APPLICATION OF SILICON COMPILATION TECHNIQUES TO A ROBOT CONTROLLER DESIGN

by

Syed Khalid Azim

Memorandum No. UCB/ERL M88/35

24 May 1988

APPLICATION OF SILICON COMPILATION TECHNIQUES TO A ROBOT CONTROLLER DESIGN

by

Syed Khalid Azim

Memorandum No. UCB/ERL M88/35

24 May 1988

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

APPLICATION OF SILICON COMPILATION TECHNIQUES TO A ROBOT CONTROLLER DESIGN

by

Syed Khalid Azim

Memorandum No. UCB/ERL M88/35

24 May 1988

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Application of Silicon Compilation Techniques to a Robot Controller Design

Ph.D. **Syed Khalid Azim** **Department of EECS**

Robert W. Brodersen

Chairman of Committee

ABSTRACT

The very high level of integration possible with today's integrated circuits technology and the availability of powerful IC CAD tools have created tremendous possibilities for developing electronic systems hitherto considered impractical or not cost-effective. In this thesis we present the design methodologies, architecture, and circuit designs used for applying silicon compilation techniques to design problems requiring custom IC solutions.

Our approach for rapidly generating application specific ICs is based on a pre-defined architecture model of a processor. This model consists of a powerful, generic control unit integrated with application specific data paths and associated local memories. The design of the processor is composed of highly modular and parameterized macrocells hierarchically assembled together according to descriptions in a set of structural description files. The actual generation of a customized version of the processor's layout is performed by a silicon compiler package, beginning with a high-level description of the algorithm being implemented.

We successfully applied the design methodologies, architecture, and circuits developed in this project for automatically generating a custom processor that implements an adaptive control algorithm for a two-axis robot arm. The chip

has been fabricated in 2 micron SCMOS technology. Tests show the chip can operate at up to 15 MHz with a power consumption of 200 milliwatts. We believe our techniques provide a very viable approach for rapid and cost-effective realization of application specific ICs.

This research was support by Defense Advanced Research Projects Agency, Contract N00039-87-C-0182.

Acknowledgement

A large number of people have contributed in many different ways towards the successful completion of the project described in this thesis. My greatest indebtedness is of course to my advisor, Professor Robert W. Brodersen. Inspite of his numerous responsibilities, he was very accessible and technical discussions with him always seems to provide new insights to problems.

A project of this scope and nature is not possible without the collaborative efforts of many others. In designing the hardware, I benefited immensely from months of often passionate discussions in group meetings with Professor Broderson, Professor Hilfinger, Ken Rimey, and C-S Shung. For the CAD support, which was indespensible for this project, I am thankful to Rajeev Jain, Brian Richards, C-S Shung, and Mani Srivastava. This project also broke new grounds of sorts in inter-departmental collaboration on campus. Professor M. Tomizuka of Mechanical Engineering department responded very enthusiastically when I first approached him with the idea of doing a custom processor for the robot control algorithm which his group was developing. My thanks are due to him and also to his student George Anwar. Within our own group, thanks are due to Phil Schrupp for building the tester board, Lars Thon and Gautam Doshi for revising some of the structural description files. Bill Baringer, Mani Srivastava, and Dave Merrill contributed some of the cells used in this project. Alex Lee designed the basic clock generation circuit and along with Jane Sun maintained the pad cell library in the face of continually changing technologies.

Finally, I must thank my close friends and family without whose support this endeavour would not have been possible. Special thanks to my wife, Lucy, whose support and incredible patience played no small role in surviving graduate school.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Cost-Effective Design of Integrated Circuits

The development of integrated circuits (IC) technology has always been very strongly influenced by the needs of electronic system designers and manufacturers. In the past, this has led to higher levels of integration at progressively lower cost per transistor. With the levels of integration now approaching a million transistors per chip and the availability of powerful IC CAD tools, major innovations have taken place and continue to take place in IC design concepts and methodologies. This in turn has provided new opportunities as well as challenges for the system designers. In this thesis we show, through an actual design of a robot control chip, how current IC technology and state-of-the-art CAD environment can be exploited for a cost-effective solution to a specific problem. Our design methodology, using customizable core processors, is based on application specific IC (ASIC) design techniques while at the same time trying to retain the benefits of full custom, standard ICs. This approach matches well with currently available IC CAD techniques, as discussed in the next section. Any IC design methodology must of course be judged by its ability to address the system designers' concerns and to provide a healthy return for the semiconductor manufacturer.

From a system designers point of view the most important considerations in designing a product are:

- Access to low cost components.
- Ability to provide unique features in order establish a distinct market presence.
- Ability to respond to market needs very quickly by being first with new system products.

Traditionally, system designers have relied on semiconductor manufacturers to provide low cost, mass produced, standard IC components such as rams, TTL parts, microprocessors, etc. Successful commodity digital signal processors and microprocessors are able to attract a large clientele of system designers and software expertise that can be quickly tapped for developing new products. Emulator boards allow rapid prototyping of systems. Moreover, since the components already exist, risks and uncertainities involved with a custom designed chip is eliminated. Because of these and other advantages, the demand for 'high-end' IC components, such as highly complex 16/32 bit processors, continues to grow.

Nevertheless systems designed with standard components have some major disadvantages. They do not provide any protection against competition from similar products by other manufacturers as can be seen by the mushrooming of IBM-PC compatibles and clones. Moreover, the system cost can be high because boards full of large number of standard components are needed even though the capabilities of the individual components are not fully utilized. Large component counts also lead to large power requirements. On the other hand, for the semiconductor manufacturer, standard components are an extremely competitive business. Huge investments in time and resources are required to get to the market in time with high performance IC components using state-of-the-art technology. At the same time competitive pressures continue to drive down prices; only a handful of semiconductor manufacturers are able to survive in the standard IC market. Consequently a large and growing number of semiconductor manufacturers have turned towards ASIC technology.

Extensive use of CAD tools combined with gate-array, standard-cell, or similar technologies allow ASIC manufacturers to provide system designers with relatively small volumes of custom IC components designed for a specific application. The advantages of ASICs have been extensively discussed in recent literature. (see for example [Koem86] [MdLki82]). Among the advantages are short design cycles, distinct product features through proprietary custom ICs, lower cost by integrating the functions of a large number of standard components in a single custom chip, etc. ASICs, however, have their limitations too.

ASIC products have short life cycles as the novelty of the product wears off or the market niche becomes saturated. On the other hand any ASIC product showing signs of longevity is certain to draw competition from other ASIC manufacturers. Moreover, most commercially available ASIC design tools are aimed at gate-level design. These tools do not support high-level architecture design and development of processor type circuits which may require large resources. Furthermore, additional resources are needed for processor type circuits in order to develop application software, compilers, etc. Such investments may not be justified for a custom IC of the complexity of a processor but targeted only at a specific application area. One solution, therefore, is to mix standard ICs and ASICs for a cost-effective system product.

Integration of standard ICs and ASICs can take on many forms depending on the sophistication of the CAD environment. The simplest approach of building systems with general purpose microprocessors as the main work horse and ASICs for proprietary circuits, interface circuits, and glue logic, is already being used in products such as personal computers. Some of the ASIC/semiconductor houses have gone one step further by providing customized interfaces and peripherals on their successful processor chips [Sim88]. Others are developing variations of a successful processor family by deleting functions not required and/or adding newer ones aimed at a specific application area. In this thesis, we have further advanced this approach by developing a methodology aimed at not only adding custom circuits to a core processor but also providing the ability to extensively customize the processor itself. High-level silicon compilation tools automatically generate customized processors

from behavioral descriptions of algorithms. We envision this approach a precursor for designing fully integrated systems with customized processors, custom glue logic, memory, and interconnects, all on a single substrate. The success of these approaches depend to a great extent on incorporating the design methodologies inside silicon compilers.

1.2 Silicon Compilation of DSP circuits

The term *silicon compilation* has been widely used in recent years to describe generation of IC layouts from high-level descriptions of circuit behavior. A silicon compiler is generally accepted to mean an IC design methodology embedded inside a CAD environment. The design methodology usually involves a series of transformations which result in a physical layout from an abstract description of the behavioral intent. These transformations may involve converting a high-level language (e.g. 'C' programming language) description of an algorithm into a microarchitectural specification, followed by a translation into logic-domain description, and finally mapping into a physical layout. The key point is to hide and automate the translation process from the user so that system designers can interact with the design system at a level which does not require expertise in circuit and layout design.

Silicon compilers may differ in their design methodologies and capabilities; however, most of them can be described in terms of a common, basic frame work. A detailed description is given in [GaDon88]. Examples of several silicon compilers are found in [Gaj88]. We give below a simplified description of silicon compilers within the context of research described in this thesis.

A top down silicon compilation process normally starts with a behavioral description in the form of an algorithm. A DSP algorithm, for example, gives a sequence of transformations on variables and data structures in order to implement some signal processing function such as Fourier Transforms. The algorithm is usually described in a programming language such as Silage [Hil85] or 'C' and is independent of any particular hardware realization. An algorithmic description

is next transformed into an implementation at the microarchitectural level. This implementation consists of *structural* elements – registers, ALUs, sequencer, etc. – and a sequence of microoperations – register transfer operations, arithmetic operations, control state transitions, etc. – describing the manipulation of data on the architecture. This transformation from algorithm to microarchitecture, sometimes referred to as *synthesis*, is perhaps the most difficult part of the compilation process since a wide range of architectural choices exist. Synthesis usually starts with a data flow representation of algorithms followed by attempts to allocate hardware based on heuristics or algorithms. Automating this step, however, is feasible only within a limited design space.

The interaction between high-level architectural decisions and low-level physical implementation strategies is very complex. An automatic exhaustive search of the design space for satisfying all the constraints of area, cost, and time is currently not feasible. A more practical approach is to restrict the design space to some target architecture. Current state-of-the-art silicon compilers vary in their synthesis strategies from providing programs for automatically synthesizing the target architecture, to open systems where the user completely defines his own architecture. Many research compilers [JaNoHa85] [Jain86] fall in the former category whereas commercial compilers (for example [Sil86]) concerned with realizing practical designs, fall in the latter category. An example of a silicon compiler targeted to a specific architecture is GE's Bit-Serial Silicon Compiler (BSSC) [JaNoHa85]. It uses a bit-serial architecture. Each operator in the algorithm is mapped into a predetermined hardware element. Other than optimizing the number of delay elements used for synchronization, no other optimization is performed. Another example of a silicon compiler targeted to a specific architecture is Cathedral-II [Rab88]. It uses a bit-parallel multi-processor architecture. A rule-based software maps higher-level program constructs into a specific implementation of the architecture. Another example of a targeted silicon compiler is LagerI [RaPoBr85]. LagerI uses a fixed architecture consisting of parameterized macrocells.

We believe, use of pre-defined architecture currently provides the most viable option to the architecture synthesis problem. Based on designer's experience

and expertise, an architecture model can be defined for a class of application. For DSP applications, several architectural choices are available: These include level of pipelining in the data paths; clocking schemes; support for complex, control flow type operations; programmable processors versus dedicated, hard-wired processors; etc. Once an expert designer selects the architecture, high-level compilers can then be written to map algorithmic descriptions (internally represented most likely in the form of data flow graphs) into the target architecture. The compiler writer chooses the most appropriate techniques to apply in performing the mapping. The quality of a compiler will be judged on the level of customization, variations, and optimizations allowed in the final implementation. This approach is also in congruence with the customizable, core processor based approach found suitable from a cost-benefit point of view as discussed in section 1.1. Whereas the transformation from algorithm to microarchitecture is regarded as synthesis, the mapping of the microarchitecture into a physical layout is sometimes referred to as *silicon assembly.*

The first step in the silicon assembly process is transformation of the microarchitecture into a logic level description. This is followed by a transformation into a circuit description represented in the physical domain as layouts. In many cases, the series of transformations from the microarchitecture's structural elements, such as muxes, registers, etc., to physical layout is a one-step process because of previously performed mappings, stored in the form of cell libraries. In general, the logic to layout transformation may use any of the available methodologies such as gate arrays, standard cells, macrocells, etc. Thus we see the silicon compilation process as a series of transformations aimed at converting an application idea into a physical layout.

In the next chapter of this thesis we describe a specific robot control application and the hardware implementation issues. Based on this discussion we define a processor architecture in chapter 3. This is followed by a discussion on the methodology for silicon compilation and the silicon compiler framework in chapter 4. The details about the circuit design are given chapter 5. Since design for testability is a very important design consideration, entire chapter 6 is devoted to discussing design of testable circuits. This is followed by a description of the im-

plementation of the robot control algorithm on the processor. Finally, in the last chapter we present our conclusions.

Chapter 2

Digital Control of a Two-Axis Robot Arm

2.1 Robot Control Systems

> A robot is a general-purpose machine system that, like a human, can perform a variety of different tasks under conditions that may not be known *a priori* [Dav85].

An industrial robot (figure 2.1) may consist of several mechanical linkages connected through joints such that the linkages can rotate or slide about the joints, usually within certain limits.

In typical applications, the robot moves an object such as a tool or a part, from one point to another point in space along some desired trajectory. The motion of the robot is usually controlled by a *controller*. The design of these controllers are pre-dominantly based on digital control techniques. A complete robot control system may consist of several levels of hierarchy.

> The upper levels of the hierarchy are concerned with global issues of control: the selection and high-level decomposition of goals and objectives, the recognition of complex patterns and relationships between objects, and the generation of plans and schedules which span extended periods of time. The middle levels of the hierarchy are occupied with more immediate issues of tactics and task sequencing: the decomposition of complex tasks into elemental movements and

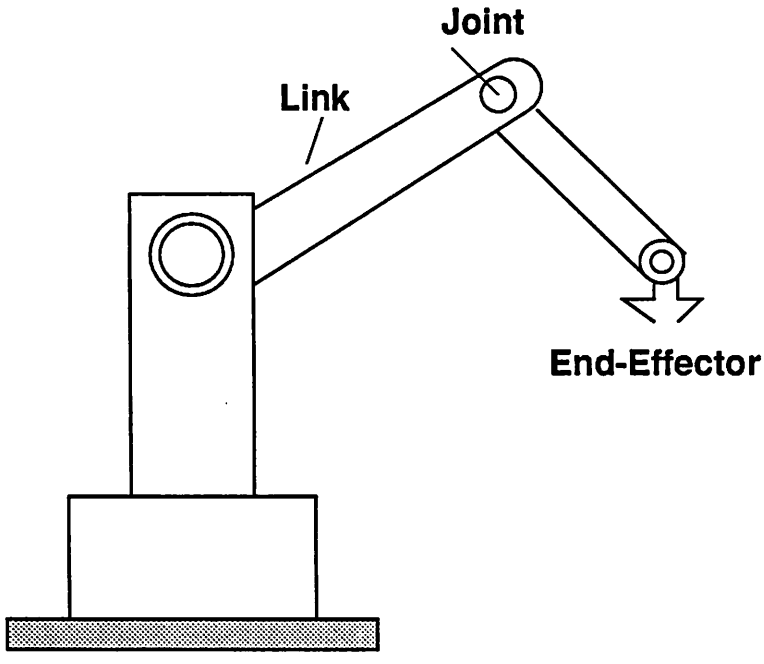


Figure 2.1: A robot arm consists of links and joints

trajectories, the recognition of simple patterns and primitive features, and the comparison between expectations and observations on a second by second basis. The lowest levels of the hierarchy perform the high-speed computations of coordinate transformation and servo control: the generation of forces, velocities, and accelerations, the measurement of tracking errors, and the monitoring of safety and overload conditions [AlBaFi].

The robot control system shown in figure 2.2 groups all the upper and middle levels of the control hierarchy into a single box labeled as *Path Planning and Profile Generation.* The output of this box are time-sequence of points specifying the desired trajectory for each joint.

The box labeled as *controller* in figure 2.2 represents a digital feedback controller which performs the low level control of each joint. It samples sensory data (such as position, velocity) from the robot joints and the desired trajectory data from the upper-level control modules at regular intervals. The controller then computes a control output for reducing the error between the desired joint position and its actual position. This control output determines the amount of torque required on the joint to reduce the position error. The time required to read in the

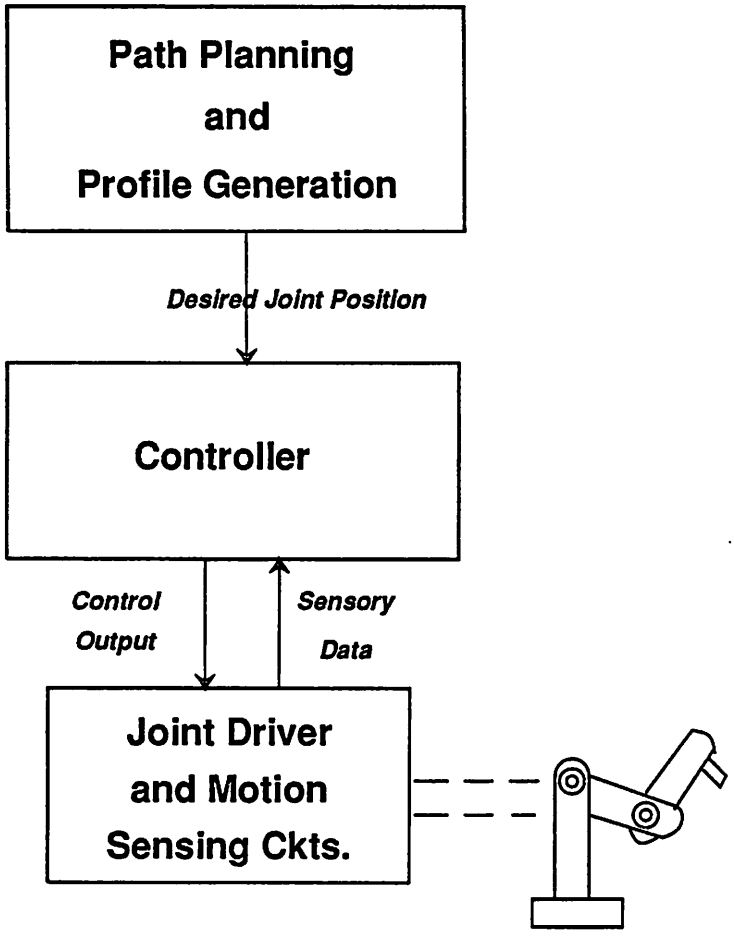


Figure 2.2: Robot control systems are hierarchically organized

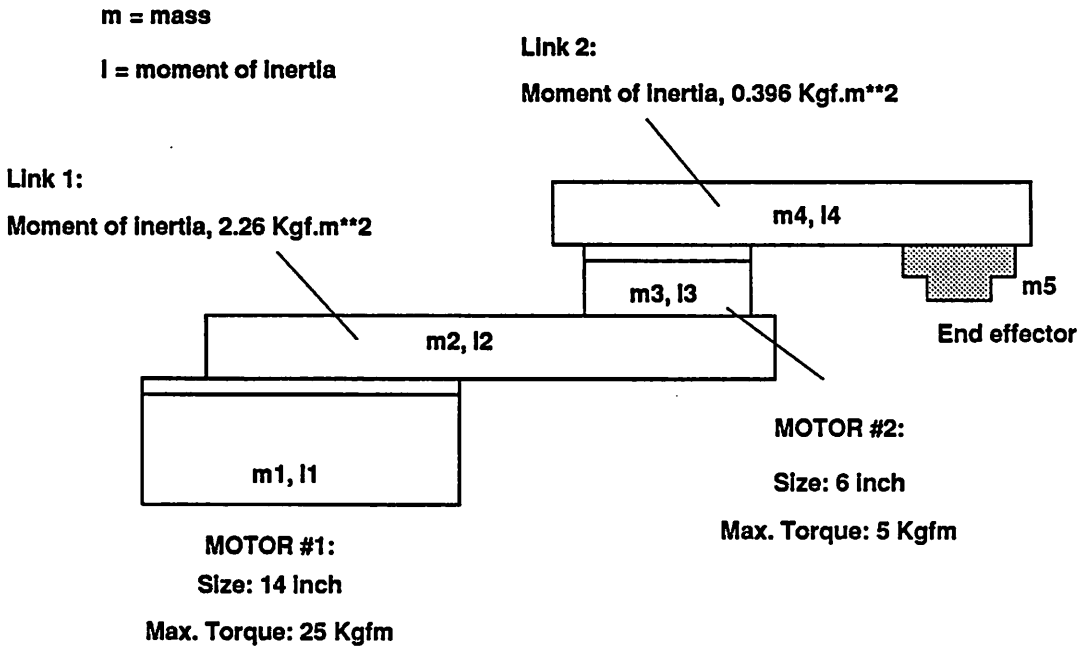


Figure 2.3: A two-axis, direct-drive robot arm from NSK/Motornetics Corp., Santa Rosa, California

sensory data, compute the control output, and apply the new control output to the joints often limits the sampling rate of the controller.

This thesis deals with the design and implementation of the low-level controller. In this chapter we first present the algorithms used for controlling the robot arm and then discuss the issues pertaining to implementation of the controller. The design of the controller is targeted towards a two-axis robot arm (figure 2.3) with both axes having rotary motion in the horizontal plane. Each joint is driven by a direct-drive motor.

The use of direct-drive motors makes payload variations and the link's inertia variations due to changes in robot's arm configuration a significant issue in the design of controllers for high performance applications. Another factor affecting the design of the controller is inherent non-linearities of the robot arm dynamics, which makes standard feedback control techniques inadequate for precision control at high speeds.

2.2 Non-linearities in Robot Arm Dynamics

The general form of the torque equation [Sny85] for a n-joint robot arm is described below:

$$\begin{aligned} q_{[i]} = \; & \textstyle\sum_{j=1}^{n} m_{[ij]} \times \ddot{\theta}_{[j]} + \\ & \textstyle\sum_{j=1}^{n} D_{cent[ijj]} \times \dot{\theta}^2_{[j]} + \\ & \textstyle\sum_{j=1}^{n} \sum_{k=1,k\neq j}^{n} D_{cor[ijk]} \times \dot{\theta}_{[j]} \times \dot{\theta}_{[k]} + \\ & c_{lin[i]} \times \dot{\theta}_{[i]} + d_{g[i]} + d_{coul[i]} \end{aligned} \tag{2.1}$$

In the above equation, i and j represent the ith and jth link respectively. $q_{[i]}$ is the torque on the ith joint. The first product-term, $m_{[ij]} \times \ddot{\theta}_{[j]}$ represents the inertial torque; $m_{[ij]}$ being the inertia and $\theta_{[i]}$ being the angular displacement. When $i = j$, the term gives the torque due to self-inertia of the link; when $i \neq j$, the term gives the torque due to inertial coupling between links. The second product-term represents the torque due to centripetal forces on joint i caused by motion in link j. The third product-term represents the torque due to Coriolis force. It is caused by the combined motion of links j and k and therefore does not exist for a single-joint motion. The last three product-terms represent disturbance forces due to friction and gravity. $d_{g[i]}$ represents gravitational force and is a function of the angular position of the link in the vertical plane. Gravitational force do not affect motion restricted to the horizontal plane only. $c_{lin[i]}$ is a coefficient of viscous friction, whereas $d_{coul[i]}$ represents Coulomb friction.

The torque equation for the two-axis robot arm, shown in figure 2.3, has been derived by [Tsai87] on the basis of Lagrangian formulation [Paul81]. This torque equation, which gives a mathematical model for the dynamic behavior of the arm, is described below.

$$\begin{pmatrix} q_1 \\ q_2 \end{pmatrix} = \begin{pmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{pmatrix} \times \begin{pmatrix} \ddot{\theta}_1 \\ \ddot{\theta}_2 \end{pmatrix} + \begin{pmatrix} V_1 \\ V_2 \end{pmatrix} + \begin{pmatrix} d_1 \\ d_2 \end{pmatrix} \tag{2.2}$$

q_1, q_2 represent the torques; V_1, V_2 represent the centripetal and Coriolis forces; and d_1, d_2 represent the frictional forces on joint 1 and 2 respectively. The expressions

for the various terms in equation 2.2 are:

$$\begin{aligned} m_{11} &= I_1 + I_2 + I_4 + l_1^2\,(m_2/4 + m_3 + m_4 + m_5) \\ &\quad + l_2^2\,(m_4/4 + m_5) + l_1\,l_2\,Cos\theta_2\,(m_4 + 2m_5) \qquad (2.3) \\ m_{12} &= I_4 + l_2^2\,(m_4/4 + m_5) + l_1\,l_2\;Cos\theta_2\,(m_4/2 + m_5) \qquad (2.4) \\ m_{21} &= m_{12} \qquad (2.5) \\ m_{22} &= I_3 + I_4 + l_2^2\,(m_4/4 + m_5) \qquad (2.6) \\ V_1 &= -(m_4/2 + m_5)\;l_1\,l_2\;(2\dot{\theta}_1\dot{\theta}_2 + \dot{\theta}_2^2)\;Sin\theta_2 \qquad (2.7) \\ V_2 &= (m_4/2 + m_5) l_1\,l_2\,\dot{\theta}_1^2\,Sin\theta_2 \qquad (2.8) \end{aligned}$$

m_5 is the payload being moved by the robot arm; l_1 and l_2 are the lengths of link 1 and link 2 respectively; and I_n represents the moment of inertia of the n*th* member of the arm (see figure 2.3) The Coulomb friction component of the friction term, d, is described below,

$$d_{coul} = \begin{cases} d_{cm} \times sign[\dot{\theta}] & if\ |\dot{\theta}| > 0 \\ d_{cm} \times sign[q] & if\ |\dot{\theta}| = 0\ and\ |q| \geq d_{cm} \\ q & if\ |\dot{\theta}| = 0\ and\ |q| < d_{cm} \end{cases} \qquad (2.9)$$

where d_{cm} is a friction constant. *The key point to note from the torque equations described in this section is the non-linear nature of the robot arm dynamics.* The inertia, Coriolis and centripetal forces, and Coulomb friction terms are all functions of the payload, the angular position of the joints, and the angular velocities of the joints. Moreover, the dynamics of one joint is affected by the motion of the other joint. Furthermore, the trend towards using direct-drive motors in modern, high-performance robots makes the robot arm dynamics sensitive to payload variations.

D.C. motors generally used in industrial robots develop high speeds but relatively low torque. Therefore, gears are used to increase torque output. This also reduces the speed and the load inertia seen by the motor as described by the following equations,

$$q_{(applied\ to\ load)} = N \times q_{(applied\ by\ motor)} \qquad (2.10)$$

$$\dot{\theta}_{load} = 1/N \times \dot{\theta}_{motor} \qquad (2.11)$$

$$m_{(seen\ by\ motor)} = \quad m_{(at\ the\ load)}/N^2 \tag{2.12}$$

where N is the gear ratio and m is the inertia. In a multi-axis robot arm, the load inertia may also include inertia of other links which will vary as a function of the link's relative position. As described by equation 2.12, the inertia of the load seen by the motor is scaled down by N^2 from its actual value. Therefore, the effect of load variations on the torque required from the motor (see equation 2.1) is greatly diminished. However, gears result in back-lash (from wear and tear), and contribute to increased friction. These problems are overcome in some of the new, high-performance robots by using direct-drive motors. They produce high torques without requiring gears, but load variations now have a more pronounced effect on the motor. This requires compensation by the controller.

Commercial robots use standard feedback control techniques such as PID [Astrm84] control which do not compensate for non-linearities in the controlled system's behavior. Consequently, adaptive control techniques have become an important area of investigation for providing satisfactory control of high performance robot arms.

2.3 Adaptive Controllers for Robots

Adaptive control is a feedback control technique for controlling partially known and/or time-varying dynamic systems to operate in a consistent and desired manner. The adaptive controller automatically adjusts its parameters for properly controlling the dynamic process. Such controllers are useful in robot control where the complex non-linear dynamics of the robot system limits the performance attainable from fixed gain PID (proportional, integral, and derivative) controllers. With the growing need for high performance robots, many research groups, in the last several years, have focused on applying adaptive techniques to robot control [DuDe79] [KoiGu81] [HorTom82]. In this section we describe a model reference adaptive control (MRAC) scheme developed by Tomizuka and his group [Hor86] [Tom86]. The complete algorithm, shown in block diagram form in

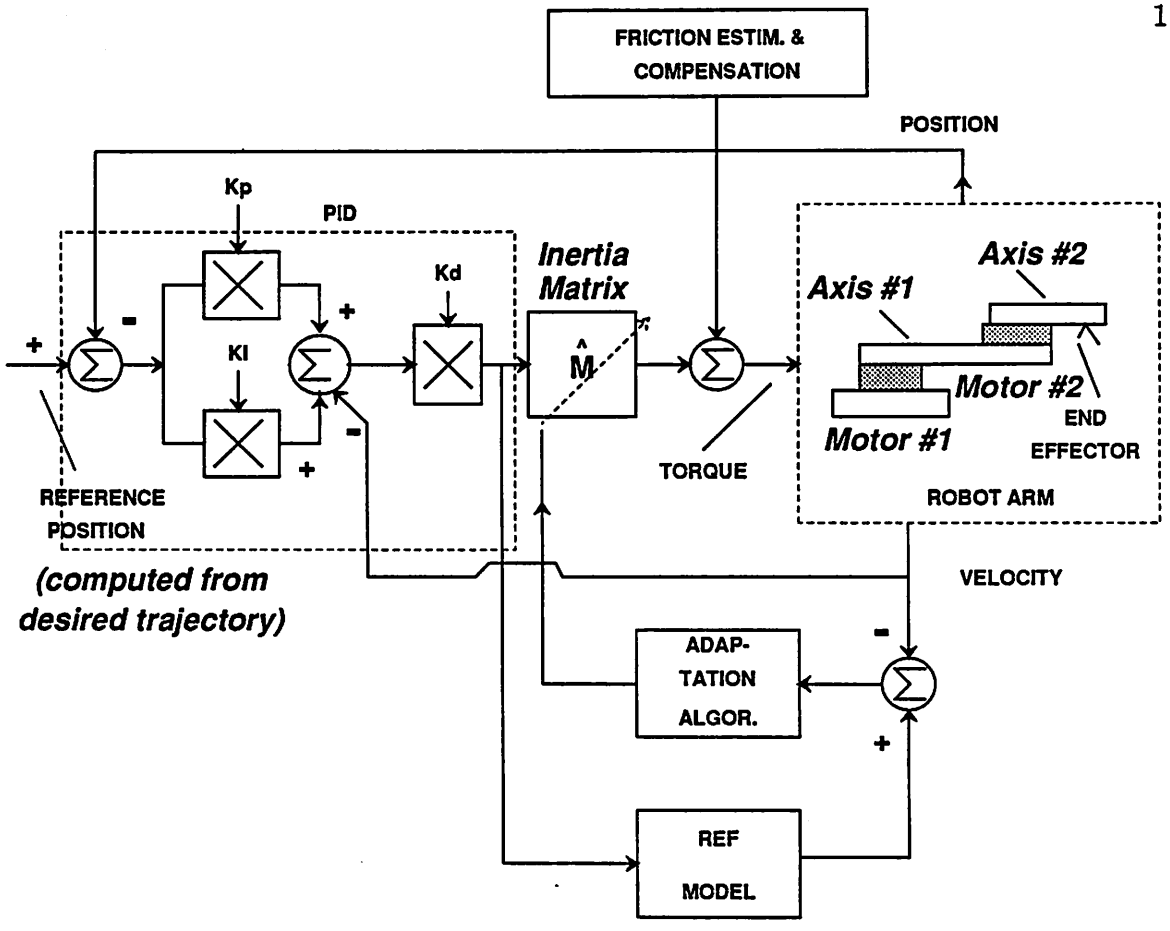


Figure 2.4: Block diagram of the control algorithm showing PID section, adaptive section, and friction compensation

figure 2.4, consists of two major sections: an adaptive control section including friction compensation; and a PID control section.

