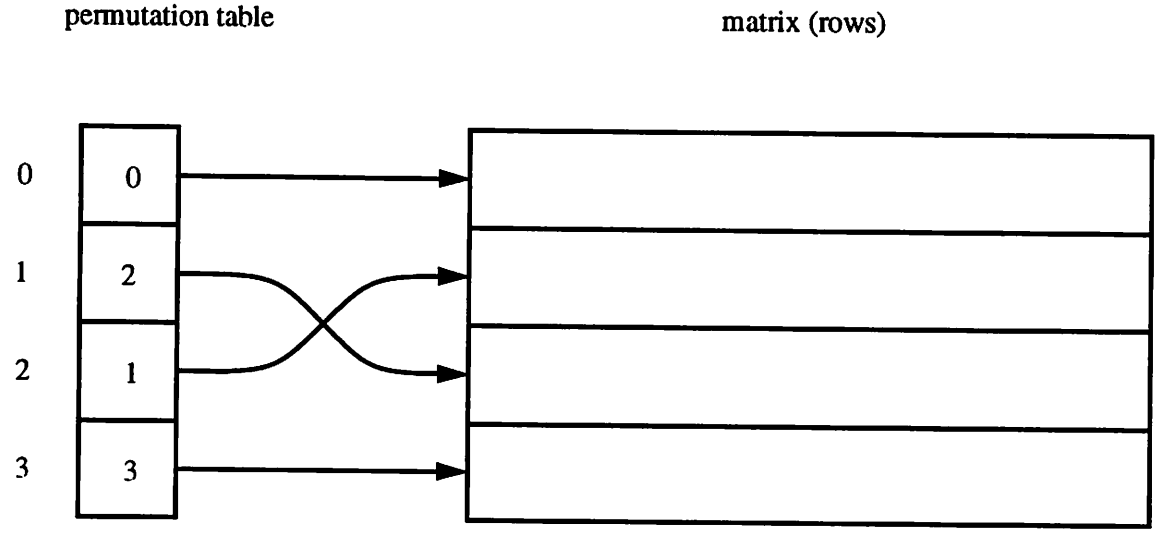


Figure 8-8 Using a permutation table to translate row addresses instead of swapping rows

concatenation of the row address and the column address. Using a pseudo-C notation, this means that the address of a[i][j] (known i C notation as &a[i][j]) is

$$\&a[i][j] = i \bullet j$$

where • is used to denote concatenation of the bit patterns. As long as the number of rows and the number of columns are powers of 2, this method will not create any holes in the address space. For example, we can use 4 bits for i and 4 bits for j to form a complete 8 bit i•j address.

It is preferable to store the right-hand side(s) **b** along with A in the same RAM and access them as a[][n], a[][n+1], ... , a[][n-1+r]. Now, the RAM size must usually be doubled when one needs a bigger size, say from 256 to 512 elements, but sometimes it is possible to simply cut out from the layout the unused parts and add on exactly the number of extra words (elements) needed. However, it is still necessary to increase the address size by 1 bit, and this increase must happen in the most significant bit (MSB) of the address.

To address the extra column(s), we would like to tack on additional address bits to **j**. However, this does not correspond to increasing the total address at the *MSB* side. The solution is to rearrange the bits as shown in Figure 8-9 before applying them to the memory. For simplicity, one can assume that the number of right-hand sides (RHSs) are also a power of 2 (1,2,4,....).

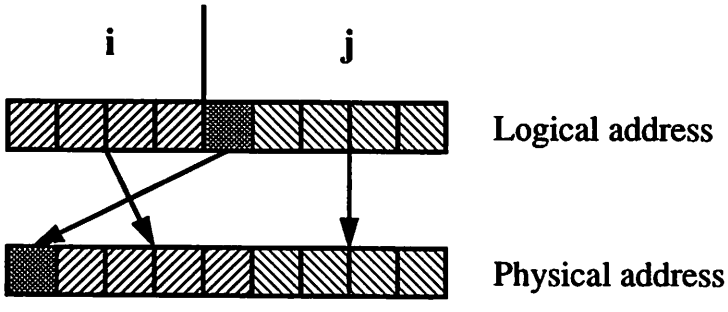


Figure 8-9 Rearranging the address bits to allow right-hand sides to be stored as additional columns in the matrix a[][]

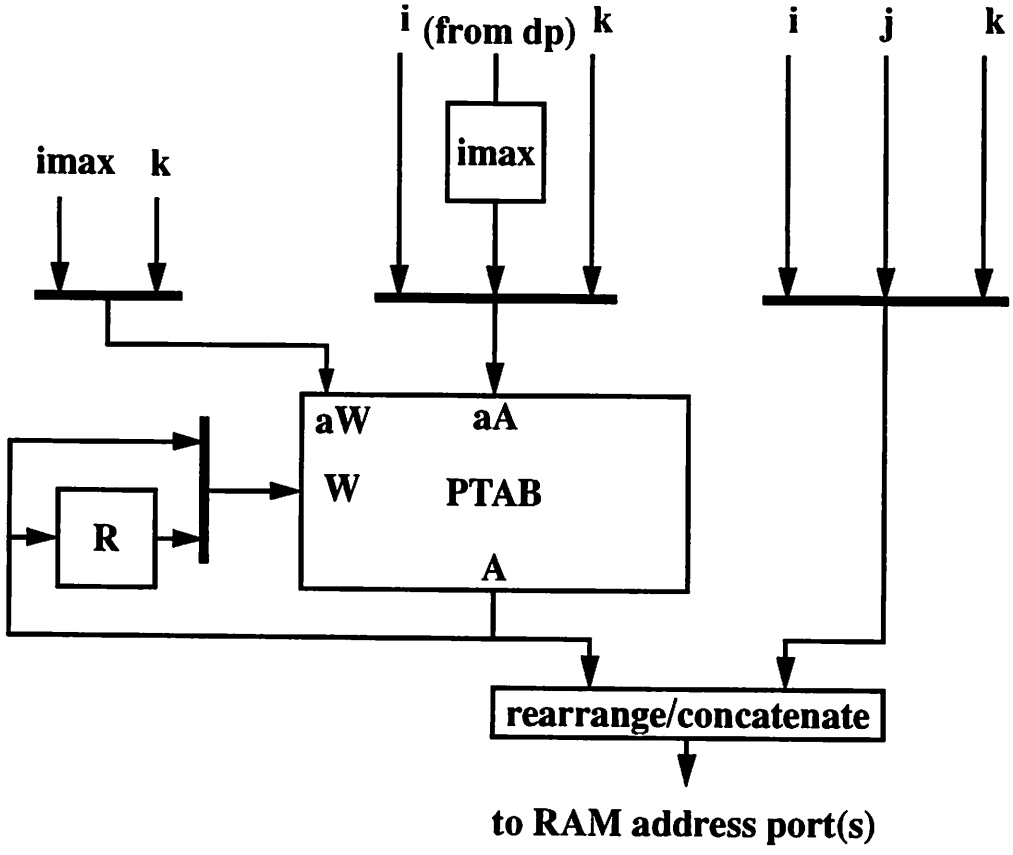


Figure 8-10 Address composition from row and column components. The PTAB row address translation table is included. The circuitry shown is replicated 2 or 3 times in the SMAC architecture, except that the translation table can be shared

Address composition

From studying the Gauss/LU algorithm one can see that only certain combinations of the indices i,j,k occur as row and column indices. Figure 8-11 shows the version of Gauss/LU which is intended for the SMAC algorithm. Specifically, the combinations [ik][ijk] are the ones that are used. This is reflected in the block diagram in Figure 8-10, which shows the hardware for composing a complete memory address from the row/column components and the multiple variables that are required. The special registers **imax** and **R** are used for the pivot update operation:

$$R = ptab[k] \qquad ptab[k] = ptab[imax] \qquad ptab[imax] = R \tag{8-1}$$

```
int linsol (n, r, a)                                                    1
     int        n, r;           /* r= #right-hand sides */             2
     fptype     a[N][NPR];      /* augmented with right-hand sides */  3
{                                                                       4
    fptype      m, prod;                                               5
    int         i, j, k, ir, kr, npr= n+r, row[N];                     6
                                                                        7
    /* Assume pre-pivoted matrix */                                    8
    for (i=0; i<n; i++) row[i]= i;                                      9
                                                                       10
    /* Forward elimination, k is the diagonal index */                 11
    for (k=0; k<n-1; k++) {                                            12
        /* Replace pivot with reciprocal */                            13
        a[kr][k]= 1.0/a[kr][k];                                        14
                                                                       15
        /* Compute all the m-factors and store in a[ir][k] */          16
        for (i=k+1; i<n; i++) {                                        17
            a[ir][k]= a[ir][k]*a[kr][k];                               18
        }                                                              19
        /* Eliminate */                                                20
        for (i=k+1; i<n; i++) {                                        21
            m= a[ir][k];                                               22
            for (j=k+1; j<npr; j++) {   /* Note k+1 NOT k */           23
                a[ir][j]= a[ir][j] -m*a[kr][j], update_max_pivot();    24
            }                                                          25
        }                                                              26
    }                                                                  27
                                                                       28
    /* Compute the last reciprocal 1/a[n-1][n-1] */                    29
    a[kr][n-1]= 1.0/a[kr][n-1];                                        30
                                                                       31
    /* Back substitution on augmented matrix.                         32
       Note u(i,i)==already reciprocal */                              33
    for (i=n-1; i>=0; i--) {            /* Row in a and b */           34
        for (j=n; j<npr; j++) {                                        35
           prod= a[ir][j];             /* a[ir][n+j]==b[ir][j] (rhs) */36
            /* 0 iterations when i=n-1, that is k=n */                 37
            for (k=i+1; k<n; k++) {     /* Col in a, Row in b */       38
                prod-= a[ir][k]*a[kr][j];                              39
            }                                                          40
            a[ir][j]= prod*a[ir][i];    /* Pivot already reciprocal */ 41
        }                                                              42
    }                                                                  43
}                                                                      44
```

Figure 8-11 Version of the Gauss/LU algorithm which works on augmented multiple right-hand sides. This is the form of the algorithm which SMAC is based on. Row index translation and pivot searching and updating are not shown explicitly in the code, as it is assumed these are taken care of behind the scenes by special hardware. The back substitution part of the algorithm is not interleaved in the above formulation

Operation type	Operations
Constants	ar=0,ar=n-1
Tests	ar<n-1,ar<n,ar<npr,ar>0
Arithmetic	ar1=ar2,ar1=ar2+1,ar++,ar--

Table 8-2 Instruction set for address computation unit

The Read/Write ports are called A/W and the corresponding addresses are aA/aW.

Address computation

From studying the algorithm in Figure 8-11 one can determine which operations are needed in the address computation unit. The necessary instruction set is shown in Table 8-2, using the name **ar** to denote one of the registers **i,j,k**. The block diagram for the address unit is shown in Figure 8-12. One possible circuit implementation of an individual register is also shown.

8.5 Loop control and instruction sequencing

The sequencing of the loops in the Gauss/LU algorithm is one of the more complex task of the SMAC architecture. As an illustration, let us first look at how the loops might be administered on a commercial μP or DSP architecture.

If one compiles the code of Figure 8-11 using a standard C compiler, each **for**-loop of the generic type will be transformed into more convenient (from a hardware viewpoint) formulations

```
1.  for (k=0; k<n-1; k++) {BLOCK}
2.  k=0; while (k<n-1) {BLOCK; k++}
3.  k=0; A: if (!(k<n-1)) goto AEND; BLOCK; k++; goto A; AEND:/**/;
```

and then converted to machine instructions. The loop test seen in (3) is serial in nature and means that several cycles (likely 4 cycles in this case) is used after each loop iteration to determine whether there is another iteration. In a superscalar architecture, it may be possible to execute, say,

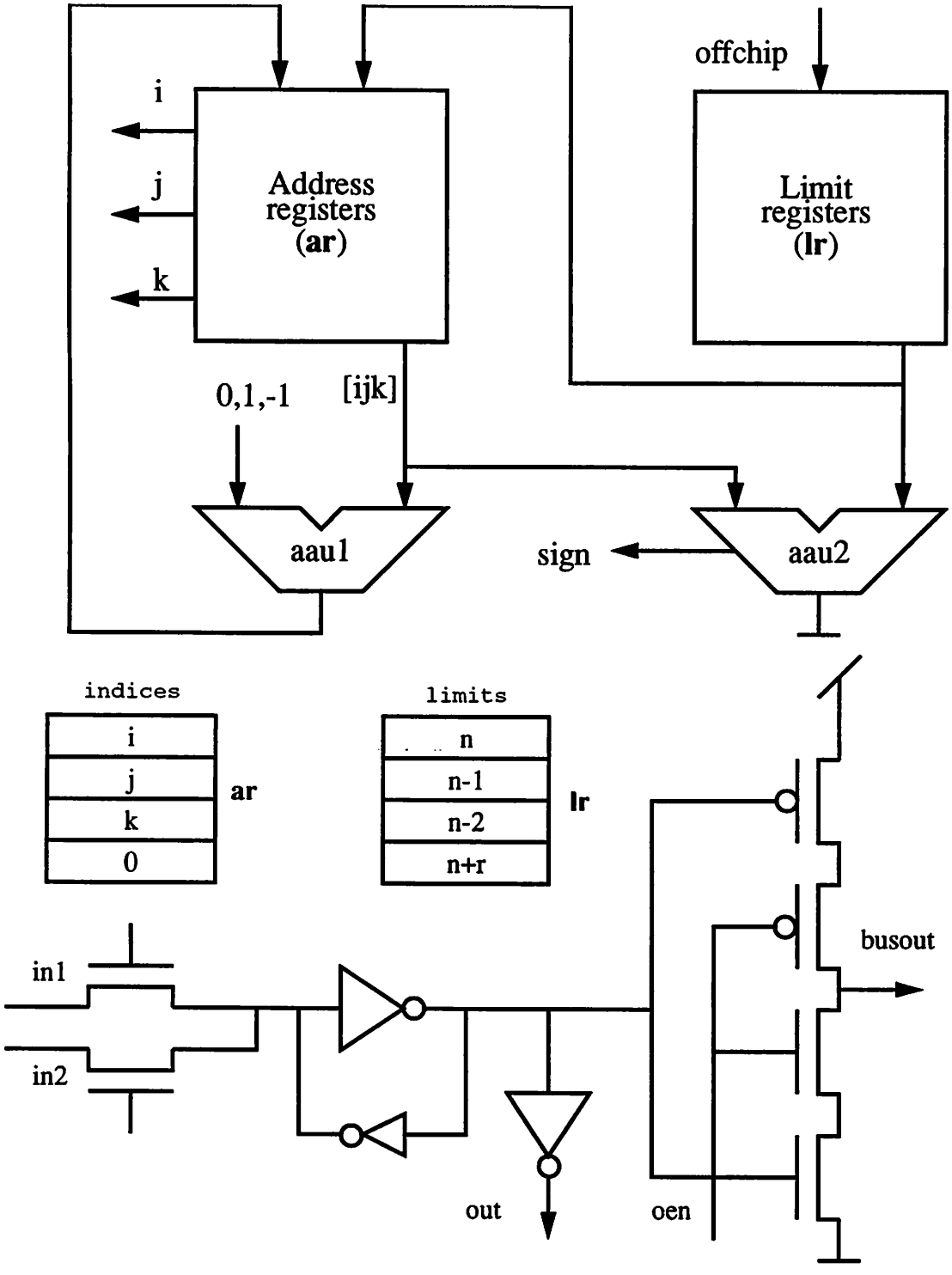


Figure 8-12 (a) Address generation unit of SMAC (b) Contents of register files and (c) Possible circuit implementation

the instructions

```
k++; goto A;
```

in the same cycle. One could even imagine precomputing k+1, compare it to n, set a flag, and know beforehand whether to jump backward or go forward at the end of the loop. These are techniques that to some extent are used in compilers, or can be forced by the user by writing the loop increments and tests using more explicit, machine-near constructs.

The above comments apply to a single loop. If there are nested loops, the problem becomes considerably more complex. A superscalar (say, 2-3 instruction issue) machine will not be powerful enough to produce a 0-overhead loop branch. The problem is especially apparent when 2 or more loops end at the same location, as seen in lines 25-26 and 42-43 of Figure 8-11. In such cases, there is in essence a multiway branch to be taken depending on the value of *several* flags. None of the superscalar architectures are capable of such a job, specifically because the instruction set is intended also for scalar implementations.

What is necessary in SMAC is a multiway branch controller, analogous to the one used in the C-to-Silicon system (section 3.8 on page 62). At the same time, it is most likely not useful actually to program SMAC using a compiler such as the RL compiler. The compiler would not be able to understand our *address composition* scheme (page 225), which is a key to performance.

Controller structure

The controller structure for SMAC can follow the general picture outlined in section 3.9 (p. 67). Since the actual number of instructions in SMAC is rather small, it may make sense to use just one finite state machine (FSM) and not also an additional "control store". The details will not be worked out here, but Table 8-3 contains a small part of the state table to indicate how the branching at the end of each loop is handled, in the case of (up to) 3 nested loops, which is what is needed for the Gauss/LU algorithm. The variables (bits) **is_last(ABC)** are status bits from the address unit, which pre-computes whether the current iteration of each loop will be the last one.

is_last (ABC)	action	is_last (ABC)	action
000	goto C	100	goto C
001	goto B	101	goto B
010	goto C	110	goto C
011	goto A	111	quit loops

Table 8-3 Branching logic for triple-nested loops, assuming the program has already entered the loop. Can be minimized before implementing it in an FSM

The only difficulty with this scheme is that it assumes that all loops will be executed at least once. The loop in line 38 of Figure 8-11 is sometimes executed only once. This case will require some additional attention.

8.6 Building blocks for implementing SMAC

In the following sections, the focus is changed from SMAC architecture to SMAC implementation issues. To create an implementation of SMAC requires a large amount of circuit design. Many of the critical parts of SMAC are not readily available in the LAGER library, and in other cases it is necessary to design new blocks for higher speed. The goal is to create a set of building blocks that will allow implementations with clock speed in the 100MHz(+) range, using standard 1.2μm SCMOS design rules from MOSIS.

The LAGER Silicon Assembly System is employed throughout this design effort, and all blocks are designed as parameterized automatic module generators, so that they can be created in variable sizes on demand. This ensures that the blocks can be used also in other applications. The blocks have generally been implemented using the TSPC (True Single Phase Clocking) design style, [yuan87][yuan89][afghahi90], which will be described below.

The next few sections describe the implementation of several critical circuit blocks (and corresponding automatic module generators) for SMAC. These include

- A heavily pipelined multiplier (**pmult**) with parameterized size.
- Circuits for floating point adder and multiplier datapaths, including operand normalization and renormalization, with parameterized sizes, in the **dpp** design style.
- Datapath pipeline latches.
- A 3-port SRAM (**regfile, regfilew**) with parameterized size and decoding.
- A high-speed PLA (**hpla**) with fully parameterized contents and FSM capability.

Several of the blocks have been fabricated and tested, and the test results will be presented.

8.7 TSPC latch design

Some of the most common latch and clocking technologies use either a 2-phase non-overlapping clock, or a single phase clock but with some form of local inversion. TSPC [yuan89] is a latch and clocking technology which uses a single clock phase that is never inverted. This technology has the advantage that only 1 clock signal needs to be distributed around the chip, which saves area and reduces the clock load. TSPC also saves transistors because there are fewer clock drivers, and no local inversion is necessary.

The basic TSPC latches are shown in Figure 8-13. There are two main types: The p^2-latch and the n^2-latch. The p^2-latch is transparent when clk=0 and latched when clk=1. The n^2-latch is transparent when clk=1 and latched when clk=0. To understand how the latches work, consider the n^2-latch as an example. When clk=1, the clocked transistors are conducting and the latch is essentially two inverters in series (transparent). When clk=0, the first stage of the latch will block a 1 input because the clocked transistor is not conducting, and the latch as a whole is blocked. If the input is a 0, the first stage will invert it into a 1, but this 1 will in turn be blocked by the (identical) second stage.

The p^2-latch works in an analogous fashion: When clk=0, the clocked transistors are conducting and the latch is two inverters in series (transparent). When clk=1, the first stage of the latch will

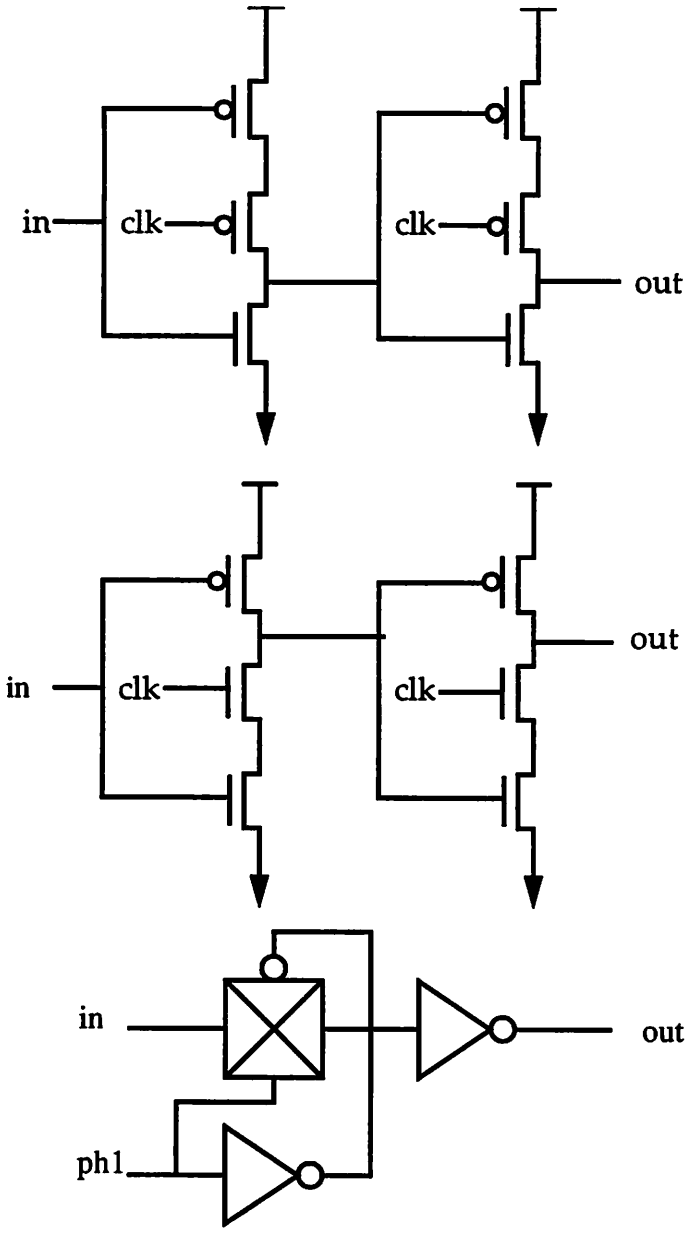


Figure 8-13 (a) TSPC p^2-latch (b) TSPC n^2-latch (c) 2-phase latch

block a 0 input. If the input is a 1, the first stage will invert it into a 0, but this 0 is blocked by the second stage.

Note that the TSPC latches have 6 transistors, which is the same amount as the standard 2-phase non-overlap latches if we count the extra inverter that is needed for local inversion of the appropriate phase.

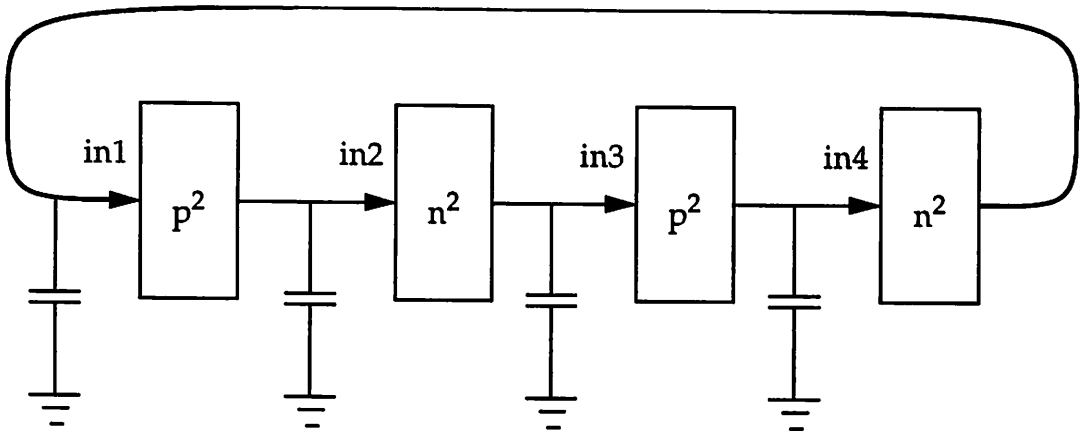


Figure 8-14 Circular shift register for testing sensitivity of TSPC latch operation to clock slope

TSPC-based systems are less sensitive to skew problems than 2-phase systems, and special techniques such as distributing the clock against the data flow direction can be used to minimize the problem. On the other hand, TSPC systems depend on having a small edge rise/fall time, because a clock with an intermediate value will set both a p^2-latch and a n^2-latch in a semitransparent state. If two such latches are connected directly in series with no delay in between, the signal may race through both latches during one edge if the edge rate is not sufficient.

The specific rise/fall times needed are best determined by simulation of a test circuit, for example the circular shift register shown in Figure 8-14. The minimum slopes depend on the node loading as well as the transistor sizing. A typical simulation result for a 1.2 μm CMOS process (HP CMOS34) was 3.0 ns for minimum-sized devices.

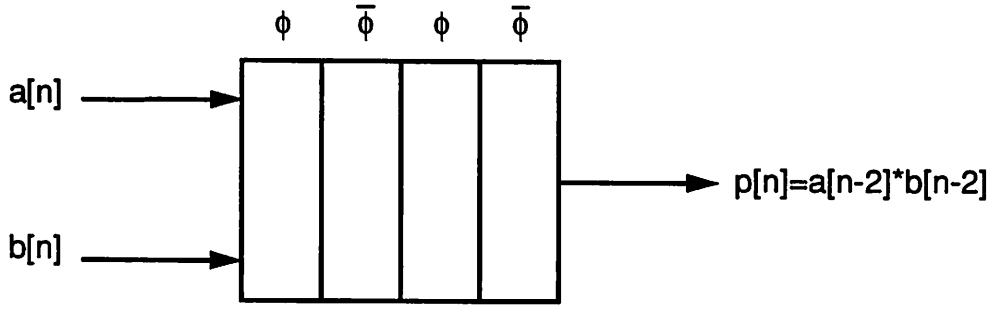


Figure 8-15 Pipelined multiplier using per-phase latch stages

8.8 Pipelined high-speed multiplier (pmult)

The design goals of **pmult** were determined by the fact that the module is intended as a building block for custom floating point units:

- Pipelined design for high throughput
- Minimal pipe latch overhead
- Sufficient regularity to allow automatic module generation
- Parameterized size

Figure 8-15 shows the basic principle of pipelined multiplication, which in this case means that a new set of operands can be applied each clock cycle, and that a new result will become available each clock cycle.

Multiplication **a•b** of two numbers consists of adding the appropriate multiples of **a** to itself, as illustrated in the familiar multiplication parallelogram in Figure 8-16. Each row in the multiplication array is typically a carry-save adder which passes on the carries generated to the next stage below, to avoid having a carry propagating up along each stage. At the bottom of the array, there will be a Carry Vector and a Sum Vector which must be summed using a carry-propagate adder. This is operation is often referred to as *vector merging*.

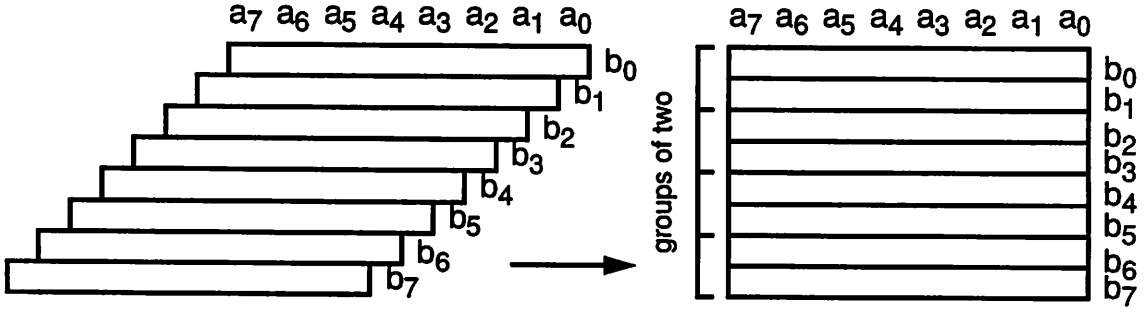


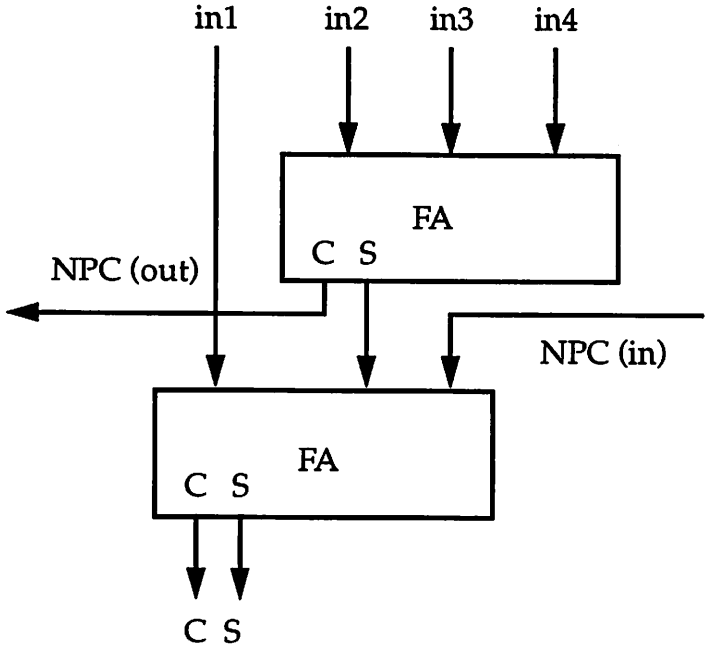
Figure 8-16 Multiplication: The parallelogram of partial products. The parallelogram is typically rectified (reshaped into a rectangle) to save area. Grouping of multiple partial products is optional

If a multiplier is implemented on a chip, the parallelogram is normally pushed into a rectangular form, and the alignment mismatch between the stages is taken care of by special inter-stage routing.

Pipelining and compressors

Carry-save multiplier arrays can be pipelined between each stage if desired. However, the heavier the pipelining the larger the latency of the unit. For a 24x24 multiplier, the latency would be roughly 12 clock cycles of there is one pipe stage for each partial product, with one corresponding clock phase. Since many algorithms, such as Gauss/LU, have a limited pipelining margin, it makes sense to try to add up more than 1 partial product in between each pipe stage.

One possible design is that each pipe stage combines the previous Sum/Carry vectors with 2 additional partial products to generate the next Sum/Carry vectors. This method implied that there will be 4 bits (of equal binary weight) to add at each position, requiring at least 3 bits of output to encode the result. The **4:2 compressor** (also known as a 4:2 adder or 5:3 counter) is a logic circuit



Signal	Weight
in1-4	1
NPC(in)	1
NPC(out), C	2
S	1

Figure 8-17 4:2 compressor made from 2 full adders

which performs the desired task. The 4:2 compressor takes 4 inputs (and a carry) of weight 1, and outputs a Sum of weight 1 and two carries of weight 2. The second carry (named NPC for Non-Propagating Carry) is fed into the neighboring compressor, but will not propagate any further, as can be deduced from the logic diagram.

The basic cell for the multiplier array contains the compressor, 3-bits of pipeline latches and the "and" functions necessary to form the partial products $a \cdot b_i$ and $a \cdot b_{i+1}$, as shown in Figure 8-18. The inputs in1/in2 are used for the Sum/Carry bits from the previous stage, and the in3/in4 inputs are used for the multiplicand (**a**). The wires pp1bar and pp2bar carry the bits b_i, b_{i+1}, of the multiplier **b**.

Using a pipelined array of 4:2 adders requires that the stages must be wired up as shown in Figure

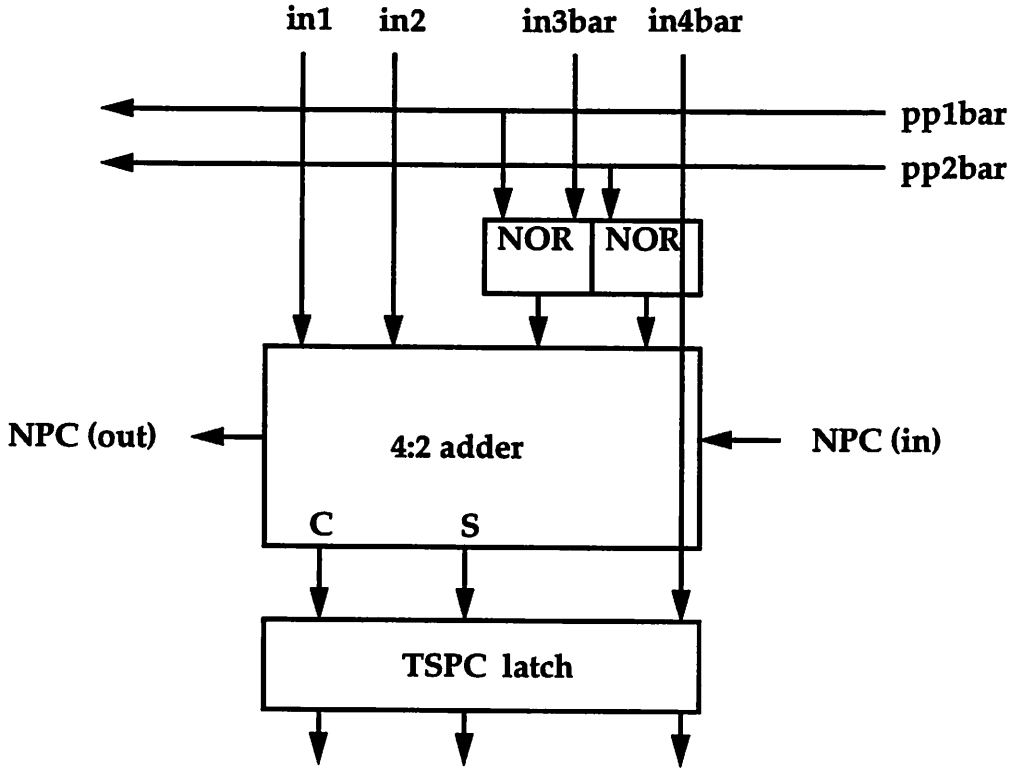


Figure 8-18 Basic cell of pipelined multiplier array. The cell exists in a p^2-latch version and an n^2-latch version. The S,C and a_i bits are pipelined through the latch

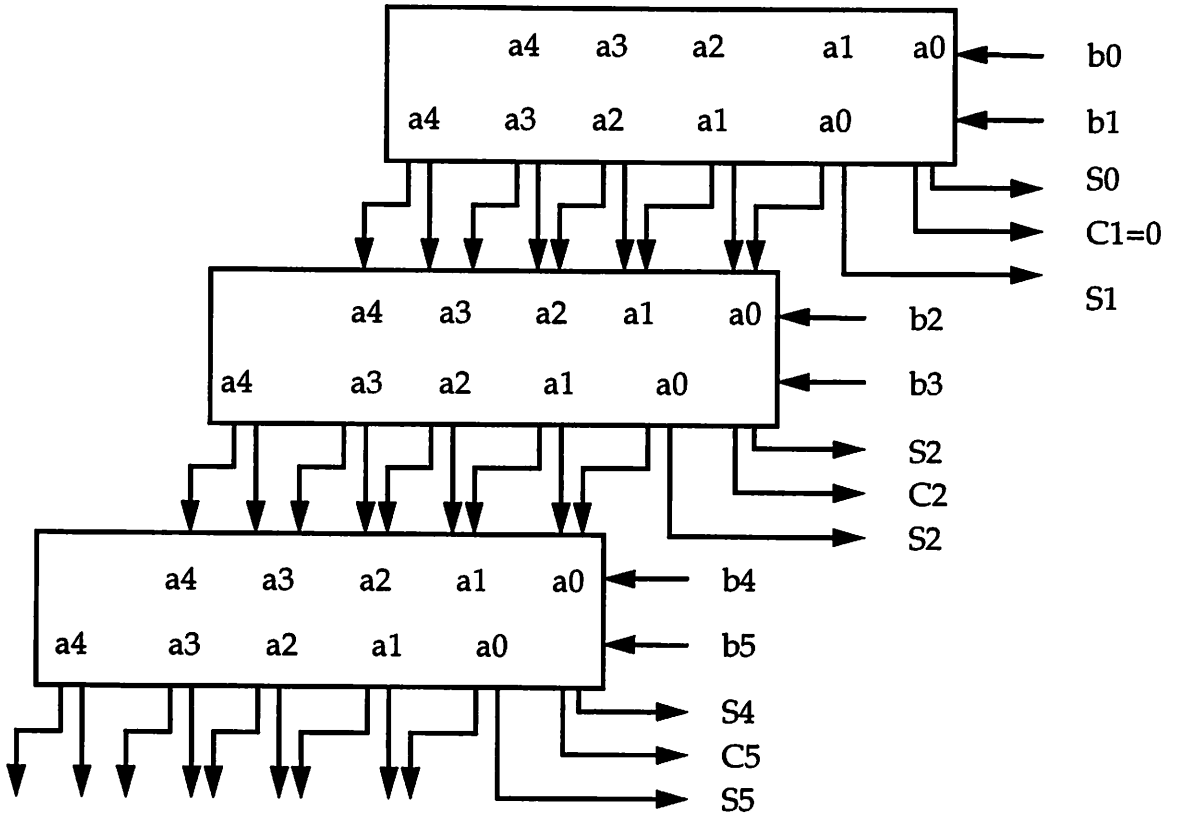


Figure 8-19 Organization of pipelined multiplier array based on 4:2 compressors. Each stage produces 2 sum bits and 1 carry bit which are routed out on the right-hand side

8-19. Each stage produces 2 sum bits and one carry bit (S_{2k}, S_{2k+1}, C_{2k}) that are routed out to the right-hand side of the array. These vectors of carries and sums must be added using a carry propagate adder, analogous to the vector merging that is going to take place on the bottom of the array.

Operand and result pipelining (input and output delays)

Since there is a pipe delay between each of the stages in Figure 8-19, it is necessary to delay the inputs (b_{2k}, b_{2k+1}) correspondingly. Also, the results (S_{2k}, S_{2k+1}, C_{2k}) arrive in a staggered fashion and need to be delayed. The inputs must be delayed more at the bottom of the array, and the outputs more at the top. This is a convenient property, because it means that the I/O delays can be

arranged into a square piece of layout, as indicated in the preliminary floorplan shown in Figure 8-20. This floorplan gives a regular and rectangular structure to the layout. However, the vector-merge addition spans 2N-1 bits for an NxN multiplier, and is the critical delay path. The length of the carry-propagate can be reduced to length N by adding up the Sum and Carry bits on the Right Hand Side (RHS) as the bits become available, *before* passing them into the output delays. This approach is shown in Figure 8-21. Since the RHS bits arrive in a staggered fashion, the RHS adder can be pipelined as shown in Figure 8-22. Each stage of the pipeline consists of adding together (S_{2k}, S_{2k+1}, C_{2k}) with the carry-in from the previous stage. This operation requires 2 Full Adders, much like the 4:2 compressor circuit, and the 4:2 compressors and the RHS adder cells will therefore be matched in their speed, which is convenient.

Because the stages in the main array operate on alternate clock phases (low or high clk), the RHS merger as well as the input/output delays must be carefully assembled from latches of the correct polarity (n-type or p-type).