2.3.1 A Model Reference Adaptive Control Algorithm

The MRAC algorithm includes estimation of the robot system's parameters. In the MRAC scheme studied in this dissertation, parameter estimation is based on the error between the robot's response and the output of a reference model. Based on these estimates, torque is computed to achieve the desired motion. The following equations in discrete time domain describe the MRAC algorithm. The

centripetal and Coriolis terms are neglected in these equations.

$$\begin{aligned}
Ref.\ model: &\quad \theta_{vm[i]}(k+1) = \theta_{v[i]}(k) + T.u_{[i]}(k) && (2.13)\\
Vel.\ error: &\quad e_{v[i]}(k) = \theta_{vm[i]}(k) - \theta_{v[i]}(k) && (2.14)\\
Dead\ band: &\quad if\ e_{v[i]}(k) \leq V_{db},\ e_{v[i]}(k) = 0 && (2.15)\\
Inertia & && (2.16)\\
Adaptation: &\quad \hat{m}_{[ij]}(k) = \hat{m}_{[ij]}(k-1) + K_{m[ij]} \times \\
& \qquad T.e_{v[i]}(k).u^T_{[j]}(k-1) && (2.17)\\
Friction & && (2.18)\\
Estimation: &\quad (see\ below) && (2.19)\\
Cntl.\ torque: &\quad q_{[i]}(k) = \hat{m}_{[ij]}(k).u_{[j]}(k) + \\
& \qquad \hat{d}_{f[i]}(k) && (2.20)
\end{aligned}$$

As illustrated in figure 2.4 the input to the adaptive section is an acceleration quantity, $u_{[i]}(k)$, required to achieve the desired motion of the joint. The reference-model is a simple integrator whose output is the model velocity $\theta_{vm[i]}$. This velocity is compared with the actual velocity of the robot arm, $\theta_{v[i]}$ and the resulting error is used to compute $\hat{m}_{[ij]}$, the estimated inertia. The velocity error is set to zero when it goes below the dead band thresh hold limit, V_{db}. $K_{m[ij]}$ is a coefficient of adaptation and T is sampling period. Finally, the required control torque, $q_{[i]}$ is computed using the estimated inertia. The torque expression also includes a Coulomb friction compensation term, $\hat{d}_{f[i]}$, which is adaptively estimated.

Adaptation for $\hat{d}_{f[i]}$:.

$$\begin{array}{lllll}
\text{if} & \theta_{v[i]} \geq V_f, & & & \hat{d}_{f[i]}(k) = \hat{d}_{f[i]}(k-1) + K_f \times e_{v[i]}\\
\text{else if} & \theta_{v[i]} \leq -V_f, & & & \hat{d}_{f[i]}(k) = \hat{d}_{f[i]}(k-1) - K_f \times e_{v[i]}\\
\text{else if} & |\theta_{v[i]}| < V_f & \& & u_{[i]} \geq 0, & \hat{d}_{f[i]}(k) = \hat{d}_{f[i]}(k-1) + K_f \times e_{v[i]}\\
\text{else} & |\theta_{v[i]}| < V_f & \& & u_{[i]} < 0, & \hat{d}_{f[i]}(k) = \hat{d}_{f[i]}(k-1) - K_f \times e_{v[i]}
\end{array} \qquad (2.21)$$

V_{db} and V_f are thresh-hold limits for applying dead band and friction compensation respectively.

Assuming the MRAC algorithm is working perfectly, the robot arm appears as a linear system. Since the MRAC controller effectively results in the

dynamics from $u_{[i]}$ to velocity, $\theta_{v[i]}$ (see figure 2.5), appear as a pure integrator, the position loop dynamics from $u_{[i]}$ to position, $\theta_{[i]}$, is characterized by a double integrator. Consequently, a non-adaptive outer loop consisting of a PID type controller is applied for accurately controlling the joint's position.

2.3.2 A PID Controller for Position Control

A PID controller for one joint of the robot arm is shown in figure 2.5. The robot joint is modelled by a double integrator since the MRAC algorithm is assumed to perfectly compensate for the non-linearities, joint interaction, and friction. A PID controller consists of three actions, which act upon the error between the desired reference position and the actual position of the joint. The three actions are as follows:

(i) Proportional or P action,

$$u_p(k) = K_p \times e_p(k) \tag{2.22}$$

where $e_p(k)$ is the position error. For many applications a simple P action is adequate. However, it results in steady state error. This error can be reduced by increasing the gain K_p, but this can lead to oscillatory behavior and instability. The steady state performance can be improved by adding an I action.

(ii) Integral or I action,

$$u_I(k) = K_I \times \sum e_p(k) \tag{2.23}$$

The addition of the I action reduces steady state error (becomes zero for a step input). However, too large a gain K_I can lead to the problem of reset windup [Astrm84] due to the controller output exceeding the saturation limit of the joint drivers. Moreover, I action also reduces the stability of the system. In order to improve stability, a D action is often added.

(iii) Differential or D action,

$$u_d(k) = K_d \times \dot{e}_p(k) \tag{2.24}$$

The D action improves the stability of the system by adding a damping effect. Since, $e_p(k)$ can be large at the beginning of a sample period (when the reference position

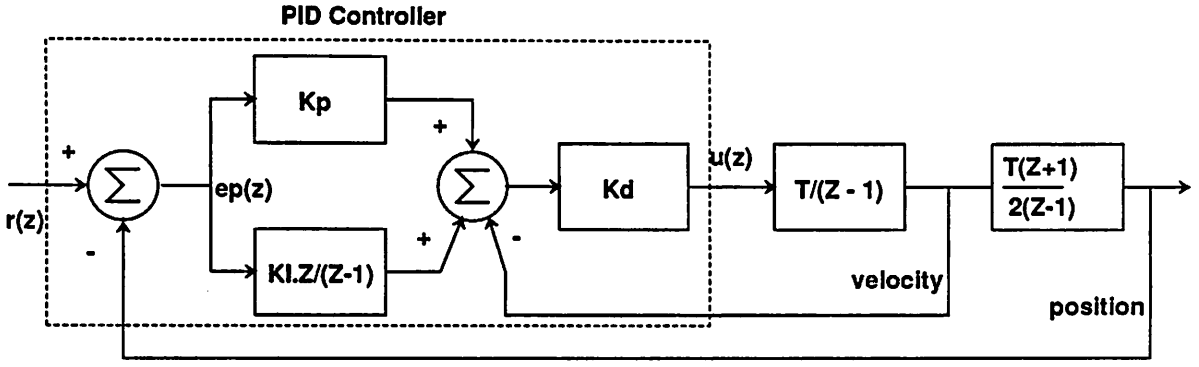


Figure 2.5: PID controller for a robot joint modelled as a double integrator. T is the sample period.

is updated), $K_d \times \dot{e}_p(k)$ can result in a large contribution to the control output. Therefore, an alternative form of the D operation is sometimes used in which the D action is performed on the current position, rather than on the position error.

$$u(k) = K_d \times [u_I(k) + u_p(k) - \dot{\theta}(k)] \tag{2.25}$$

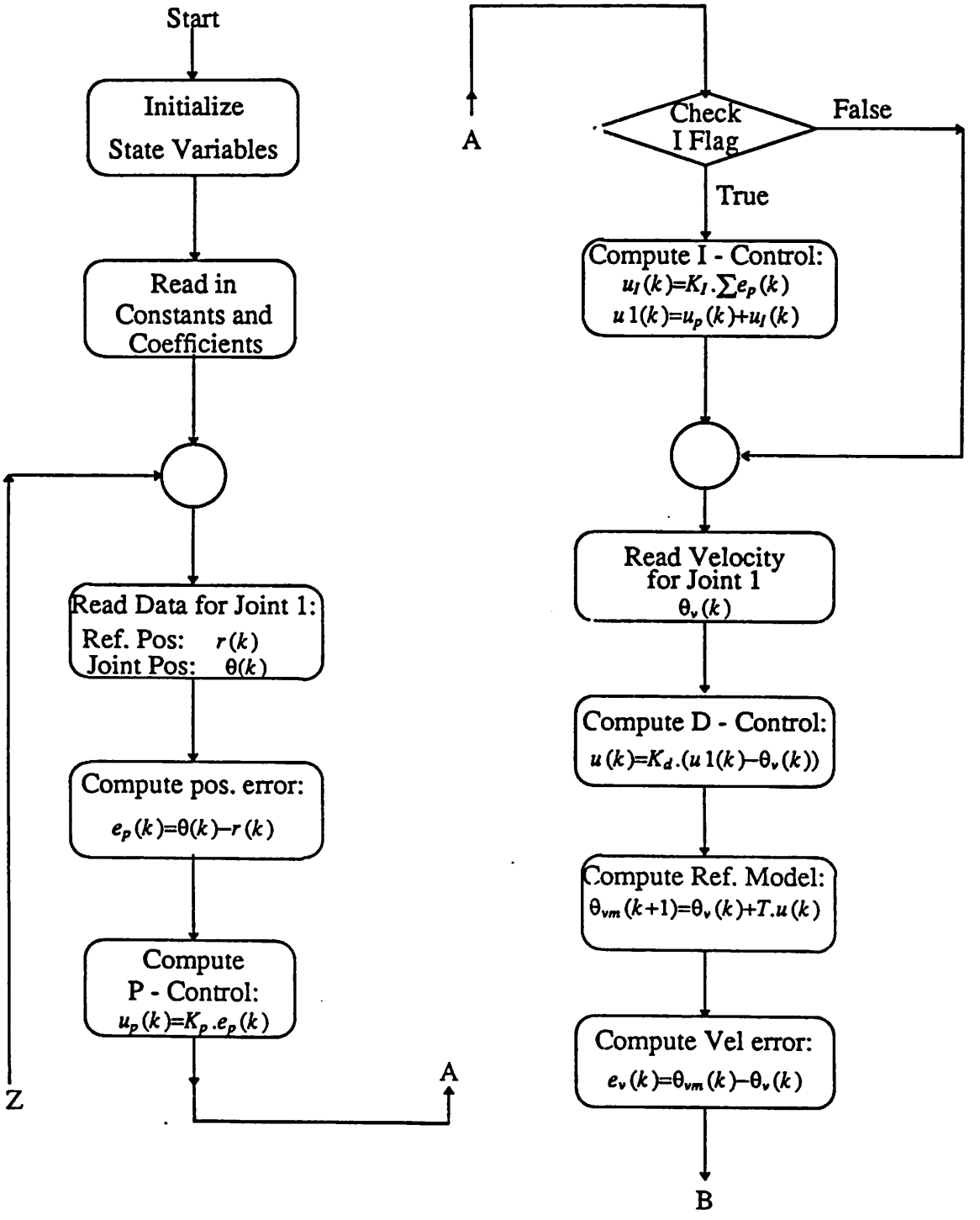
This is the form of D action used in the PID controller shown in figure 2.5, where $u(k)$ is the acceleration required by the joint to achieve the desired input reference position, $r(k)$. In figure 2.5, all the quantities are represented in Z domain. Also note that in the PID algorithm shown in the figure, the effective P-gain, K_p, and I-gain, K_I, are given by K_dK_p and K_dK_I respectively.

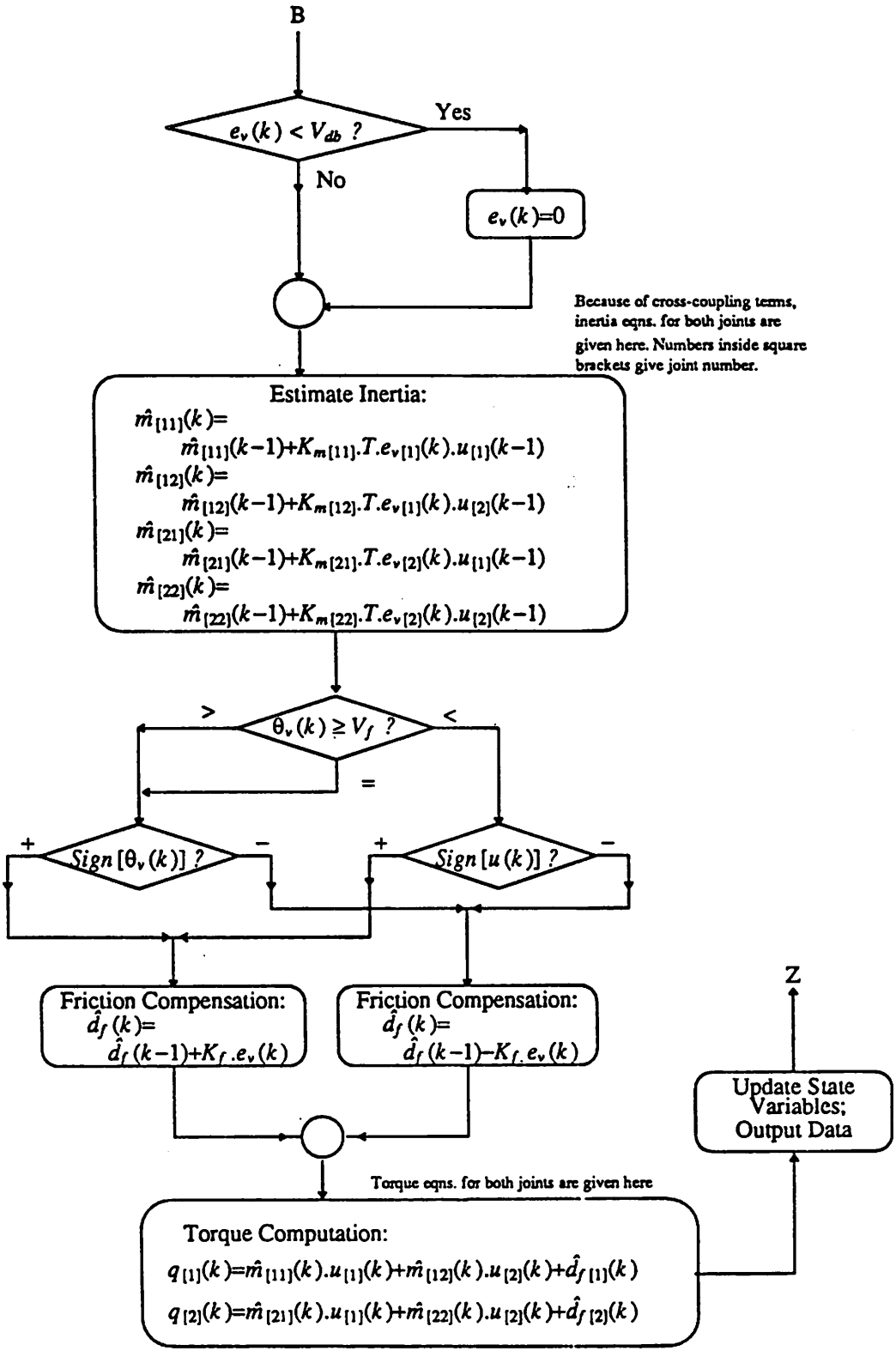
In sections 2.3.1 and 2.3.2 we have presented the adaptive control algorithm and the PID control algorithm required for accurately controlling the robot joints. In the next section we shall discuss the issues related to efficiently implementing the algorithms.

2.4 Issues in Implementing the Robot Controller

In order to specify the requirements for designing a custom hardware for the robot controller, we examine the implementation issues from three points of

Flow chart for PID and adaptive controller for one joint of the two-axis robot arm




B
$e_v(k) < V_{db}$?
Yes
No
$e_v(k)=0$
Because of cross-coupling terms, inertia eqns. for both joints are given here. Numbers inside square brackets give joint number.
Estimate Inertia:
$\hat{m}_{[11]}(k)=\hat{m}_{[11]}(k-1)+K_{m[11]}.T.e_{v[1]}(k).u_{[1]}(k-1)$
$\hat{m}_{[12]}(k)=\hat{m}_{[12]}(k-1)+K_{m[12]}.T.e_{v[1]}(k).u_{[2]}(k-1)$
$\hat{m}_{[21]}(k)=\hat{m}_{[21]}(k-1)+K_{m[21]}.T.e_{v[2]}(k).u_{[1]}(k-1)$
$\hat{m}_{[22]}(k)=\hat{m}_{[22]}(k-1)+K_{m[22]}.T.e_{v[2]}(k).u_{[2]}(k-1)$
$\theta_v(k) \geq V_f$?
>
<
=
+
−
Sign $[\theta_v(k)]$?
+
−
Sign $[u(k)]$?
Friction Compensation:
$\hat{d}_f(k)=\hat{d}_f(k-1)+K_f.e_v(k)$
Friction Compensation:
$\hat{d}_f(k)=\hat{d}_f(k-1)-K_f.e_v(k)$
Z
Update State Variables; Output Data
Torque eqns. for both joints are given here
Torque Computation:
$q_{[1]}(k)=\hat{m}_{[11]}(k).u_{[1]}(k)+\hat{m}_{[12]}(k).u_{[2]}(k)+\hat{d}_{f[1]}(k)$
$q_{[2]}(k)=\hat{m}_{[21]}(k).u_{[1]}(k)+\hat{m}_{[22]}(k).u_{[2]}(k)+\hat{d}_{f[2]}(k)$

view:

(i) Efficient implementation of the algorithm.

(ii) Interfacing with the system hardware.

(iii) Testing and debugging.

2.4.1 Efficient Implementation of the Algorithm

The flow chart for the entire control algorithm appears on the preceding pages. In this section we shall identify commonly used operations in this algorithm and discuss their implications for designing the hardware.

Multiply-Accumulate

The equations describing both the adaptive control algorithm (equations 2.13 - 2.21) and the PID algorithm (equations 2.22 - 2.25) reveal a large number (22 multiplications in all) of *multiply-accumulate* type operations. This is typical of signal processing algorithms, making a hardware parallel multiplier an essential part of commercial DSPs. In designing a custom processor for implementing the two-axis robot controller, the question of including a hardware multiplier must be carefully examined on the basis of required sampling rate.

Simulations and experimental studies reported by [Hor86] as well as our own simulations, show higher sampling rates improve the performance of the controller. A TMS32010 based implementation (see figure 2.7) achieved a sampling period of 0.7 ms [Anwr87]. For the custom implementation we have targeted a sampling period of 0.5 ms.

Table 2.1, based on data reported in [Vos86], shows the area speed trade offs for two forms of multiplication: (i) using a parallel multiplier requiring only one cycle for a $n \times n$ multiplication; (ii) using shift and accumulate operations requiring n cycles for $n \times n$ multiplication. In general parallel multipliers are the fastest whereas shift and accumulate based multiplication without any booth decoding are the slowest. The actual cost of using a parallel multiplier is even higher than what the table shows. This is because a processor using a multiplier still needs a

	Area (mm^2) in 3 μ CMOS	Time (ms) for 22 multiplies
Parallel 20 × 20 Hardware Multiplier	6.5	0.0022
20 bits Shifter/Accum. Data Path	1.25	0.044

Table 2.1: Area-speed trade-off: shift/accumulate vs. parallel multiplication

data path for other arithmetic operations. On the other hand with shift-accumulate multiplication, the data path resources are shared with other arithmetic operations. The time required to complete 22 multiplications using iterative shift-accumulate operations, although slower, take only about 10% of the sample period without requiring the overhead of a parallel multiplier. We, therefore, conclude that *a parallel multiplier is not necessary for implementing the two axis-robot control algorithm.*

Conditional Operations

Apart from the large number of multiplications, the other dominant characteristic of the robot control algorithm is the use of conditional operations. The dead band expression (eqn. 2.15) and the Coulomb friction compensation (eqn. 2.21), both require *if ... then ... else* type operation. The flow chart of the control algorithm shows another use of conditional operation: optionally skipping the I control part of the PID controller, using an external control signal, *IFlag.*

if *IFlag* is true
    then compute I Control,
else go to
    D Control.

Program loops for efficiently implementing the control algorithm also require conditional operation. The iteration count of the loop is usually maintained in a loop counter. At the end of each iteration the counter is tested and a decision is made to repeat the loop or to branch to a new section of the program.

In commercial DSPs, e.g. TMS32010, conditional operations are performed with branch instructions requiring two cycles, such as,

BANZ: branch on aux. reg not zero

BGEZ: branch if accum ≥ 0

BZ: branch if accum $= 0$

Clearly, in applications requiring frequent use of conditional operations, they must be supported efficiently.

Subroutines and Loops

Subroutines and loops are generally used to provide programming efficiency by conserving program rom space. The LagerI [Pope85] architecture had very limited support for subroutines and loops. This was not a major problem for some of the applications for which the LagerI architecture was used. Partitioning the application program among several processors operating in parallel, kept the size of the program (typically less than 100 instructions) in each individual processor relatively small. Parallel processing, however, costs chip area due to multiple processors and additional hardware for inter-processor communication, not to mention the need to maintain synchronism between different processors. For relatively low sampling rate applications, this type of overhead is not always justified.

In the robot control application, assuming a 100 ns instruction cycle time, a 0.5 ms sample period allows 5000 instructions to be executed per sample. Since the 22 multiplications (see section 2.4.1) require $22 \times 20 = 440$ instruction cycles, we can safely assume the control algorithm for both the joints can be implemented on a single processor. This also avoids inter-processor communication problems. Since at low sampling rate a large number of instructions can be executed on a single processor, the program rom size can become very large. Therefore, support for sub-routines and loops is very important for reducing rom size.

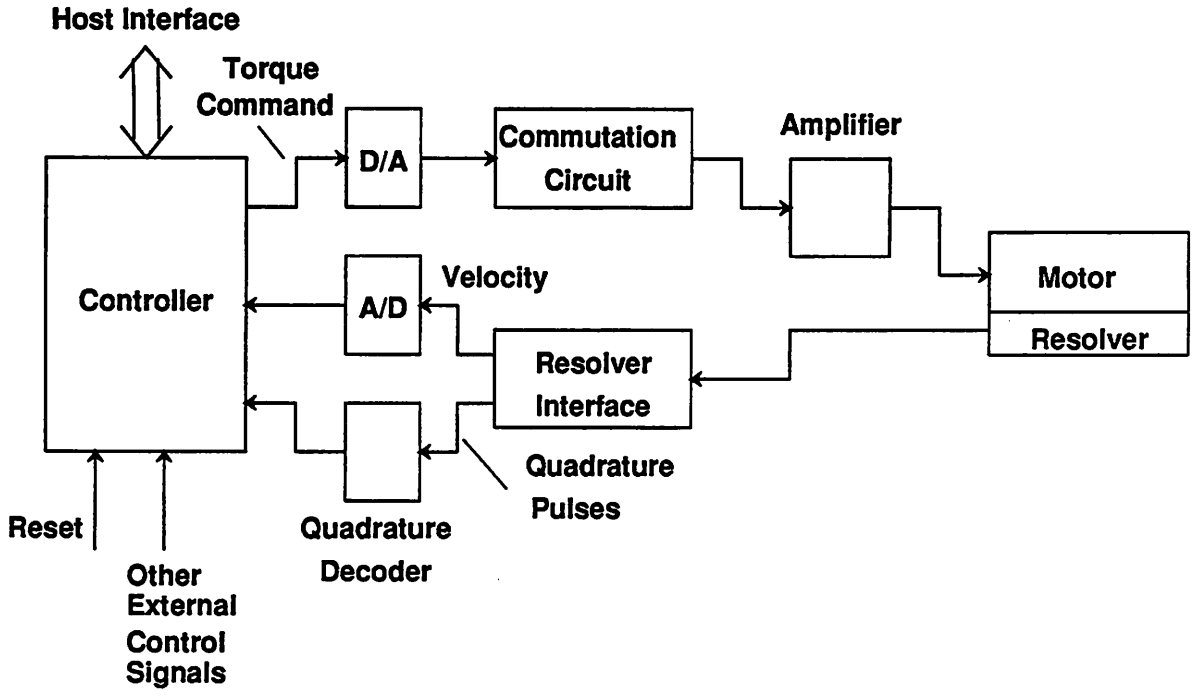


Figure 2.6: Robot controller with external signals and I/O circuits for one joint

2.5 System Interface

Since board-level design can be time-consuming and costly, an effective custom solution for a specific application must aim at a high-level of system integration. This involves integrating peripheral circuits; customizing the processor to potentially include the peripheral circuits; and providing convenient I/O interfaces on the processor. Figure 2.6 shows a block diagram of the robot controller along with its peripheral and I/O circuits. In the discussion below, we classify these circuits into two categories: circuits provided by the manufacturer; and circuits provided by the user.

2.5.1 Circuits Provided by the Manufacturer

The direct-drive motor from NSK/Motornetics comes with an interface unit [NSK86] which contains three main circuit components:

(i) *Commutation circuit* which converts an analog torque command or velocity command signal into a three-phase signal for driving the motor.
(ii) *Power amplifiers* to boost the power of the three-phase signal from the commutation circuit, before feeding it to the motor windings.
(iii) *Resolver interface circuit* which converts a three-phase output signal from a resolver [ILC73] into an analog velocity signal as well as standard two phase quadrature pulses [Snyd85]. The latter can be used to keep a digital count of increments in the motor's angular position. The resolver unit is an electro-mechanical transducer coupled to the motor shaft which generates a three-phase amplitude modulated signal. The modulation varies as a function of the position of the shaft.

2.5.2 Circuits Provided by the User

The user must provide the circuits to connect the controller with the interface unit. These include the following for each joint: a D/A for converting the digital output of the controller into an analog command signal; an A/D for converting the velocity signal into digital form; and a two-phase quadrature decoder-accumulator to keep a digital count of the angular position. Since the output of the resolver interface circuit generates 153,600 counts per revolution corresponding to a resolution of 8.5 arc-seconds, the decoder must be able to hold a count of 18 bits (note the joint motion is limited to $\pm\pi$).

In addition to interfacing with the motor, the controller must also communicate with a host computer. The host computer provides the coefficients and constants needed by the controller as well as the reference position data corresponding to the desired trajectory. All the interface circuits discussed so far perform input/output operations. Certain applications may also require the ability to directly manipulate the control flow of the program through external signals.

The robot controller application may potentially use three external control signals. A *sync* signal, a *reset* signal, and an I-control select signal (see section 2.4.1). A sync signal is useful for synchronizing the sample period of the controller with an external signal, such as a global synchronizing signal or a completion signal from

an I/O device. The sync signal can also be generated from a timer to ensure each sample period is of a specified, fixed duration. A reset signal, on the other hand, is used to force the controller to start up in a known state. This is important in robot control applications where an unknown initial state can be hazardous to the operation of the mechanical parts. An external control signal is also required for selecting or turning off the I-control section of the PID controller.

The discussion in this section shows the need for considerable amount of hardware overhead for integrating the controller into the robot system. In fact, I/O requirements were primarily responsible for performance bottle-neck and hardware overhead in implementing [Anwr87] the controller using a typical TMS32010 board (see figure 2.7 for a block diagram of the controller). Over 70% of the sample period (0.7ms) was used for performing I/O and data transfers between the various processors. The PC-AT #1 executes the P-control section of the PID controller while the TMS32010 executes rest of the algorithm. The P-control which acts inside the position loop may execute slower than the adaptation and D-control which are in the velocity loop. The second PC-AT provides communication between PC-AT #1 and TMS32010. In addition, PC-AT #2 performs most of the I/O operations. Since the robot has two joints, a second set of A/Ds, D/A, and quadrature decoder is also provided. The controller shown in figure 2.7 may not be the most efficient design; nevertheless, this is an actual system put together by a robotics group. The design clearly underlines the fact that application designers tend to use off-the-shelf, easily configurable components rather than spend effort on board-level design. The interface unit provided by the manufacturer of the motors also adds to the overall system's cost and size. Clearly, a custom circuit that meets the specific I/O and system interface requirements of the application would significantly reduce the size, cost, and design effort needed for the complete system.

2.5.3 Decoding Position Information from Resolver Output

An example of special I/O circuit for the robot control application is the *resolver* interface circuit for measuring the joint's position. Each joint of the robot

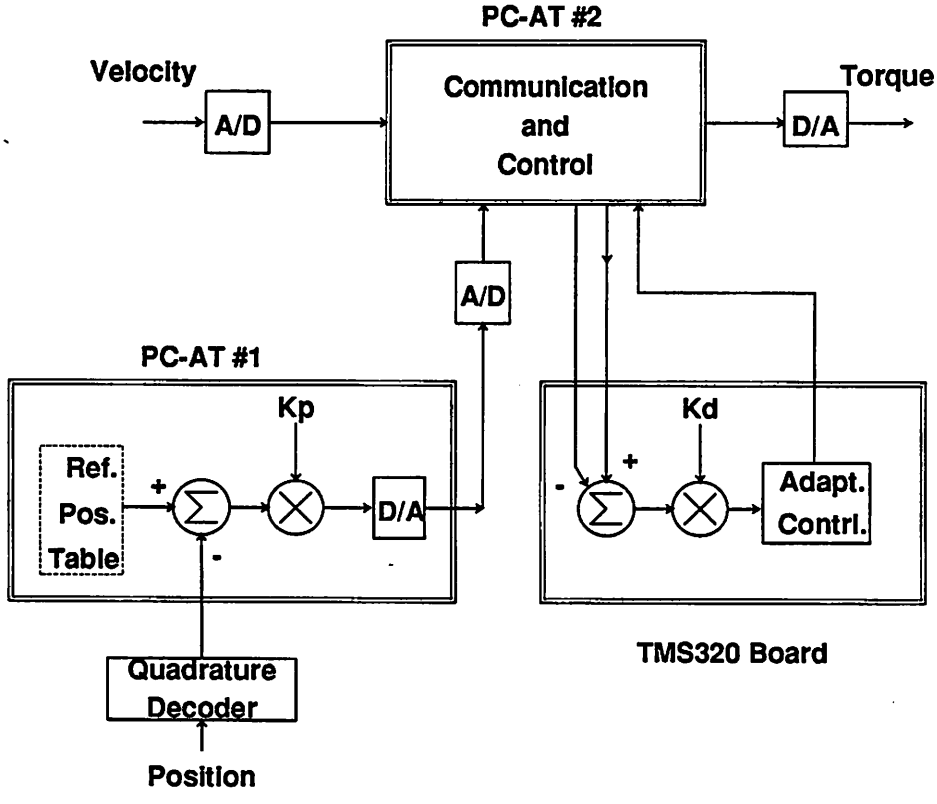


Figure 2.7: A TMS32010 based system implementing the adaptive control algorithm.

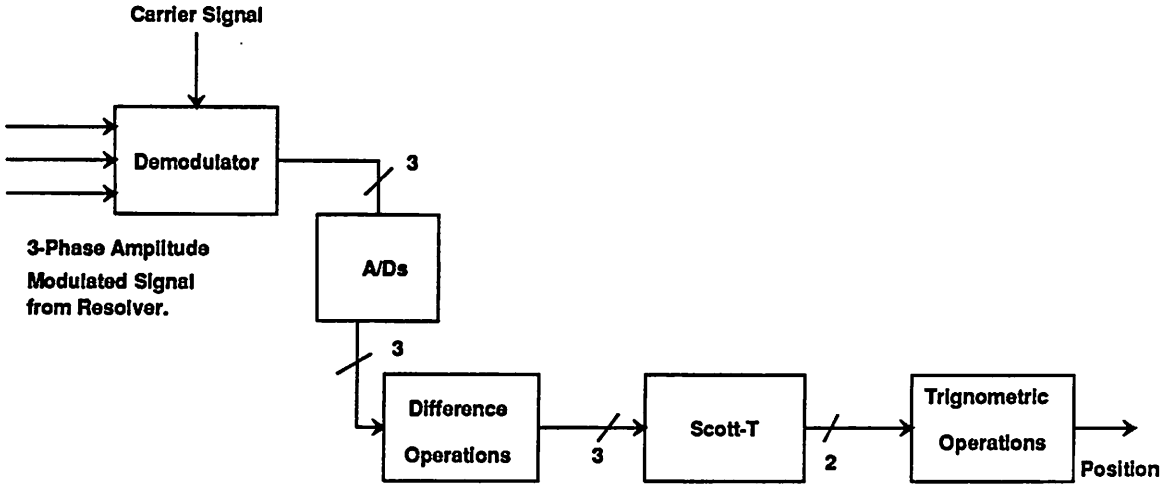


Figure 2.8: Block diagram showing the operations required to obtain the position information from the resolver output.

arm from NSK/Motornetics is fitted with a Reactasyn resolver [Ira84]. The resolver generates a three-phase amplitude modulated signal on a 6Khz carrier signal. The relative position of the joint is encoded in the modulating signal's phase. One revolution of the joint results in 150 cycles of the modulating signal. Thus the relative joint position within a 1/150 th sector of a complete rotation can be determined. By keeping count of the digital pulses from the decoded signal, the absolute position with respect to some reference position can also be found. Furthermore, since the maximum speed of the joint is 1 revolution per second, the maximum frequency of the modulating signal is 150 cycles/sec. The robot arm from NSK/Moternetics uses a fairly complicated hardware [Ira84] for obtaining the position information in digital form from the three-phase resolver output. The main components of the hardware are op-amps, electronic Scott-T [ILC73] circuit, and resolver-to-digital converter (RDC) [ILC73]. A more cost-effective solution, however, can be obtained using custom VLSI. We discuss below a possible custom implementation for generating digital position data from the resolver signal.

Figure 2.8 shows the block diagram of the position decoder. The input signals to the demodulator are,

$$
\begin{aligned}
V_a &= K_a(1 + m_a Sin\theta)Sin\omega_c t \\
V_b &= K_b(1 + m_b Sin(\theta - 120))Sin\omega_c t \\
V_c &= K_c(1 + m_c Sin(\theta - 240))Sin\omega_c t
\end{aligned} \tag{2.26}
$$

where θ is the angular position we wish to obtain; ω_c is the carrier frequency; K_a, K_b, K_c are gain values; and m_a, m_b, m_c are the modulation indices. These three signals from the resolver are first demodulated and then digitized. By taking differences of the sampled signals we obtain the following signals,

$$
\begin{aligned}
V_{ab} &= V_a - V_b = E.Sin\theta \\
V_{bc} &= V_b - V_c = E.Sin(\theta - 120) \\
V_{ca} &= V_c - V_a = E.Sin(\theta - 240)
\end{aligned} \tag{2.27}
$$

where we have assumed $K_a = K_b = K_c = K$, $m_a = m_b - m_c = m$, and $E = (\sqrt{3}).K.m$ Next the Scott-T transformer performs the following operations to generate a two phase signal.

$$
\begin{aligned}
V_x &= V_{ab} = E.Sin\theta \\
V_y &= V_{bc} - V_{ca} = (\sqrt{3}).E.Cos\theta
\end{aligned} \tag{2.28}
$$

Finally, we obtain θ by trigonometric manipulation.

$$
\theta = Tan^{-}1(V_x/V_y') \tag{2.29}
$$

where $V_y' = V_y/(\sqrt{3})/E$. All the above algebraic and trignometric operations can be performed in a microcoded processor such as the one described in this dissertation.

2.6 Algorithm Development, Testing and Debugging

Although extensive simulations of the algorithm and the hardware must precede an actual custom implementation, *in situ* development efforts is still necessary for ensuring satisfactory performance. This is especially true for robot control applications where the robot system cannot be precisely modelled. Therefore, the hardware should allow the user to change the coefficients and constants such as the P, I, and D gains; dead band limit; etc. Another desirable feature is the ability to experiment with different forms of the algorithm, such as turning off the I action of the PID controller.

The controller chip may also fail to perform satisfactorily because of design and/or process problems. Consequently the design must facilitate testing and debugging of the chip, once it is fabricated. This requires the ability to get the processor into a known state as well as the ability to observe and control internal states of the processor for isolating problem areas.

Chapter 3

Architecture Design of the Custom Digital Signal Processor

3.1 Overview of Digital Signal Processors

Digital signal processors are electronic components useful for building systems that are intended for performing intense arithmetic computations. DSPs achieve high through-puts through various architectural techniques, which include parallel functional units, pipelining, dedicated hardware for computationally intensive operations, multiple memories, and multiple busses.

Most general purpose DSPs are built around the so called Harvard architecture. Instead of a single memory, as in traditional Von Neumann architectures, the Harvard architecture has separate program memory and data memory. This, combined with separate program and data busses, allows simultaneous access to program instructions and data. The TMS32010 [TMS83] from Texas Instruments, one of the first generation DSPs on a single chip, uses the Harvard architecture with a modification that allows data transfer between the program bus and the data bus for greater flexibility. Some of the more recent DSPs (for example DSP56000 [Mo86]) have gone for an architecture based on three memory units: one program memory and two data/coefficient memories. With the ability to access two pieces of data in the same cycle this type of architecture is very efficient in implementing the

multiply-accumulate operation,

$$y(k) = b \times x(k) \tag{3.1}$$

However, the implicit assumption that the two operands, b and $x(k)$ reside in different memories may not always be true. In addition to providing multiple memory units operating in parallel, DSPs also have separate address computation hardware.

Address computation units are dedicated data paths for computing the memory addresses of operands in parallel with the operations of the main data path. The address units usually have their own alu, registers, pointers, and stack. A very popular feature of the address units is built-in hardware for handling modulo addressing useful in accessing and updating circular buffers and data queues. Whereas the address unit increases through-put by allowing address computation in parallel with the operations of the main data path, pipeline stages in the processor allow several instructions to be processed simultaneously.

The TMS32020, for example, has a three stage pipeline:

Instruction Fetch

Instruction Decode

Instruction Execute

With this scheme, execution of three instructions can proceed simultaneously. Operations in the execution stage include *add, multiply, shift, load register,* etc. The more advanced TMS320C30 has four stages of pipeline for increasing through-puts. One disadvantage of pipeline, though, is pipeline bubbles caused by conditional branch operations [LeeJa84]. Since the result of a branch operation cannot always be predicted in advance, the pipeline must be flushed before a new instruction following a branch can be executed. The conditional branch operations are also used in executing program loops. The TMS32020 makes the flushing operation transparent to the user by specifying two instruction cycles for all conditional branches. The TMS320C30, on the other hand, permits delayed branch [LiFrSi87]. Another disadvantage of pipelines is seen in executing program loops. Every instruction

in the loop must be fetched and decoded even though the same instruction or set of instructions are repeated on each iteration of the loop. In order to overcome this problem, many commercial DSPs use special techniques. The TMS32020 has a repeat counter which allows a single instruction to be repeated up to 256 times without the need for fetch and decode except on the first iteration. DSP56000 from Motorola allows loops of multiple instructions without any instruction cycle overhead for change-of-flow. This is achieved at the cost of a loop counter and a loop-end address register as well as making use of a stack. Although most commercial DSPs provide some way of efficiently executing loops, other control flow type instructions such as conditional branch and subroutine call/return still require certain number of instruction cycles to execute. As discussed in chapter 2, the robot control application requires efficient support for control flow type operations. In this chapter the architecture of the robot controller is discussed, which among other things, supports efficient execution of control flow operations.

3.2 Architectural Model for the Custom Processor

Based on the analysis of the robot controller application, presented in chapter 2, an architectural model for the custom processor has been defined. This model comprises of three main parts:

- One or more application specific *data paths* for executing arithmetic and logical operations.
- A high-performance, generic *processor control unit* that determines the operations in the data paths and the sequence in which they are executed.
- *Memory units* associated with the data paths for data storage, such as RAMs and register files.

In developing an implementation of the architecture model we have tried to take advantage of the existing LagerI architecture [Pope85]. However, as described in this

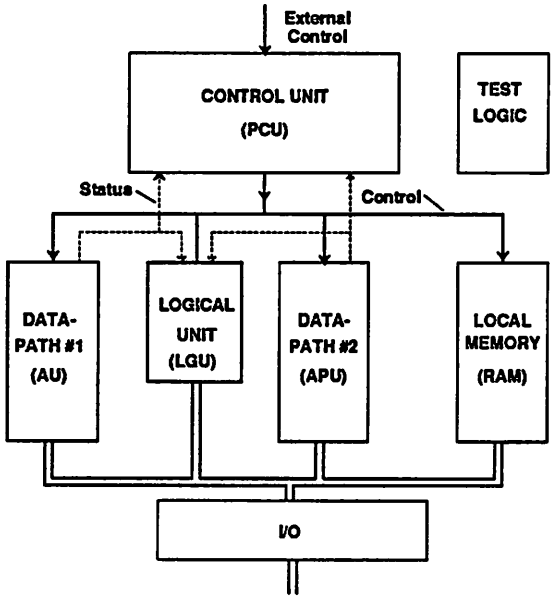


Figure 3.1: Processor architecture for the robot controller

chapter, major changes and additions have been made for better performance. The actual implementation of the architecture not only aims at efficiently supporting the robot control algorithm, but also making the various functional blocks highly modular and customizable so that they can be easily adapted for other applications.

As shown in figure 3.1, the architecture [Azim88] [1] consists of several customizable functional blocks listed below:

- Processor Control Unit (PCU) - includes instruction memory
- Arithmetic Unit Data Path (AU)
- Address Processing Unit Data Path (APU)
- Data Memory (RAM)
- Logical Unit (LGU)
- Test Logic

[1] In some Lager literature, this is referred to as the KAPPA architecture.

- I/O

The instruction memory or *control store* inside the control unit holds the application program and issues a new instruction on every clock cycle. This instruction specifies the operations to be executed in the data paths, memory, and other functional blocks except Test-logic. The Test-logic block is a dedicated hardware block used during testing and does not operate under program control. A description of the test-logic and the testing strategy is given in chapter 6, whereas the rest of the hardware blocks are described in this chapter.

3.3 Processor Control Unit

The processor control unit (PCU) controls the instruction by instruction execution of the application program. Whereas the design of the data paths determine how well the arithmetic operations are executed, the PCU's design determines how well the control flow operations are executed. The LagerI architecture used a very simple design for the control unit. It consisted of a program rom containing horizontal instruction words, a program counter to sequence through the instructions, and a sub-program counter to execute multiple iterations of any one section of the rom code. Sub-routines that can be called from different sections of the main program and branch operations were not supported. Clearly, the requirements of the robot controller, as discussed in chapter 2, dictates a major redesign of the control unit. The design of the PCU is aimed at achieving two major goals: (i) efficient implementation of the control flow operations; (ii) Support for automatic generation of the control unit from high-level behavioral descriptions of the processor. In the following sub-sections, the hardware design of the PCU is described, whereas in chapter 5 the automatic generation of the control unit is described.

3.3.1 A Conceptual Framework for the Control Unit Design

The design of the control unit presented here evolves naturally from a finite state diagram representation of algorithms. Figure 3.2 shows a partial state

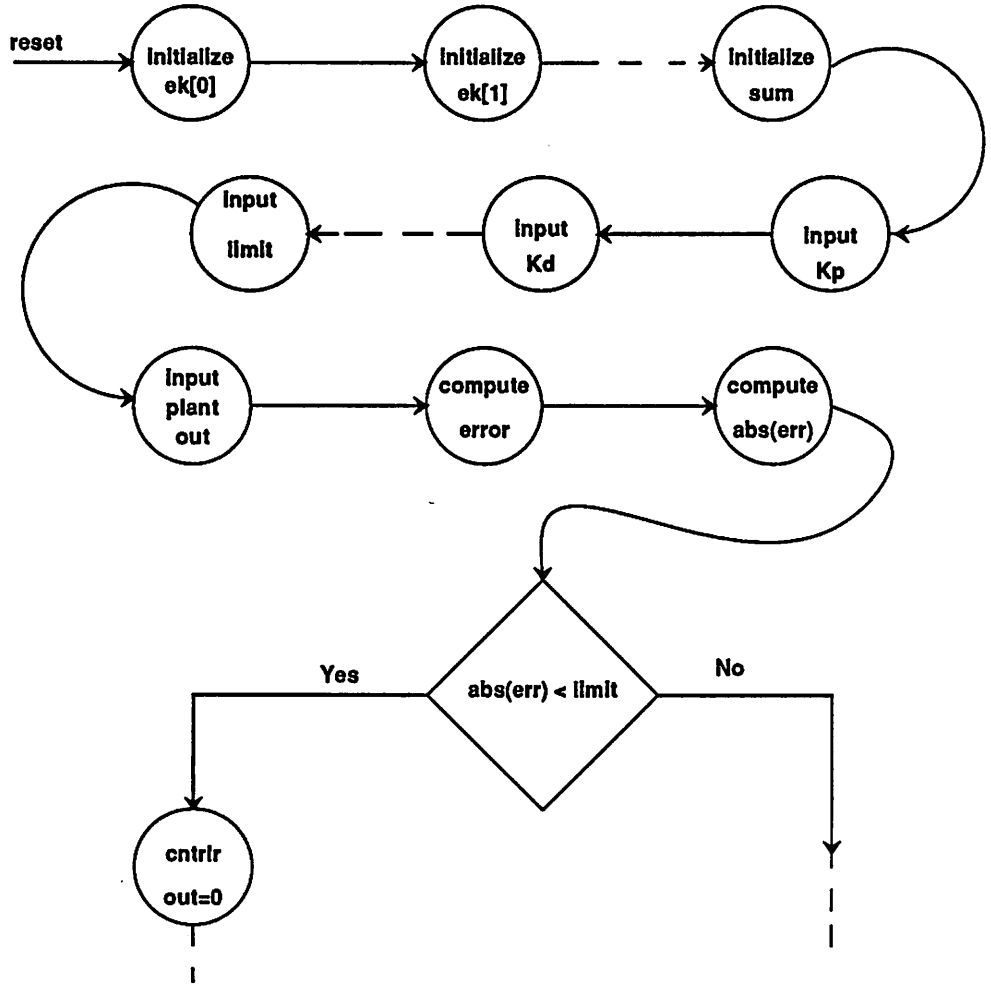


Figure 3.2: A partial state transition diagram for a PID control algorithm

transition diagram of a PID controller.

The corresponding state table is shown in table 3.1. The output of the machine represents the next state and the actions taken during each state. When the conditions necessary for making a state transition are satisfied (usually meaning completion of all the desired actions), the indicated transition takes place. One could conceptually identify each state with a machine instruction and therefore, the entire state table can be regarded as a machine-level program. Such a state table can be implemented on a finite state machine (fsm) using combinational logic elements and flip flops or with pla's and flip flops. This, in principle, can serve as

Present State	Conditions	Next State	Action
INIT ek[0]	task complete	INIT ek[1]	initialize ek[0]
INIT ek[1]	task complete	INIT SUM	initialize ek[1]
INIT SUM	task complete	READ Kp	initialize sum
READ Kp	task complete	READ Kd	read in Kp
...	...	...	...
READ LIMIT	task complete	GET PLANT-OUT	read in error limit
GET PLANT-OUT	task complete	COMPUTE ERR	read plant output
COMPUTE ERR	task complete	COMPUTE ABS(ERR)	compute error
COMPUTE ABS(ERR)	task complete and (abserr < limit)	CNTRLR OUT=0	compute abs(err)
COMPUTE ABS(ERR)	task complete and (abserr $\geq$ limit)	...	...

Table 3.1: State table for the PID state transition diagram.

the control unit.

A straight forward fsm based implementation, however, is inefficient for most applications. It does not take advantage of program-characteristics such as ordering of the state table, repetition of a group of state transitions, etc. Practical realization of the state machine is greatly facilitated by ordering the state table such that most of the state transitions are effected by simply going to the next entry in the state table. Such an ordered state table is characterized by two types of transitions: (i) sequential transitions; and (ii) out-of-sequence transitions. In practice, programs are generally composed of blocks of sequential state transitions followed by an out-of-sequence state transition. This leads to a program model based on separation of the sequential and out-of-sequence state transitions as shown in figure 3.3. The blocks of sequentially executing states can be identified with blocks of straight-line machine instructions specifying data manipulation operations (e.g. add, load), whereas the out-of-sequence transitions can be identified with control flow operations (e.g branch).

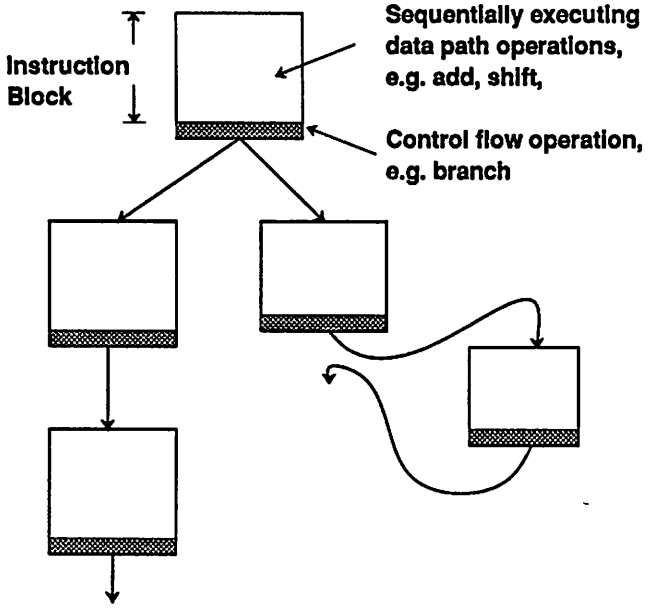


Figure 3.3: Program model showing data path operations and control flow operations.

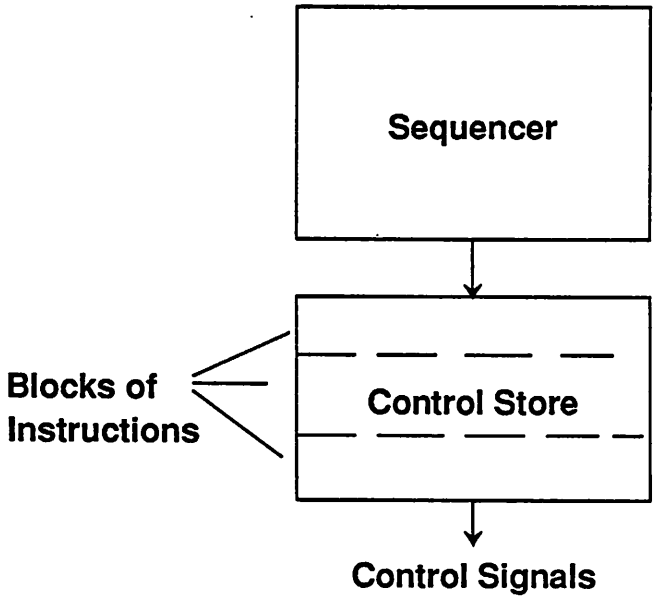


Figure 3.4: Hardware model for the control unit.

The program model described here leads to a hardware model of the control unit. This model (figure 3.4) consists of a *sequencer* which embodies the control flow structure of the program and a *control store* which holds blocks of straight-line machine instructions specifying data path operations. Based on this hardware model, a simple control unit can be designed.

3.3.2 Functional Description of the Control Unit

An implementation of the basic control unit is shown in Figure 3.5. The sequencer is implemented with a finite state machine and a program counter (pc), whereas the control store is implemented with a pla. The fsm's output for the basic control unit has two fields: a new-state field and an address field which points to an instruction block in the control store. The program counter sequences through the individual instructions of the specified block. These instructions are stored as horizontal control words whose bit-fields are control signals for executing data path, memory, and I/O micro-operations. As shown in figure 3.6, the control word has three main fields: data path control bits; EOB2; and EOB. The end-of-block bit, EOB, in the control word is set in the last instruction of each block in the control store. This bit resets the program counter. Since the program counter always counts up and is reset to zero by EOB, no additional hardware or control is required to determine when the program counter should be reset.

The size of the program counter should be large enough to sequence through the largest block. Since the size of the blocks vary, the control store may have sections of unused locations. This is not a major problem as the unused locations can be deleted during layout generation. The control store's address word-size however, may be slightly larger than the minimum required. The address word-size is given by,

$$A_s = log_2(N_b) + log_2(B_{max}) \tag{3.2}$$

where N_b is the number of blocks and B_{max} is the number of instructions in the largest block. Irrespective of the size of the block, on execution of the last instruction

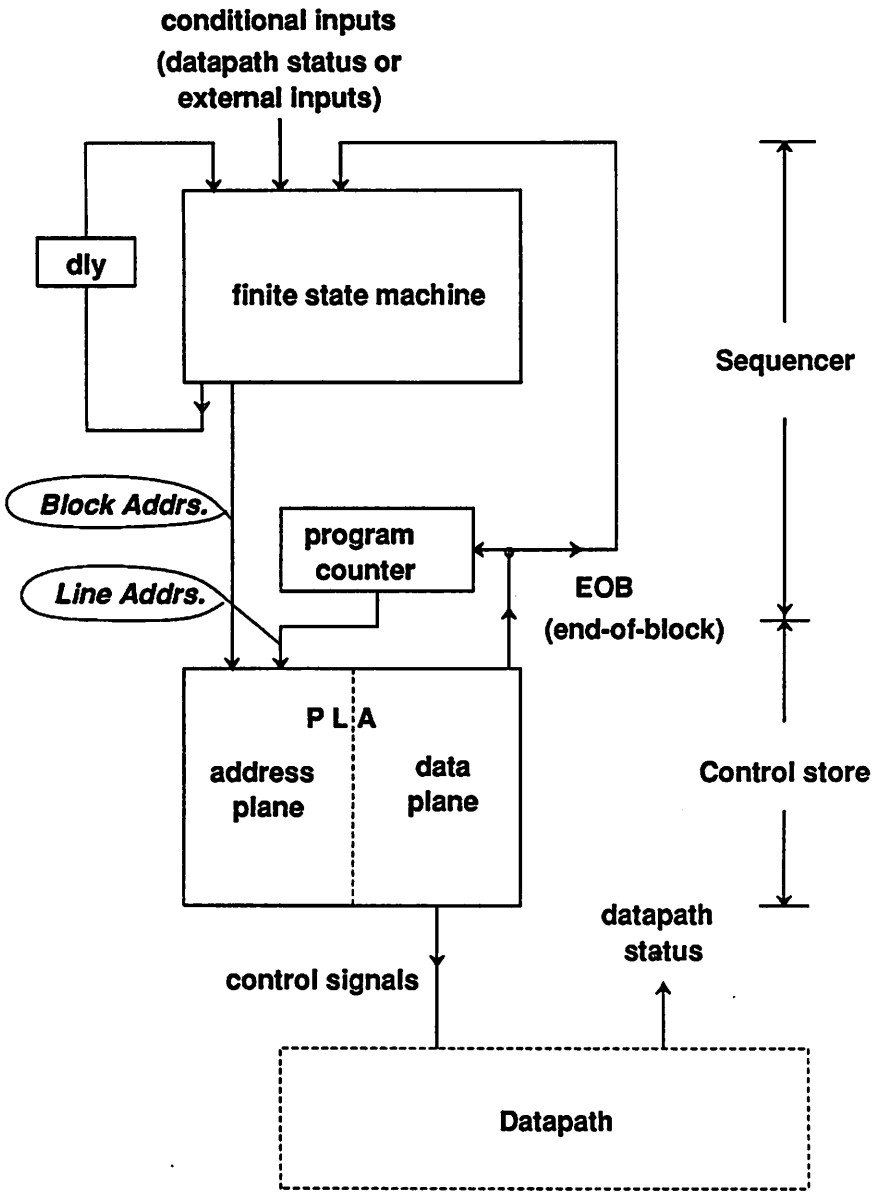


Figure 3.5: Block diagram of the basic control unit. The higher order bits (block address) of the control store address are provided by the fsm and the lower order bits (instruction line address) are provided by the program counter.

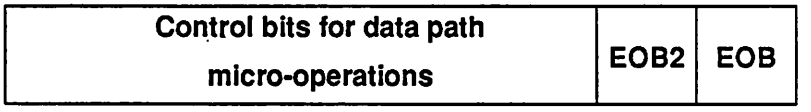


Figure 3.6: Bit-fields of the control-word. EOB specifies end of an instruction block whereas EOB2 is used during subroutine returns.

of a block, the EOB bit in the control word triggers a state transition in the fsm.

The state transitions in the fsm actually take place in two steps: first, the EOB causes the fsm to go into a *change-over* state; second, as EOB goes away after one cycle, the fsm goes into a new *hold* state. Upon transition, a new block address pointer is specified in the change-over state as well as the new hold state. This allows the first instruction of the new block to be executed immediately following the cycle in which EOB is asserted (details about the circuit implementation is discussed in chapter 5). When an instruction block has only one instruction, the hold state is eliminated. The use of change-over state is shown in figure 3.7 and as the fsm state-table shows, each control flow operation requires at least two entries.

As described in the preceding paragraphs, the use of a finite state machine for implementing the control flow operations as state transitions, permits change-of-flow from one block of instructions to another without requiring any instruction cycle overhead. This is in contrast with traditional implementation of control instructions, which like all other instructions, require a certain number of cycles for fetching, decoding, and executing each instruction. The control instructions implemented in the basic control unit shown in figure 3.5 are multi-way branches and subroutines.

Multi-way Branches

Multi-way branch simply means selecting one of N possible state transitions, based on input signals to the fsm. Therefore, such a branch is also executed without any instruction cycle overhead. The change-over states and the correspond-

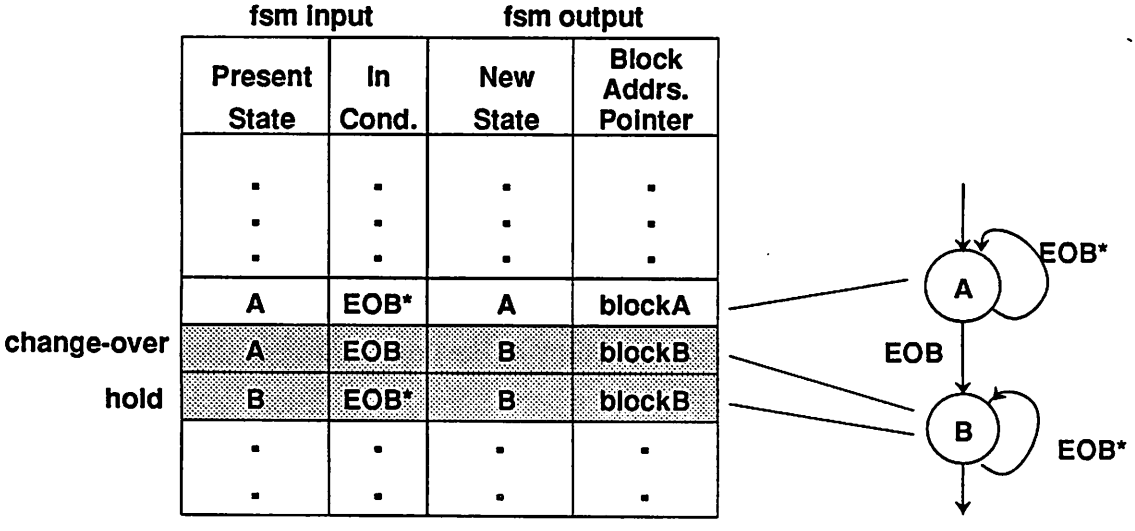


Figure 3.7: Fsm state transitions and state-table. Each transition requires two entries in the table: change-over and hold. In the change-over state, which lasts for only one cycle following EOB, the first instruction of blockB is executed. The remaining instructions of blockB are executed during hold state of B.

Present State	Input Conditions	New State	Block Addrs. Pointer
.	.	.	.
A	EOB.X1	B1	blockB1
A	EOB.X2	B2	blockB2
A	EOB.X3	B3	blockB3
.	.	.	.

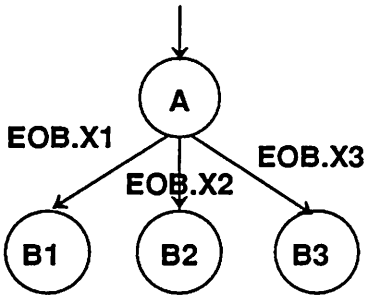


Figure 3.8: Change-over states and the corresponding state table entries for a multi-way branch. Entries for the hold states are not shown.

ing state table entries for a multi-way branch are shown in figure 3.8. An important feature of the control unit to note here is the ability to have external control signals directly effect state transitions without any software or major hardware overhead as in traditional interrupt schemes. This feature is similar to TMS320 family's $\overline{BIO}$ control input. Unlike the TMS320 however, any desired number of external control inputs can be provided by specifying them during layout generation. On the other hand the state transitions take place only at the end of a block; whereas in traditional interrupt schemes, the machine can be interrupted at any instruction during execution of a program.

Subroutines

For executing subroutines, the block pointer in the fsm's output points to the desired subroutine block (figure 3.9). In this way the same block of instructions can be specified from different states, thus avoiding the need for having multiple copies of the same block. Further more, the transitions to and from a subroutine state may also be multi-way, conditional branch operations. The reader should note, however, the various states pointing to the same subroutine cannot be merged, since their next state transitions are not necessarily same. Moreover, nested subroutines

Present State	Input Cond.	New State	Block Addrs. Pointer
•	•	•	•
A	EOB	S1	blockS
S1	EOB*	S1	blockS
S1	EOB	R1	blockR1
•	•	•	•
C	EOB	S2	blockS
S2	EOB*	S2	blockS
S2	EOB	R2	blockR2
•	•	•	•

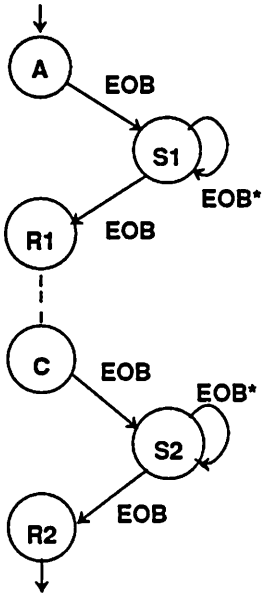


Figure 3.9: Implementation of limited subroutine capability without using stacks. Figure showing both subroutine states S1 and S2 pointing to the same instruction block, blockS. Because they return to different states R1 and R2 respectively, the two states cannot be merged.

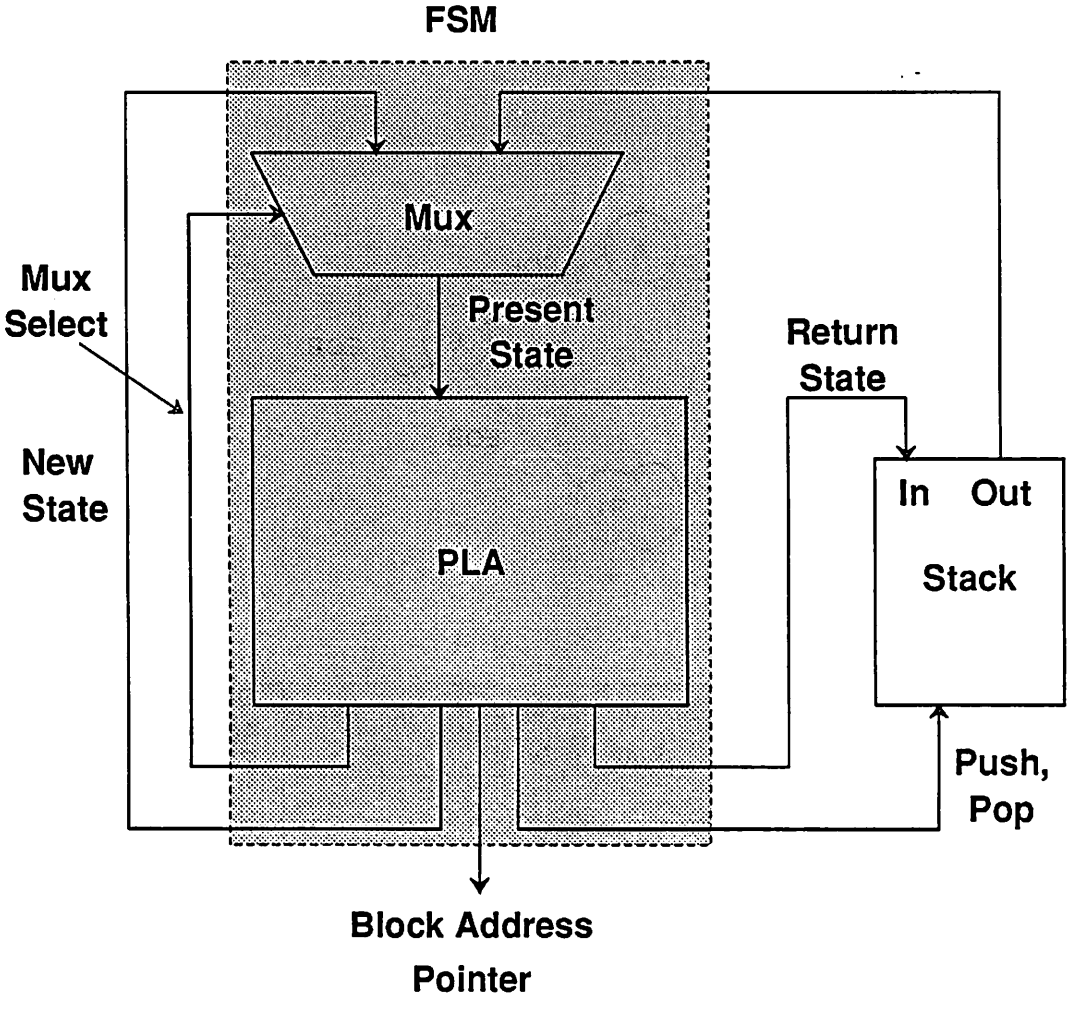
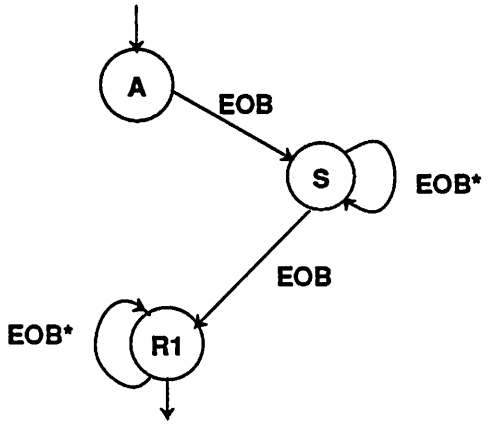


Figure 3.10: Figure showing the addition of a mux and a stack to the PCU for supporting subroutine. The control store and program counter are not shown.

cannot be handled – they must be flattened. In order to reduce the number of states and allow nested subroutines, a hardware stack is added. This permits merging of all the states pointing to the same subroutine.