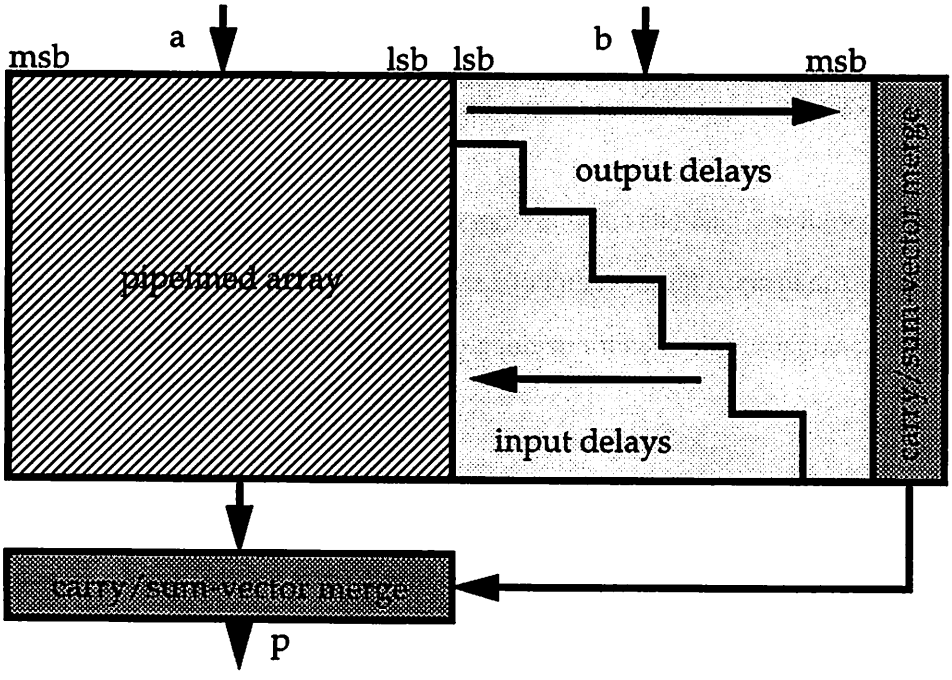


Figure 8-20 A possible floorplan for the pipelined multiplier

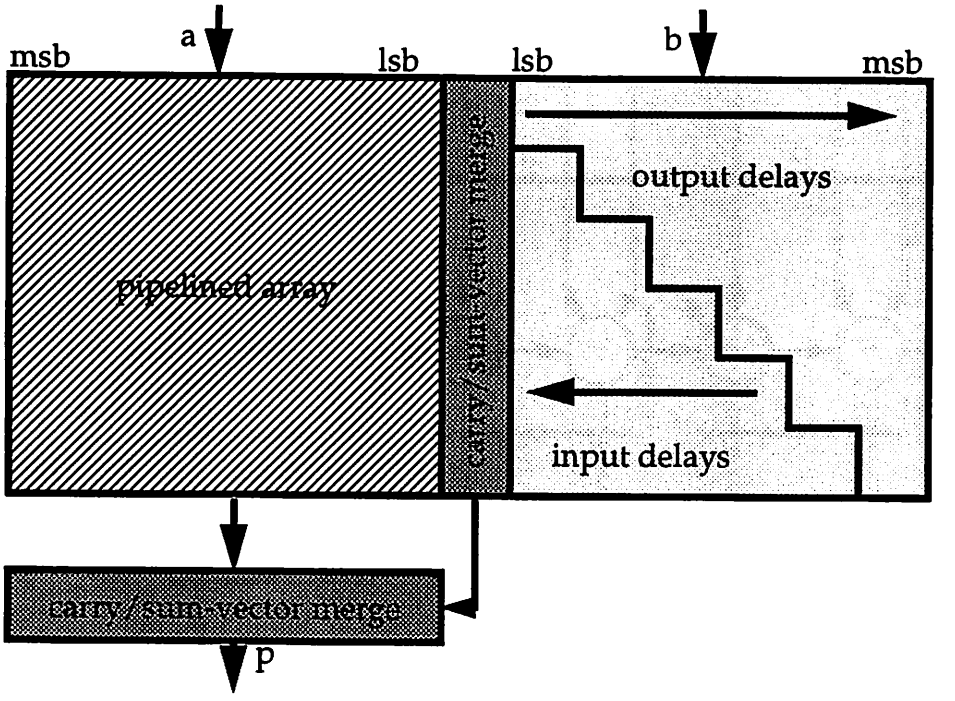


Figure 8-21 Final floorplan for **pmult**. The RHS vector merge can be done by a simple pipelined ripple-carry, leaving only an NxN bit vector merge at the bottom

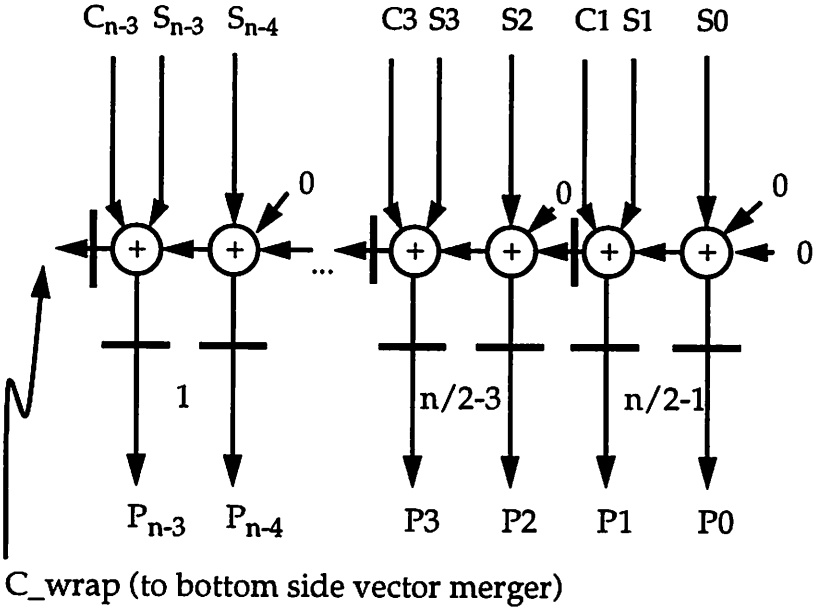


Figure 8-22 Logic diagram of pipelined Right-Hand Side (RHS) vector merger. The output delay latches are also shown, with numbers indicating the number of stages for an NxN multiplier

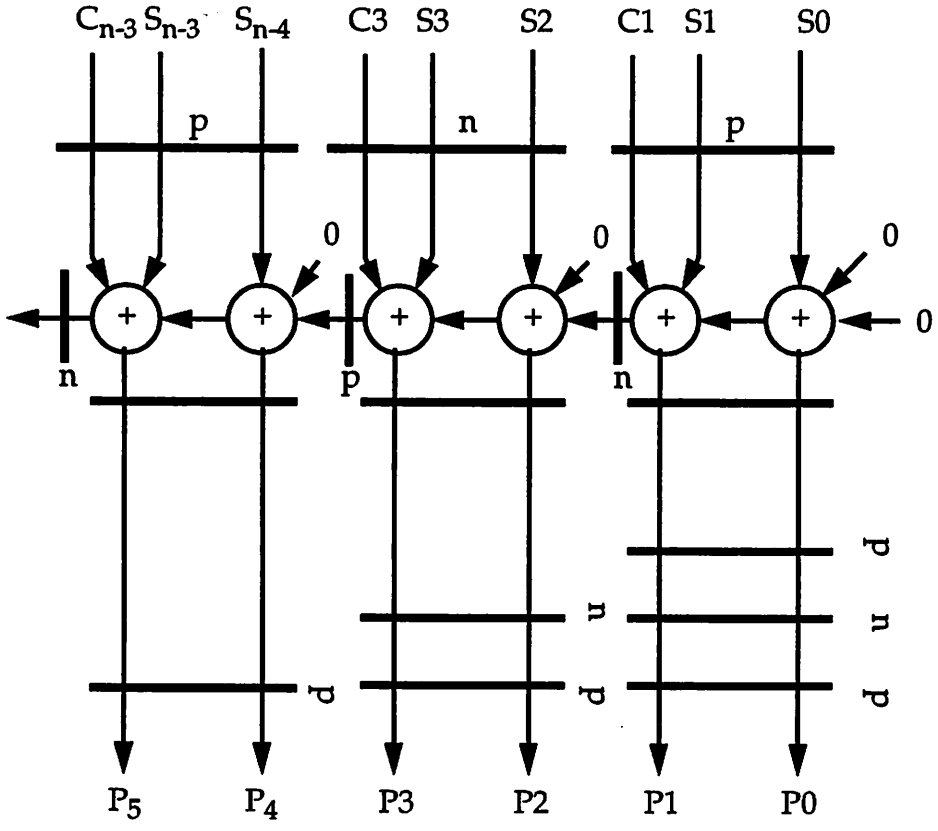


Figure 8-23 The RHS merger and the I/O latches must start with latches of the appropriate polarity, so as to fit with the timing of the main array

Vector merger

The bottom-side vector merger must be a fast adder that can add two 32-bit numbers within the course of 1 clock phase, which is 5ns or shorter for clock frequencies of 100 MHz or higher. The functionality required is shown in the logic diagram of Figure 8-24.

Fortunately, such an adder exists in the LAGER library (cs3.sdl). It is a carry-select adder with pass-gate logic for high speed operation. The cs3 adder can be hooked up to the main array using the Flint Place-and-Route tool.

If the cs3 adder were not fast enough, it is also possible to pipeline the final carry-propagate stage of the multiplication.

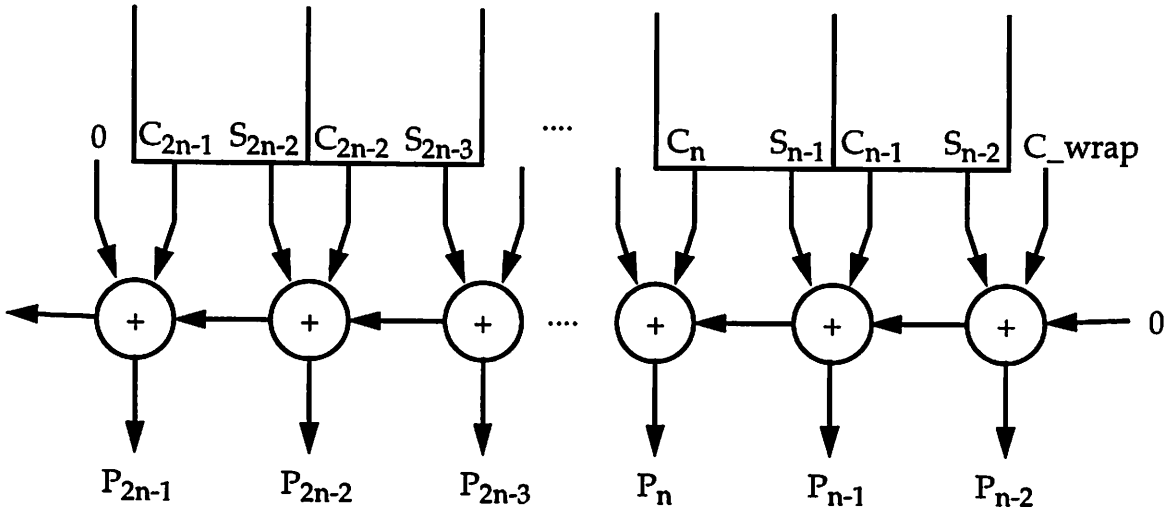


Figure 8-24 Logic function of the bottom-side vector merger

Tiling and circuit implementation

The design of **pmult** is strongly influenced by the requirement that the layout should be amenable to automatic module generation. The layout design of **pmult** involved creating the necessary leafcells, as well as making them fit together (partly with the help of routing cells) so that the resulting layout could be tiled by the TimLager layout generator (cf. section 3.11).

The tiling procedure for **pmult** is fairly complex, consisting of 255 lines of C code. The details of the code will not be presented here. However, Figure 8-25 shows a tiling example for a small (8x8) multiplier. This example is large enough to show the tiling in its full generality. There is a total of 23 different leafcells that are used to form the multiplier, not counting the cells in the cs3.sdl carry-select adder.

Simulation results

A 24x24 test layout of the multiplier was automatically generated, including the cs3.sdl VMA (vector-merge adder). The layout was extracted and simulated with IRSIM, yielding a clock speed of 108 MHz using parameters derived from the HP CMOS34 process.

compRT	compRT	compRT	compRT	compRT	compRT	compRT	compRT	raddRT	1	2	3	3	3
pmult	pmult	pmult	pmult	pmult	pmult	pmult	pmult	raddn	yinv	in2out	outdp	outdn	outdp
compR	compR	compR	compR	compR	compR	compR	compR	raddR	4	5	6	7	7
nmult	nmult	nmult	nmult	nmult	nmult	nmult	nmult	raddp	yinv	indp	in2out	outdn	outdp
compR	compR	compR	compR	compR	compR	compR	compR	raddR	4	8	5	6	7
pmult	pmult	pmult	pmult	pmult	pmult	pmult	pmult	raddn	yinv	indn	indp	in2out	outdp
compR	compR	compR	compR	compR	compR	compR	compR	raddR	4	8	8	5	6
nmult	nmult	nmult	nmult	nmult	nmult	nmult	nmult	ncomp_LR	yinv	indp	indn	indp	in2out

Figure 8-25 Tiling example for 8x8 **pmult** multiplier. The main array contains 9 different leafcells, and the RHS/ input/output section contains 14 different cells. The numbered cells have the following names:
1=invRT, 2=in2outT, 3=outdRT, 4=yinvR, 5=in2outL, 6=in2outR, 7=outdR, 8=indR.
Semantics: R=route or right, L=left, T=top

Test chip

A test chip with the 24x24 multiplier has been designed, fabricated and tested. Because of the high-speed operation of the chip, it is not practical to apply inputs and observe outputs at full speed. Instead, a test architecture was developed that allows alternating sets of inputs to be applied from internal registers at full speed, while the alternating outputs are latched into two different result registers. The result registers are static and will hold the results for later external observation at low clock speed. Figure 8-26 shows the test chip architecture. For high speed testing, the PLA is used as a sequencer to control the alternate application of two pairs if inputs to the multiplier array, and the latching of the results as they emerge at the end of the pipeline.

The input-side test circuits are shown in Figure 8-27. The registers store two operands which can be alternated as inputs to the pipeline by toggling the multiplexer control signal (this is done by the PLA). The inputs can also be fed straight to the array by controlling the 2nd multiplexer. On the