For incorporating a subroutine stack, a mux is included in the PCU circuit to select the new state as shown in figure 3.10. Moreover, four extra fields must be added to the fsm output: a subroutine return-state field, a push-stack control bit, a pop-stack control bit, and a mux-select control bit (see figure 3.11). While making a transition to a subroutine state, the return-state is pushed on to the stack. On



Present State	In Cond.	New State	Return State	Mux Select	Pop	Push	Block Addrs. Pointer
•	•	•	•	•	•	•	•
A	EOB	S	R1	fsm	0	1	blockS
S	EOB*	S	X	fsm	0	0	blockS
S	EOB	D	X	stack	1	0	blockD
R1	EOB*	R1	X	fsm	0	0	blockR1
•	•	•	•	•	•	•	•

Dummy Change-over

Figure 3.11: State transition diagram and table illustrating use of stack for subroutines. Notice a dummy change-over state has been used for returning from the subroutine S. This replaces the usual change-over state required for making a transition to state R1. X = don't care; Mux Select = fsm, means select fsm's new state output; Mux Select = stack, means select the stack output; Push/Pop = 1, means enable signal; and Push/Pop = 0, means disable signal.

the other hand, while returning from the subroutine state, the fsm first goes into a dummy change-over state during which the stack is popped and the mux is switched to select the return-state output from the stack. The instruction block pointed to by the dummy state contains a single no-op instruction. However, the EOB bit is not set during this instruction in order to avoid an unwanted state transition. Instead, the EOB2 bit is set for keeping the program counter reset. Following the dummy state, the fsm goes into the return-state. The insertion of the dummy state results in one extra cycle when returning from subroutines (using stack). This also causes an additional entry in the state-table for each subroutine. However, no change-over state is required for the return state itself. As a result, the total number of entries in the state-table for invoking subroutine states is reduced. The savings is even greater when nested subroutines are present, as explained below.

Let P_i be the number of times the ith subroutine is called (a subroutine may be a single subroutine state or a unique group of states treated as a subroutine). Also assume N_i to be the number of states (*excluding any nested subroutine[s]*) in each unique group of subroutines. The contribution made by the ith subroutine to the total number of entries in the state-table can be computed as follows:

P_i: Add one table entry for the change-over state, for each transition (call) to the subroutine.

1: Add one table entry for the hold state of the first state in each subroutine.

1: Add one table entry for the change-over state for returning from the subroutine

$-P_i$: Subtract one table entry in order to compensate for the change-over state counted for each return state. (remember subroutine return states do not have a change-over state.)

$2(N_i - 1)$: Add two table entries for every state, excluding the first state, in the subroutine.

Thus the number of state-table entries due to subroutines is:

$$\begin{aligned}\sum_{i=1}^{k} C_i &= \sum_i (P_i + 1) + (1 - P_i) + 2(N_i - 1) \\ &= \sum_i 2 + 2(N_i) - 2 \\ &= \sum_i 2.N_i\end{aligned}$$

C_i is the number of entries in the state-table corresponding to each subroutine i, and k is the number of unique subroutines. As the equation shows, the number of state-table entries is independent of the number of times a subroutine is called or the level of nestings. Moreover, since the number of state entries for the top-level states is also equal to $2 \times N_{top}$, the total number of state-table entries is easily found by counting the number of states in each unique group of states (including both the top-level and the subroutines) and multiplying by two. This is illustrated in figure 3.12. Note, in the above analysis we have assumed return from subroutines never jumps across different levels of nesting. If this is not the case then dummy states must be inserted in order to ensure a return state for each call.

One drawback of using a stack for subroutines is the difficulty of executing a multi-way branch while returning from a subroutine, since only one return-state address can be pushed or popped from the stack during a state transition. Multiple return addresses may be provided with a wider stack, multiple return-state fields in the fsm output, and additional circuit complexity. Whereas the addition of a stack to the basic control unit provides a mechanism for handling subroutine states, another important extension of the basic control unit is a hardware loop counter.

Loops

As discussed in section 3.1, many commercial DSPs provide support for executing zero instruction-cycle overhead loops. However, zero overhead loops are useful only for short loops and result in negligible improvement in loops with a large number of instructions and/or involving complex control flow operations. Consequently, we have provided a hardware loop counter in the PCU for executing loops involving a single block of instructions. For more complex loops, data path registers may be used for loop counters, as described later. The PCU loop counter adds another bit, called the Incr bit, in the fsm output. (See figure 3.13.) This

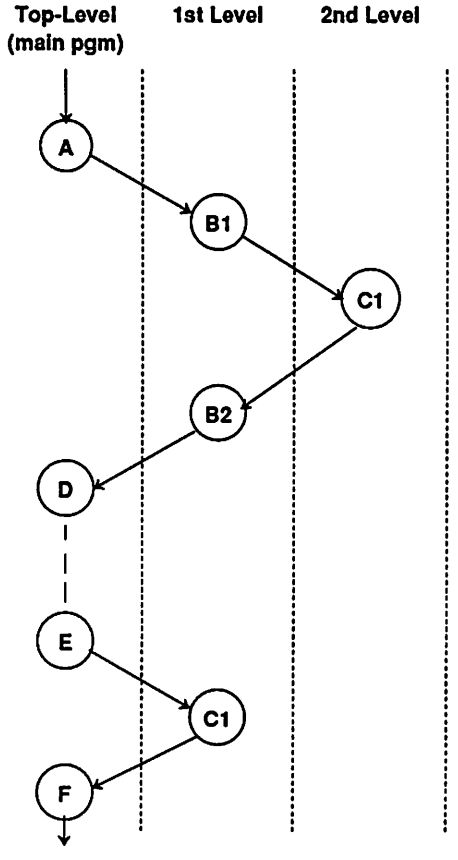
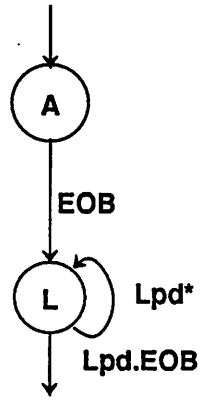


Figure 3.12: State diagram showing two levels of subroutine nesting. Total number of state entries is 4×2 (for top-level states) $+2 \times 2$ (for subroutine group B1-B2) $+1 \times 2$ (for subroutine C1) $= 14$. A flattened state diagram would require $8 \times 2 = 16$ state entries.



Present State	Input Cond.	New State	Loop Incr	Return State	Mux Select	Pop	Push	Block Addrs. Pointer
•	•	•	•	•	•	•	•	•
A	EOB	L	0	X	fsm	0	0	blockL
L	EOB*	L	1	X	fsm	0	0	blockL
L	EOB.Lpd*	L	1	X	fsm	0	0	blockL
L	EOB.Lpd	B	0	X	fsm	0	0	blockB
•	•	•	•	•	•	•	•	•

Figure 3.13: State diagram and table for executing loops. The Loop Incr. signal from fsm's output sets the loop counter in increment mode. When the specified number of iterations of the block is completed the loop-done signal, Lpd, is asserted by the counter, causing state transition out of the loop state.

Incr bit is set when the instruction block addressed by the fsm is to be executed more than once using the PCU loop counter. At the end of block the loop counter is incremented by one. A count-detect logic tests the output of the loop counter. When the pre-specified count is reached, a loop-done signal is asserted which triggers a new state transition. For each unique loop count to be detected, a separate count-detect logic and an associated loop-done signal is provided (see chapter 5 for details). We decided to use count-detect logic instead of using the counter in decrement mode and using carry-out signal. This avoids the need for setting the counter to an initial value from the fsm, which requires an additional field in the fsm output plane, as well as additional routing between the fsm and the counter. We also note in figure 3.13, the loop state, L, requires an additional row (for a total of three rows) in the state table. In many signal processing applications, in addition to loop counters, a counter for controlling the length of the sample period (in terms of number of cycles) may be needed. For this reason, the PCU design includes a timer.

Timer

The timer is built around a counter which counts up once every instruction cycle from an initial count value. This initial count is provided by a register which can be loaded from the data path under program control. Normally, the fsm executes the entire program once per sample period, at the end of which the fsm goes into a wait state. Meanwhile the timer keeps counting and when the maximum count is reached, a control signal is asserted and the counter resets to its initial count. The control signal from the timer also causes the fsm to restart the program. As shown in figure 3.14, a control bit is added in the fsm output in order to reset the timer. Usually the timer is reset during initialization states, which are done only once on start-up. The ability to reset the timer from the fsm output may be exploited for other uses too. For resetting the PCU, an external reset signal is provided which forces the fsm to go into a reset state and also resets the program counter.

A complete block diagram of the PCU is shown in figure 3.15. In the next

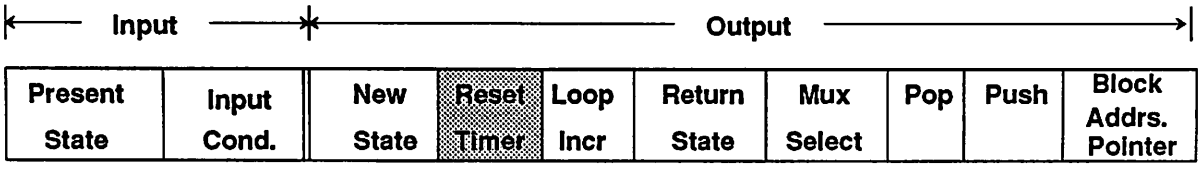


Figure 3.14: Fsm input and output fields. Timer reset control bit is high lighted.

section we compare the design of the PCU with alternative designs for implementing program-control functions. After describing, the control unit used in the Cathedral design system [Rab88] for custom DSP circuits, we examine a stand alone micro-sequencer chip recently announced by Altera Corporation [Alt87]. This chip is very interesting because its design has many similarities with our approach. Finally, implementation of control flow functions in two commercial DSP chips are discussed.

3.3.3 Comparison with Alternative Approaches to Control Unit Design for DSPs

In considering alternative designs for implementing program-control functions, we examine the following operations.

- Generation of next-instruction address
- Branch
- Loops
- Subroutines

(The description about the designs of the various control units have been obtained or deduced from information given in data sheets, manuals, and papers. These descriptions corroborate the hardware behavior; however, they may differ in actual implementation details.)

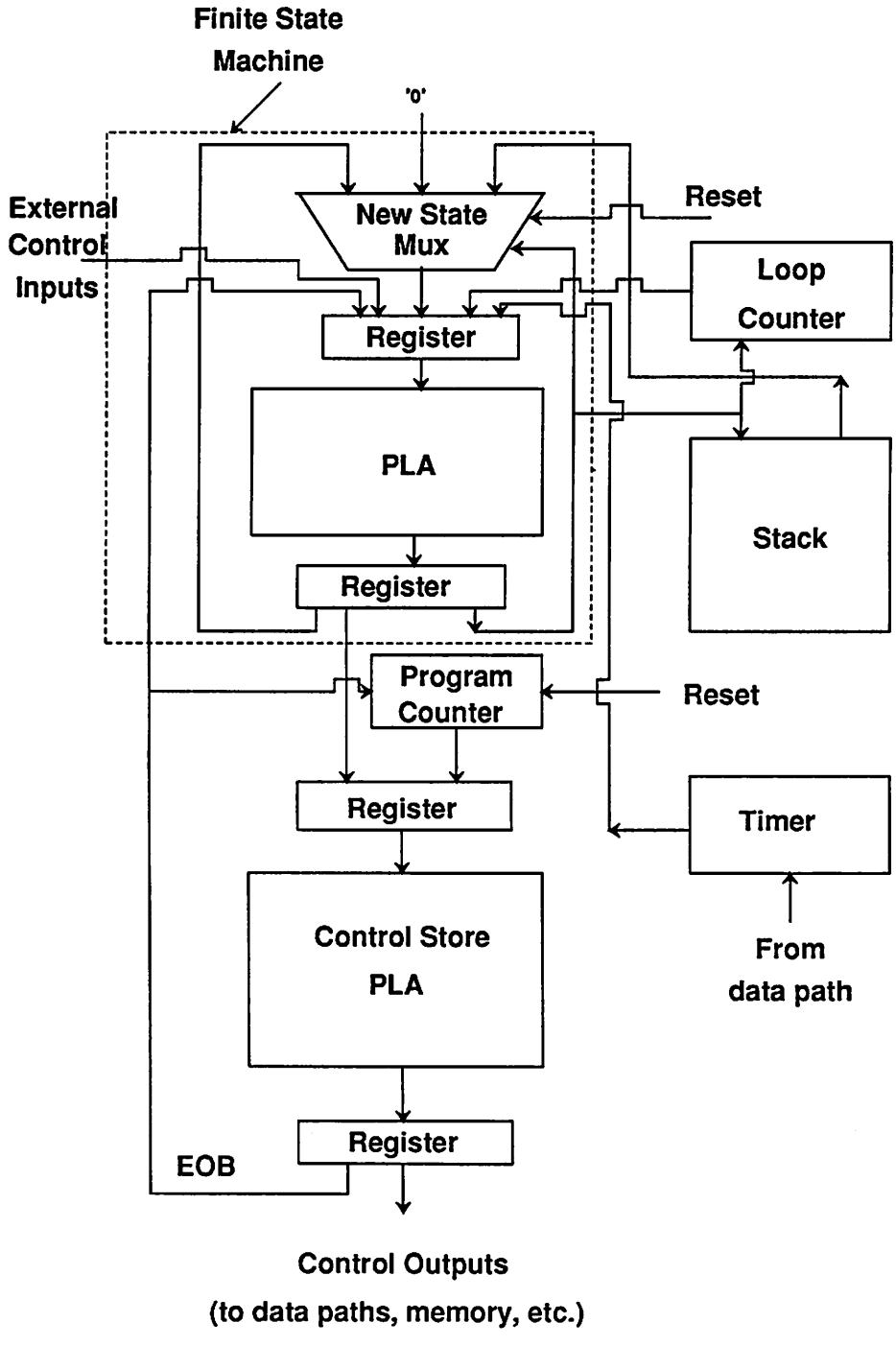


Figure 3.15: Block diagram of the complete PCU.

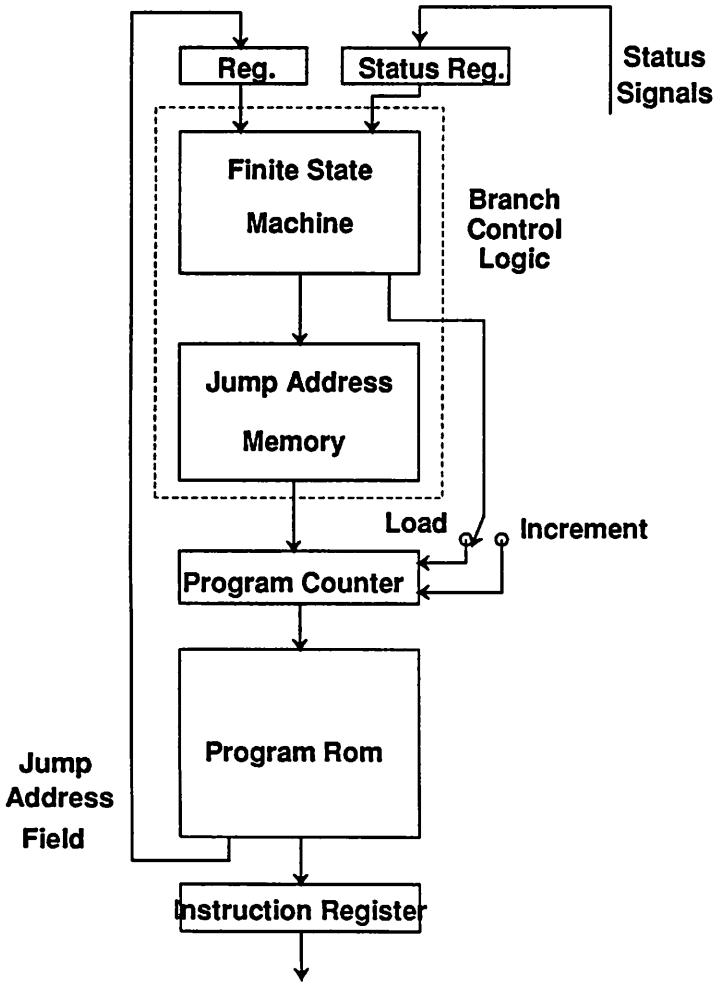


Figure 3.16: Control unit used in the Cathedral design system for DSP circuits.

Customizable Control Unit of the Cathedral Design System

The Cathedral design system is aimed at generating custom DSP circuits from a library of customizable functional blocks. An important functional block is the control unit which is described here. The block diagram of the control unit is shown in figure 3.16. The next-instruction address is usually obtained by incrementing the program counter. The addressed instruction is then loaded into an instruction register from where the control signals are applied to the data paths. For a change-of-flow operation, the branch control logic consisting of a finite state machine and a jump-address memory is used to generate the next-instruction ad-

dress. In order to execute a branch, the next-instruction address is encoded in the jump address field (JAF). The address of the branch target is eventually provided by the jump address memory and loaded into the program counter. The finite state machine is used to evaluate conditional branches and multi-way branches based on status signals from data path or other sources. Furthermore, the fsm also evaluates boolean expressions as in the logical unit (see section 3.5) of the robot controller. One of the fsm's output is used to choose between loading the program counter with a new address or incrementing the program counter for the next sequential address. A major difference between the Cathedral control unit and the PCU is in the pipeline scheme.

The Cathedral control unit has a two stage pipeline with the branch control logic forming one stage; the program counter and the rom forming the other stage. All branch addresses must be loaded into the program counter first. As a result a conditional branch takes two cycles. For unconditional branches, the branch instruction may be issued two cycles earlier (look-ahead) for avoiding the two-cycle penalty. In the PCU, on the other hand, the program counter and the finite state machine (implementing the control flow instructions) are in the same pipeline stage and jointly provide the rom's address. Consequently, conditional branches take only one cycle because of the data path pipeline stage. On the other hand unconditional branches take effect immediately without requiring any look-ahead scheme.

Unlike, all the other control units discussed in this section, no hardware loop counter is provided. Consequently program loops are implemented using the branch operation and registers/counters in the data paths. This may lead to instruction-cycle overhead in tight loops. Loop unfolding technique, however, is used by the compiler for avoiding the instruction-cycle penalty although such an approach is too inconvenient for direct programming by the user.

Subroutines are implemented by setting a special bit in the status register before branching to the subroutine. One bit is dedicated for each subroutine call. Thus during return, the status bits can be evaluated to choose the correct return (branch out of the subroutine) address. Since each subroutine call adds a bit to the status register, it can grow very large. Moreover, like branch, subroutine call/return

operations also require one extra cycle.

EPS448, Stand Alone Microsequencer

The EPS448 from Altera Corporation is designed as a stand alone microsequencer (SAM). Figure 3.17 shows a block diagram of the sequencer. The major blocks of the SAM are branch control logic, microcode memory (EPROM), loop counter, and stack. The microcode memory holds 'instructions' just like the control store of the PCU. Unlike the PCU, however, 20 of the 36 output bits of the memory are reserved for internal SAM operation. The bit-fields of the memory word are shown in figure 3.18. These 20 bits include next state addresses, op code, and tristate control of outputs. The remaining 16 bits are available for user-defined outputs to control, for instance, operations of a data path. The memory holds a total of 448 words whose addresses are generated by the branch control logic, which functions much like the sequencer of the PCU. However, the manner in which the next state address is generated has many important differences with the PCU's next state address generation mechanism.

The SAM does not use a program counter to generate the next state address; instead two possible next state addresses are specified in the D and Q fields (figure 3.18) of the microcode memory word itself. The mux inside the branch control logic selects one of the two addresses based on the status of the loop counter's Zero Flag and/or the microcode memory output's Op-code field. This restricts the SAM to executing a two-way branch for 192 of the 448 words of the microcode memory. Also, over 50% (20/36 bits) of the microcode memory is used for generating the next state address as opposed to only two bits (EOB and EOB2) in the PCU's control store. On the other hand, the SAM does not restrict change-of-flow operations at only block boundaries, as in the PCU. Looping and stack operations can be specified on any instruction. For multi-way branch, the remaining 256 (of the 448) microcode memory locations are organized into 64 groups of four words. A branch select EPLD (erasable programable logic device) selects one of the four words based on external inputs. Thus the external inputs can select only up to one

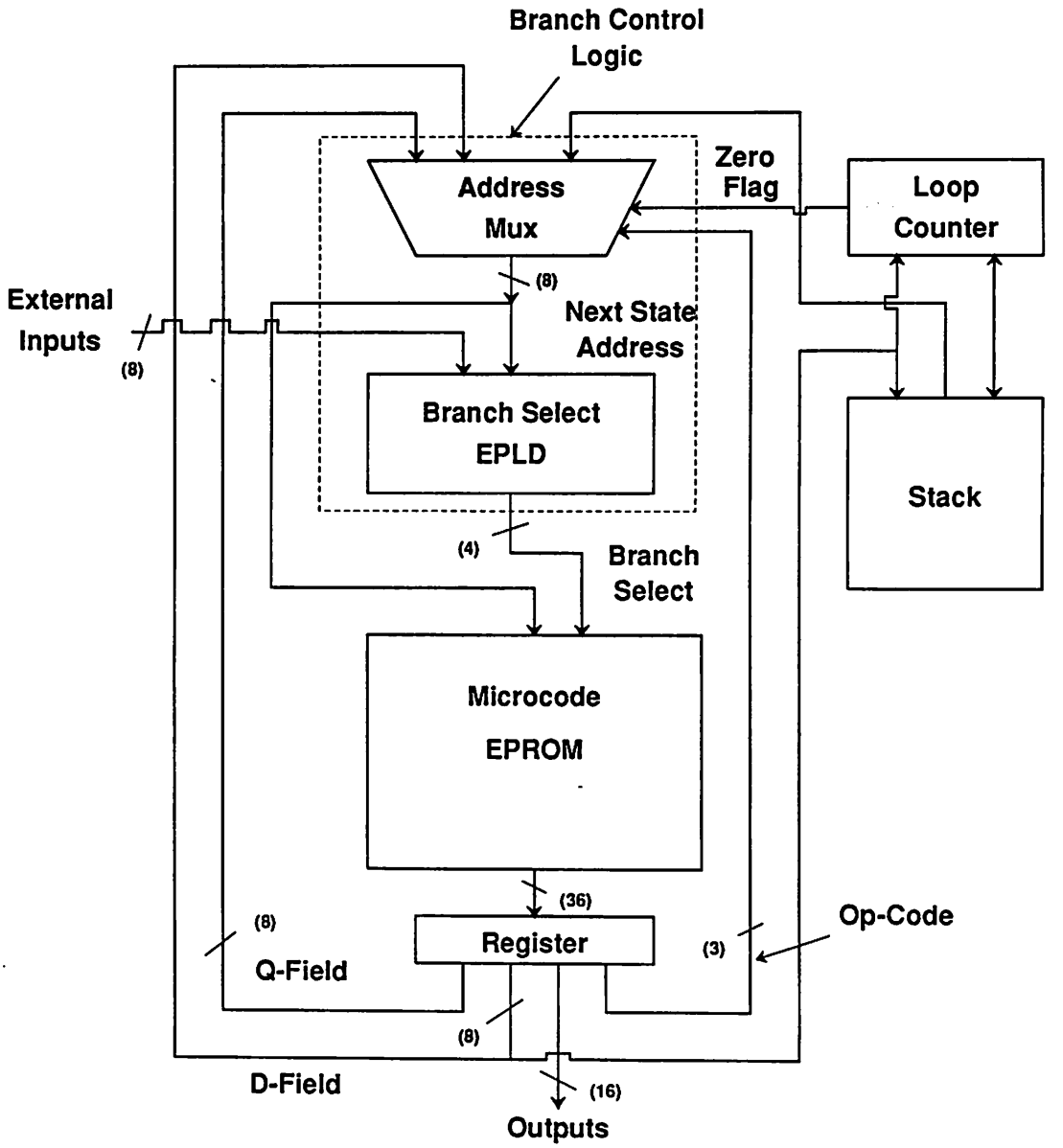


Figure 3.17: Block diagram of the recently introduced Altera EPS448 stand alone microsequencer (SAM). The operation of the SAM has many similarities with the PCU's operation

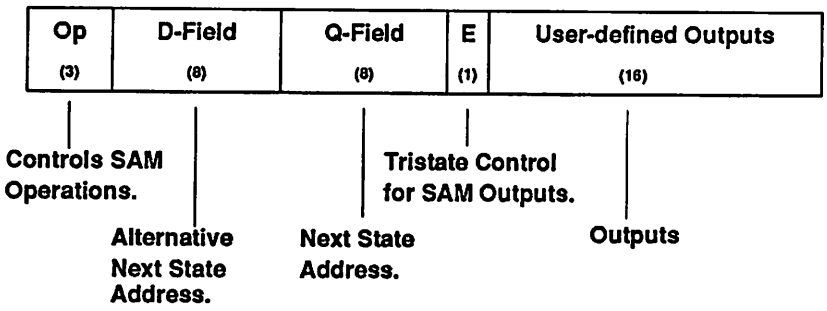


Figure 3.18: Bit-fields of SAM's microcode memory word.

of four possible next states in the microcode memory space 192-448.

The loop counter, which operates in decrement mode, is used for looping. A desired constant value specified in the D-Field must be loaded into the register associated with the loop counter, before entering the loop. On completion of the specified loop counts, the zero flag is set which causes a next state transition out of the loop. For nesting loops as well as subroutines, the stack can be used. All the SAM operations such as push stack, pop stack, and branch take place in one clock cycle. However, like the state transitions of the PCU, the SAM operations take place in parallel with the operations specified in the user-defined outputs of the while at the same time generating an output to control data path operations, differs from sequencing operations done in general purpose DSPs.

TMS32020

A block diagram of the control section of the TMS32020 is shown in figure 3.19. Unlike the PCU or the SAM, each instruction must be fetched from the program memory and decoded before executing it. During the fetch cycle, the instruction addressed by the program counter (PC) is loaded from the program rom into the instruction register. This instruction is decoded during the decode cycle and finally the specified operations are executed in the execute cycle. Normally, on every cycle a new instruction address is generated by incrementing the program counter. *However, when a change-of-flow is required, the next-instruction address*

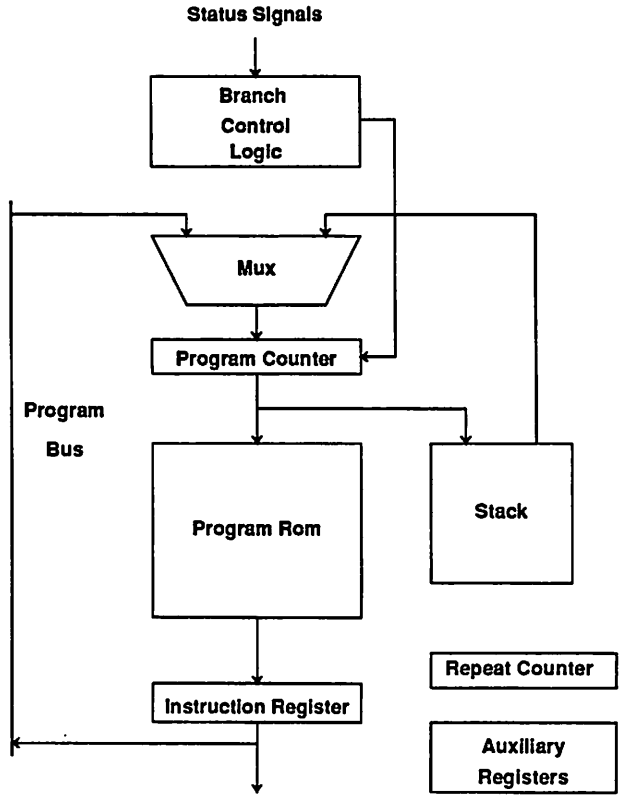


Figure 3.19: Block diagram of the TMS32020 control section.

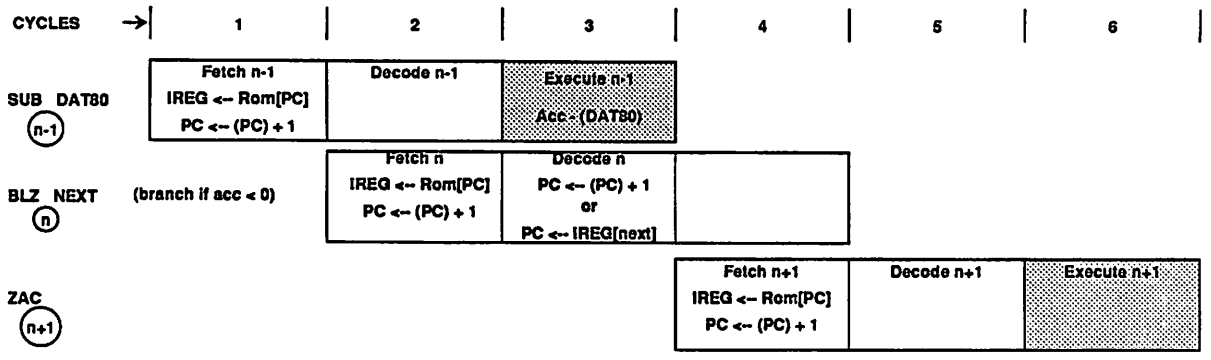


Figure 3.20: Execution of branch instruction in TMS32020. During cycle 3, based on evaluation of the branch conditions, PC either gets the address of the next sequential rom instruction or the address of the branch target. Note that between the execution of the (n-1)th instruction and the (n+1)th instruction, the branch operation takes up two cycles. (The actual increment in the PC during fetch cycle may be more than one, depending on the number of words required by the current instruction.)

is specified in a separate instruction. This is once again different from the PCU or the SAM where the next-address is always generated in parallel with the data path operations. Execution of all control flow instructions on the TMS32020 require two cycles. As an example, the execution of a branch instruction is illustrated in figure 3.20. During execution of the branch instruction, the branch conditions are evaluated in the branch control logic. Accordingly the program counter is either incremented if branch is not taken, or if the branch is taken, the branch target address is loaded into the program counter from the instruction register. Unlike the PCU, the branch mechanism described above allows only a two-way branch. The status signals for determining the branch may come from such sources as the data path (accumulator), auxiliary registers, etc. The auxiliary registers are used for, among other things, looping.

The desired loop count is loaded into one of the auxiliary registers before entering a loop. By using the *BANZ* (branch on auxiliary register not zero) instruction at the end of each iteration of the loop, the auxiliary register is decremented

and tested by the branch control logic. If the auxiliary register is zero, the program flow branches out of the loop. The BANZ instruction thus adds two cycles to each iteration of a loop, which can be a substantial penalty for tight loops. This penalty can be avoided for a single-instruction loop by using a hardware repeat counter. When the repeat count is in effect, no new instructions are fetched; instead, the same decoded instruction is repeatedly executed till the repeat counter goes to zero. At that point, normal program flow resumes.

Subroutine call and return instructions also take two cycles to execute on the TMS32020. No special mechanism is available to avoid this instruction cycle overhead. While executing a subroutine call, the next-instruction address (return address) is pushed on the stack and the subroutine's address is loaded into the program counter from the instruction register (see figure 3.19). During return from subroutine, the stack is popped and the return address is loaded back into the program counter.

DSP56000

The DSP56000 from Motorola is another example of a general purpose DSP chip. A program address generation (PAG) hardware computes a new instruction address every cycle, which, like the TMS32020, is decoded and executed in the following two cycles. The next-instruction address can be the next sequential address or an out-of-sequence address if a change-of-flow, such as branch, is involved. As in TMS32020, only two-way branches are allowed. Each branch (JUMP) or subroutine call/return instruction costs four cycles. On the other hand special hardware is provided for executing multiple-instruction loops with zero cycle overhead. The hardware includes a loop counter (LC); a loop-end address (LA) register to hold the address of the last instruction in a loop; and a stack to store the starting address of the loop. The stack is of course used for other purposes also such as subroutine call/return. Before entering the loop, a *DO* instruction is executed to initialize the LC, LA and to push the loop-start address on the stack. Although the DO instruction costs six cycles, the loop itself does not require any cycles for change-of-flow at

the end of each iteration of the loop. On every instruction cycle, the next-instruction address generated by the PAG is compared with the loop-end instruction address. If a match is found, the following instruction's address is obtained from the stack (loop-start address), instead of by simply incrementing the current instruction's address. As can be seen by the above description, considerable amount of hardware has been dedicated for providing multiple-instruction loops. This can be contrasted with the PCU, which requires only a loop counter. Moreover the PCU uses the same general scheme for implementing all types of control flow instructions without any cycle overhead.

We conclude this section on comparison of various control unit designs by giving a summary in figure 3.21 and figure 3.22.

3.4 Arithmetic Unit Data Path

The Arithmetic unit (AU) is the main data path for performing arithmetic computations. In our design of the data path we have tried to maintain the simplicity and compactness of the LagerI data path. However, several important modifications have been made to remove some of the deficiencies encountered in LagerI.

multiplication

In order to reduce truncation error during multiplication, the shifter which was put before the adder in LagerI data path has now been placed after the adder. As a result the worst case truncation error in implementing an $n \times n$ shift/accumulate multiplication has been reduced from

$$\text{error} > n \times (\textit{magnitude of one lsb})$$

to

$$\text{error} = \textit{magnitude of just one lsb}.$$

Moreover, the shifter, which used to be one stage of a three stage pipeline in LagerI has been merged with the adder stage. As described in chapter 5, this has been

	General Purpose Programable DSPs		Stand Alone micro-sequencer (prog. logic device)	Custom Circuits
	TMS32020 Texas Instr.	DSP56000 Motorola	EPS448 Altera	DSP (robot contr.) U.C Berkeley
INSTRUCTION LOOPS				
	Single Instr. / Multiple Instrs.			
Hardware Overhead	- repeat counter / - aux reg [5]*, - aux reg. ptr*, - aux arith. unit*.	- loop counter, - loop end address reg., - stack*	- counter	- loop counter
Cycles	zero cycle / 2 cycles	zero cycle	zero cycle	zero cycle
CONDITIONAL BRANCH				
Cycles	2 cycles	4 cycles	depends on number of data path pipeline stages	1 cycle delayed br.
Multiway Branch ?	NO; 2-way only	NO; 2-way only	upto 4-way branch	YES
SUBROUTINE				
Hardware Overhead	stack*	stack*	stack	stack**
Cycles for CALL	2 cycles	4 cycles	zero cycle	zero cycle
Cycles for RETURN	2 cycles	4 cycles	zero cycle	one cycle
Cond. Subr. Call?	NO	YES	NO	YES

* Hardware resources shared with other functions.

** Subrs. can also be implemented by trading the stack for additional states in the fsm.

Figure 3.21: Implementation of control flow operations in different architectures.

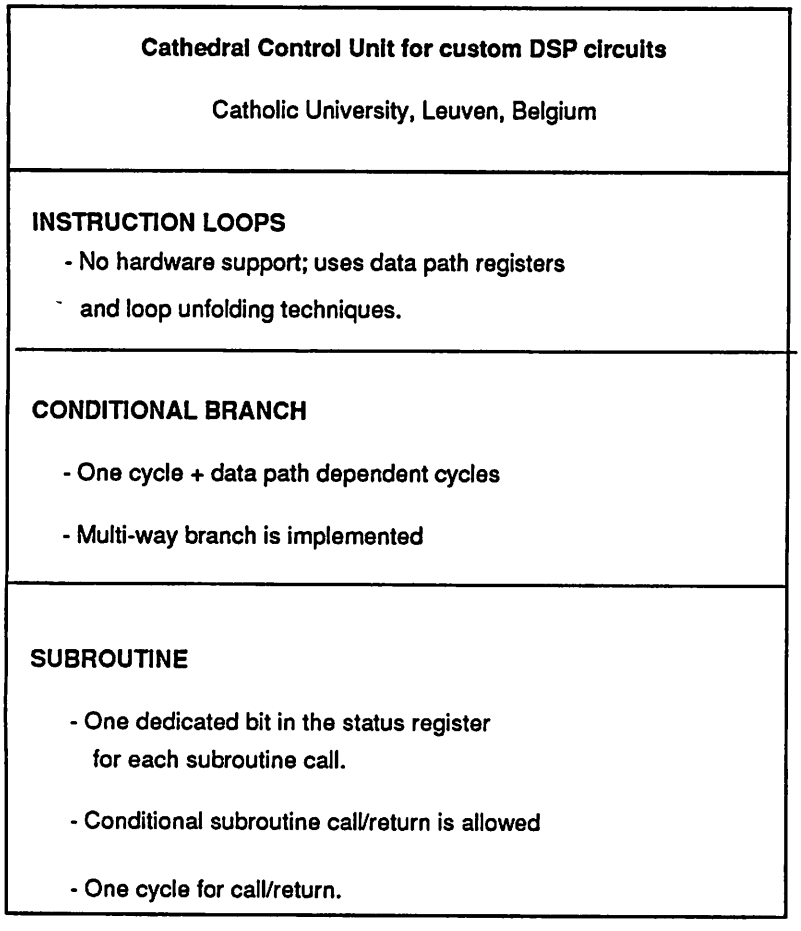


Figure 3.22: Summary of characteristics of the control unit for the Cathedral design system.

achieved without necessarily increasing the critical path delay. In order to perform the iterative shift/accumulate multiplication, the multiplier is loaded into a parallel-serial coefficient register. On each iteration, product of the multiplicand word and a multiplier bit (starting with LSB first) is added to the partial sum in the accumulator after the accumulator's contents have been shifted right. The operations performed during each iteration of the multiplication is summarized below.

shift right coefficient register,
B input = accumulator shifted right by one bit,
A input = (LSB of coefficient) $\times$ multiplier,
accumulator = A + B, disable saturation;

As described above the shift operation is now done on the accumulator's contents. This complicates the saturation of the adder's output upon overflow. The adder's output, in case of overflow, should normally be saturated at $2^{n-1} - 1$ (positive maximum) or -2^{n-1} (negative maximum), n being the number of bits. For certain cases (e.g. shift/accum. multiplication), however, the number should not be saturated since a right shift would restore the number back to the bound of $\pm 2^{n-1}$. Consequently, the hardware has been designed to allow the user, under program control, the option to leave the adder's output unsaturated.

Figure 3.23 shows the section of the data path that executes all the arithmetic operations. In addition to add and shift, the data path supports several other operations. These include ABSOLUTE of A input, NEGATIVE of A input, INCREMENT B input, etc. The NEGATIVE operation gives a true 2's complement negative result, unlike LagerI, which did a 1's complement.

Data Transfers

Three types of data transfers are involved in the data path: with other functional units on the processor; with memory; and with external devices. In LagerI all data transfers to and from other functional units/processors on the chip took place through serial lines with the objective of reducing routing for data

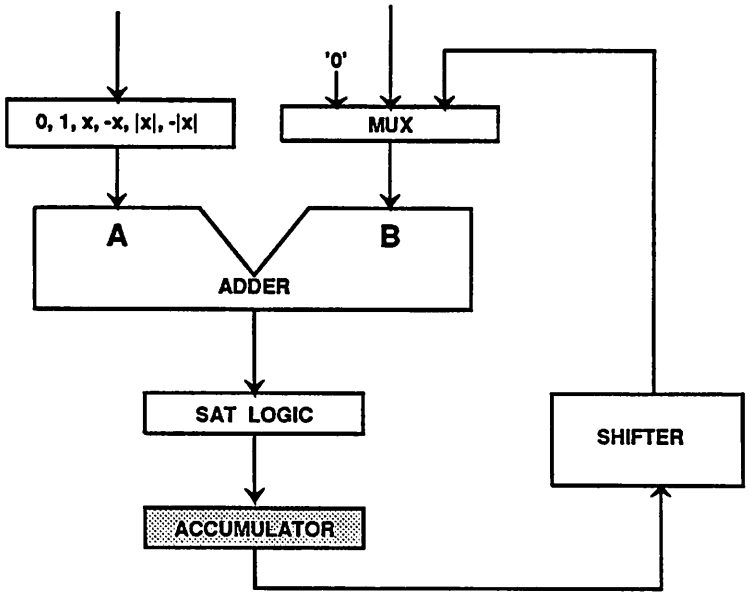


Figure 3.23: Section of the AU data path which performs add, shift and other arithmetic operations. The coefficient register used in shift/accum. multiplication is not shown here.

busses. Another consideration was to alleviate the synchronization problem in multi-processor configuration. Consequently, each program variable involved in I/O communication had a parallel-serial converter at the transmit-end, a serial-parallel register at the receive-end, and a dedicated serial interconnect. This approach, however, often led to large overhead in I/O circuits.

Although serial lines take up less area for routing than parallel lines, this advantage diminishes as the number of I/O variables increases, with a corresponding rise in the number of dedicated serial lines. When the number of I/O variables is equal to the word-size of the variable then the number of data lines required for serial as well as parallel transfers is exactly same. Moreover serial transfers involve an N cycle delay, where N is the number of bits. The hardware needed to convert from serial to parallel form and vice-versa also adds to the hardware cost. As examples of LagerI applications in table 3.2 show, the overhead can be considerable.
In order to avoid proliferation of I/O circuits, a different scheme for transferring data to and from the AU has been adopted in the robot controller.

Application	Number of I/O Variables (with serial transfer)	Relative Sizes I/O ckts: Data Path
Correlator	7	1:1
Pitch-Tracker	2	2:5
Decision FBE	4	1:2

Table 3.2: Relative sizes of I/O circuits – for serial data transfer – and data path for several LagerI chips. Note the relatively large area used by I/O circuits.

As shown in figure 3.24, all data I/O are done over parallel lines. Instead of providing dedicated registers, as in LagerI, for holding I/O variables, all the variables are stored in a local ram, which is a much more efficient storage unit. All external I/O is done through a parallel port. A port address bus selects the desired I/O device. Since external data cannot be written directly to the local memory in one cycle, two general purpose registers, R_0 and R_1, are also provided as shown in figure 3.25. These registers form one of the three pipeline stages of the data path. The memory output register, MOR, forms the other pipeline stage besides the arithmetic stage discussed earlier. As shown in figure 3.26, following the second stage, data may be stored in the registers while at the same time a new data may be read into the MOR register. The data in registers R_0 and R_1 may be moved into the memory when an idle cycle is available. On the other hand, the data path also allows the accumulator data to be directly moved into the memory via the Mbus, by-passing the third stage. The Mbus is the main bus going across the data path, and also connects with the address processing unit as well as the coefficient register.

3.5 Logical Unit

The Logical Unit (LGU) implements boolean operations. Consider the following,

if $x > y$ then $\{action\ list\}$.

Although this operation can be implemented with the PCU, for those cases where

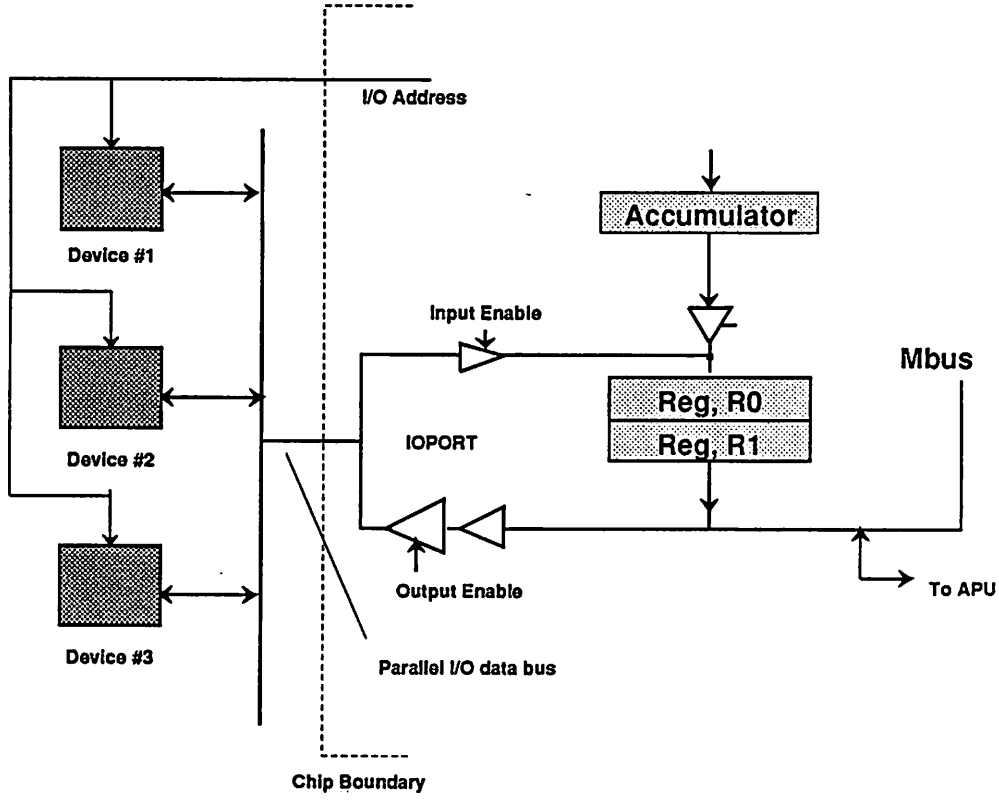


Figure 3.24: Section of data path showing I/O bus and registers. All data transfers to and from the data path are done through parallel lines.

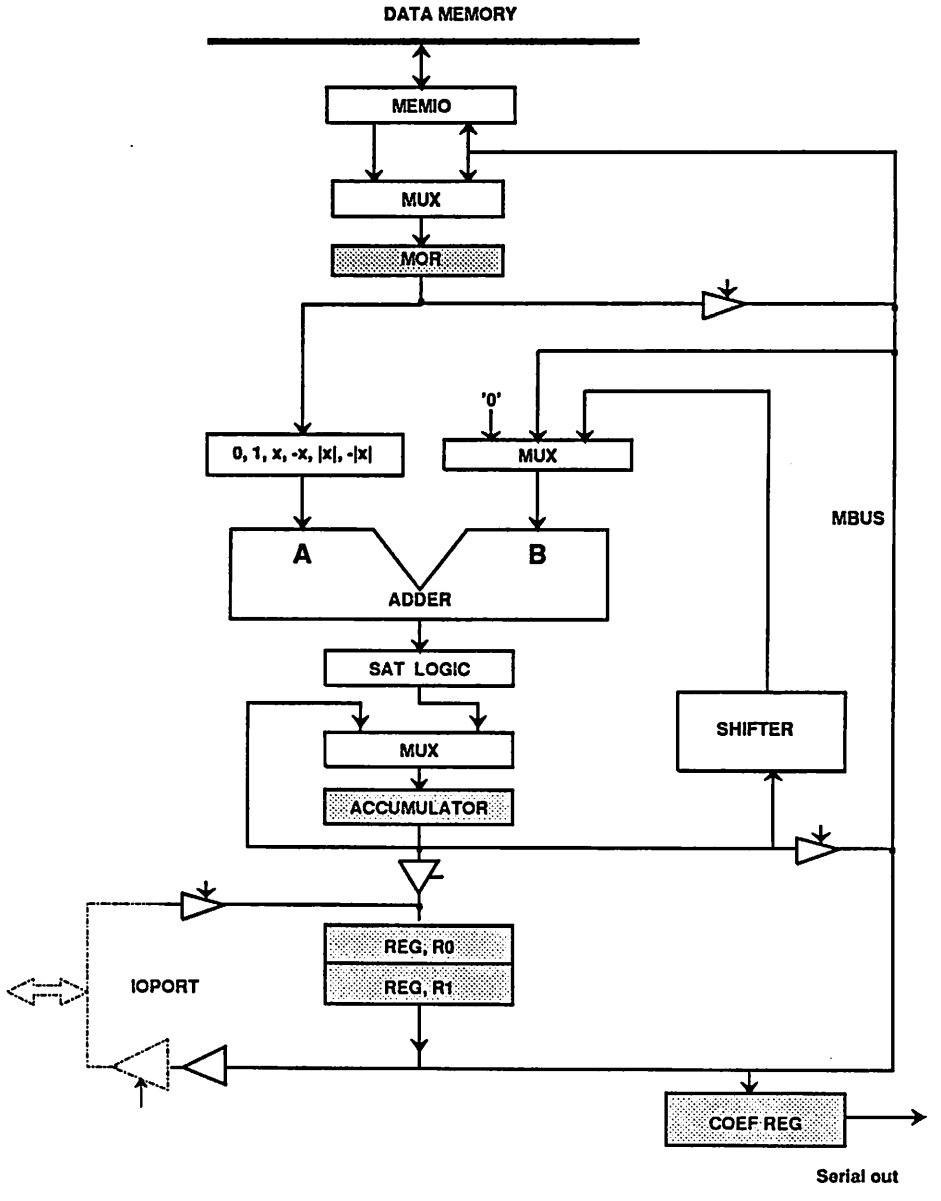


Figure 3.25: Block diagram of the data path.

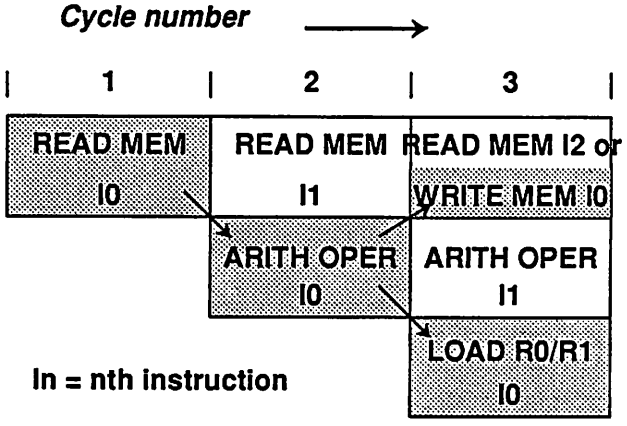


Figure 3.26: The three pipeline stages of the data path.

the *action list* is a single operation, the LGU may be used. The design of the LGU is based on a finite state machine, as described in [Pope85]. The state of the fsm is set by input conditions, such as the sign bit from the data path or even external signals, and the fsm's past state. This allows the fsm's output, called condition code, to be a boolean function of past and present input conditions. The condition code signal, CC, is then used for executing two data path instructions:

write conditionally (to local memory)

accumulate conditionally

Using the LGU avoids creating a large number of small program blocks in the instruction memory as well as more states in the fsm of the PCU. However, the execution of conditional operations with the LGU costs instruction cycles which may be avoided with the PCU.

3.6 Memory Unit

Closely tied with the data path is a local memory which stores program variables and I/O data as mentioned in section 3.4. The memory is organized in word-parallel form without any column decoding, the word size being same as that

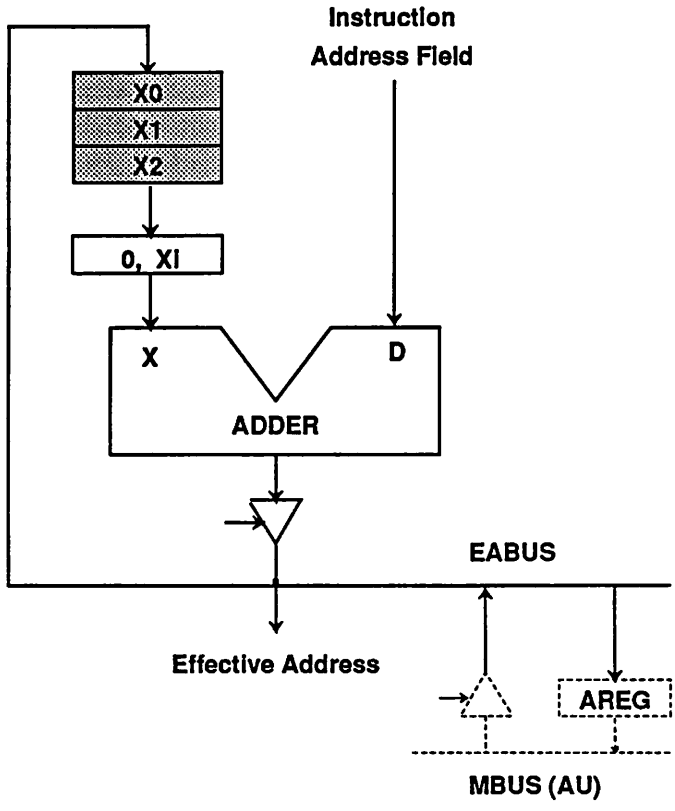


Figure 3.27: Address processing unit block diagram.

of the data path. As in LagerI, any memory location can be configured as a read-only constant. However, the user now also has the option of specifying constants in the address field of instructions as described below.

3.7 Address Processing Unit Data Path

All address computations for variables in the local memory are done in an address processing unit (APU). Since the APU operates in parallel with the arithmetic unit, no instruction cycle overhead is required for address computations, thus increasing through-puts. Like the AU, the APU is also based on a bit-slice architecture. The number of bit-slices being strictly dependent on the largest ram address to be computed, is usually much smaller than for the AU. The main components of the APU, shown in figure 3.27, are three registers and an adder. The hardware is

adequate for supporting several addressing modes commonly encountered in digital signal processing applications. In addition the registers can also be used as loop counters.

3.7.1 Addressing Modes

In the description of the addressing modes given below, the following symbols are used:

$EA = effective\ address.$

$IAF = address\ field\ in\ the\ control - word.$

$X_i = ith\ register$

Immediate Addressing

$$EA = IAF(= constant)$$

A constant may be directly specified in the address field of an instruction. Using the bus connection between the APU and the AU (see section 3.7.3), the constant may be moved into the AU.

Direct Addressing

$$EA = IAF$$

The X input of the adder is forced to zero so that the address field of the instruction directly forms the effective address.

Indirect Addressing

$$EA = X_i$$

The address is supplied by one of the APU's registers and the instruction's address field is set to zero.

Indexed Addressing

$$EA = X_i + IAF$$

The effective address is formed by adding the address specified in the instruction and the contents of one of the registers. The same method may be used for pointer mode addressing by loading a pointer base-address in one of the registers and adding offsets to address the variables.

3.7.2 Loop Counting

The APU registers can also be used as loop counters. Whereas for short loops the PCU loop counter is very useful, for large and complex loops the APU registers may be used as loop counters. These registers allow nesting of loops. At the same time the contents of the registers are also available for address computations. The loop counter register may be incremented or decremented by setting the instruction's address field to $+1$ or -1. The modified count value is then stored back into the register.

$EA = X_i + 1;\ X_i = EA$ or

$EA = X_i - 1;\ X_i = EA$

3.7.3 Data Transfer between APU and AU

Although the APU and the AU are dedicated for address computations and arithmetic computations respectively, sometimes data may have to be transferred between the two. For example, the address may be computed in the AU and then moved into one of the APU's registers, or some of the variables used in the APU may be saved in the local memory for minimizing the number of registers needed in the APU. Because of the need, a parallel connection is provided between the EABUS in the APU and the AU's MBUS. The hardware shown in dotted in figure 3.27 is part of the AU. Since the word length of the two data paths may be different, the following convention is adopted, assuming APU-word-size $<$ AU-word-size: (i)

When moving data from APU to AU, data is sign extended; (ii) When moving data from AU to APU, data is truncated on the MSB side. Clearly, any loop count or address value should not exceed the word size of the APU.

Chapter 4

The Silicon Compiler Environment

In order to create a physical layout of the target architecture described in chapter 3, the Lager-III [Sh87] silicon compiler environment is used. Using this CAD environment, a customized layout of the architecture is automatically generated for a specific algorithm. Details about the design of the CAD tools and the compilers are given in [AzShBr88] [ShJSB87] [Shung88]. The work reported in this thesis deals with the application of the CAD environment to the robot control problem. This involves development of the architecture model which provides the frame work for control generation and synthesis; developing the design methodology; integrating the design of the architecture inside the CAD environment through structural description files and front-end routines for generating layouts of specific macrocells. First, we present an overview of the Lager-III system.

4.1 Overview of Lager-III

Lager-III has two independent parts, illustrated in figure 4.1. At the lower level a silicon assembly subsystem assembles a customized layout. At the higher level compilers map algorithmic descriptions into hardware specifications for the processor. The input to the high-level compiler is a program written in Silage [Hil85],

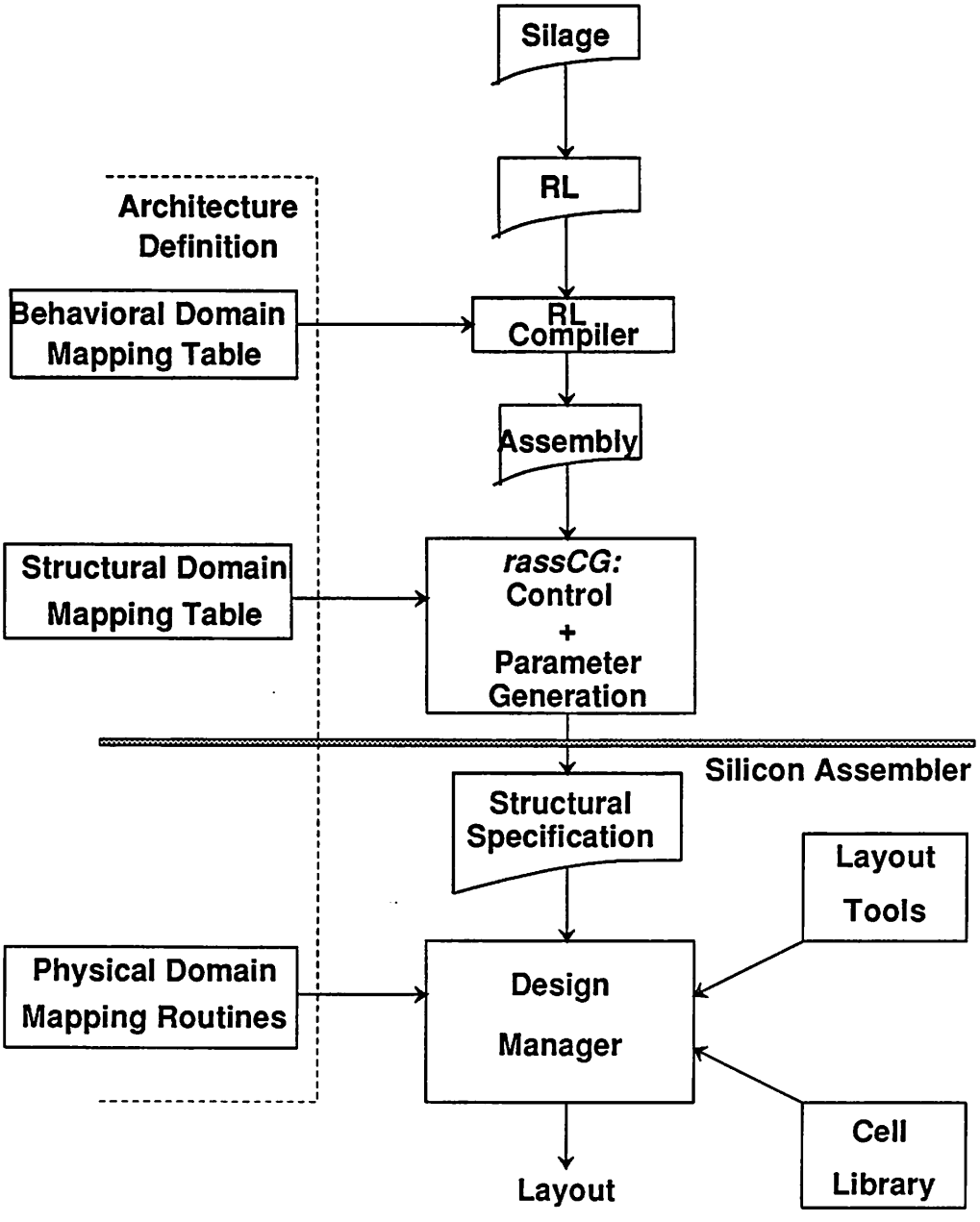


Figure 4.1: Lager-III silicon compiler system.

which is an applicative language for representing signal flow graphs, or RL [Rim88], which is a procedural language like 'C'. Next, the RL program is compiled into an architecture specific *assembly language* program. This assembly program describes the algorithm at the microarchitecture level. Various sections of this program are listed below; detailed specifications are given in appendix A

- list of variables
- list of boolean operations to be performed in the LGU's (logical unit) finite state machine (dfsm)
- control flow state-transitions
- list of loop sizes (number of iterations) to be detected by the loop counter
- width of the arithmetic unit data path
- list of control states in which timer is to be kept reset
- maximum size of the sampling period in terms of number of instruction cycles
- list of blocks of instructions, each instruction consisting of multiple microoperations performed in parallel

The user may enter the design system by writing his own assembly language program or let the RL compiler [Rim88] generate the assembly language program from a 'C' description.

In order to do the compilation, the RL compiler requires knowledge about mapping the behavioral description of the algorithm in RL into a behavioral description in assembly language. This knowledge is provided in the form of a list of allowed microoperations, legal data transfers, hardware resources used by each microoperation, and a code-generation table. In order to target a different architecture, the knowledge-base must be altered. The next level of transformation involves converting the microarchitecture-level behavioral description (assembly language program) into a microarchitecture-level *structural specification.*

The structural specification generated by the control and parameter generation software, *rassCG* (see figure 4.1), includes hardware parameter values (e.g. number of ram words, data path word-size), control-words in the rom specifying the microoperations in each cycle, and an fsm table specifying the state-transitions. As in the behavioral domain mapping, a structural domain mapping table is required by rassCG. These tables for the robot control processor and their relationship to the architecture model are discussed in section 4.2. After generation of the structural specification they are sent to the silicon assembly part of Lager-III for layout generation.

At the center of the silicon assembler in Lager-III is the Design Manager [ShJSB87] (DM). It is a program for managing the layout generation process. The designer provides DM with a set of hierarchical *structural description files* (sdl) describing the processor in the structural domain. Each sdl file describes a macrocell which may be part of a parent macrocell and may itself contain other macrocells as children. The description in sdl files includes connectivity information, list of sub-cells used, a list of hardware parameters, type of layout generator required, and a simulation model. DM calls up the appropriate layout tools for generating each hardware block, provides them with the necessary parameter values, and keeps track of the connectivity information for the routing tools. In this project, the following layout tools were used.

(i) *TimLager:* [ShJSB87] Assembles layout of macrocells by tiling together leafcells. The macrocell designer must also write a 'C' procedure describing the tiling scheme for the macrocell. The 'C' procedure allows the layout to be parameterized with regard to size, type of cells used, etc.

(ii) *Data Path Compiler* (dpc): [Ma87] Assembles bit-slice data paths. The configuration of the data path with regard to type of cells used is specified in the sdl file; the width of the data path is specified as a parameter.

(iii) *Standard Cell Generator:* [ShJSB87] Assembles rows of standard cells for implementing logic functions. A front-end program, eqn2sdl [Ma87], allows the designer to specify the logic in the form of boolean equations.

(iv) *Flint:* [RaPoBr85] [ShJSB87] An interactive place and route program for con-

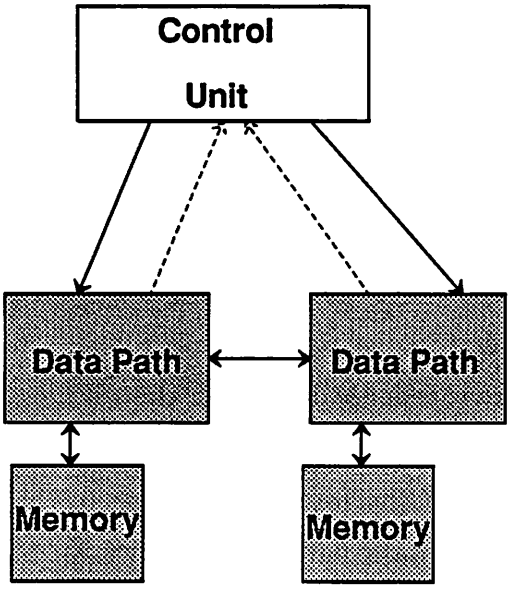


Figure 4.2: Architecture model consisting of a central control unit, application specific data paths, and memory.

necting up macrocells generated using one of the last three tools.
In addition to the layout tools, DM also needs leafcell libraries for the various macrocells required to complete the design.

4.2 Relationship Between Architecture Model and Silicon Compilation

In chapter 3 we briefly discussed the architecture model followed by a detailed description of the complete architecture. In this section we shall elaborate on the choice of the architecture model from the point of view of silicon compilation. We shall also describe the link between the assembly language program and the structural specification of the processor.

The architecture model is based on separation of control flow and data path operations and consists of a single control unit, multiple data paths, and a memory unit, as shown in figure 4.2. As described in chapter 3, we have designed

a generic, self-contained control unit which is capable of supporting a small but functionally complete set of control flow operations. The processor can be built around the control unit by adding one or more application specific data paths. This approach also facilitates adapting the compilers and rassCG program to a new processor design. Since the control unit design remains fixed, the rules for mapping the control flow operations into the control unit's fsm table and the manner in which the control signals for the data paths are specified in the rom, remain fixed. On the other hand, the data path's microarchitecture is redefined through the sdl files. This simply requires a new mapping table for describing the association between the primitive microoperations and the control-bits of the rom. In addition to the rom's output controlling the microoperations in the data paths, a second form of interaction takes place in the form of status signals from the data paths feeding back into the finite state machine inside the control unit. These signals are used for executing conditional operations such as a conditional branch. An arbitrary number of status signals (e.g. sign bit, carry-out from ALU, etc.) feeding into the control unit's fsm can be easily specified through the sdl files. The above discussion shows that application specific processors can be conveniently assembled by using the same basic architecture model with a fixed control unit design, a fixed scheme for mapping microoperations into the control-word but with freedom to customize the data paths for the target application. Once the microarchitecture is described through sdl files, a customized layout is generated from the structural specifications.