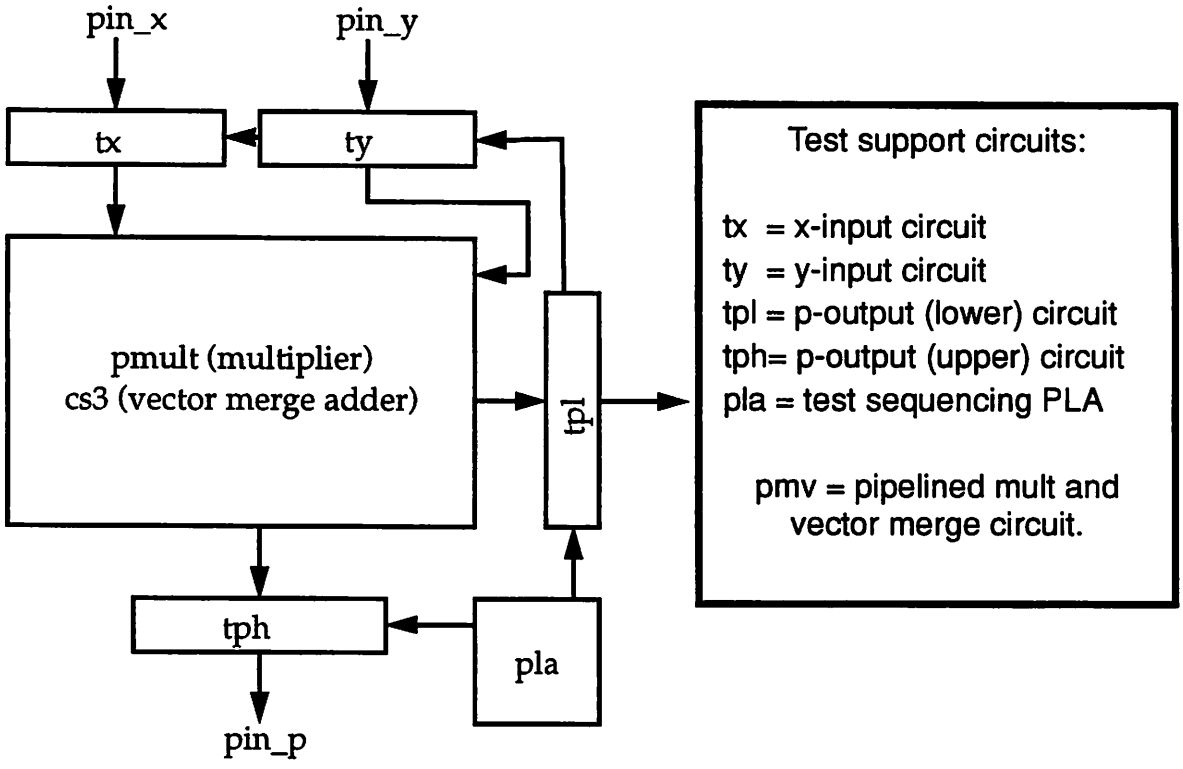


Figure 8-26 **pmult** test chip architecture

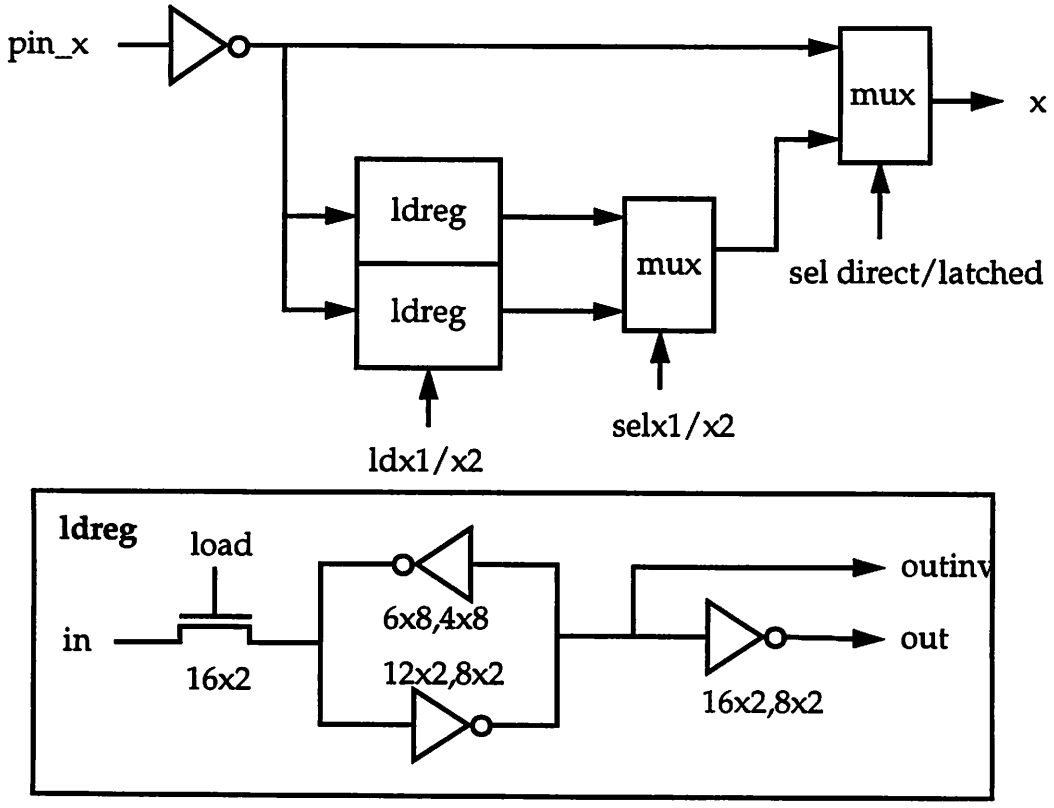


Figure 8-27 Input side test circuits for pmult

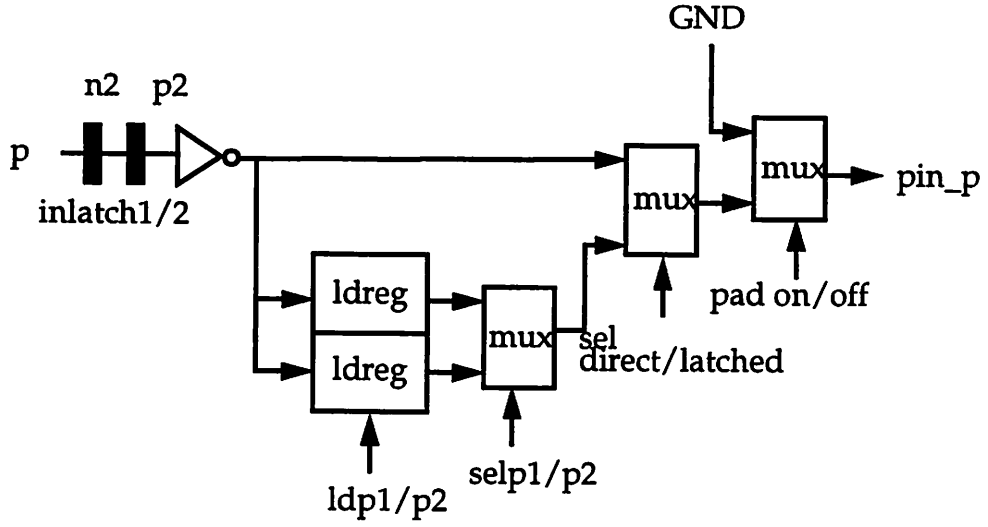


Figure 8-28 Output side test circuits for pmult

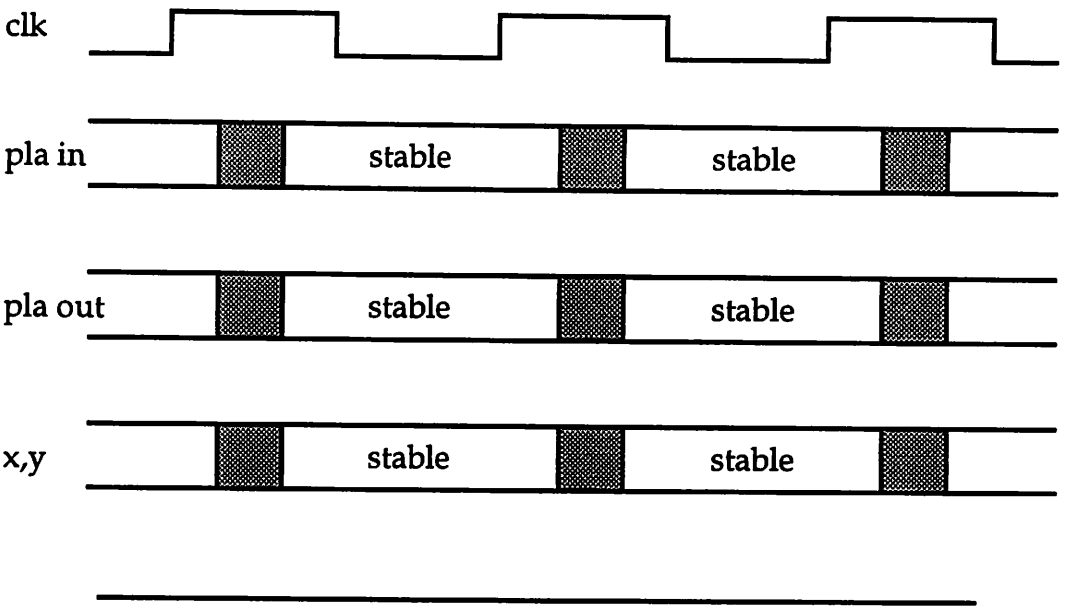
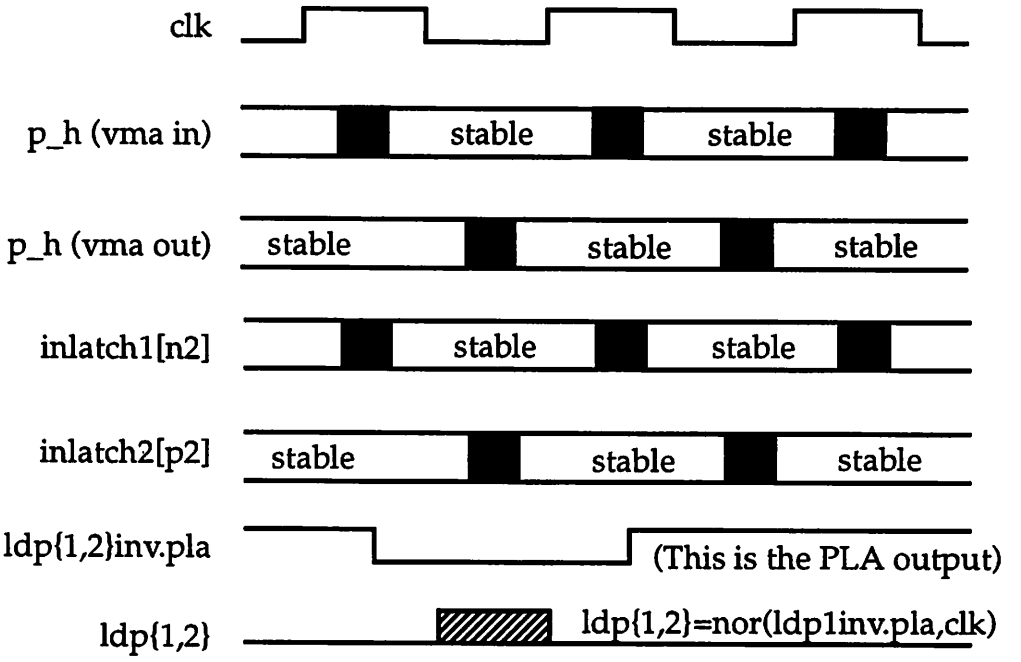


Figure 8-29 Timing of the input-side test circuits. The alternation of two sets of inputs (x1,y1) and (x2,y2) can be controlled by a PLA which selects the appropriate source register. The PLA output must change during clk=1



ldp{1,2} = nor (ldp1inv.pla, clk) = and (ldp1.pla, clkbar)

Figure 8-30 Timing of the output-side test circuits

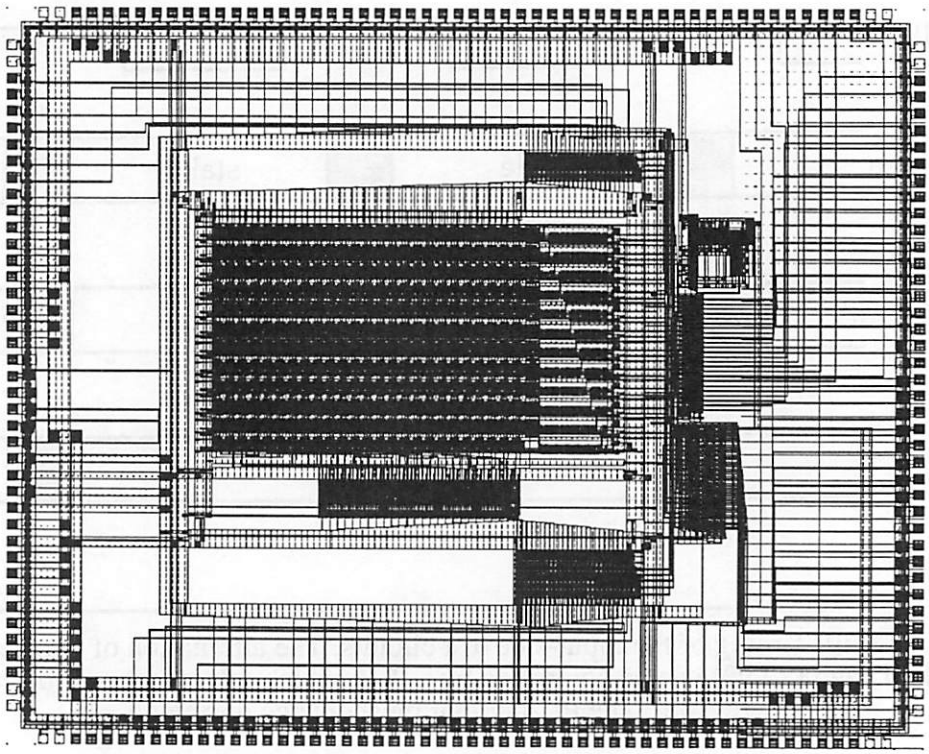


Figure 8-31 CIF plot of pmult multiplier testchip (**pmvt24c**). The chip is severely pad-limited. The size is 16702x12932λ^2, or 10021x7759μm^2

output side (Figure 8-28), the results can either be passed straight to the pins under manual control, or they can be latched into the registers under PLA control.

Test results

The chip (**pmvt24c**) was tested using an HP 16500 pattern generator and logic analyzer. I found that the chip had one serious problem, namely that the clock drivers were not strong enough to create the necessary slope to avoid races in the TSPC latches (cf. section 8.7). However, by turning the supply voltage down to 3V, it was possible to make the chip work (probably because the latches slowed down more than the clock drivers), and the functionality was verified at f=30MHz. This test was done straight through the pads, as the PLA did not work at this voltage (the PLA will be discussed in a later section). As a result, the speed is not as high as what could be achieved with the internal test generation/acquisition and at full supply voltage. I believe that the 108 MHz speed predicted by simulation can be achieved with better clock drivers.

8.9 Floating point datapath building blocks

The **pmult** block described in the previous section is the cornerstone of the floating point unit design, and is intended for multiplying the mantissas of two floating point numbers. In addition, an exponent datapath, plus circuits for mantissa normalization, are needed. The building blocks needed are mostly a subset of the functions needed to create a floating point *adder*, so in this section we will concentrate on blocks needed to create high-speed, pipelined, parameterized floating point adders.

All the blocks described in this section are created in the **dpp** design style (cf. section 3.11), which is a key to easy parameterization. The strategy is that one can use **dpp** to create two separate datapaths: one for the mantissa and another one for the exponents. The mantissa datapath is shown in Figure 8-32, and the exponent datapath in Figure 8-32. The notation **x.{s,m,e}** is used to denote the *sign*, *mantissa* and *exponent* part of the number x, respectively.

Table 8-4 contains a list of the new datapath blocks designed for the floating point library. Some of

Name	Function
bufS	Buffer with selectable size
fmux	Fast mux21
fnorM	Fast NOR of M-out-of-N datapath bits (0-detect, floating point normalize)
fshift	Fast logarithmic up/down shifter stages (1,2,4,8,16)
fxor	Fast controlled inverter (XOR with N data inputs, 1 control input)
invS	Inverter with selectable size
ldreg	Static (weak feedback) register with load
n2	n-squared tspc latch
p2	p-squared tspc latch
shift_tc	shift.c from Lager/cellib/dpp but with tc=top control slice

Table 8-4 New datapath cells for floating point

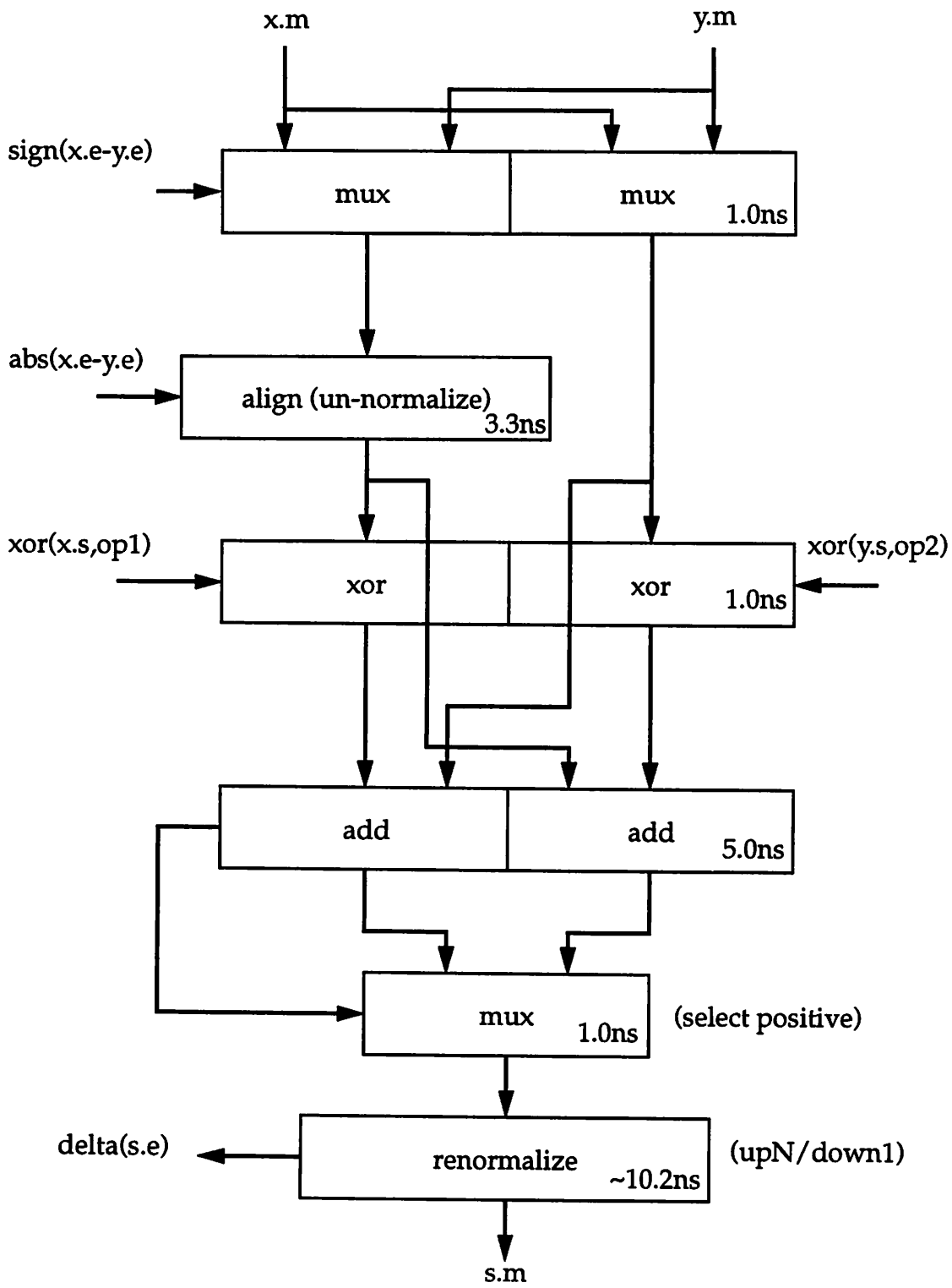


Figure 8-32 Mantissa datapath for floating point adder.
The adder computes the function $s=f(x,y)=op_1(x)op_2(y)$ where op_1, $op_2=+/-$.
Delays are estimates based on new library cells and N=24 bits mantissa

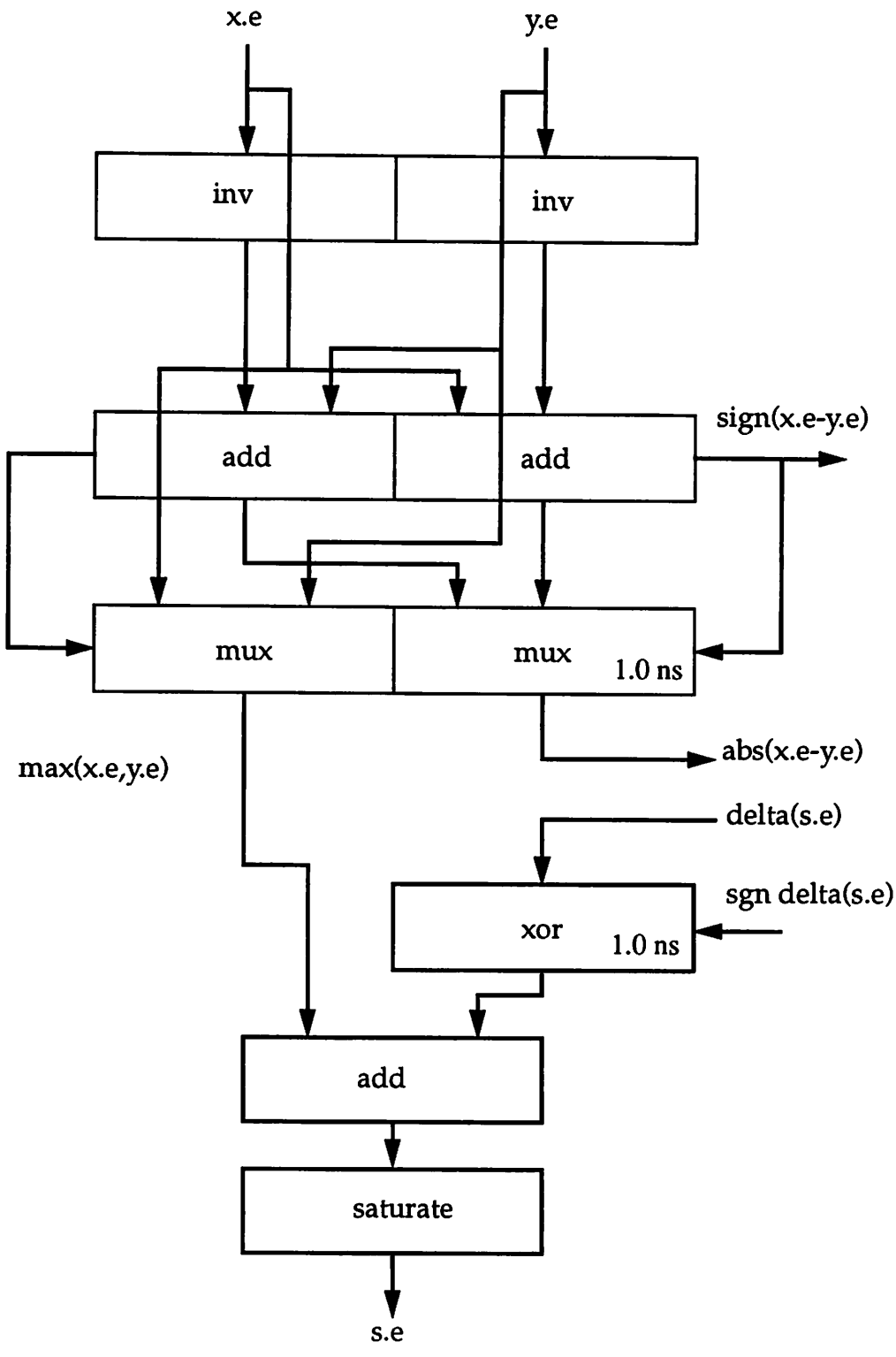


Figure 8-33 Exponent datapath for floating point adder.
The adder computes the function $s=f(x,y)=op_1(x)op_2(y)$ where op_1, op_2=+/-.
Delays are estimates based on the new library cells

the blocks are really a family of blocks with the same function, such as the fshift block, which contains logarithmic shifter stages for up or down shifting and 1,2,4,8,16 bits of shift.

It appears that alignment and normalization operations are in general the most time-consuming and complex part of floating point hardware design. Mantissa alignment refers to the operation of shifting down the mantissa and correspondingly increase the exponent of the operand with the smallest exponent so that the exponent becomes the same as for the larger operand. This is a necessary precursor to adding the two mantissas, and corresponds to the concept in *fixed* point arithmetic that it only makes sense to add numbers that have the same scale. Figure 8-34 shows the approach used in this work, which is to use a logarithmic shifter. The log shifter has the advantage that no decoding of the shift amount is necessary.