4.2.1 Structural Specifications

The structural specification of the processor consists of four parts: hardware parameters; rom control-words specifying the operations in each instruction; fsm table for the state-transitions; and fsm table specifying the boolean operations in the LGU.

Hardware parameters

Hardware parameters are used for customizing the layout. These parameters include word length of the data path, number of ram words, etc. A complete list of the parameters for the robot control processor is given in appendix B.

Rom Control-words

The rom control-word has a bit-field assigned to every control signal or group of signals used for controlling the operation of the data paths, memory, and any other functional unit. This is illustrated in figure 4.3 for a small subset of a data path. Each primitive microoperation in the data path is associated with a particular set of control signals as listed in appendix C for the robot control processor. Appendix D gives a list of the control signals and their field assignments in the rom word. The control signals are explained in chapter 5.

Control FSM Table

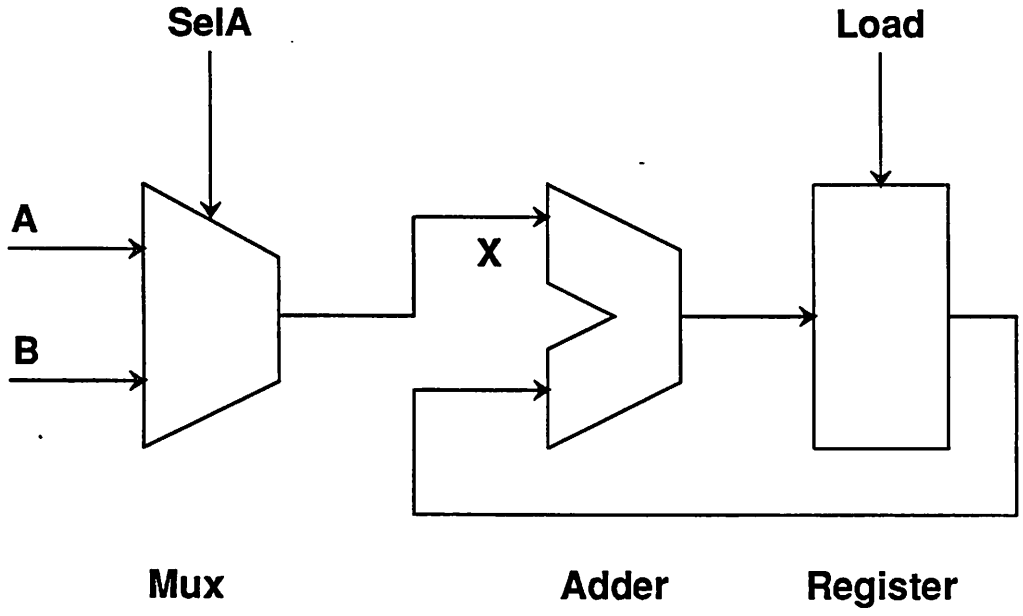
The control fsm (cfsm) table describes the state transitions. In chapter 3, we have presented a detailed discussion on mapping control flow operations such as branch, subroutine call, etc., into the fsm table. Part of rassCG program implements those mapping rules.

Logical Unit FSM Table

The decision fsm (dfsm) table specifies condition codes as boolean functions of various input signals. A number of functions can be stored and any one of them can be invoked from a particular instruction. When a function evaluates true, the corresponding condition code is asserted.

4.3 Customization of the Hardware

The structural specifications provide the input to Design Manager for generating a customized layout of the processor based on the structural descriptions.



Mapping microoperations to control signals	
microoper.	control signals
X = A	SelA = 1
X = B	SelA = 0
Reg = Reg + X	Load = 1

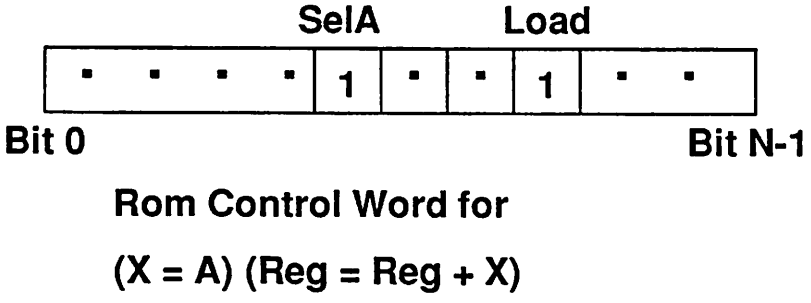


Figure 4.3: Microoperations and corresponding control signals for a subset of a data path. The control-bits for SelA and Load are asserted in the rom-word for executing the instruction ((X=A) (Reg=Reg+X))

Various levels of customization is possible through the structural specifications and the sdl files as discussed below.

Level #1 - Software Programmability:

This is the simplest form of customization in which a new assembly language program is written for a new application on an existing processor. This results in reprogramming of the control rom and the finite state machine in the control unit.

Level #2 - Hardware Programmability:

Hardware programmability means the hardware can be configured during layout generation according to the values of the various parameters. The parameters are a) word size of the data paths, b) word size of the ram, c) number of words in the ram, d) type of operations in the logical unit, e) number of unique loops in the PCU, f) depth of the stack inside PCU as determined by the number of levels of nested sub-routines, g) maximum count of the timer inside PCU as determined by the maximum sample interval, and h) maximum count of the program counter inside the PCU.

Level #3 - Customization of the hardware configuration:

The microarchitecture can be re-configured to match the requirements of the application. These modifications in the hardware include a) choosing the number and type of functional elements in the data paths (e.g. number of registers); b) bus structure (inter-connections between elements) of the data paths; and c) selecting the hardware functions (timer, loop counter, stack) of the control unit.

Level #4 - Inclusion of I/O and interface circuits:

Application specific I/O, interface circuits, and 'glue' logic can be included with the processor for reducing system cost and facilitating system integration.

In the above description, level 1 through level 4 represent progressively increasing levels of customization of the hardware. Customization at levels 1 and 2 require no effort by the user other than writing the RL or assembly language program. Customization at level 3 requires modifying the sdl files and also providing new mapping tables to the compiler and rassCG. Customization at level 4 not only requires modifying the sdl files and mapping tables for compiler and rassCG but also possibly designing some of the peripheral circuits.

Chapter 5

Circuit and Layout Design

A processor's high level representation, such as the microarchitecture, must eventually be reduced to transistor-level circuits and finally mapped into a physical layout. Both circuit level design and layout design are very time-consuming and tedious jobs. Consequently most silicon compilers have fairly powerful tools for assembling the layout from some intermediate level representation – net list, logic diagram, etc. – of the processor. Before a silicon compiler is capable of generating the layout, however, the design methodologies, layout styles, cell libraries, etc. must be defined and incorporated into the CAD environment. In this chapter we describe in detail the circuit design, use of appropriate layout tools, and organization of the cell hierarchy for the robot controller chip.

5.1 Design Approach

The robot controller is a macrocell based microcoded processor. In designing custom circuits, a designer may choose between a hard-wired circuit which is fully dedicated to one specific algorithm or a microcoded processor that can be customized. In hard-wired circuits, the operations and the sequence in which the operations are performed are fixed and defined by the hardware configuration. These circuits are usually data path intensive with simple control and are able to provide high through-put rates [Reu86] [Bar87]. Consequently they are typically

used in applications where desired sampling rate is close to the clock rate. Since each circuit is unique, however, considerable design effort is required for defining the architecture. Moreover, the hard-wired nature of the circuits results in minimum sharing of hardware resources and hence larger chip area. Therefore, for relatively low sampling rate applications, a microcoded processor is a better choice.

The robot control processor consists of six macrocells as discussed in chapter 3 – AU, APU, PCU, LGU, RAM, and Test-Logic. The following considerations guided our design of the cells:

(i) Use of highly parameterized, modular macrocells.

(ii) Bit-slice architecture for data path macrocells.

(iii) Minimize number of leafcells in the macrocells and share cells.

(iv) Leafcells designed to be easily adapted for a variety of situations.

(v) Conservative, straight forward circuit design in order to minimize circuit design time and problems due to complex circuit techniques which are often process dependent. All circuit designs are done in CMOS.

(vi) Use of scalable CMOS (SCMOS) technology in order to allow some degree of technology-independence.

5.2 Organization of the Processor

The processor consists of a series of hierarchical structural description files (sdl) reflecting the cell hierarchy. The hierarchy of the sdl files are shown in figure 5.1. At each level of the hierarchy the child cells are assembled together with Flint, a place and route tool. One of the design decisions we had to make was the amount of hierarchy in the processor's layout design.

The amount of hierarchy in the design is a trade-off between convenience of design and compactness in layout. Ideally, a completely flat design would provide the greatest freedom in moving the cells around for the most compact layout. Moreover a flat design results in fewer files. However, the place and route tool may be limited in the number of cells and interconnections it can handle simultaneously.

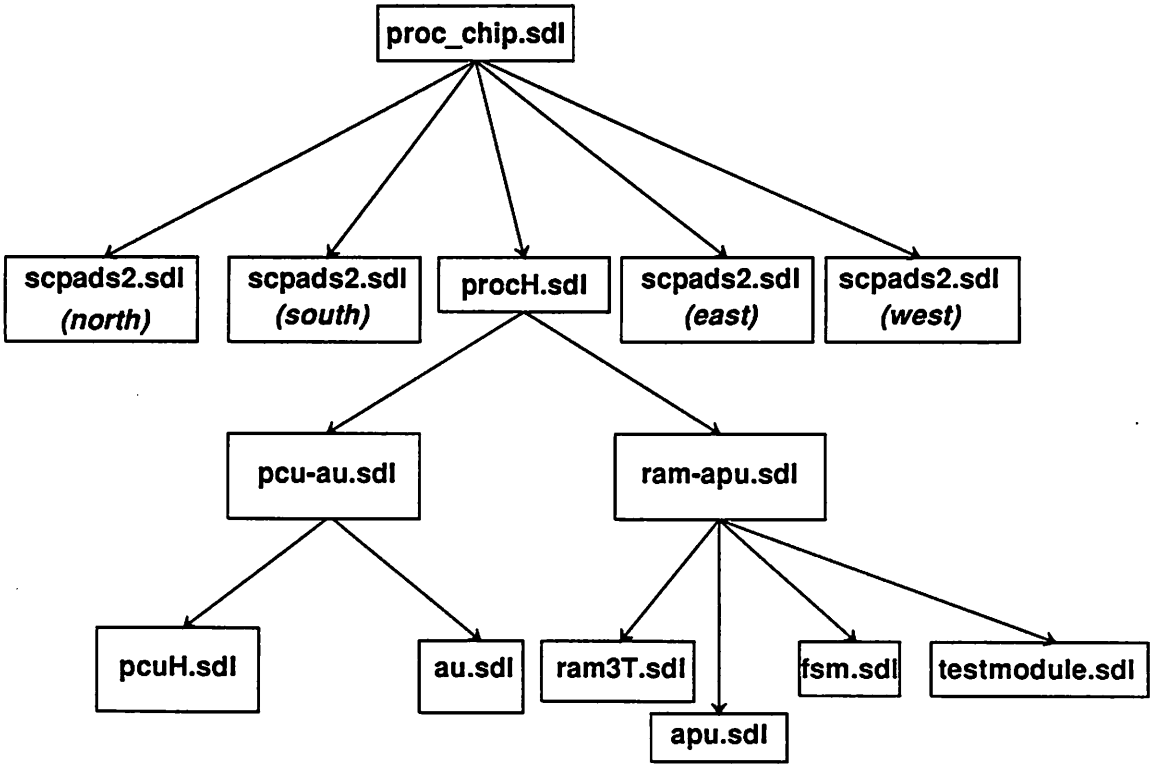


Figure 5.1: The structural hierarchy of the robot control processor. In this figure, the five cells at the bottom are the five major macrocells in the processor.

Apart from overcoming this limitation, hierarchy also simplifies the design task in many ways:
(i) Allows divide and conquer strategy; smaller cells can be easily debugged and verified both with respect to the sdl files as well as the actual layout.
(ii) Subcells and their sdl files can be easily re-used within the design and also in other designs.
(iii) Subcells can be easily added or deleted from the design.
(iv) The design can be easily shared between different designers or design groups.
(v) Simulation and verification of the layout/design can be speeded up by taking advantage of the hierarchy.
One possible compromise is to use hierarchy during design and verification but flattening out the sdl files during layout generation. For the robot controller chip we maintained a hierarchical structure both during design and layout generation because of limitations of the place and route tool.

As shown in figure 5.1, the top-level sdl file for the entire chip is called proc_chip.sdl. It is composed of five sub-cells, four of which are the four pad groups around the periphery of the chip. Since the North pad group does not have any real pads, it is deleted in the final layout for minimizing chip area. Inside the space bounded by the pads is the core processor described by procH.sdl. The top level layout showing the pad groups and the core processor is shown in figure 5.2. In order to make the routing problem manageable for Flint, the procH.sdl is assembled from two subcells pcu-au.sdl and ram-apu.sdl. Finally, at the bottom of the hierarchy are the five macrocells labeled with their generic names. These macrocells are themselves composed of even smaller child cells as described later. First, however, we digress briefly for describing the global timing and interconnections of the processor.

5.3 Global Timing and Interconnections

The processor uses a two phase non-overlapping clocking scheme as shown in figure 5.3. In a global sense, the processor architecture has a two stage pipeline:

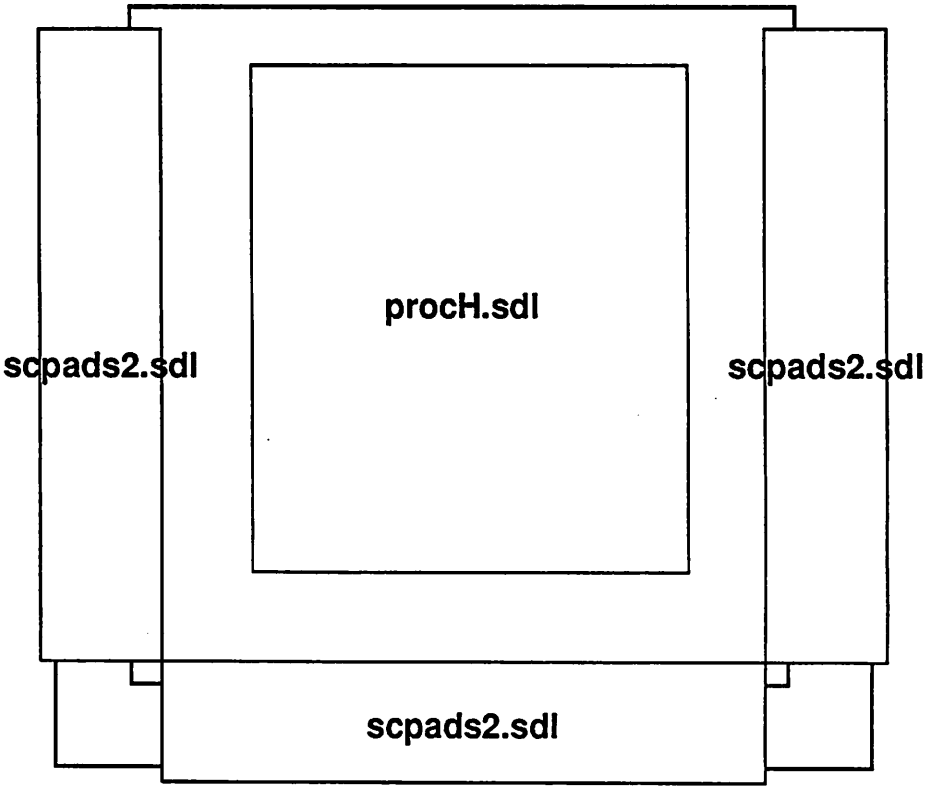


Figure 5.2: Top-level view of the robot controller chip layout showing three pad groups surrounding the core processor cell. The North pad group is deleted since it does not have any actual pads in this example.

the evaluation and generation of the control-word followed by operations in the data paths and memory. Multiple pipeline stages, however, exist within the data paths as described later. At the end of phase 2 of every clock cycle a control-word becomes valid. This control-word is latched on the falling edge of phase 2 and is available in the next cycle. During phase 1 and phase 2 of the next cycle this control-word is used to control the microoperations in the data path. The control-word also provides an address input to the APU where the ram address is computed during phase 1. The ram is also precharged on phase 1 and read or written during phase 2. At the end of a ram read operation, the output of the ram is latched inside a memory output register which is part of the AU data path.

The top-level interconnection network of the processor is shown in figure 5.4. The individual blocks (macrocells) of the processor are described next. In this description the details about the scan path testing circuits – scan register, scan latch, and Test-logic – are not given here; they are described in chapter 6. The structural description files of the hardware are given in appendix E.

5.4 Processor Control Unit (PCU)

5.4.1 Macrocell Organization

The control unit consists of six child macrocells: cstore-Macro; cfsm-Macro; pc-Macro; lpc-Macro; stack-Macro; and timer-Macro as shown in figure 5.5. Names in italics and enclosed with [] in the figure are instance names of the cells. The six PCU macrocells are assembled together using Flint. The boundary terminals of the PCU macrocell are shown in figure 5.6 and are described below. In the figure, the numbers inside [] give the actual bus widths for the robot controller processor. In general, however, these widths are parameterized.

TIMERINBUS: This is a data bus which connects with the global EABUS for loading the timer register inside the PCU from the AU data path. The number loaded in the timer register determines the sample period in terms of number of instruction cycles.

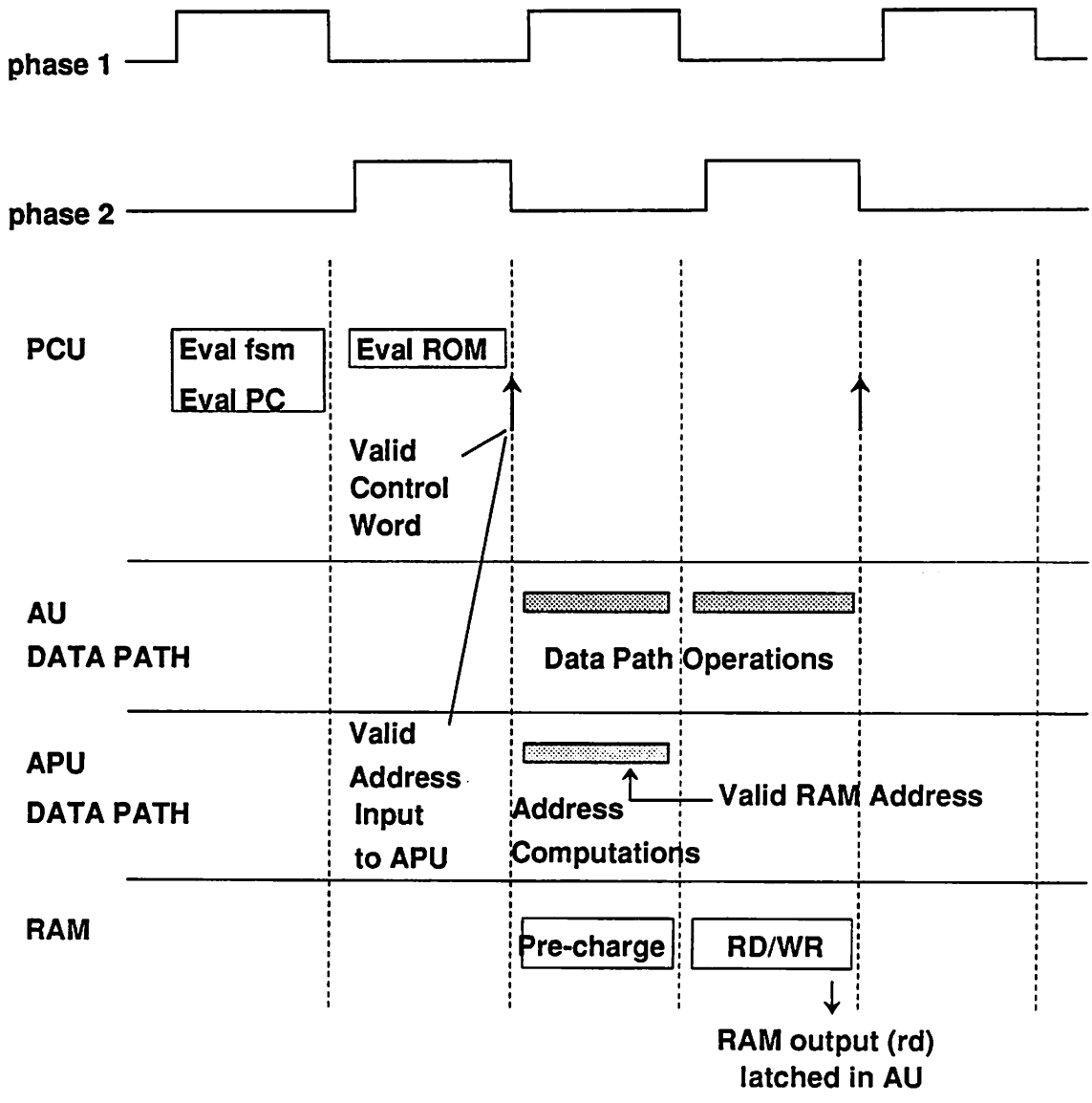


Figure 5.3: The global timing scheme of the processor is based on a two phase non-overlapping clock. On each cycle a new control-word is generated which determines the data path and ram operations in the following cycle.

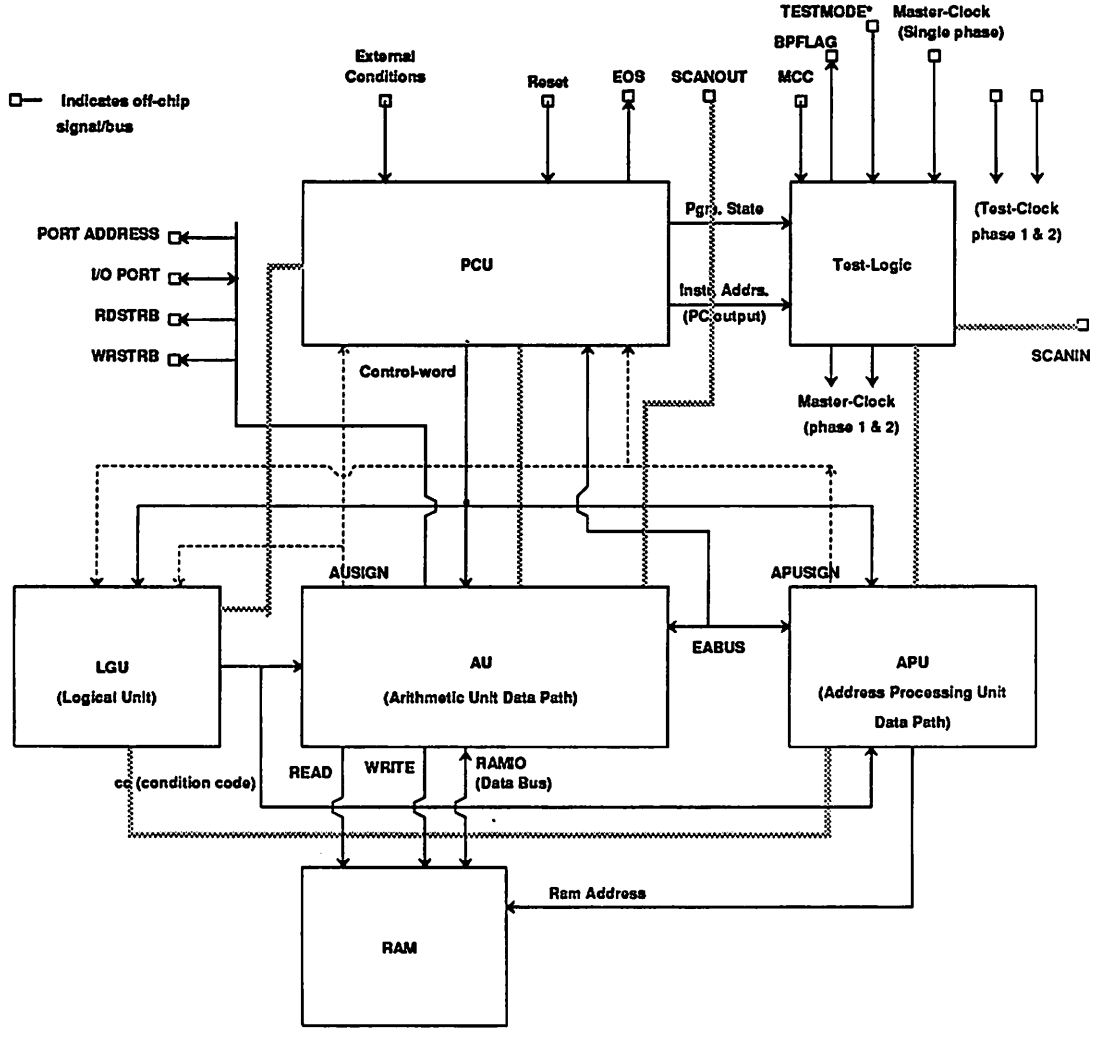


Figure 5.4: Top-level interconnection network of the processor. Dotted lines indicate data path status signals. Scan path connections are shown as cross-hatched lines.

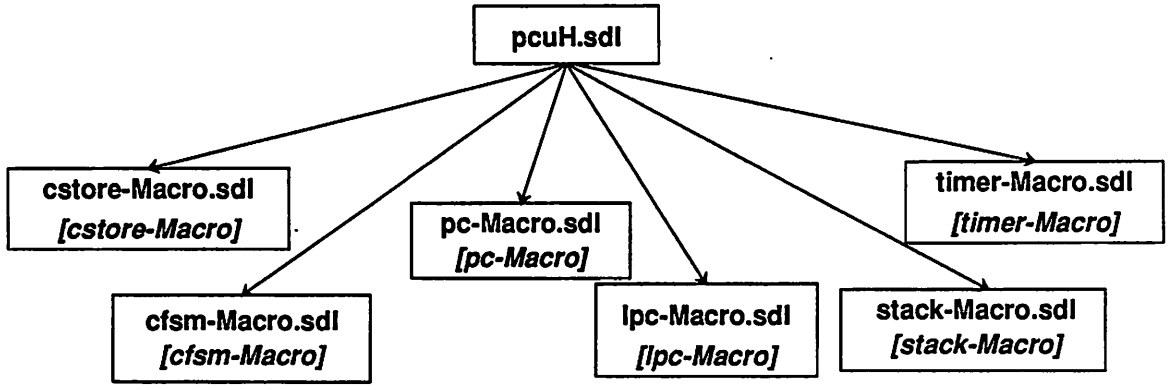


Figure 5.5: Structural organization of the PCU macrocell.

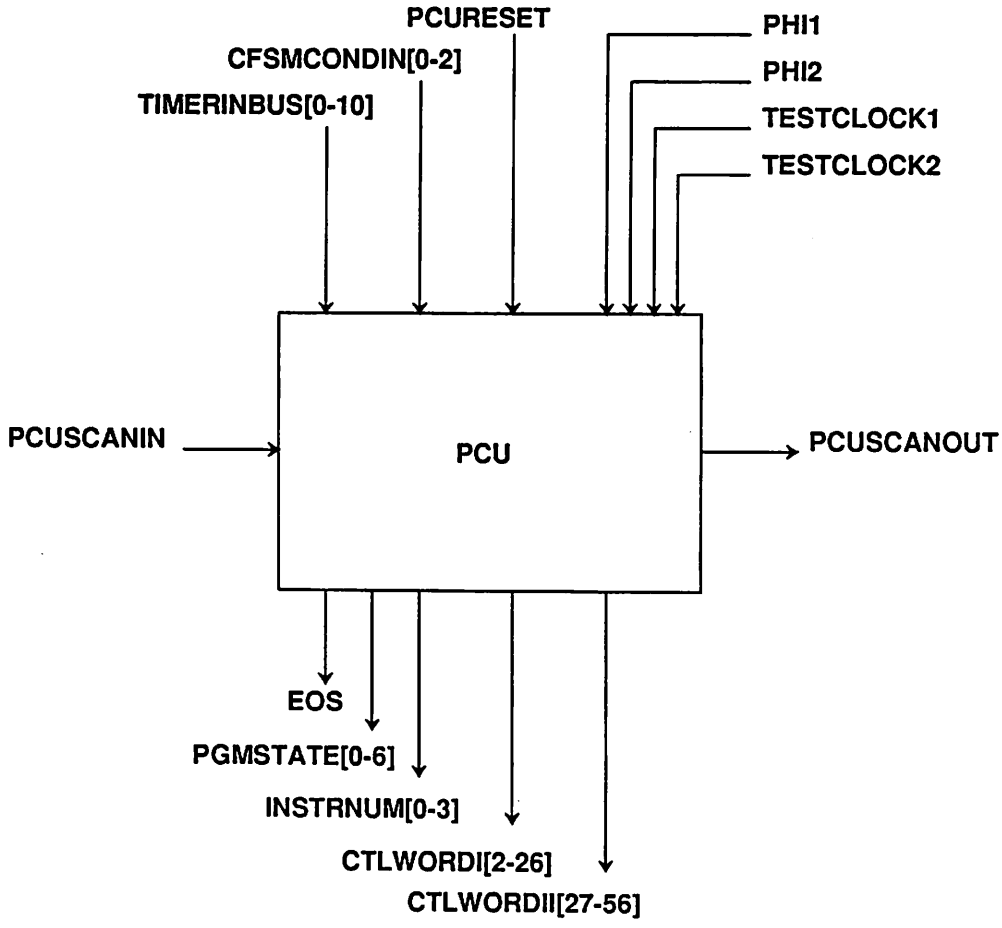


Figure 5.6: Boundary terminals of the PCU. The number inside [] give the actual bus widths for the robot controller design.

CFSMCONDIN: These are the control signals going into the PCU and are tested for executing conditional branch operations. CFSMCONDIN[0] is the APU sign bit, CFSMCONDIN[1] is the AU sign bit, and CFSMCONDIN[2] is an off-chip signal used for optionally skipping the integration operation of the PID section of the control algorithm. This is synonymous with the IFLAG signal referred to in chapter 2.

PCURESET: This is the reset signal for the PCU and is connected with the external RESET signal. It initializes the program counter to zero and the finite state machine to state zero.

PHI1, PHI2: These are the phase 1 and phase 2 respectively of the two-phase master clock generated inside the Test-logic macrocell.

TESTCLOCK1, TESTCLOCK2: These are phase 1 and phase 2 respectively of a two-phase test clock. Details about using the test clocks are described in chapter 6.

PCUSCANIN, PCUSCANOUT: Single-bit scan-in and scan-out lines respectively for testing purposes. Details are described in chapter 6.

EOS: This is the end-of-sample signal and is asserted when the timer completes its count of the length of the sample period. The signal is used within the PCU for starting a new iteration of the program at the end of a sample. EOS is also available externally for monitoring purposes.

PGMSTATE: These are the state bits of the FSM inside the PCU and are available for monitoring by the testing circuits as described in chapter 6.

INSTRNUM: These are the program counter output bits and provide the low-order bits of the current instruction's address being executed; high order address bits are provide by the block address. INSTRNUM is also used by the test circuits.

CTLWORDI, CTLWORDII: These are the control-signals generated by the control-store and specify the microoperations to be performed during each cycle. For convenience in placement during layout, the control-word is split into two smaller sections referred to as CTLWORDI and CTLWORDII.

5.4.2 Circuit Design

The main circuit components of the PCU are PLA, counters, mux, registers, and some random logic. All the six child macrocells of the PCU are built from these basic circuits.

cstore-Macro

The cstore-Macro consists of three child cells: PLA and two scan-latches. The PLA holds the control-words which specify the microoperations. By using a PLA we can take advantage of minimization programs and also delete unused locations. The PLA itself doesnot have any input or output latches; external scan-latches are provided for latching the data. The address inputs are assumed to be latched at the source (cfsm-Macro and pc-Macro), whereas for the outputs, latches are provided within the cstore-Macro. In order to provide greater freedom for producing a more compact layout, the output latch is divided into two, approximately equal sections – CSTOREOUTREGI and CSTOREOUTREGII. The overall block diagram of the cstore-Macro is shown in figure 5.7.

The block address from the FSM provide the high order bits of the pla address whereas the instruction address from the program counter provide the low order address bits. The outputs from the pla are active high and are latched by the two latches. The least two significant bits of the control-word from the pla are EOB and LDTIMER signals. The complemented output is used from the latch for LDTIMER as required by the timer register. The pla and the latches use only phase 2 of the clock. Thus, the control-word becomes valid during phase 2 and is latched on its falling edge. In addition to the normal clocks, both the latches have a two-phase test clock for serially shifting scan data in and out through SCANIN and SCANOUT terminals. The detailed design of the scan latches are described in chapter 6. We now discuss the design of the pla.

The PLA circuit is shown in figure 5.8. The output lines are precharged when CLOCK (in the case of cstore-Macro, CLOCK=phase 2) is low and evaluated when CLOCK is high. Each minterm line uses a weak PMOS pullup. When a valid

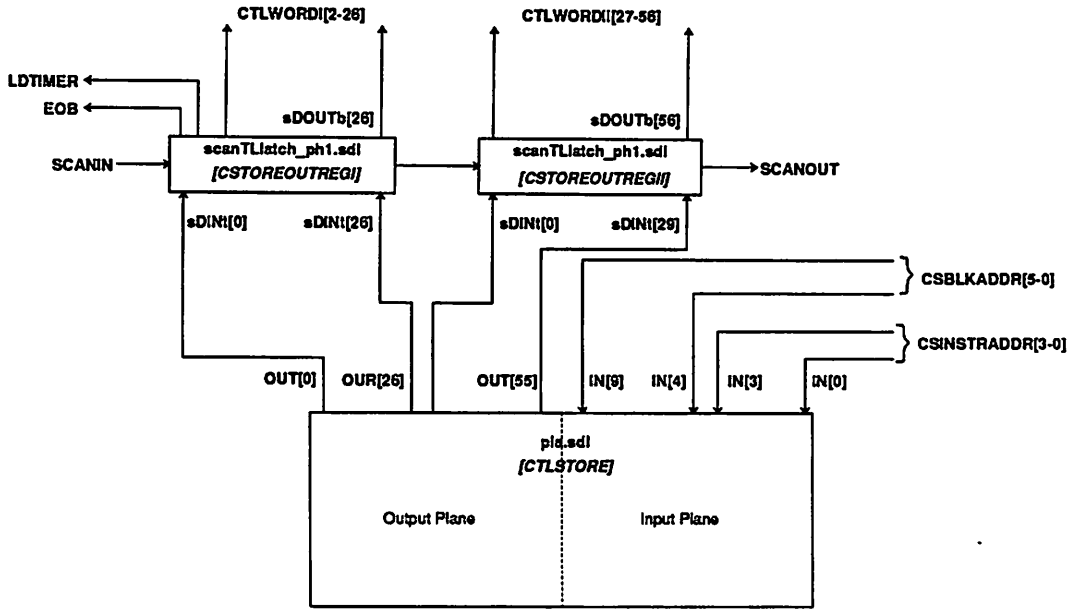


Figure 5.7: The cstore-Macro consists of a pla and two scan-type latches.

input is set up, all the minterm lines are pulled low except the one selected. As a result the output lines tied to the selected minterm line go high; outputs are active high. There is also a weak pullup on the output lines for preventing charge sharing problems. The timing diagram for the operation of the PLA is shown in figure 5.9. The PLA and the latch circuits are generated using the TimLager tiler program since it provides the most compact layout for array type structures.

cfsm-Macro

The cfsm-Macro (figure 5.10) consists of four child macrocells: PLA, two scan-latches, and a mux with a built-in latch. The PLA implements the control-flow state transition table. As in cstore-Macro, the outputs are latched by two latches CFSMOUTREGI and CFSMOUTREGII. The first one latches the state outputs whereas the latter one latches the various signals used for controlling the operation of the PCU. These are: block-address for the control-store, LPINC for incrementing the loop-counter, and TIMER_RST2 for reseting the timer. The block diagram in figure 5.10 does not show any control signals for stack operation since

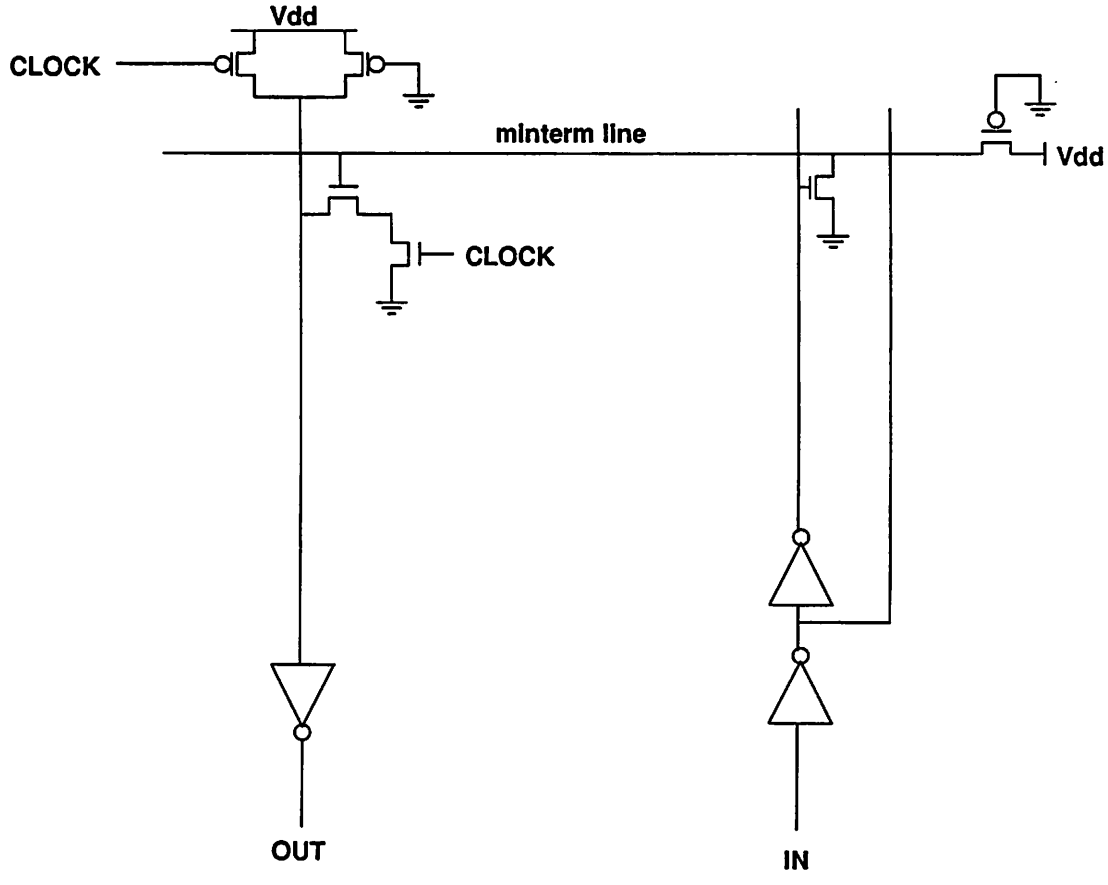


Figure 5.8: PLA circuit showing one input line, one minterm line and one output line. The active high outputs become valid when CLOCK, which may be connected to phase 1 or phase 2, goes high.

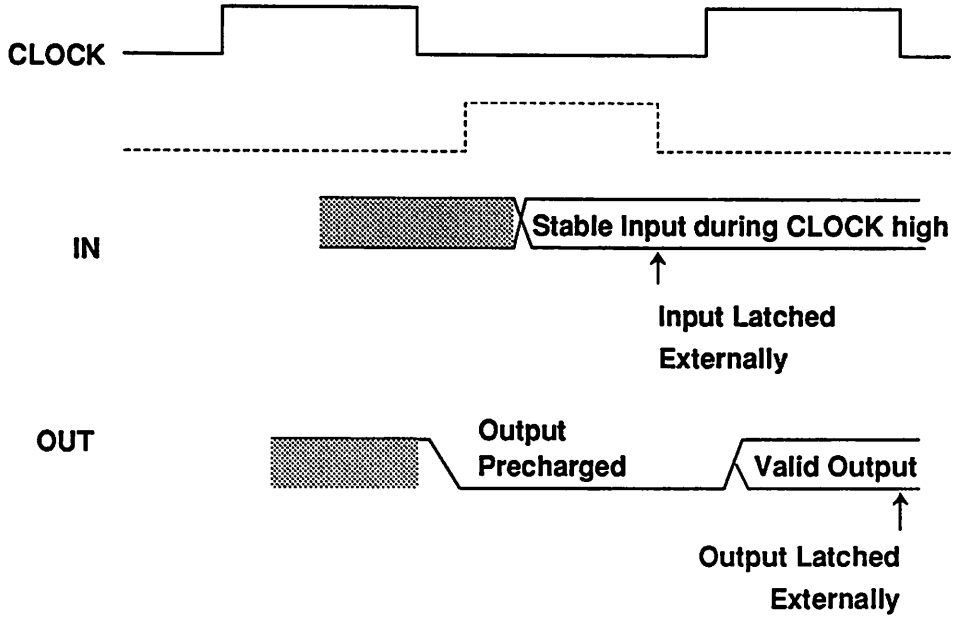


Figure 5.9: Timing diagram showing the operation of the control-store.

in the robot control processor, non-stack based subroutines (see chapter 3) were used. In order to feed the state outputs back into the PLA for FSM action, the output from CFSMOUTREGI are latched again, using the second phase of the clock, in a combined mux/latch circuit called CFSMREG. CFSMOUTREGI and CFSMREG together form a state feedback register. The mux selects between two inputs: either the fsm's state output or the output from the stack. The stack output is normally connected to the CFSMREGIN2 terminals but in this case these terminals are grounded since a stack is not used. The mux control signal, SEL_IN2, is also grounded which causes the mux to always select the state outputs. As shown in figure 5.10, the mux output forms part of the PLA's input vector. The remaining inputs are: ENDLOOP signals, which detect completion of specific loop counts - in this case two different count values; EOB, end-of-block signal from the control-store; COND, which are the conditional signals connected to APU sign, AU sign, and the external IFLAG pin; and EOS, end-of-sample signal from the timer.

The details of the circuit for the PLA has been discussed earlier in the section on cstore-Macro. We now describe the circuit of the mux_latch shown in

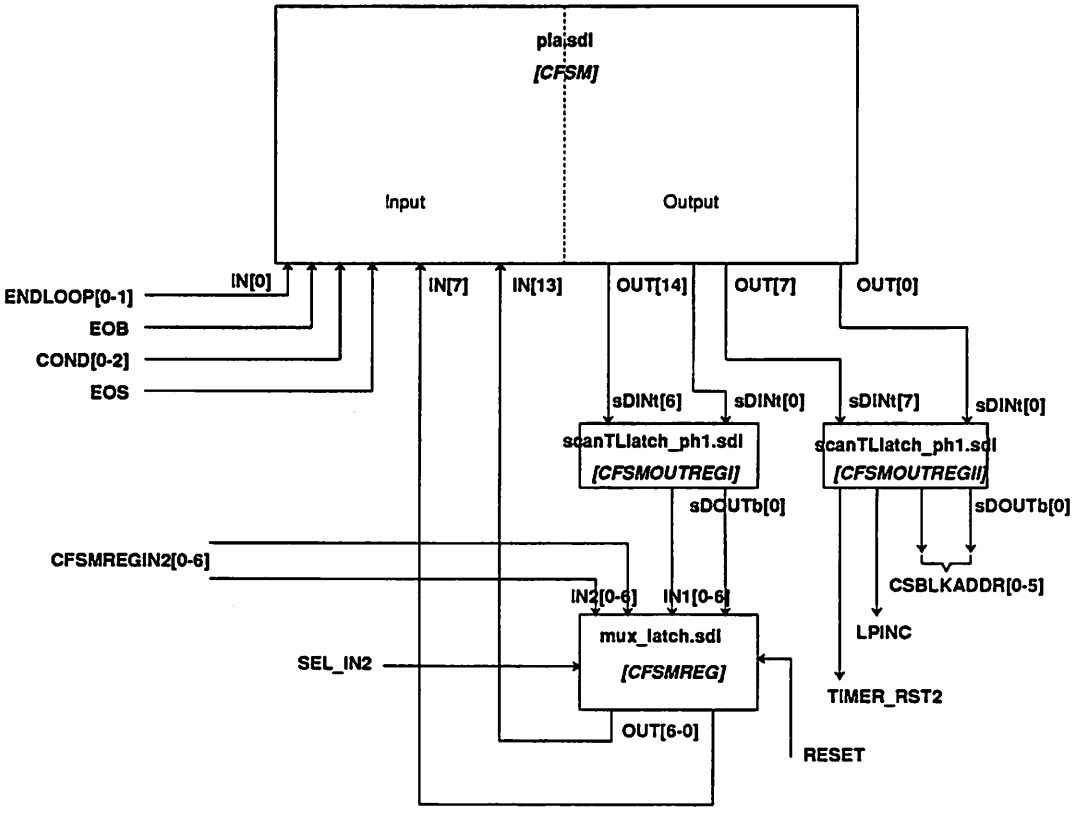


Figure 5.10: Block diagram of the PCU's finite state machine.

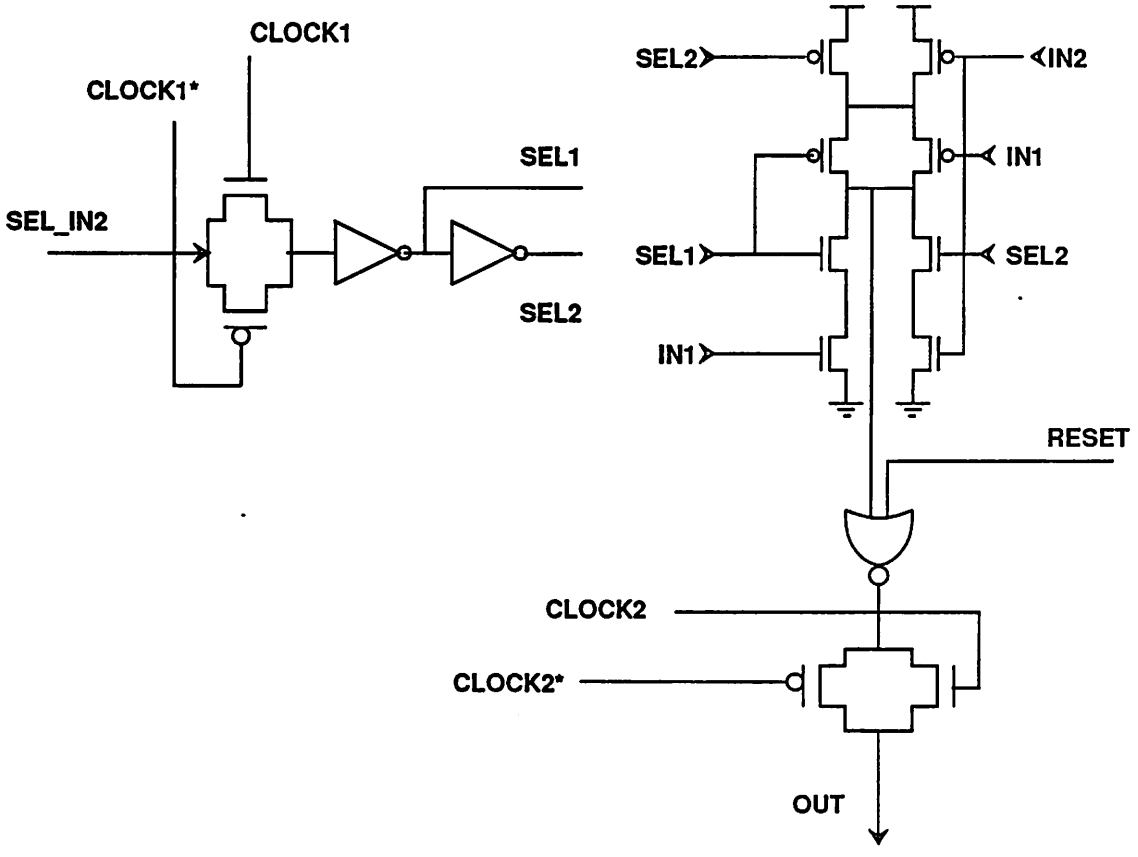


Figure 5.11: Mux_latch circuit is used to latch either the state output from the FSM or the return state from the stack. Mux_latch also provides means for forcing the FSM into zero state for reset purposes.

figure 5.11. Basically, it consists of a 2X1 mux whose output is latched on CLOCK2. The mux output switches to IN2 when SEL_IN2, latched on CLOCK1, is high. The nor gate provides a mechanism for forcing the output to zero when RESET is applied. Since the mux output is applied to the FSM input, the FSM is forced into zero state when RESET is asserted. Without the nor gate, the alternative would be to allow a state transition to the reset-state from every state in the program. This of course would result in a very large number of additional entries in the state table.

The cfsm-Macro uses both the clock phases for its operation. CLOCK1

is tied to phase 1 and CLOCK2 is tied to phase 2 of the master clock whereas the PLA's CLOCK is tied to phase 1. Thus the cfsm-Macro's timing scheme is as follows. The FSM is precharged when phase 1 is low and is evaluated when phase 1 is high. On the falling edge of phase 1 the FSM's outputs are latched. On the following phase 2, the state output of the FSM is transferred to the FSM's input through the mux_latch. Thus at the end of phase 2 the FSM is ready to start evaluating a new state.

The layout generation for the child macrocells of the cfsm-Macro are done with the TimLager tiling program. On the other hand, the child macrocells are connected together with Flint. All the latches, mux_latch, and the PLA are parameterized. For example, the number of conditional inputs to the FSM is a parameter in the csfm-Macro.sdl file. The layout and interconnections are automatically adjusted to take into account the number of conditional inputs.

Program Counter (PC)

The pc-Macro consists of two child cells: a counter and a latch as shown in figure 5.12. The counter consists of alternating odd and even counter leafcells tiled together with TimLager. In addition, a single leafcell containing some random control-logic is also tiled with the counter as shown in figure 5.13. The counter leafcells consist essentially of a half-adder and carry circuit (figure 5.14). The carry-in of the LSB slice is tied to Vdd so that the counter always operates in an increment mode. The previous output of the half-adder is fed back into the adder input after a single clock-cycle delay through a dynamic register. All the storage nodes in the counter are dynamic. An alternative input into the dynamic register is the initial-count. For initialization, the counter can be forced to a pre-specified initial value by applying a reset signal. Three different reset signals can be applied: RESET, EOB, and EOB2. In order to tie any unused reset terminal to ground, the ZEROpc terminal, which is connected to GND, is provided. The initial-count is specified during layout generation. Based on the initial-count value, the *init-count* terminal of each counter cell is connected to either GND or Vdd. The outputs become valid

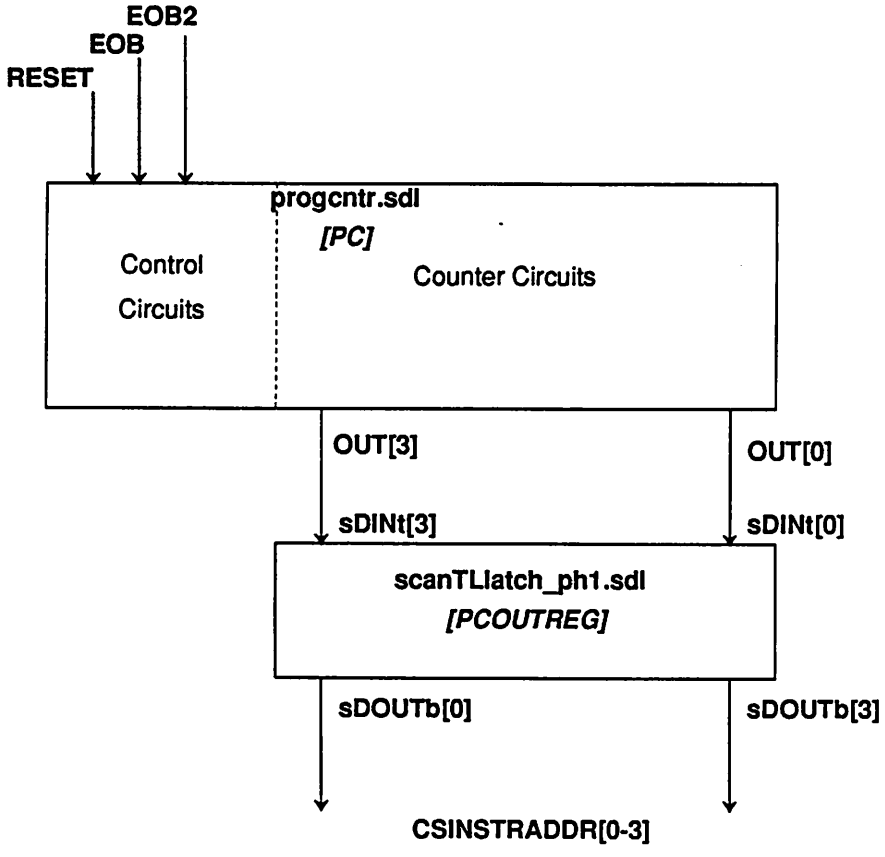


Figure 5.12: pc-Macro consists of two child cells: counter and latch.

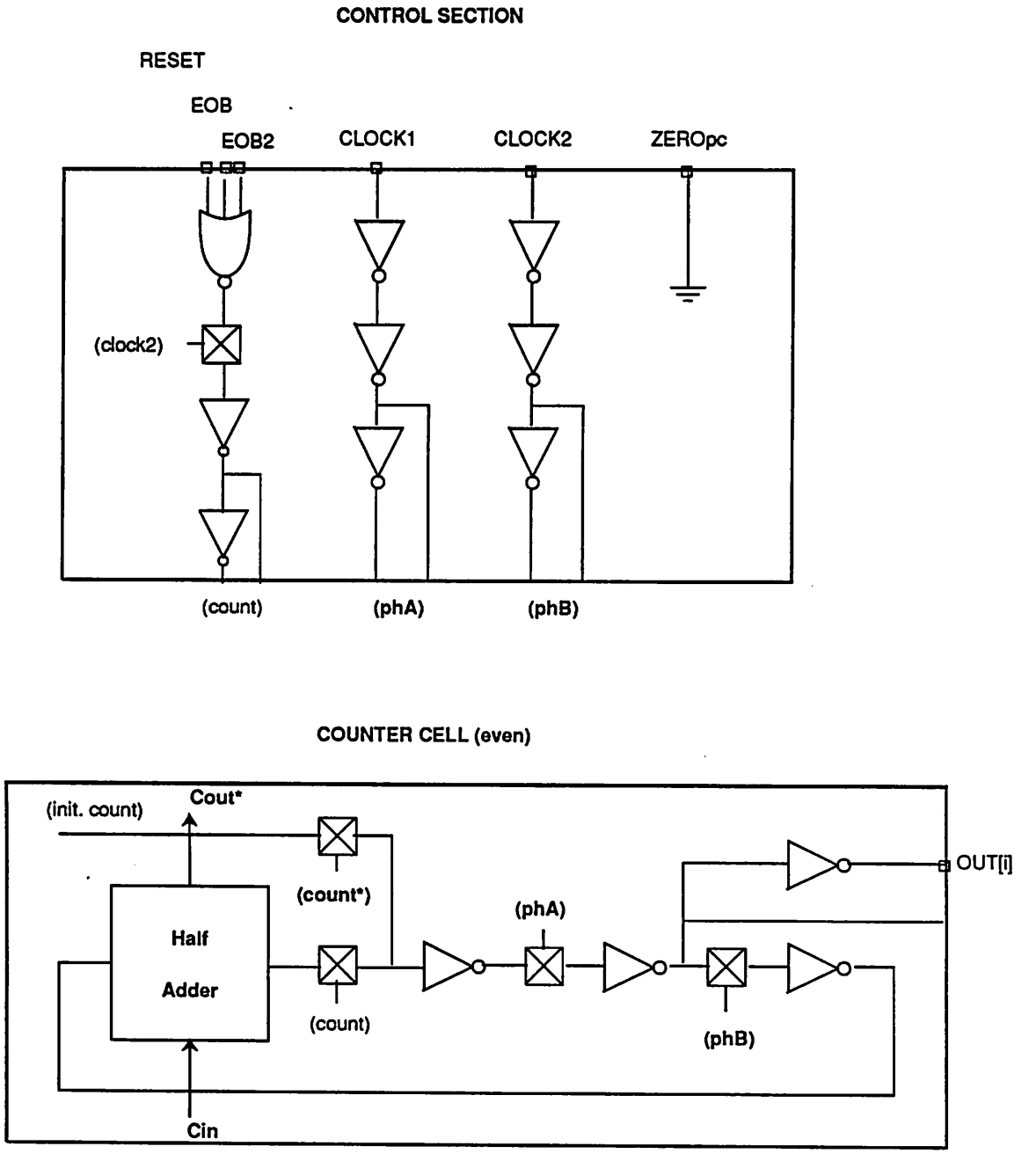


Figure 5.13: Control and counter circuit for the program counter.

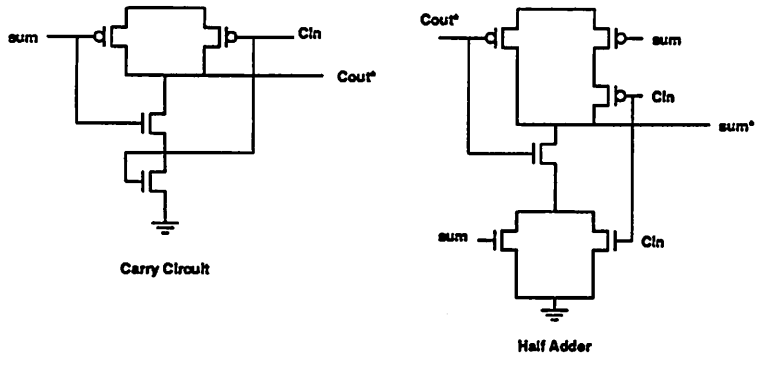


Figure 5.14: Half-adder and carry circuits for the even-slice of the counter. The sum input is the previous sum whereas sum* is the complement of the new sum. The odd-slice circuits are just the complement of the even-slice.

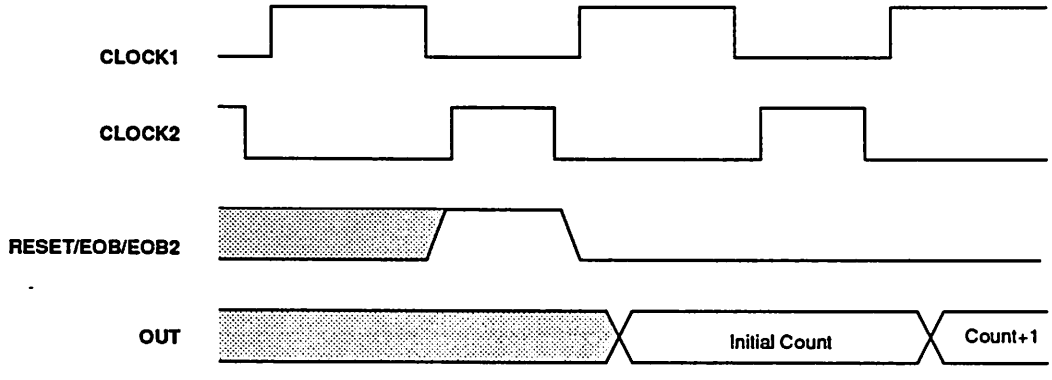


Figure 5.15: Timing diagram for the program counter.

on CLOCK1 as shown in the timing diagram in figure 5.15.

Loop Counter (LPC)

The loop counter macrocell, lpc-Macro, is very similar to the program counter in design and layout organization. It consists of a loop counter and a latch. Only the control section of the counter is different as shown in figure 5.16. The LPINC output from the control unit's FSM is held high during any state where looping is required. This enables the loop counter. Actual increment in the loop

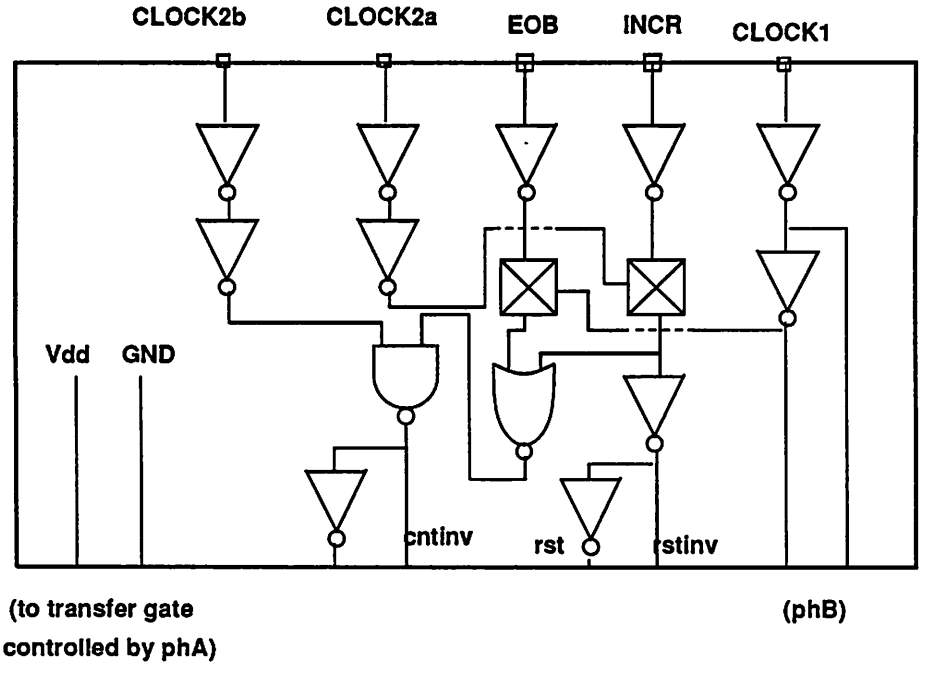


Figure 5.16: Control circuits for the loop counter.

counter, however, takes place only when the EOB signal is asserted at the end of a block of code. The counter resets when LPINC is low. In order to implement this behavior, the phA latch in the counter cell (see figure 5.13) is always turned ON. As shown in the timing diagram in figrure 5.17, the counter increments on phase 1 of the master clock. The count-detect signals also becomes valid during phase 1 when the appropriate count is reached. The output of the counter and the count-detect signals are latched by a scan-type latch, LPCOUTREG, on phase 1. Another difference between the loop counter and the program counter are the count-detect circuits. When the loop counter reaches a pre-specified count, the corresponding count-detect circuit, hard wired to detect the particular count, asserts a detect signal called ENDLOOP. This signal is used by the FSM for making the next state transition. Each count to be detected has its own detection circuit, which is a simple nand gate as shown in figure 5.18. The number of nand gates and the count each one detects are determined during layout generation. Figure 5.18 also shows

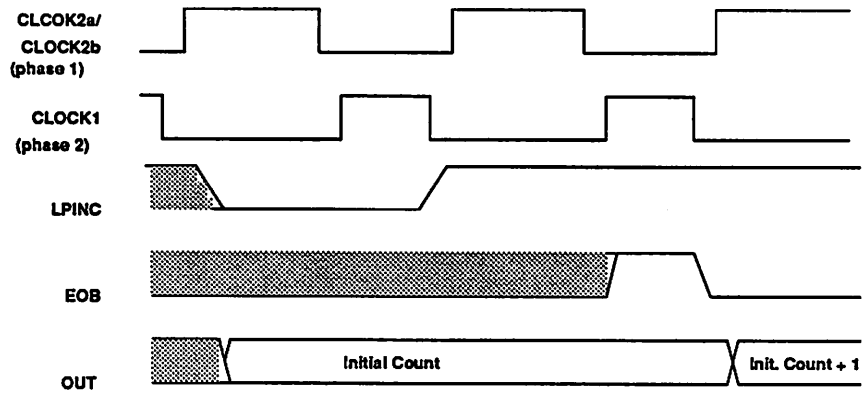


Figure 5.17: Timing diagram for loop counter.

the schemetic of a cell, several of which are tiled together to form a complete nand gate.