The normalization operation at the end of an addition means to shift the mantissa up so that there are no leading zeros, and adjust the exponent down correspondingly. When the width of the mantissa is fixed (not a parameter), handmade custom layout such as the one used in

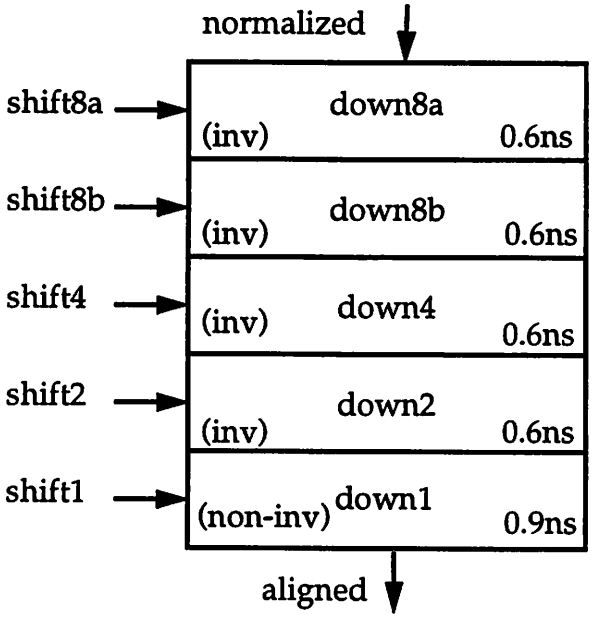


Figure 8-34 Mantissa alignment in 3.3ns using a logarithmic shifter

[hu87][bose88] is advantageous. The approach taken here (Figure 8-35) is not the fastest method, but it is much easier to parameterize. The idea is first to look for a 1 among the first 16 bits, and shift up if there is none, then look for a 1 among the first 8 bits of the (possibly shifted) result, and so on. However, since a 16-bit NOR is very slow, I have instead used two stages of 8-bit NOR, since 8-8-4-2-1 shifts are sufficient to normalize a 24 bit mantissa. Moreover, in the figure is shown that the two 8-bit NOR operations are done in parallel (on the uppermost 16 bits of the

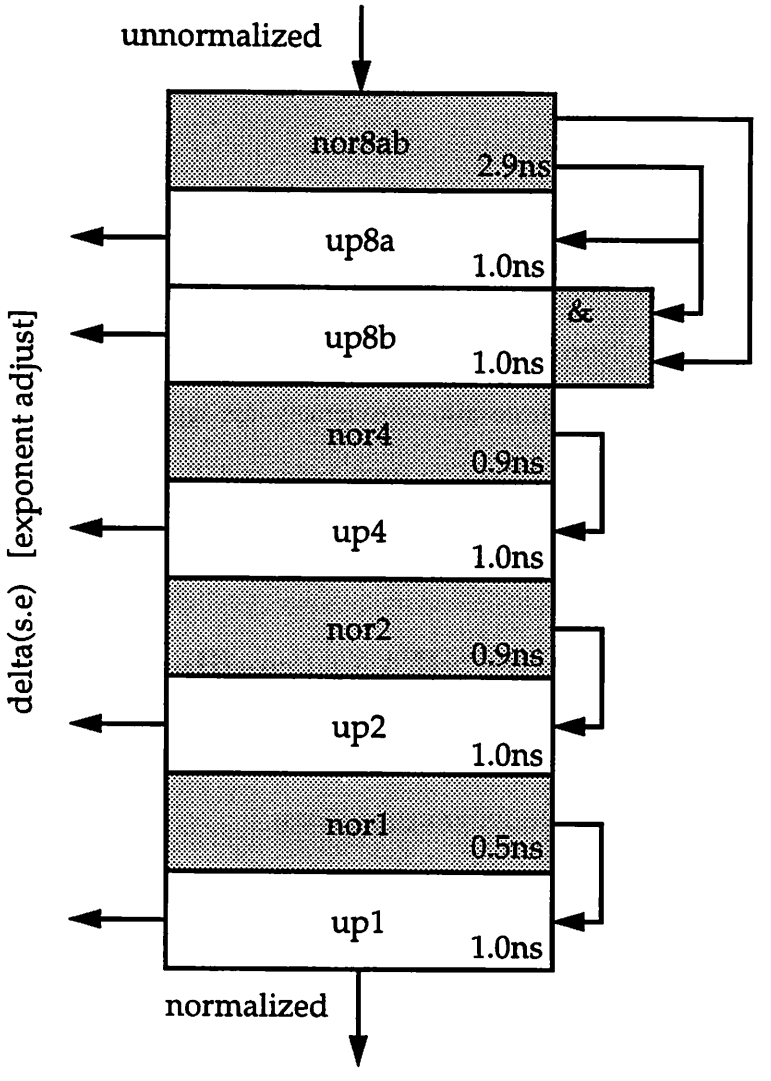


Figure 8-35 Mantissa normalizer built around logarithmic NOR-based 1-detectors and logarithmic shifter stages. Total delay is 10.3ns (estimated)

input), and the 2nd shift of 8 is performed only if *both* of the NOR operations did not detect a 0. The delay of the extra AND gate is inconsequential since it takes place while waiting for the delay through the first shifter stage.

Pipelining

Pipelining the datapaths (not shown in the figures) is easily accomplished by inserting TSPC latch datapath stages. The optimal number and placement of the stages depends on the mantissa and exponent size, but it can be expected that about 4 stages (2 clock cycles) will be sufficient for 24+8 bits mantissa+exponent formats when targeting f=100MHz.

Test chips

No test chips containing the new floating point library have been fabricated. However, an earlier chip (**tcr24**) was designed and fabricated that contained a normalizer using the same logarithmic normalizer idea as shown here. This circuit did not use the trick with the parallel 8-bit NOR evaluations, and the circuit design was also considerably less aggressive. The chip was fabricated in VTI 2µm CMOS and had a delay of 27ns from pad to pad. It is expected that a 1.2µm version based on the above method will perform at 10ns.

8.10 High speed 3-port SRAM (regfilew)

The **regfilew** high-speed 3-port SRAM is based on a fixed-size (32 words x 64 bits) handmade layout described in [iris92], with the following additions:

- The hand-tiled leafcells have been modified for automatic tiling (alignment, overlap boxes, labelling).
- The design has been made parameterizable.
- Additional cells for tileable Vdd, GND and clk routing to the edges of the layout.
- Construction of parameterized tiling procedure.
- Optional 2-1 column decoding.

A block diagram of regfilew is shown in Figure 8-36. The address lines are precharged high while clk=1, and the outputs A,B are valid during clk=0. The write input must be precharged high during clk=1 and then conditionally pull down before the end of clk=0. Circuit-wise, regfilew is identical to the circuits of [iris92], with the exception of the addition of column decoding. Figure 8-37 shows the storage cell and the read/write circuits, including the modification for 2-1 column decoding.

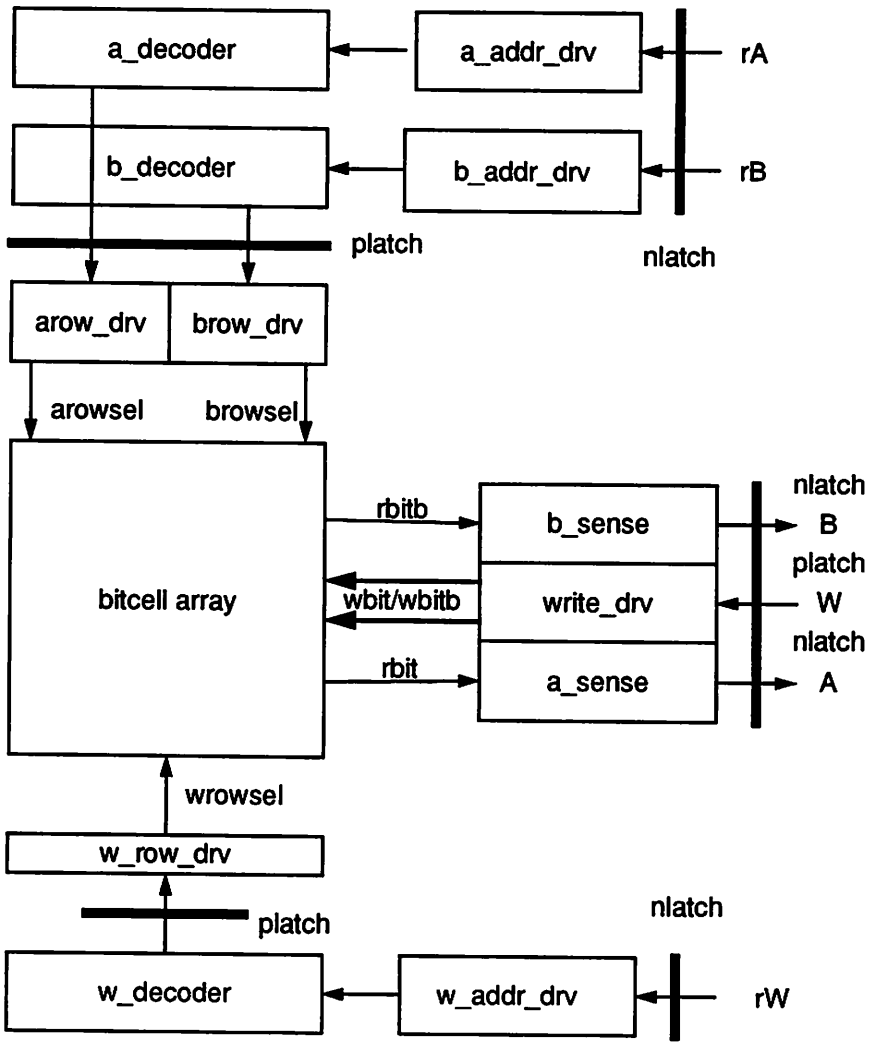


Figure 8-36 Block diagram of **regfilew**. A,B are read ports and W is the write port. The terminals rA,rB,rW are the address signals

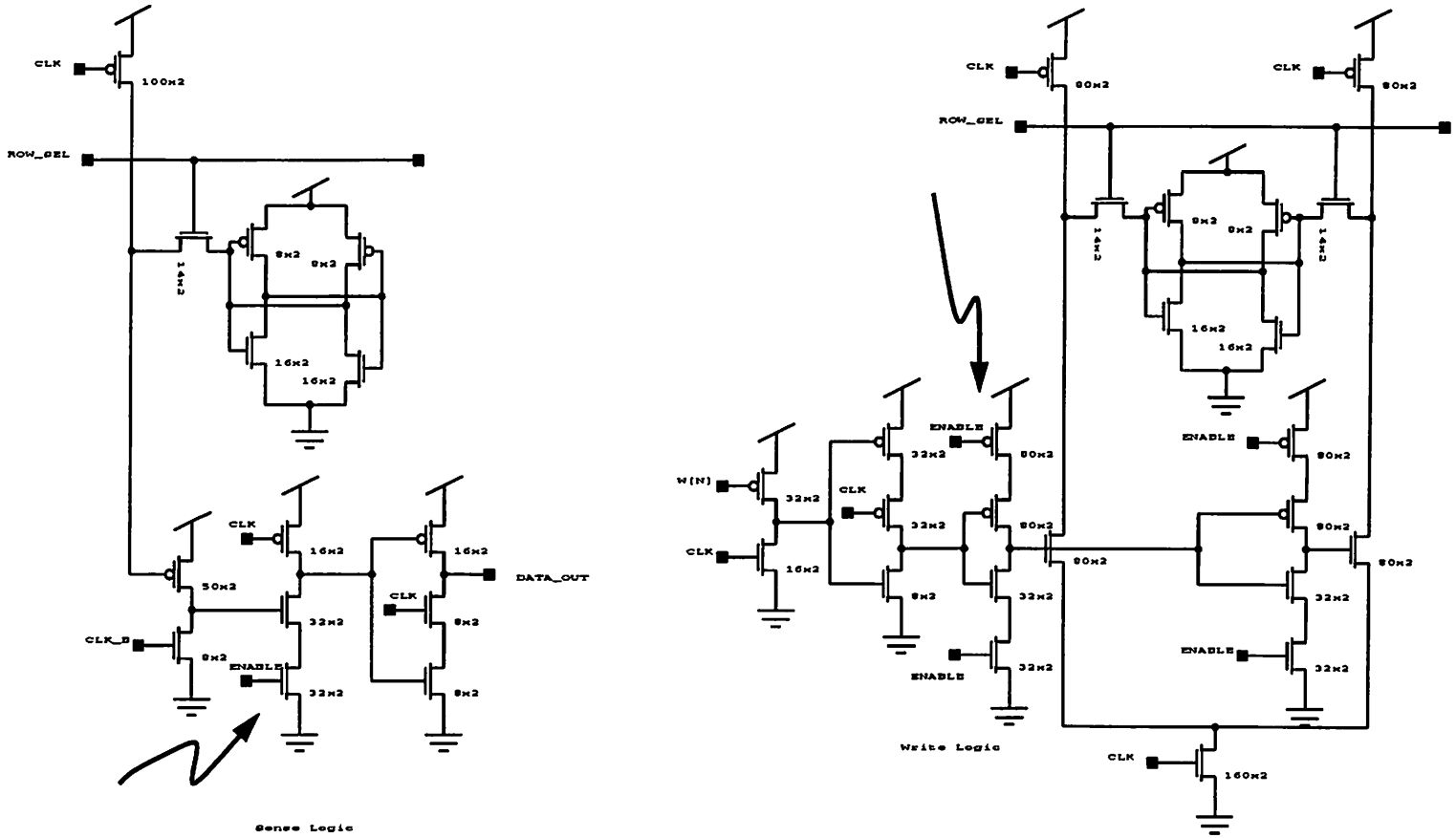


Figure 8-37 Circuit diagrams of **regfilew** (a) Storage cell and read port (b) Storage cell and write port. The transistors marked with an arrow are additions for 2-1 column decoding

Floorplan and tiling

Regfilew is constructed from 24 leafcells and 9 additional hierarchical subcells. The TimLager module generator was used for the automatic tiling. The tiling procedure consists of 558 lines of C code and uses 3 levels of tiling hierarchy. The block arrangement at the top level of a 256x32 bit version is shown in Figure 8-38.

Test chip and results

A CIF plot of the **RW256C** test chip is shown in Figure 8-39. In addition to the 256x32 regfilew instance, it contains various precharge circuits for the inputs and some additional registers and multiplexers to keep the I/O pin count down. The pads saved in this manner were allocated for

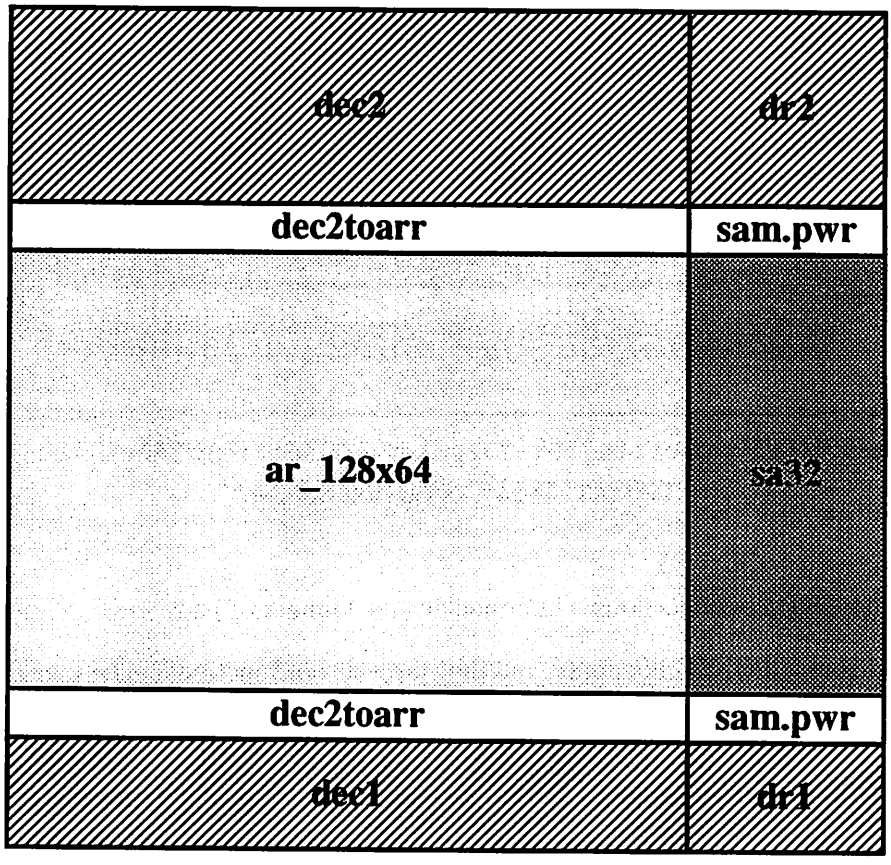


Figure 8-38 Floorplan for the top level of 256x32 **regfilew** layout, based on a 128x64 array and 2-1 column decoding

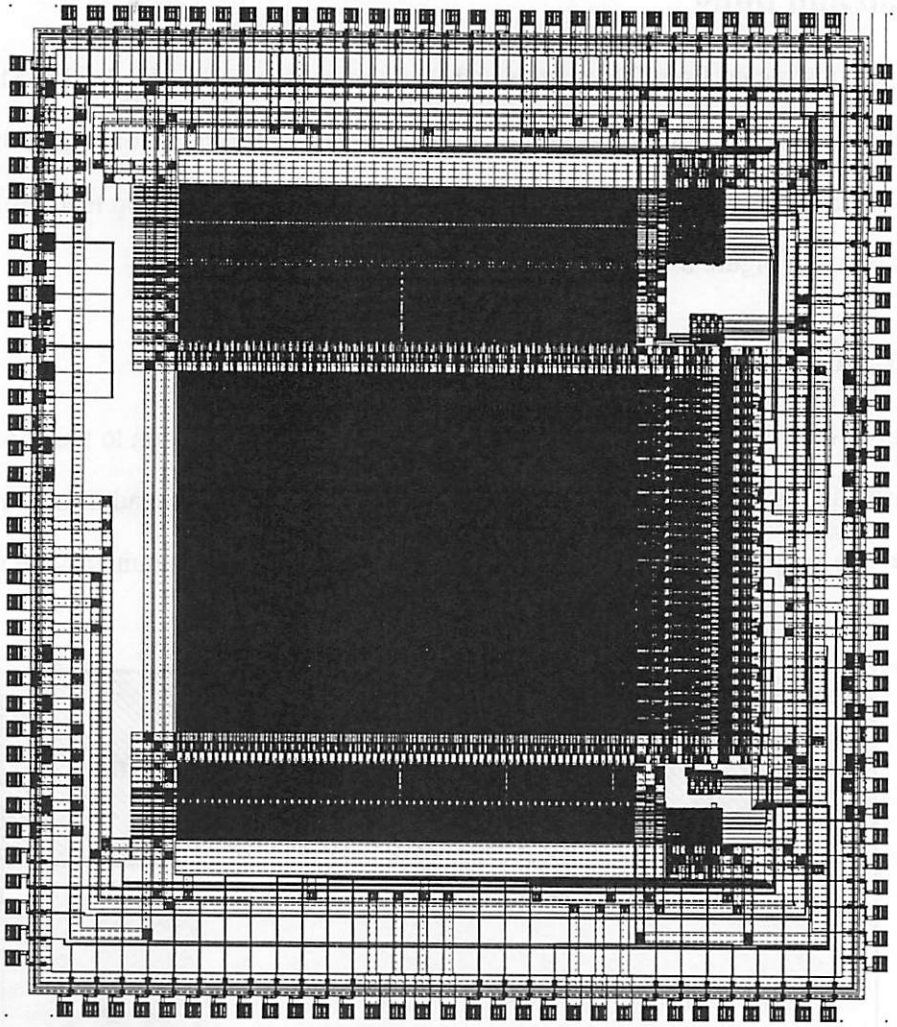


Figure 8-39 CIF plot of the 256x32 regfilew test chip (**regw256c**).The size is $10032x11482\lambda^2$, or $6019x6889\mu m^2$

additional Vdd/GND/clk connections. Unfortunately, the number of pads were still not sufficient to provide a clean Vdd/GND/clk supply, and the chip was not functional. The first chip (64x32 version) described in [iris92] had the same problem. That chip has been fabricated in a 2nd version, using the same SRAM core but with a large amount of decoupling capacitance (made using gate oxide) on the chip itself. This 2nd chip was functional at 180MHz, indicating that regfilew can indeed be expected to perform at high speeds. The IRSIM simulation of the 256x32 regfilew predicts at maximum clock speed of 170MHz.

8.11 High speed PLA (hpla)

The **hpla** high-speed PLA (Programmable Logic Array) is based on the handmade design in [iris92]. Irissou also wrote a special-purpose layout generator that generates MAGIC layout of the PLA, but in this work the TimLager generator and the OCT framework was used. The modifications made were

- Leafcells modified for automatic tiling.
- Parameterized size and contents of the PLA.
- Additional cells creates Vdd, GND and clk buses with terminals at the border of the layout.
- Construction of a parameterized TimLager tiling procedure.

The circuit schematic for hpla is shown in Figure 8-41. The inputs do not need to be precharged, but must be stable during clk=0. Outputs change during clk=1 and are stable during clk=0, just like the inputs (Figure 8-42). The latency through the PLA is 1 clock cycle, which means that the

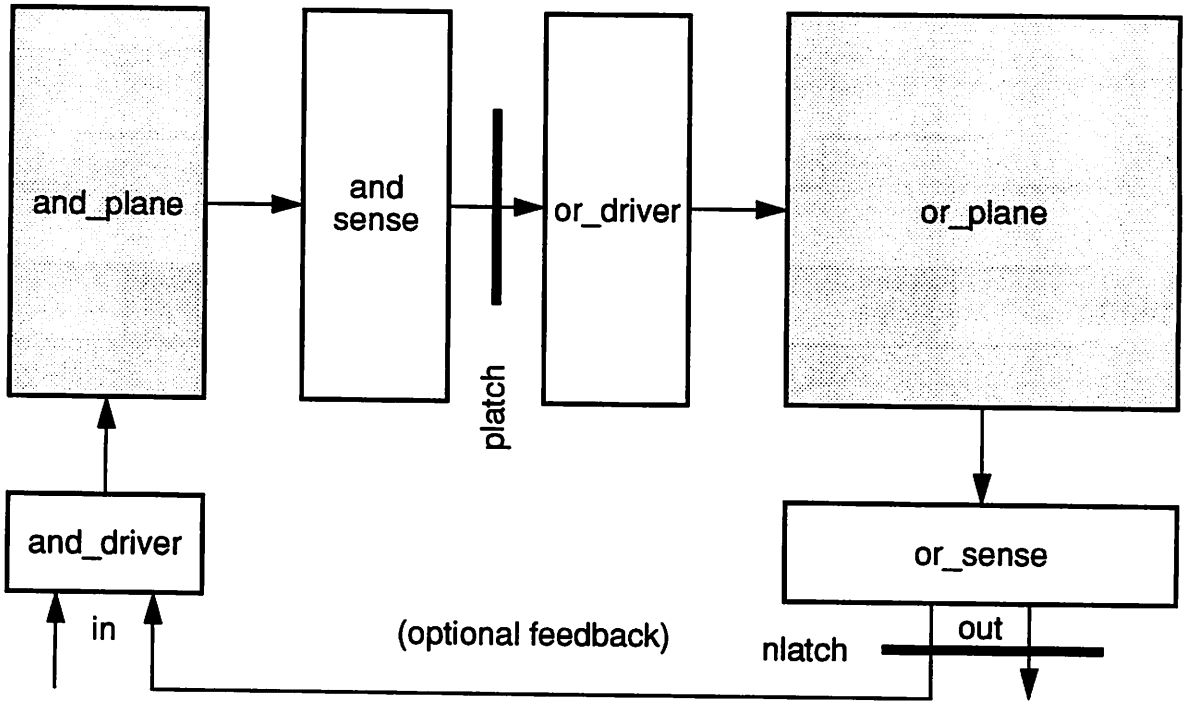


Figure 8-40 Block diagram of **hpla**

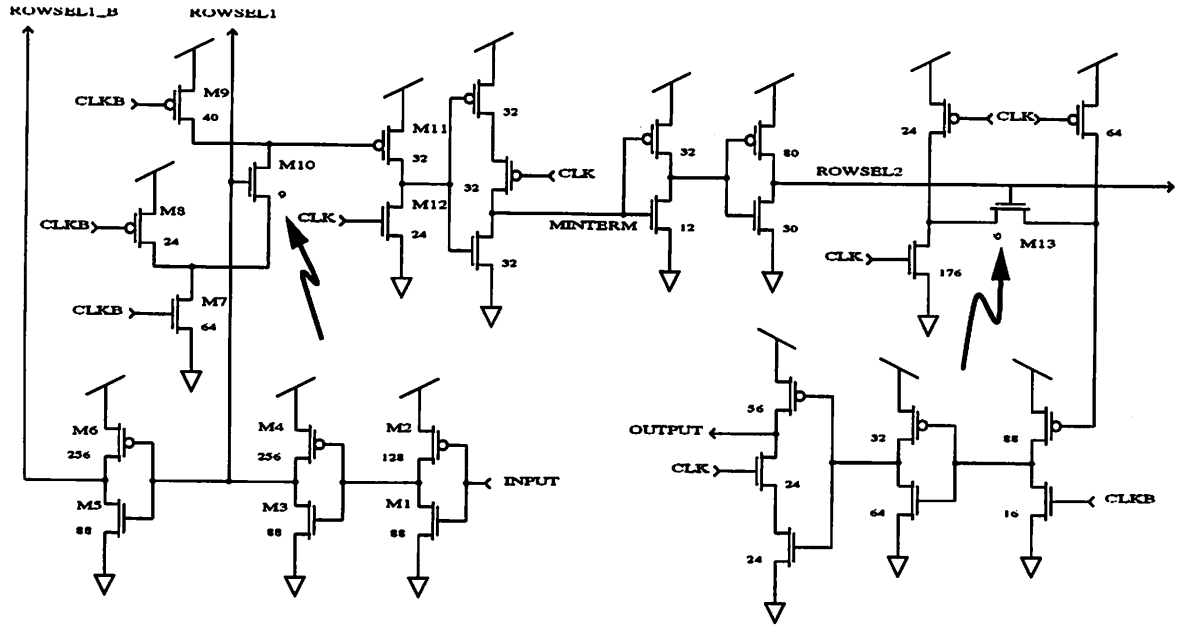


Figure 8-41 **hpla** circuit schematic. The marked transistors correspond to the minterm patterns in the inplane and the outplane

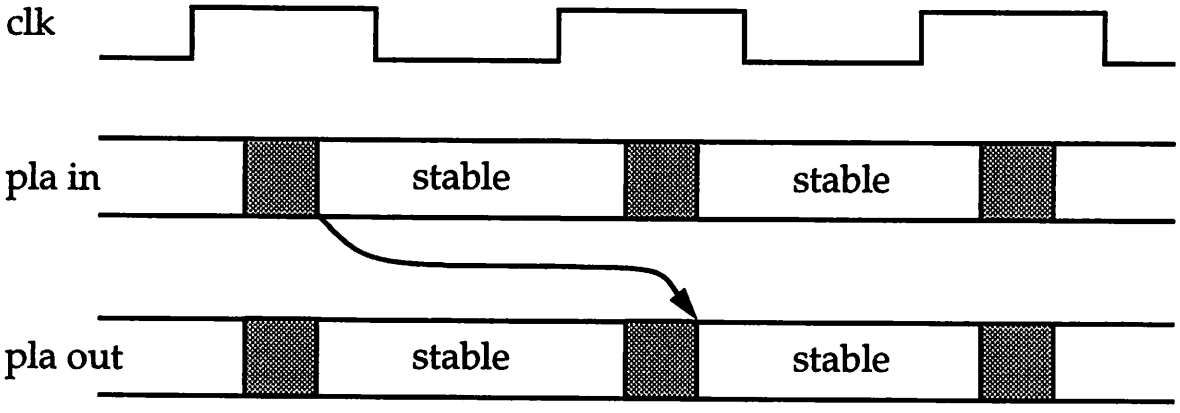


Figure 8-42 Timing and latency of **hpla**

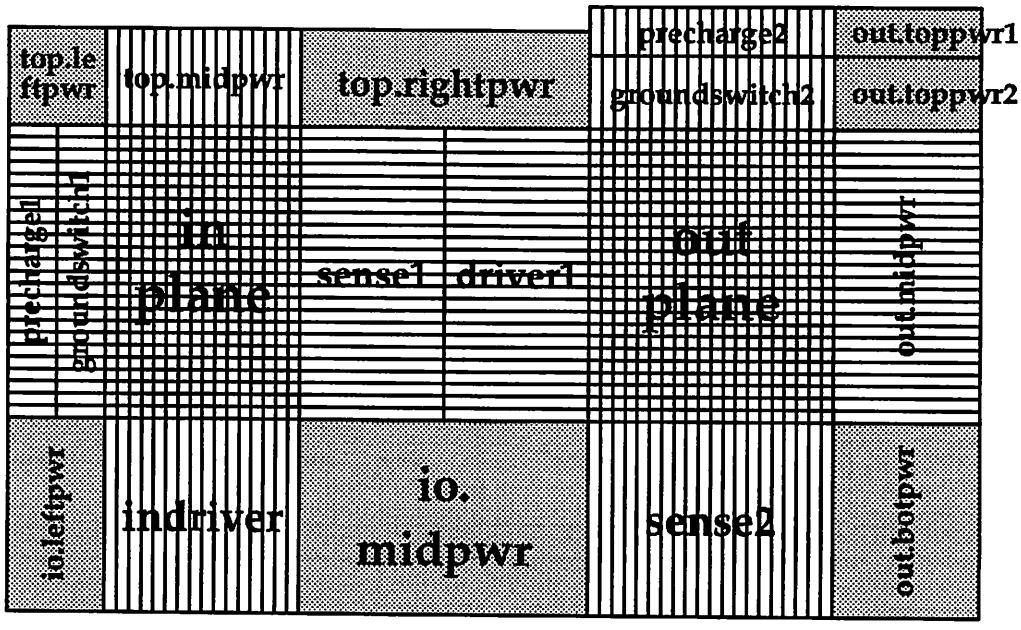


Figure 8-43 Floorplan and tiling of **hpla**. Each square denotes a leafcell

outputs can be fed straight back as inputs to create a state machine. This is very convenient for controller applications.

Floorplan and tiling

The general tiling pattern of hpla is shown in Figure 8-43. The design contains 33 leafcells, 16 of which are different variations of the basic 2x2 bitcell used in the in/outplanes. Because some of the cells in the right half of the layout do not align with cell boundaries in the left half (the *precharge2* cell), the halves were originally tiled separately and then put together at a 2nd level of hierarchy. This inconvenience (and others like it) led to the development of a new and more general set of tiling primitives for TimLager [richards92]. The new tiling primitives (Table 8-5) makes it possible to place any corner of a new cell relative to any corner of a previously placed cell. These primitives are very useful for tiling structures that do not follow the customary left-to-right, bottom-to-top scheme that TimLager was originally designed for. They also makes it possible to

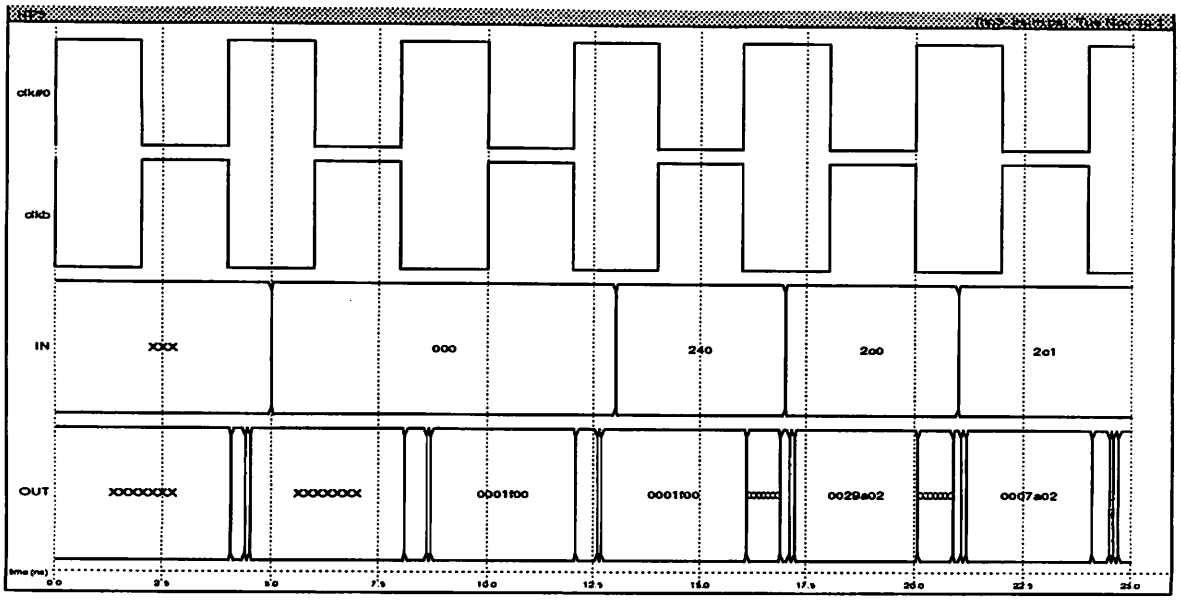


Figure 8-44 IRSIM simulation of a **hpla** design at f=250MHz.
inputs•minterms•outputs=10•50•28

tile most designs in a straightforward and intuitive manner, whereas earlier it was often necessary to rotate some subblocks to make them fit into the tiling paradigm. The tiling procedure hpla.c consists of 319 lines of C code.

Simulation

Figure 8-44 shows an IRSIM simulation of an hpla instance of size 10•50•28 at f=250MHz. There appears to be still a good margin on the clock speed.

Function name	Function semantics
Addcell()	Generalizes Addup()/Addright(). Place new cell at any corner of current bbox.
NewBound()	Push a new, 0-sized bounding box on top of a bbox stack.
MergeBounds	Merge the two bounding boxes at the top of the bbox stack.

Table 8-5 New TimLager tiling primitives used in **hpla**

Test and fabrication results

The pmult testchip (pmvt24c) was intended to serve as the testchip also for the hpla design. As mentioned earlier, the clock drivers on pmvt24c were not sufficient, and hence the chip did not provide any data on the speed. Again, the 2nd generation of the testchip from [iris92] showed that the speed is at least 180MHz for the given 10•50•28 example.

8.12 Pads and clock distribution

Special care must be taken in designing (and using) pads for high-speed chips. The pad ring for a high-speed chip must provide adequate

- Driving ability (delay, slope) for signals arriving on chip.
- Driving ability for signal going off chip.
- Vdd and GND connections, with special emphasis in handling high peak currents without excessive Vdd and GND bounce (due to pin and bonding wire inductance, mostly).
- Clock driving ability, and especially sufficient clock slope for TSPC circuits.

The pads used in the aforementioned test chips were procured from Prof. Wawrzynek's group. The output pad was designed for 100MHz signal operation, with 2ns rise/fall time into 8nH/30pF and a ground bounce of less than 0.5V. Each output pad requires a pair of Vdd/GND pads to supply it. The pads were reworked into LAGER format so that they could be used with TimLager and Padroute (section 3.11 on page 76). The members of the **pads12** library are listed in Table 8-6. There are separate Vdd/GND pads for the padring itself and the chip core, the idea being to keep the core power clean even if the pad power lines are bouncing.

The clock drivers are also based on pads, with large pullup (pulldown) transistors residing in special clock driver pads that are bonded to Vdd (GND) and then tied together onchip to form a gigantic inverter that drives all clock lines.

Name	Description	Name	Description
analog_12	Plain analog pad, no driver	gndpad_12	GND pad for padring supply
clkn_12	2nd stage clock driver pulldown (n) transistor	in2_12	Input pad with true/inverted outputs
clkndrv_12	1st stage clock driver pulldown (n) transistor	in_12	Input pad
clkp_12	2nd stage clock driver pullup (p) transistor	out_12	Output pad
clkpdrv_12	1st stage clock driver pullup (n) transistor	thru_12	Space pad for enlarging pad ring
corner_12	Corner piece of pad ring	vddcore_12	Vdd pad for chip code
gndcore_12	GND pad for chip core	vddpad_12	Vdd pad for padring supply

Table 8-6 The pads of the **pads12** family

Test results

It was found that the above pad scheme was not sufficient for the speed and current requirements in the test chips, nor the test chip described in [iris92]. Part of the reason is that the bonding wire inductance was higher than anticipated (15nH versus 8 nH), and another part was that it was prohibitively expensive to spend the number of Vdd/GND pads necessary to get reasonably high clock slopes and reasonably low power supply bounce.

For the 2nd design, [iris92] successfully used a different approach based on placing large decoupling capacitors onchip. The capacitors were made by creating large transistors with the gate tied to Vdd and the source/drain tied to GND, and fitting them underneath the power supply lines. With local decoupling, it did not make sense to use pads as clock drivers. Instead, the clock drivers were placed on the chip, still using the approach that all clock lines are driven from one central point. This method was pioneered by [dobber92].

8.13 Summary

The design of the SMAC Small MAtrix Computer architecture has been presented. The architecture contains a number of innovations aimed specifically at matrix computations, in particular

- An address generator with split row/column addresses that are combined into a physical memory address, while at the same time allowing the address components to be used directly as loop counters.
- Row pivoting based on a hardware permutation table (as opposed to swapping of row contents) eliminates pivoting overhead.
- Parallel pivot searching during the elimination steps obliterates the search time overhead.
- "Soft" pivot searching based on comparing exponents results in considerable hardware savings.

In addition to these innovations, SMAC uses a multiport memory structure especially tailored to the Gauss/LU algorithm, and it targets pipelined floating point units so that the Pipelining Margin of the algorithm can be fully applied towards increasing the throughput of the processor implementation.

On the hardware side, the most critical modules needed to implement SMAC have been designed:

- A heavily pipelined multiplier (**pmult**) with parameterized size.
- Circuits for floating point adder and multiplier datapaths, including operand normalization and renormalization, with parameterized sizes. All are in the **dpp** library design style.
- High speed datapath pipeline latches (TSPC) and multiplexers.
- A 3-port SRAM (**regfile, regfilew**) with parameterized size and decoding.
- A high-speed PLA (**hpla**) with fully parameterized contents and direct FSM capability.

All the blocks have been simulated for speeds in the 110 MHz to 170 MHz range. Test chip results reported in [iris92] confirm speeds of 180MHz for some of the blocks.

CHAPTER 9

SUMMARY AND CONCLUSION

The basic premise of this dissertation is that

- It is likely that the benefits of ASIC implementation for DSP computations can be duplicated in the area of Numerical Processing.
- The knowledge base, CAD tools, design methods and architectures developed for Application Specific Digital Signal Processor design are to some extent applicable also in the Numerical Processing domain.
- Because Numerical Processing pose different computational demands, *additional* innovations and developments need to be made in all the aforementioned areas in order to realize the gains of ASIC implementation.

The dissertation contains two main parts (Chapters 3-4 on C-to-Silicon/PUMA and Chapters 5-8 on ConsolC/SMAC) that present different approaches to Numerical Processor design. The next few sections summarize the work described in these chapters and then present the conclusions along with some directions for future investigation in the area of Application Specific Processors for Numerical Algorithms.

Both the PUMA chip and the SMAC architecture are based on a study of algorithms for the Inverse Position-Orientation (IPO) computation for robots with 6 revolute joints. IPO is used as the common thread, while at the same time providing two radically different examples of numerical computation tasks. Taking the designer's perspective, Chapter 2 presents a survey of IPO computational methods and mathematical background with special emphasis on the facts that are important to the system and chip designer.

9.1 The C-to-Silicon system and the PUMA chip

The C-to-Silicon system is a powerful design tool for Applications Specific Processors for Numerical Processing and DSP. The system supports easy architecture exploration and performance evaluation at a the architecture level, without having to perform detailed logic and layout level design. High-level algorithm simulation is also supported. C-to-Silicon uses the LAGER Silicon Assembly System to perform layout and simulation tasks, resulting in a very powerful and general system. It has been demonstrated that C-to-Silicon is flexible with respect to the range of architectures and algorithms that can be implemented.

C-to-Silicon is the result of an integration effort that pulls together an assortment of tools to form a complete design system that spans the range from the algorithm description down to mask layout and fabrication. The design goals for C-to-Silicon system were to

- Use a high-level "C" language for algorithm specification
- Allow architecture exploration without detailed hardware design
- Separate the hardware *implementation* from algorithm and hardware design
- Simplify concurrent design of hardware/architecture/software
- Eliminate machine language coding altogether
- Support simulation at all abstraction levels
- Provide accurate performance data without detailed hardware design

The successful design of the PUMA chip, as described in Chapter 4, demonstrates that the design goals have been met. PUMA is a 100,000 transistor CMOS chip that executes an algorithm described by 260 lines of "C" statements, computing all IPO solutions to the PUMA 560 industrial robot in real time.

C-to-Silicon has subsequently been applied to other chip designs, notably as part of work on analog-to-digital (A/D) converters [mmar92]. This is a DSP application where the C-to-Silicon processor performs filtering tasks related to oversampled A/D converters.

9.2 Matrix computations

Many (most, some would say) numerical algorithms can be reduced to a core of matrix computations when viewed at a detailed level. One of the reasons behind this fact is that matrix computations are both reliable and computationally tractable as long as they are properly formulated. Hence, it is a popular and powerful approach to try to reduce more general numerical problems into matrix problems, using the theoretical background of multivariable functions and linear algebra. It is therefore important to investigate matrix computations when considering application specific architectures for numerical algorithms.

The homotopy continuation method for solving systems of n polynomial equations in n unknowns is a good example. It boils down to Newton's method applied to a set of continuation paths, which in essence means that the computation consists of evaluating functions and derivatives, and solving linear systems of equations. The general Inverse-Position Orientation (IPO) computation for 6R robots can be cast in this form.

ConsolC was developed as a tool for experimenting with homotopy continuation algorithms, especially as they apply to solving the general IPO equations. Experimentation was in turn motivated by the need to obtain detailed knowledge about the numerical and structural properties of the algorithms, for the purpose of determining efficient computing architectures. It is clear that

the matrix operations are in fact the key to rapid execution of continuation algorithms, and that the numerical precision can be maintained using reasonable wordlengths. The remaining parts of the computation can easily be parallelized and executed on standard processors.

The SMAC architecture was developed in response to the need for high-speed, efficient computation of solutions to linear equations. Evaluation of a wide range of commercial architectures showed that they are not efficient for solving small linear systems, and that an order of magnitude can be gained in speed using roughly the same amount of silicon.

The speed and efficiency gains in SMAC are the result of a number of innovations aimed specifically at matrix computations, in particular

- An address generator with split row/column addresses that are combined into a physical memory address, while at the same time allowing the address components to be used directly as loop counters.
- Row pivoting based on a hardware permutation table instead (as opposed to swapping of row contents) eliminates pivoting overhead.
- Parallel pivot searching during the elimination steps obliterates the search time overhead.
- "Soft" pivot searching based on comparing exponents results in considerable hardware savings.

In addition to these innovations, SMAC uses a multiport memory structure especially tailored to the Gauss/LU algorithm, and it targets pipelined floating point units so that the *Pipelining Margin* of the algorithm can be fully applied towards increasing the throughput of the processor implementation.

On the hardware side, the most critical modules needed to implement SMAC have been designed:

- A heavily pipelined multiplier (**pmult**) with parameterized size.
- Circuits for floating point adder and multiplier datapaths, including operand normalization and renormalization, with parameterized sizes. All are in the **dpp** library design style.
- High speed datapath pipeline latches (TSPC) and multiplexers.

- A 3-port SRAM (**regfile, regfilew**) with parameterized size and decoding.
- A high-speed PLA (**hpla**) with fully parameterized contents and direct FSM capability.

Tests and simulations have demonstrated speeds in the 110MHz to 180 MHz range.

9.3 Conclusion and directions for further investigation

The main questions to be answered by this dissertation are the following: Is Numerical Processing really different from DSP? If so, can the gains produced by ASICs for DSP also be realized in the Numerical Processing application area? While these are a very broad and complex questions to answer, the results presented here affirm the differences and show that the prospects are in fact promising.

In Chapter 7, it was shown that all but the most expensive commercially available processors have considerable inefficiencies in performing matrix computations such as the Gauss/LU algorithm. If price is considered, *all* the commercial alternatives studied are inefficient. The reason for the inefficiency is indeed that the processors are optimized for other types of operations than the ones found in Numerical Processing. Some distinguishing properties of numerical algorithms were established in Chapter 6, in particular in the areas of required memory bandwidth, permutation lookups and the available pipelining margins. The investigation shows that an order of magnitude improvement can be realized (for small matrices) by utilizing improved architectures and design methods.

Such advances are not without cost, especially because the tools and techniques for Numerical Processor design are much less developed than their counterparts in the DSP arena. However, it is predictable that NP design tools and techniques can follow the successful path exemplified by DSP and develop to a level where it is just as easy to design an Application Specific Numerical Processor as it is to design an a corresponding DSP chip today.

The main challenges appear to be in the following areas:

- **Applications.** Additional applications of embedded Numerical Processing should be investigated. Some examples are listed in Chapter 1.
- **Languages.** While RL (or C) is a sufficient programming language for DSP and fixed point Numerical Processing, an architecture such as SMAC cannot easily be programmed in any of the existing high-level languages. Among the most serious challenges is the development of constructs which can efficiently express the types of addressing that is typically used in matrix computations. Languages such as dspC [ad92] and NumericC [ansi92] are examples of ongoing efforts in this area. Their applicability in the Numerical Processing arena should be investigated.
- **Tools.** With the advent of appropriate programming languages, it will become possible to extend systems such as C-to-Silicon so that they can be applied more naturally to numerical problems.
- **Architectures.** While solving linear equations is a common task, there are also other forms of matrix computations that are prevalent in Numerical Processing. Examples such as singular value decomposition (SVD), eigenvalue computations, orthogonalization and sparse matrix computations come to mind. Some of these computations may fit in well with a SMAC-like architecture, whereas others may require additional or different innovations.

ASICs for Numerical Processing require the development of new sets of languages, tools and architectures. While some of the issues have been addressed in this dissertation, the above list indicates that there are many others that require further investigation.

BIBLIOGRAPHY

[afghahi90] Morteza Afghahi and Christer Svensson. A Unified Single-phase Clocking Scheme for VLSI Systems. *IEEE Journal of Solid-State Circuits*, pages 225–233, Feb 1990.

[anna86] Marco Annaratone, Emmanuel Arnould, Thomas Gross, H T Kung, Monica S Lam, Onat Menzilcioglu, Ken Sarocky, and Jon A Webb. WARP Architecture and Implementation. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 346–356. IEEE, 1986.

[ansi92] ANSI. *Numerical C, draft X3J11.1*. American National Standards Institute, 1992.

[att88] AT&T. *WE DSP32C Digital Signal Processor Information Manual*. AT&T Documentation Management Organization, Dec 1991.

[azim88] Syed Khalid Azim. *Application of Silicon Compilation Techniques to a Robot Controller Design*. PhD thesis, UC Berkeley, May 1988. UCB/ERL memo M88/35.

[blahut85] Richard Blahut. *Fast Algorithms for Digital Signal Processing*. Addison-Wesley, 1985.

[bose88] Bidyut Kumar Bose. *VLSI Design Techniques for Floating-point Computation*. PhD thesis, UC Berkeley, December 1988. UCB/CSD report 88/469.

[canny88] John F Canny. *The complexity of Robot Motion Planning*. MIT Press, 1988.

[chen92] Deveraux C Chen. *Programmable Arithmetic Devices for High Speed Digital Signal Processing*. PhD thesis, UC Berkeley, May 1992. UCB/ERL memo M92/49.

[chen86] J Bradley Chen, Ronald S Fearing, Brian S Armstrong, and Joel W Burdick. NYMPH: A Multiprocessor for Manipulation Applications. In *IEEE*

International Conference on Robotics and Automation, pages 1731–1736, 1986.

[chow78] S N Chow, J Mallet-Paret, and J A Yorke. Finding Zeros of Maps: Homotopy Methods are Constructive with Probability One. *Mathematics of Computation*, 32:887–899, 1978.

[chu88] Chorn-Yeung Chu. *Improved Models for Switch-Level Simulation.* PhD thesis, Stanford University, 1988. CSL report TR-88-368.

[cody80] William J Cody and William Waite. *Software Manual for the Elementary Functions.* Prentice-Hall, 1980.

[craig86] John J Craig. *Introduction to Robotics.* Addison-Wesley, 1986.

[dahlquist74] Germund Dahlquist and Åke Bjorck. *Numerical Methods.* Prentice-Hall, 1974.

[dec92] DEC. *Alpha Architecture Handbook.* Digital Equipment Corporation, 1992.

[dobber92] Daniel Dobberpuhl et al. A 200MHz 64-b dual issue cmos microprocesor. *IEEE Journal of Solid-State Circuits*, pages 1555–1567, Nov 1992.

[drexler77] F J Drexler. Eine Methode zur Berechnung sämtlicher Lösungen von Polynomgleichungssystemen. *Numerischer Mathematik*, 29:45–58, 1977.

[duffy80] J Duffy and C Crane. A Displacement Analysis of the general spatial 7R mechanism. *Mechanism and Machine Theory*, pages 153–169, 1980.

[erdman90] Donald J Erdman and Donald J Rose. *CAzM, Circuit Analyzer with Macromodeling.* MCNC Center for Microelectronics, Jun 1990.

[fearing91] Ronald S Fearing and T O Binford. Using a Cylindrical Tactile Sensor for Determining Curvature. *IEEE Transactions on Robotics and Automation*, pages 806–817, Dec 1991.

[fraleigh83] John B Fraleigh. *A First Course in Abstract Algebra.* Addison-Wesley, 3d edition, 1983.

[gagli86] Robert D Gaglianello and Howard P Katseff. A Distributed Computing Environment for Robotics. In *IEEE International Conference on Robotics and Automation*, pages 1890–1896, 1986.

[garcia77] C B Garcia and W I Zangwill. Global Continuation Methods for Finding All Solutions to Polynomial Systems of Equations in N Variables. Technical report, Center for Matematical Studies in Business and Economics, Report no. 7755, University of Chicago, 1977.

[gnu90] Doug Lea. *User's Guide to the GNU C++ Library.* Free Software Foundation, 1990.

[golub83] Gene H Golub and Charles F Van Loan. *Matrix Computations.* Johns Hopkins University Press, 2nd edition, 1983.

[golub89] Gene H Golub and Charles F Van Loan. *Matrix Computations.* Johns Hopkins University Press, 3d edition, 1989.

[gupta92b] Rajesh K Gupta, Claudionor N Coelho, and Giovanni de Micheli. Synthesis and Simulation of Digital Systems Containing Interacting Hardware and Software Components. In *IEEE/SIGDA Design Automation Conference*, Jun 1992.

[gupta92a] Rajesh K Gupta and Giovanni de Micheli. System-Level Synthesis Using Re-Programable Components. In *Proc. of the European Design Automation Conference*, Mar 1992.

[harbison87] Samuel P Harbison and Guy L Steele. *C: A Reference Manual.* Prentice-Hall, 2nd edition, 1987.

[hepa90] John L Hennessy and David A Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufman Publishers, 1990.

[hoang92] Phu D Hoang. *Compiling Real-Time Digital Signal Processing Applications Onto Multiprocessor Systems.* PhD thesis, UC Berkeley, May 1992. UCB/ERL memo M92/68.

[horowitz84] Mark A Horowitz. *Timing Models for MOS Circuits.* PhD thesis, Stanford University, 1984.

[hu87] Timothy Hu. Circuit Design Techniques for a Floating-Point Processor. Master's thesis, UC Berkeley, 1987. UCB/CSD report 87/372.

[htkung78] H T Kung and C E Leiserson. Systolic Architecures for VLSI. In *Sparse Matrix Proceedings*, pages 37–46. SIAM, January 1978.

[hwang92] Kai Hwang. *Advanced Computer Architecture.* McGraw-Hill, 1992. Pre-publishing edition.

[intel92] Intel. *Multimedia and Supercomputing Data Book.* Intel Incorporated, 1992.

[iris92] Bertrand S Irissou. Design Techniques for High-Speed Datapaths. Master's thesis, UC Berkeley, Dec 1992.

[jag85] Hosagrahar V Jagadish. *Techniques for the Design of Parallel and Pipelined VLSI Systems for Numerical Computation.* PhD thesis, Stanford University, 1985.

[jain91] Rajeev Jain, Paul T Yang, and T Yoshino. Firgen - a computer-aided design system for high performance fir filter integrated circuits. *IEEE Transactions on Signal Processing*, Jul 1991.

[jassica85] J R Jassica, S Noujaim, R Hartley, and M J Hartman. A Bit-Serial Silicon Compiler. In *Proceedings of ICCD*, Oct 1985.

[kane92] Gerry Kane and Joe Heinrich. *MIPS Risc Architecture.* Prentice-Hall, 1992.

[kernighan78] Brian W Kergnighan and Dennis M Ritchie. *The C Programming Language.* Prentice-Hall, 1978.

[le92] Dinh Le, Milos Ercegovac, Tomas Lang, and Jaime Moreno. MAMACG: A Tool for Automatic Mapping of Matrix Algorithms Onto Mesh Array Computational Graphs. In *IEEE International Conference on Application Specific Array Processors*, pages 511–525, Oct 1992.

[lee86] Edward A Lee. *A Coupled Hardware and Software Architecture for Programmable Digital Signal Processors.* PhD thesis, UC Berkeley, December 1986.

[lettang89] Erik Lettang. Padroute: A Tool for Routing the Bonding Pads of Integrated

Circuits. Master's thesis, UC Berkeley, 1989.

[luenberger84] David G. Luenberger. *Linear and Nonlinear Programming.* Addison-Wesley, 1984.

[mac83] The Mathlab Group. *MACSYMA Reference Manual.* Laboratory for Computer Science, MIT, 10 edition, January 1983.

[manocha92] Dinesh Manocha. *Algebraic and Numeric Techniques for Modeling and Robotics.* PhD thesis, Computer Science Division, Department of Electrical Engineering and Computer Science, University of California, Berkeley, May 1992.

[mmar92] Monte Mar. *Automated Design of Signal Acquisition Modules.* PhD thesis, UC Berkeley, 1992. In preparation.

[micheli90] G De Micheli, D Ku, F Mailhot, and T Truong. The olympus synthesis system. *IEEE Design and Test of Computers*, Oct 1990.

[moreno90] Jaime N Moreno and Tomas Lang. Matrix computations on systolic-type meshes. *IEEE Computer Magazine*, pages 32–51, April 1990.

[morgan86] Alexander P Morgan. A Homotopy for Solving General Polynomial Systems that Respect M-homogenous Strucures. Research publication GMR-5437, General Motors Research Laboratories, Warren, Michigan 48090, May 1986.

[morgan87a] Alexander P Morgan. *Solving Polynomial Systems Using Continuation for Engineering and Scientific Problems.* Prentice-Hall, 1987.

[morgan87b] Alexander P Morgan and Andrew Sommese. Computing All Solutions to Polynomial Systems Using Homotopy Continuation. Research publication GMR-5692, General Motors Research Laboratories, Warren, Michigan 48090, January 1987.

[mot89] Motorola. *DSP96002 IEEE Floating-Point Dual-Port Processor User's Manual.* Motorola Inc, 1988.

[mot90] Motorola. *MC88100 RISC Microprocesor User's Manual.* Motorola Incorporated, 2nd edition, 1990.

[nara86] Sundar Narashiman, David Siegel, and John M Hollerbach. Implementation of Control Methodologies on the Computational Architecture for the Utah/MIT Hand. In *IEEE International Conference on Robotics and Automation*, pages 1884–1889, 1986.

[okamoto91] Fuyuki Okamoto et al. A 200-MFlops 100-MHz 64-b BiCMOS Vector-Pipelined Processor (VPP) ULSI. *IEEE Journal of Solid-State Circuits*, pages 1555–1567, Dec 1991.

[otten82] Ralph Otten. Automatic Floorplan Design. In *IEEE/SIGDA Design Automation Conference*, pages 261–267, Oct 1982.

[pati88] Y C Pati et al. Neural Networks for Tactile Perception. In *IEEE International Conference on Robotics and Automation*, 1988.

[pieper68] D Pieper. *The Kinematics of Mechanisms Under Computer Control.* PhD thesis, Stanford University, 1968.

[pope84] Stephen S Pope. *Automated Generation of Signal Processing Circuits.* PhD thesis, UC Berkeley, 1984.

[primrose86] E J F Primrose. On the input-output equation of the general 7r mechanism. *Mechanism and Machine Theory*, 21:509–510, 1986. This paper shows there are at most 16 solutions.

[ptolemy91] Electronics Research Laboratory. *Almagest: Ptolemy User's Manual.* UC Berkeley, 1991.

[rabaey91] J Rabaey, C Chu, P Hoang, and M Potkonjak. Fast prototyping of datapath-intensive architectures. *IEEE Design and Test of Computers*, June 1991.

[rabaey88] Jan Rabaey, Hugo De Man, Joos Vanhoof, Gert Goossens, and Francky Catthoor. Cathedral-II: A synthesis system for multiprocessor DSP systems. In Daniel D Gajski, editor, *Silicon Compilation*, pages 311–360. Addison-Wesley, 1988.

[rabaey85] Jan Rabaey, Stephen Pope, and Robert W Brodersen. An Integrated Automatic Layout Generation System for DSP Circuits. *IEEE Transactions on CAD*, pages 285–296, July 1985.

[rao85] Sailesh K Rao. *Regular Iterative Algorithms and Their Implementation on Processor Arrays.* PhD thesis, Stanford University, 1985.

[rao88] Sailesh K Rao and Thomas Kailath. Regular Iterative Algorithms and their Implementation on Processor Arrays. *Proceedings of the IEEE*, pages 259–269, March 1988.

[rb92] Robert W Brodersen, editor. *Anatomy of a Silicon Compiler.* Kluwer Academic Publishers, 1992.

[richards92] Brian C Richards. Generalized Tiling Primitives for TimLager. Personal communication, Jan 1992.

[rimey89] Kenneth Edward Rimey. *A compiler for Application-Specific Signal Processors.* PhD thesis, UC Berkeley, September 1989. UCB/CSD report 90/556.

[roy88] V P Roychowdury and T Kailath. Regular Processor Arrays for Matrix Algorithms with Pivoting. In *IEEE International Conference on Systolic Arrays*, pages 237–245, January 1988.

[roy89] V P Roychowdury and T Kailath. Regular Processor Arrays for Matrix Pivoting Algorithms. *Communications of the ACM*, 1989.

[ruetz86] Peter A Ruetz. *Architectures and design techniques for real-time image processing IC's.* PhD thesis, UC Berkeley, May 1986. UCB/ERL memo M86/37.

[salz90] Arturo Salz. *Irsim manual.* Stanford University, 1990.

[shung88] Chuen-Shen Shung. *An Integrated CAD System for Algorithm-Specific IC Design.* PhD thesis, UC Berkeley, May 1988.

[shung89] Chuen-Shen Shung et al. An Integrated CAD System for Algorithm-Specific IC Design. In *Proceedings of the 22nd Hawaii International Conference on System Science*, pages 82–91, Jan 1989.

[shung91] Chuen-Shen Shung et al. An Integrated CAD System for Algorithm-Specific IC

Design. *IEEE Transactions on CAD*, pages 447–463, April 1991.

[sparc92] SUN Microsystems. The SuperSparc Microprocessor. Technical White Paper, May 1992.

[spectrum92] Glen Zorpette (Editor). Special issue on supercomputing. *IEEE Spectrum Magazine*, pages 26–76, September 1992. See especially article by Cybenko and Kuck, p40.

[spickelmier90] Rick L Spickelmier (Editor). *Oct Tools distribution 4.0*. UC Berkeley, 1990.

[sriva92] Mani B Srivastava. *Rapid-Prototyping of Hardware and Software in a Unified Framework*. PhD thesis, UC Berkeley, May 1992.

[strang80] Gilbert Strang. *Linear Algebra and its Applications*. Academic Press, 2nd edition, 1980.

[svensson90] Lars G Svensson. *Implementation aspects of decision-feedback equalizers for digital mobile telephones*. PhD thesis, Lund Instutute of Technology, June 1990.

[terman83] Chris J Terman. *Simulation Tools for Digital LSI Design*. PhD thesis, MIT, September 1983.

[thon92] Lars E Thon and Robert W Brodersen. C-to-Silicon Compilation. In *Proceedings of the IEEE Custom Integrated Circuits Conference*, May 1992.

[thor88] CAD Group, Stanford University. *Thor tutorial*, 1988.

[ti88] Texas Instruments. *Third-Generation TMS320 User's Guide*. Texas Instruments Incorporated, 1988.

[tsai84] Lung-Wen Tsai and Alexander P Morgan. Solving the Kinematics of the Most General Six- and Five-Degree-of-Freedom Manipulators by Continuation Methods. Research publication GMR-4631, General Motors Research Laboratories, Warren, Michigan 48090, October 1984.

[volder59] J E Volder. The CORDIC Trigonometric Computing Technique. *IRE Transactions on Electronic Computers*, pages 330–334, 1959.

[walther71] J S Walther. A unified algorith for elementary functions. In *Proceedings of the 1971 Spring Joint Computer Conference*, pages 379–385. IEEE, IEEE, 1971.

[wampler89] Charles Wampler and Alexander Morgan. Solving the 6R Inverse Position Problem Using a Generic-case Solution Methodology. Research publication GMR-6702, General Motors Research Laboratories, Warren, Michigan 48090, January 1989.

[whitcomb92] Gregg Whitcomb. *BLIS Reference Manual*. EECS Department, University of California at Berkeley, 1992.

[yuan87] Jiren Yuan, Ingemar Karlsson, and Christer Svensson. A True Single-Phase Clock Dynamic CMOS Circuit Technique. *IEEE Journal of Solid-State Circuits*, pages 899–901, Oct 1987.

[yuan89] Jiren Yuan and Christer Svensson. High-speed CMOS Circuit Technique. *IEEE Journal of Solid-State Circuits*, pages 62–70, Feb 1989.

APPENDIX A

puma.k CODE

```
#File puma3.k
/*********************************************************************
Name     : puma.k
Purpose  : Inverse kinematics for Puma robot
Author   : Lars E. Thon. Copyright (c) 1987-1989. All rights reserved
*********************************************************************/

#pragma word_length     20
#pragma mult_subroutine
#pragma r_capacity 2
#pragma x_capacity 3

#include "const2.k"
#include "puma.h"
#include "common.k"
#include "indat.k"
#include "outdat.k"
#include "catan2.k"
#include "csin.k"
#include "croot.k"
#include "closed3.k"

loop() {
    indat();
    closed();
    outdat();
}
```

```
init() {

}

#File const2.k
/*********************************************************************
Name     : const.h
Purpose  : puma constants
Author   : Lars E. Thon. Copyright (c) 1987-1989. All rights reserved
*********************************************************************/

#define ONE 0.99999
#define M20 0x000fffff

/* IO addresses/control codes */
#define IO_READCOORD    0
#define IO_WRITEANGL    0

/* Powers of two */
#define toto11 2048
#define toto19 524288
#define toto20 1048576
#define toto22 4194304

/* Trigonometric constants */
#define PI              3.14159265358979323844
#define PIHALF          1.57079632679489661922
#define PIQUART         0.78539816339744830961
#define FIXPI           (0.999999)
#define FIXPIHALF       (0.50)
#define FIXPIQUART      (0.25)

/* Conversion constants */
#define M2DEG   180.0
#define M2L     (double)toto11
#define M2L2    (double)toto22

/* Cordic constants */
#define NUMIT           15                      /* 0:15 or 1:15 */
#define CROOT_AMPFACTOR 0.82978162026770026000 /* croot: mode=-1, k=1:15
*/
#define CSIN_AMPFACTOR  1.64676025786545480000 /* csin : mode=+1, k=0:15
*/
#define CSIN_STARTVALUE (1/CSIN_AMPFACTOR)

const fix ctable[17] = {                        /* For all */
    45.00000000000000000000/180,
    26.56505117707799000000/180,
    14.03624346792647900000/180,
    7.12501634890179770000/180,
    3.57633437499735110000/180,
```

```
    1.78991060824606940000/180,
    0.89517371021107439000/180,
    0.44761417086055311000/180,
    0.22381050036853808000/180,
    0.11190567706620690000/180,
    0.05595289189380367500/180,
    0.02797645261700367600/180,
    0.01398822714226501600/180,
    0.00699411367535291910/180,
    0.00349705685070401130/180,
    0.00174852842698044950/180,
    0.00087426421369378026/180
};

#File puma.h
/*********************************************************************
Name     : puma.h
Purpose  : puma constants
Author   : Lars E. Thon. Copyright (c) 1987-1989. All rights reserved
*********************************************************************/

/* Puma constants */
#define a2 431.8
#define a3 20.32
#define d3 124.46
#define d4 431.8

#define aa2 (a2/toto11)
#define aa3 (a3/toto11)
#define dd3 (d3/toto11)
#define dd4 (d4/toto11)

#define a2s (a2*a2/toto22)
#define a3s (a3*a3/toto22)
#define d3s (d3*d3/toto22)
#define d4s (d4*d4/toto22)

#File common.k
/*********************************************************************
Name     : common.k
Purpose  : Global variables etc for inverse kinematics program
Author   : Lars E. Thon. Copyright (c) 1987-1989. All rights reserved
*********************************************************************/

/*Global variables*/
bool    singular3, singular5, tooclose, outside;
fix     goal[12], tetamatrix[48];

/*More readable names for the input variables (matrix entries)*/
#define r11     goal[0]
#define r12     goal[1]
#define r13     goal[2]
```

```
#define px      goal[3]
#define r21     goal[4]
#define r22     goal[5]
#define r23     goal[6]
#define py      goal[7]
#define r31     goal[8]
#define r32     goal[9]
#define r33     goal[10]
#define pz      goal[11]

#File indat.k
/*********************************************************************
Name    : indat.k
Purpose : subroutine for data input to inverse kinematics chip
Author  : Lars E. Thon. Copyright (c) 1987-1989. All rights reserved
*********************************************************************/

/*
  The outside source must set the input pin source_ready to indicate
  it is ready   to provide data (cartesian coordinates for the robot)
*/
const volatile bool     source_ready;

indat ()
{
#ifdef KT
    /* If this is only a simulation we cannot access a chip pin ... */
    source_ready=1;
#endif

    /*busy waiting for input*/
    while (!source_ready);

    /*
      Read in 12 numbers. The external source must watch the READSTRB pin
      and also set source_ready back to 0 when it is empty (no data).
      The numbers are the consecutive _rows_ of the coordinate matrix.
    */

    r11= in(IO_READCOORD);
    r12= in(IO_READCOORD);
    r13= in(IO_READCOORD);
    px = in(IO_READCOORD);

    r21= in(IO_READCOORD);
    r22= in(IO_READCOORD);
    r23= in(IO_READCOORD);
    py = in(IO_READCOORD);

    r31= in(IO_READCOORD);
    r32= in(IO_READCOORD);
    r33= in(IO_READCOORD);
```

```
    pz = in(IO_READCOORD);

}

#File outdat.k
/*********************************************************************
Name     : outdat.k
Purpose  : subroutine for data output from inverse kinematics chip
Author   : Lars E. Thon. Copyright (c) 1987-1989. All rights reserved
*********************************************************************/

/*
  The outside destination must set the input pin dest_ready to indicate
  it is ready to accept the results (joint angles for the robot)
*/
const volatile bool     dest_ready;

outdat () {
    register int        i;

#ifdef KT
    /* If this is only a simulation we cannot access a chip pin ... */
    dest_ready=1;
#endif
    /*busy waiting for the data destination to accept*/
    while (!dest_ready);

    /*
      Transfer the 8x6 matrix in row order. The external destination
      must watch the WRITESTRB pin and also set dest_ready back to 0
      unless it is immediatley for another batch of results.
    */

#ifndef KT
    for (i=0; i < 48; i++) out (tetamatrix[i], IO_WRITEANGL);
#endif

#if defined(KT_FLOAT) && defined (SIMULATE)
    {
        int i,k;
        for (i=0; i<8; i++) {
            for (k=0; k<6; k++)
            printf("%8.2lf", tetamatrix[6*i+k]*M2DEG);
            printf("\n");
        }
    }
#endif

#if defined(KT_FIX) && defined (SIMULATE)
    {
        int i,k;
        for (i=0; i<8; i++) {
```

```
            for (k=0; k<6; k++)
            printf("%8.2lf ", tetamatrix[6*i+k]/(float)toto19*M2DEG);
            printf("\n");
        }
    }
#endif
}

#File catan2.k
/*********************************************************************
Name     : catan2.k
Purpose  : 2-argument arctangent by Cordic method
Author   : Lars E. Thon. Copyright (c) 1987-1989. All rights reserved
*********************************************************************/

fix     catan2  (yarg, xarg)
     fix        yarg, xarg;
{
    register int        k;
    register fix        x, y;
    fix                 theta;

    /* Start Cordic. The first step takes care of quadrants 2 and 3 */
    if (xarg < 0) {
        if (yarg >= 0) {
            theta = FIXPIHALF;
            x = yarg;
            y = -xarg;
        } else {
            theta = -FIXPIHALF;
            x = -yarg;
            y = xarg;
        }
    } else {
        theta = 0;
        x = xarg;
        y = yarg;
    }

    /* Scale x and y down so that they don't overflow when amplified */
    x= (x>>1); y= (y>>1);

    /*The Cordic iterations work in quadrants 1 and 4*/
    for (k = 0; k <= NUMIT; k++) {
        fix     xnew, ynew;

        if (y > 0) {
            theta += ctable[k];
            xnew = x + (y >> k);
            ynew = y - (x >> k);
            x = xnew; y = ynew;
        } else {
```

```
            theta -= ctable[k];
            xnew = x - (y >> k);
            ynew = y + (x >> k);
            x = xnew; y = ynew;
        }
    }
    return theta;
}

#File csin.k
/*********************************************************************
Name    : csin.k
Purpose : sin/cos by Cordic method
Author  : Lars E. Thon. Copyright (c) 1987-1989. All rights reserved
*********************************************************************/

csin    (sinpt, cospt, theta)
     fix        *sinpt, *cospt, theta;
{
    register fix        x, y;
    register int        k;
    bool                quad2, quad3;

    /*
      Angles in quadrants 2 and 3 are mapped into their complementary
      angles in quadrants 1 and 4. Must remember that cos/sin turns
      into sin/cos with appropriate change of sign. This is fixed at the
end.
    */
    quad2 = (theta > FIXPIHALF);
    quad3 = (theta < -FIXPIHALF);
    if (quad2)
    theta -= FIXPIHALF;
    else if (quad3)
    theta += FIXPIHALF;

    /* Assign correct starting values */
    /* Scale down to avoid intermediate result overflow */
    x = CSIN_STARTVALUE/2;
    y = CSIN_STARTVALUE/2;

    /*The Cordic iterations work in quadrants 1 and 4*/
    for (k = 0; k <= NUMIT; k++) {
        fix     xnew, ynew;

        if (theta > 0) {
            theta -= ctable[k];
            xnew = x + (y >> k);
            ynew = y - (x >> k);
            x = xnew; y = ynew;
        } else {
            theta += ctable[k];
```

```
            xnew = x - (y >> k);
            ynew = y + (x >> k);
            x = xnew; y = ynew;
        }
    }

    /*
      The prescaling (to avoid overflow) cancels this operation
      x /= 2; y /= 2;
    */

    /*Corrections for 2-3 quadrant*/
    if (quad2) {
        *cospt = -x + y;        /* -sin */
        *sinpt = x + y;         /* cos */
    } else if (quad3) {
        *cospt = x - y;         /* sin */
        *sinpt = -x - y;        /* -cos */
    } else {
        *cospt = x + y;         /* cos */
        *sinpt = x - y;         /* sin */
    }
}

#File croot.k
/*********************************************************************
Name    : croot.k
Purpose : square root by Cordic method
Author  : Lars E. Thon. Copyright (c) 1987-1989. All rights reserved
*********************************************************************/

/*
  Normal convergence is guaranteed for 0.03 < w < 2.42
  However, this is a fixed-point routine that only allows w < 1
  Some automatic scaling is necessary to avoid intermediate overflow
  (for large arguments) and marginal precision (for small arguments).

  OUTPUT:

  When    w <= 0.00: return(0)
  0.75 <= w <  1.00: Prescale by 1/4 and postscale by 2  (avoid overflow)
  0.03 <= w <  0.75: No scaling
  0.0075 <  w <  0.03: Prescale by 16 and postscale by 1/4 (improve
accuracy)
  0.00 <  w <  0.0075: Prescale by 64 and postscale by 1/16(improve
accuracy)

  The idea of prescaling and postscaling is simple; see Walther (p382)
  The maximum error is about a factor of 2, occuring for w = epsilon
*/

fix     croot   (w)
```

```
     fix        w;
{
    register fix        x, y;
    register int        k;
    bool                smallflag1, smallflag2;
    bool                bigflag;

    /*Scaling to increase precision for small arguments, and to avoid
      overflow for large arguments */

    if (w<=0) return(0);

    smallflag1= (w < 0.0075);
    smallflag2= (w < 0.03);
    bigflag   = (w > 0.74);
    if (smallflag1) w= w<<6; else if (smallflag2) w= w<<4;
    else if (bigflag) w= w>>2;

    /*Generate starting values*/
    x = w + (1/4);
    y = w - (1/4);

    /*Cordic iterations*/
    for (k= 1; k <= NUMIT; k++) {
        fix     xnew, ynew;
        if (y > 0) {
            xnew = x - (y >> k);
            ynew = y - (x >> k);
            x = xnew; y = ynew;
        } else {
            xnew = x + (y >> k);
            ynew = y + (x >> k);
            x = xnew; y = ynew;
        }
    }

    /*Postscaling*/
    if (smallflag1) x= x>>3; else if (smallflag2) x= x>>2;
    else if (bigflag) x= x<<1;

    return (x/CROOT_AMPFACTOR);
}

#File closed3.k
/*********************************************************************
Name    : closed.k
Purpose : subroutine for inverse kinematics
Author  : Lars E. Thon. Copyright (c) 1987-1989. All rights reserved
*********************************************************************/

#define EPS1 0.0005
#define EPS2 0.000005 /* Will detect (1/2)*s5^2 <0.000005 ie. t5 < 0.2 deg
```

```
*/
#undef KT_FIX_DEBUG

int     i,j,k,ind2,ind2b,ind3,ind4;
fix     teta1[2],teta2[4],teta3[2],teta23;
fix     teta4[8],teta5[8],teta6[8];
fix     c1,c23,c3,c4,c5,c6;
fix     s1,s23,s3,s4,s5,s6;

fix     px2,py2,pz2;
fix     h1,h2,h3,h3b,h4,h5,h6,h7,h8,K1,K2;
fix     q1,q2,q3,y23,x23,s5s,c4s5,s4s5;
fix     u1,u2,u3,u4,u5,u6;
fix     v1,v2,v3,v4,tmp;
fix     distance;

closed ()
{

    /*Some useful values*/
    px2= px*px;
    py2= py*py;
    pz2= pz*pz;

    /*Two solutions for teta1*/

    h1= catan2(py,px);
    h2= px2 + py2 - d3s;
    /* tooclose = (h2<0); */
    /* singular3= (h2<1/1024); */
    h3= croot(h2);

#if defined(KT_FIX) && defined (DEBUG)
    printf("#px = %8d (int) %8x (hex) %10.21f (mm )\n",
           px,px,px*M2L/toto19);
    /* More of the same is left out here */
#endif

    h3b= catan2(dd3,h3);
    teta1[0]= h1 - h3b;

#if defined(KT_FIX) && defined (DEBUG)
printf("#h3b  = %8d (int) %8x (hex) %10.21f (deg)\n",
       h3b,h3b,h3b*M2DEG/toto19);
printf("#t1[0]= %8d (int) %8x (hex) %10.21f (deg)\n",
       teta1[0], teta1[0], teta1[0]*M2DEG/toto19);
#endif

    h3b= catan2(dd3,-h3);
    teta1[1]= h1 - h3b;

#if defined(KT_FIX) && defined (DEBUG)
```

```
printf("#h3b  = %8d (int) %8x (hex) %10.21f (deg)\n",
       h3b,h3b,h3b*M2DEG/toto19);
printf("#t1[1]= %8d (int) %8x (hex) %10.21f (deg)\n",
       teta1[1], teta1[1], teta1[1]*M2DEG/toto19);
#endif

    /*Two solutions for teta3*/

    /*This value should really be computed once and for all in init()*/
    h4= catan2(aa3,dd4);
    h5= px2+py2+pz2-a2s-a3s-d3s-d4s;
    K1 = h5/2/aa2;
    K2= K1*K1;
    h6= a3s+d4s-K2;
    /* outside  = (h6<0); */
    /* singular3= (h6<1/1024); */

    h7= croot(h6);

    h8= catan2(K1,h7);
    teta3[0]= h4 - h8;
    h8= catan2(K1,-h7);
    teta3[1]= h4 - h8;

    /*
      Main loop. Each iteration computes a set of solutions.
      Four solns for teta23 => four solns for teta2,teta4,teta5,teta6.
      Later increase to eight solutions for teta4,teta5,teta6.
    */
    for (i= 0; i <= 1; i++) {
        csin (&s1, &c1, teta1[i]);
        for (j= 0; j <= 1; j++) {
            csin (&s3, &c3, teta3[j]);

            q1= -aa3 -aa2*c3;
            q2= c1*px+s1*py;
            q3= dd4 -aa2*s3;

            y23=  q1*pz -q2*q3;
            x23= -q3*pz -q1*q2;

            /*
              Certain array indices are used extensively. We compute
              them here for use in the entire loop:
            */
            ind2 = 2*i+j;
            ind2b= 2*ind2;
            teta23= catan2(y23,x23);
            csin(&s23,&c23,teta23);

            teta2[ind2]= teta23 -teta3[j];
```

```
/*
  Four solutions for teta4. No overflow problems with
  unit variables such as sin and cos because they will
  automatically be correct modulo 2 (hah!)
*/
/*Squeeze out some more common subexpressions??*/
c4s5= - r13*c1*c23 - r23*s1*c23 + r33*s23;
s4s5= - r13*s1     + r23*c1;

/* Overflow hazard because of inaccuracy,
   whenever teta5~=90deg. Hence shift down */
s5s= (c4s5*c4s5>>1) + (s4s5*s4s5>>1);

singular5= (s5s < EPS2);
if (singular5) {
    teta4[ind2b+0]= 0;
} else {
    teta4[ind2b+0]= catan2(s4s5,c4s5);
}
csin (&s4, &c4, teta4[ind2b+0]);

/*Four solns for teta5*/
u1= c1*c23*c4+s1*s4;
u2= s1*c23*c4-c1*s4;
u3= s23*c4;
u4= c1*s23;
u5= s1*s23;
u6= r33*c23;

/* To avoid inaccuracy-induced overflow if c5 or s5 are
  close to 1 in magnitude */
u1>>=1; u2>>=1;     u3>>=1; u4>>=1; u5>>=1; u6>>=1;

s5= -r13*u1 -r23*u2 +r33*u3;
c5= -r13*u4 -r23*u5 -u6;

/* Since s5,c5 are used elsewhere we need to
  scale them back up again, possibly with saturation
  We could get more fancy and recompute them if there
  was no danger of saturation */

if(s5<=-0.5) s5=-1;else if(s5>=0.5) s5=ONE;else s5<<=1;
if(c5<=-0.5) c5=-1;else if(c5>=0.5) c5=ONE;else c5<<=1;
teta5[ind2b+0]= catan2(s5,c5);

/*Four solns for teta6.*/
v1= c1*c23*s4-s1*c4;
v2= s1*c23*s4+c1*c4;
v3= s23*s4;
v4= c23*s5; /*New*/

/* To avoid inaccuracy-induced overflow if c6 or s6 are
```

```
                close to 1 in magnitude.
                The u's are already scaled down (above)*/
              v1>>=1; v2>>=1; v3>>=1; v4>>=1;

              s6= -r11*v1 -r21*v2 +r31*v3;
              c6= r11*(u1*c5-u4*s5)+r21*(u2*c5-u5*s5)-r31*(u3*c5+v4);

              /* No need to scale s6/c6 up again since they are
                only used in atan2 */

              if (s6<=-0.5)s6=-1;else if (s6>=0.5)s6=ONE;else s6<<=1;
              if (c6<=-0.5)c6=-1;else if (c6>=0.5)c6=ONE;else c6<<=1;
              teta6[ind2b+0]= catan2(s6,c6);

              /* The number of solns is doubled by symmetry of the hand */
              teta4[ind2b+1]= teta4[ind2b+0] + FIXPI;
              teta5[ind2b+1]= -teta5[ind2b+0];
              teta6[ind2b+1]= teta6[ind2b+0] + FIXPI;

              /*
                Place each solution in the array. Some indices are
                used repeatedly and are computed once each iteration:
                ind3== 4*i+2*j+k
                ind4== (4*i+2*j+k)*6+(index_to_teta)
              */
              for (k= 0; k <= 1; k++) {
                  ind3= 2*ind2+k;
                  ind4= ind3*6;
                  tetamatrix[ind4+0]= teta1[i];
                  tetamatrix[ind4+1]= teta2[ind2];
                  tetamatrix[ind4+2]= teta3[j];
                  tetamatrix[ind4+3]= teta4[ind3];
                  tetamatrix[ind4+4]= teta5[ind3];
                  tetamatrix[ind4+5]= teta6[ind3];
              }
          }
      }
  }
```