Timer

Like the program counter and the loop counter, the timer is also built around the same basic counter circuit. The timer-Macro consists of three child macrocells: timer, scanTLlatch_ph1, and scanTLreg2Port and is used for setting the sample period. A block diagram of the macrocell is shown in figure 5.19. The timer is assembled from alternating odd and even counter leafcells and a control-logic cell. The initial-count, however, is not hard-wired during layout generation. Instead, initial-count terminals are brought out and connected to the output of scanTLreg2Port (TIMERINREG), which is a scan-type register. Since the timer's even slices require inverted inputs, complementary outputs from the TIMERINREG are used for connecting with the even numbered input terminals on the timer. TIMERINREG is loaded with a desired initial-count value from the data path under program control. The LDTIMER (see figure 5.7) signal of the control-word is applied to the LOADINV terminal of TIMERINREG for loading the initial-count. Normally, the processor first reads this initial-count value from an external source

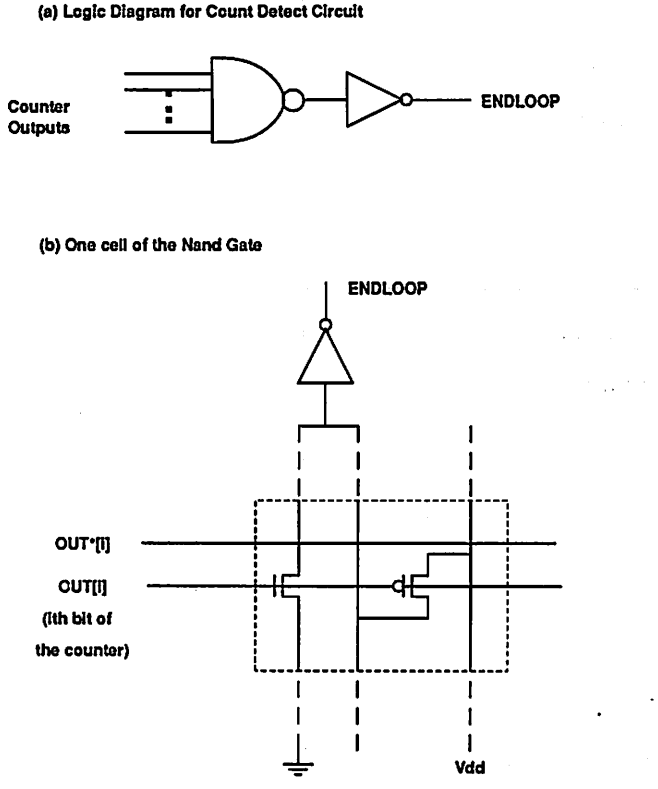


Figure 5.18: Count-detect circuits for the loop counter.

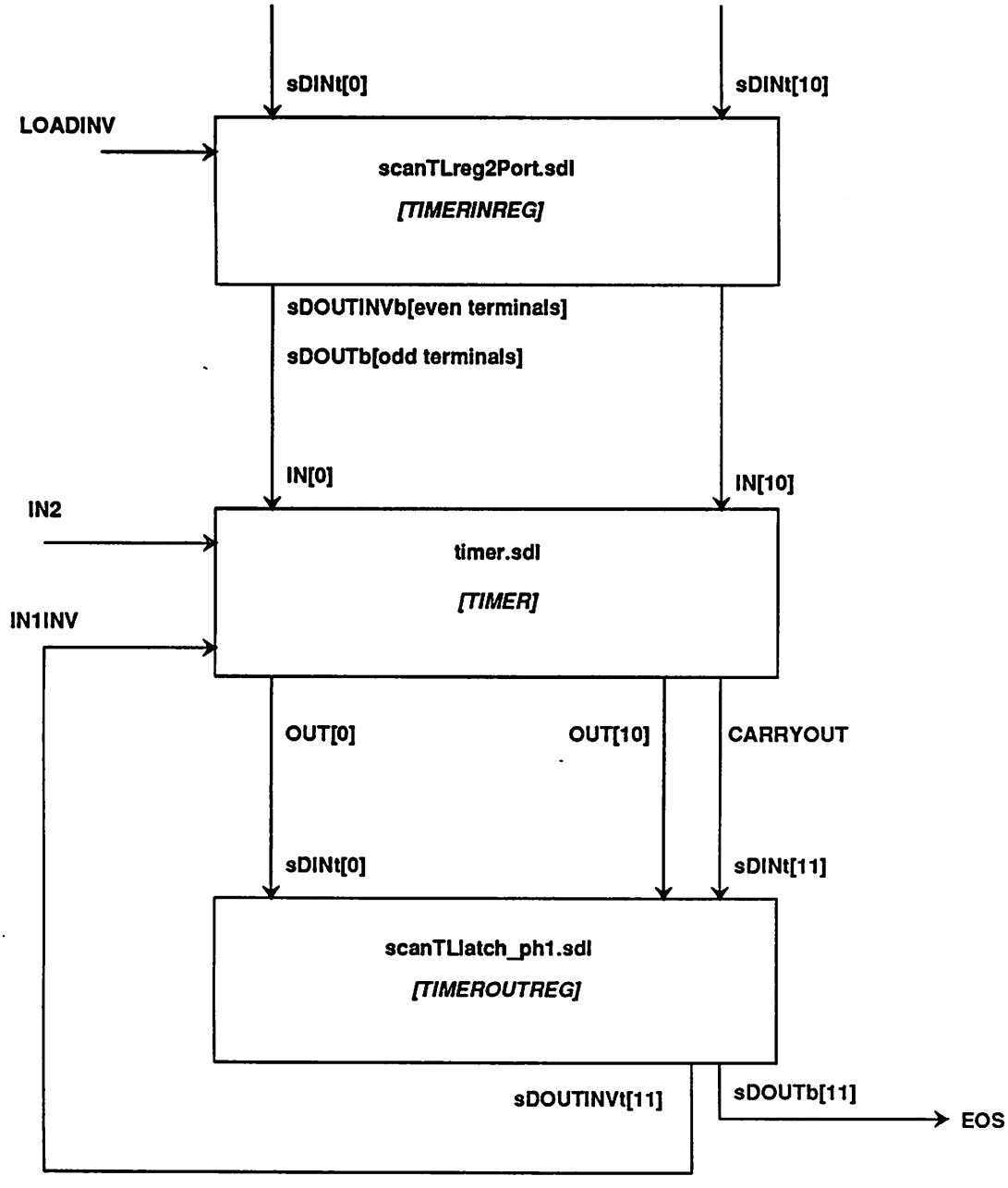


Figure 5.19: Block diagram of the timer-Macro.

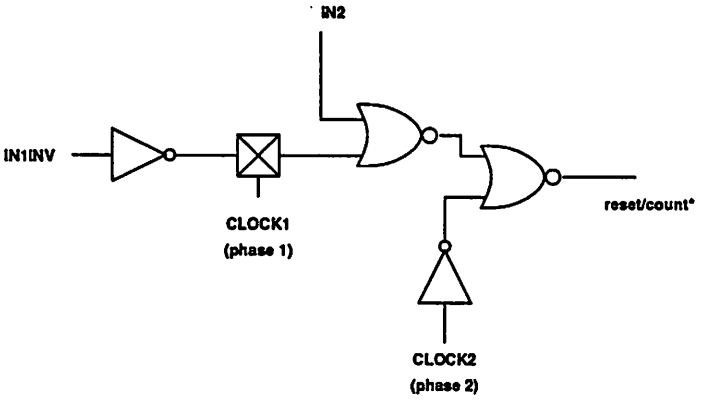


Figure 5.20: Control circuit for reseting the timer.

through the I/O ports. Thus, the user can specify a desired initial-count which in turn determines the sample period.

Since upon reset the counter is initialized to init-count, the length of the sample period is given by $(2^n - \textit{init-count})$ where n is the number of bits in the counter. At the end of the sample period the timer overflows and EOS, which is the latched carry-out signal from the timer, is asserted. This EOS is used by the FSM for making the appropriate state transition. At the same time an inverted version of EOS from TIMEROUTREG (see figure 5.19) is applied to the terminal IN1INV (see figure 5.19 and figure 5.20) of the timer. This generates a reset signal as shown in the reset-control circuit of the timer in figure 5.20. An alternative way of reseting the timer is by pulling high the IN2 terminal of the timer, also shown in figure 5.20. IN2 is actually controlled from the FSM in order to keep the timer reset during initialization states of the program. The timing diagram for the timer is shown in figure 5.21.

Stack

The stack macrocell, stack-Macro, is used for handling subroutines in the control unit although a more limited form of subroutine can also be implemented without the stack as discussed in chapter 3. Stack-Macro consists of two child macrocells: basicStack and scanTLlatch_ph1 as shown in figure 5.22. Both the

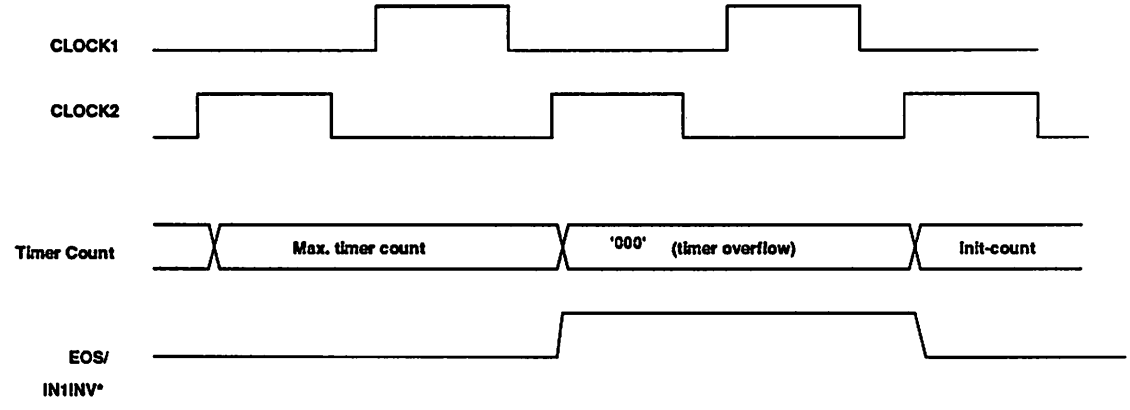


Figure 5.21: Timing diagram for the timer.

macrocells are generated through tiling with TimLager and are connected together with Flint. The width of the latch and the stack as well as the depth of the stack are all parameterized and determined during layout generation.

The operation of the stack is controlled by two control signals, POP and PUSH, from the FSM. When making a transition to the subroutine state, the FSM asserts the PUSH signal and applies the return-state to the stack's input, TOSTACKIN. When making a return transition back from the subroutine-state, the FSM asserts the POP signal which causes the stack to be popped after the data from the stack's output has been used by the FSM. The output data is latched by a scanTLlatch_ph1 latch and is available at its output terminals. The circuit diagram of a stack register cell and the control circuits are shown in figure 5.23

Figure 5.24 gives the timing diagram for subroutine call and return along with the associated state transitions. In the figure, the EOB signal at time interval t1 causes a state transition to a subroutine-state. At the same time a return-state and a PUSH signal is put out by the FSM during phase 1 of the clock cycle. In the following phase 2, the stack control circuit generates the sshd signal from the PUSH signal, causing the return-state to be pushed on the stack. At the end of the subroutine block another EOB signal is asserted during time interval t2. This causes the FSM to make a transition to a dummy state as explained in chapter 3. At the same time the FSM puts out a mux-select signal, SEL_IN2, and a POP

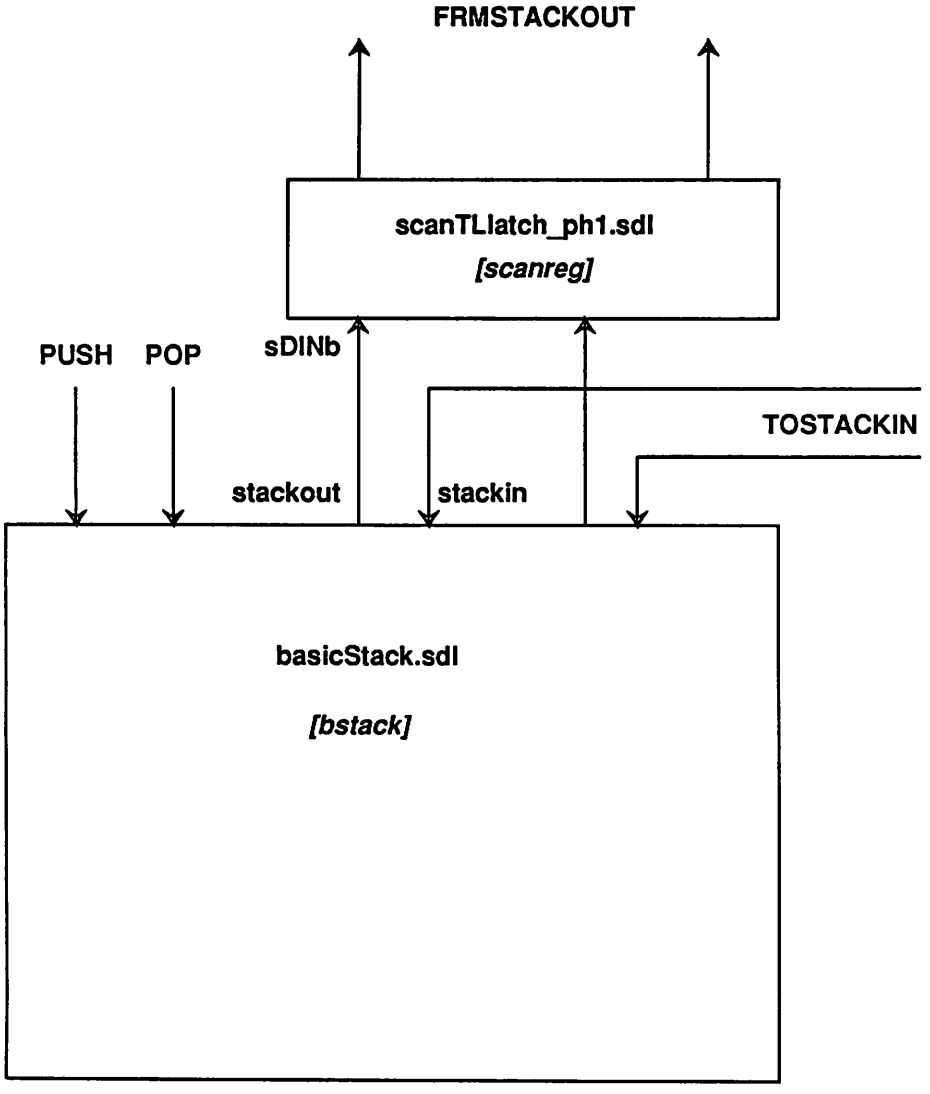


Figure 5.22: Stack-Macro consists of a basic stack and a scan-type latch.

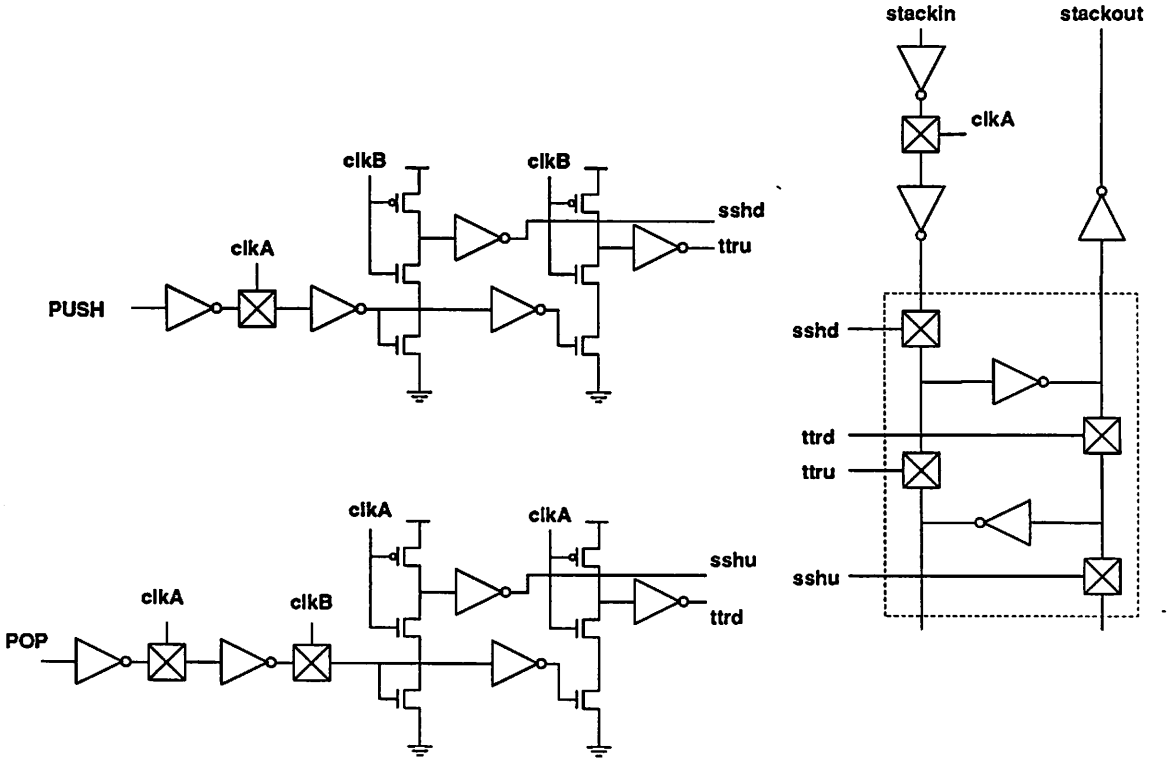


Figure 5.23: Circuit diagram of a stack cell and stack control circuits. The circuit within the dotted box contains the basic stack register cell. This cell is replicated horizontally for forming the stack word and vertically for increasing the stack depth.

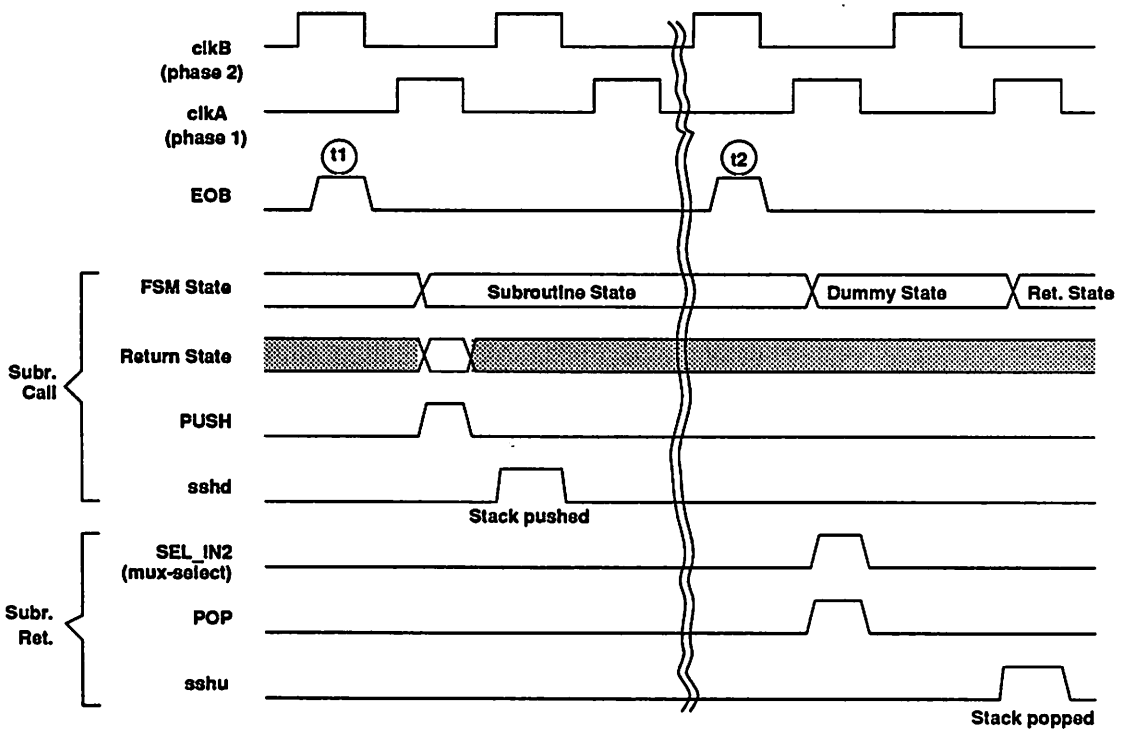


Figure 5.24: Timing diagram showing stack operations for subroutine call and return. EOB signal and FSM state transitions are also shown.

signal. SEL_IN2 causes the mux inside the FSM (see cfsm-Macro description in this chapter) to select the stack output, which is the return-state. This return-state is applied to the input of the FSM during phase 2, causing the FSM to go into the return-state on following phase 1. After the FSM has made the transition, the sshu signal is applied by the stack control circuit to the stack thus forcing the next level return-state (for nested subroutines) on to the top of the stack. Note, in the dummy state, EOB2 is asserted instead of EOB to keep the program counter reset but not cause another state transition after the fsm has gone into the return state.

In the robot control processor, the stack was not used. However, a separate test-chip was fabricated for evaluating the control unit including the stack. This chip was fully functional, verifying the control unit design.

5.5 Arithmetic Unit (AU)

The arithmetic unit is the main computational engine of the processor. The AU consists of two major macrocells: the control circuits, aucntl; and the data path, audp. An important consideration in designing the data path is to make it easily adaptable for different applications. This is achieved in two ways. First, by having a library of a small number of hand designed data path leafcells such as register, adder, mux, etc. Second, by implementing the local, random logic control circuits with standard cells. The control circuit generates the local control signals for the data path circuits from the primary control signals put out by the control-store inside the PCU. The standard cell implementation of the control logic is done by simply specifying the logic in the form of a boolean equation. The equations are then converted into an sdl file by a program called eqn2sdl [Ma87]. The Design Manager (see chapter 4) is then able to generate the layout by calling the standard cell place and route program, Wolfe. With this scheme the control behavior of the data path cells is quickly redefined by simply changing the boolean equations. On the other hand, the data path is also easily reconfigured by rewriting the data path sdl file. An example of a data path cell and its control logic implementation is shown in figure 5.25. The entire data path is composed of several data path leafcells connected together.

The circuit diagram for the arithmetic unit data path is shown in figure 5.26. Details about each section of the circuit and its associated control logic are discussed later in this chapter. The data path has a three stage pipeline as discussed in chapter 3. Within the data path certain operations take place on phase 1 of the clock and other operations occur during phase 2 as listed below.

phase 1:

enable register[0/1] output

shift

phase 2:

load register[0/1]

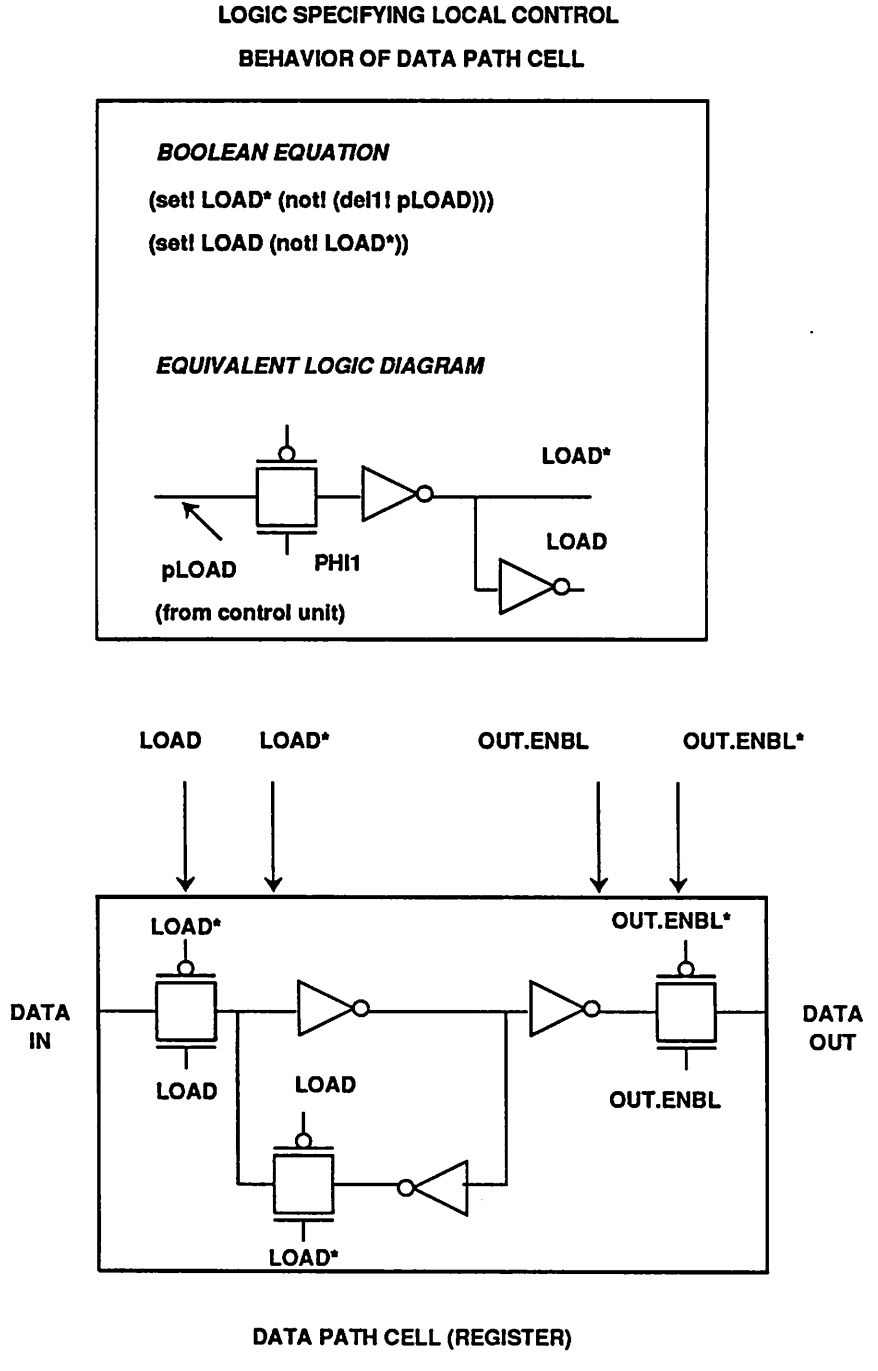


Figure 5.25: A data path cell and its control logic. The data path cell is usually assembled from hand designed bit-slice register cells where as the control logic is implemented with standard cells. The standard cell layout is generated from a boolean expression. The figure shows the generation of LOAD and LOAD* signals from the primary control signal, pLoad.

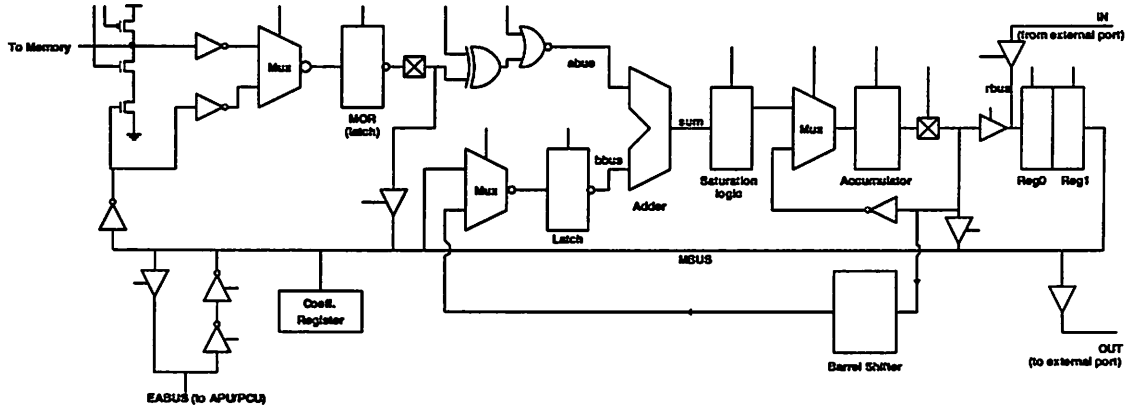


Figure 5.26: Schemetic of the arithmetic unit data path.

memory read/write
arithmetic oper./accumulate

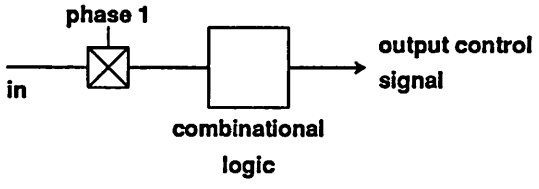
In order to ensure the interconnections between the various data path cells do not lead to any timing conflict, certain composition rules must be observed.

5.5.1 Data Path Composition Rules

Composition rules, based on timing schemes, define a strategy for connecting data path cells in order to obtain a functionally working circuit. Although the control-word in the control unit becomes valid on phase 2, the clock phase in which the execution of a particular operation takes place depends on the local control circuit of the data path. Based on the clock phase or phases in which a control signal is active, three basic types of control circuits – Type 1-2, Type 1, and Type 2 – are identified (figure 5.27). These circuit types are defined on the basis of a two phase clocking scheme in which the input signals are latched inside the control unit on the falling edge of phase 2 and therefore remain stable during the following phase 1. Since Type 1-2 circuits make available both the clock phases for completing an operation, it is the most commonly used control circuit type unless some constraint forces use of the other circuit types. In general, Type 1-2 is used for controlling

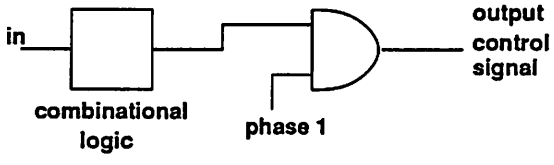
Type 1-2:

Control signal is active during both phase 1 and 2.



Type 1:

Control signal is active during phase 1 only.



Type 2:

Control signal is active during phase 2 only.

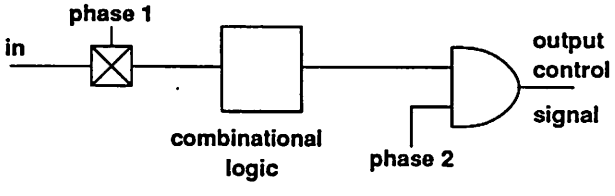


Figure 5.27: Three basic types of circuits used in the local control circuits of the data path.

data transfer between busses, triggering arithmetic/logic operations, enabling logic steering circuits (e.g. mux), etc. Type 1 circuit is usually used for enabling the outputs of two-phase registers. It ensures that registers are read only on phase 1 so that a new data may be loaded into the register on phase 2. A data path cell controlled by Type 1 circuit has at most one clock phase to complete its operation. Data path cells controlled by Type 2 circuits also have at most one clock phase to complete their operation. Type 2 circuit is usually used for loading registers since we would like to load a register only after allowing a possible read operation during phase 1. This scheme also allows time for transferring data from the source to the destination register during phase 1.

In constructing a data path, the following minimum rules must be observed.

(i) Data path cells controlled by the different control circuit types may be connected in any combination except one: A Type 1 data path cell must not appear after a Type 2 data path cell (even if there are intervening Type 1-2 cells) within the same pipeline stage. A pipeline register itself, though, may have a Type 2 circuit controlling the input (load) and a Type 1 circuit controlling the output enable.

(ii) A Type 1 data path cell using a pre-charge (obviously on phase 2) must be followed by a Type 1 latch (not necessarily a pipeline register) before being connected to the input of a Type 2 circuit.

The above rules are the minimum constraints on connecting the data path cells. The usual rules for connecting CMOS domino type circuits [Nel83] and for mixing static and dynamic circuits (dynamic circuit must not follow a static circuit within the same pipeline stage) still apply. Special rules, specific to a data path cell, may also restrict the type of cells that can be connected at the input and/or the output. Furthermore, the number and the combination in which the cells are connected within a pipeline stage will affect the critical path. We now describe in detail the construction of the data path from the basic cells.

5.5.2 Circuit Design

The data path is divided into eight sections. These are,

- Memory read/write
- Arithmetic operations
- Coefficient register
- Saturation Logic
- Accumulator
- Registers and I/O ports
- Barrel Shifter
- MBUS to EABUS Interface

In the figures for the data path's sections, dotted box show each data path leafcell, names in caps are bus names and control signal names, names within () usually refer to the clock phase being connected to the terminal, and names in italics within [] are the instance name of the leafcells (corresponding generic names can be found from the sdl file). All the primary control-signals coming from the PCU begin with the letter 'p' (e.g. pRDPORT) and all the status signals coming out of the data path start with 'd' (e.g. dSUMMSBINV).

Memory Read/Write

The memory read/write section of the data path is shown in figure 5.28. The control circuits are shown in figure 5.29. The RAMIO terminal connects directly with the bit-line of the RAM. The bit-line is precharged inside the MEMIO cell on phase 1. For writing data into the memory, WEN is held high and the data is applied to the bottom NMOS transistor in MEMIO cell. This data comes from the MBUS. The write enable signal, WEN, is generated when the control signal pW is true. Alternatively, a conditional write operation is triggered when pWC is true

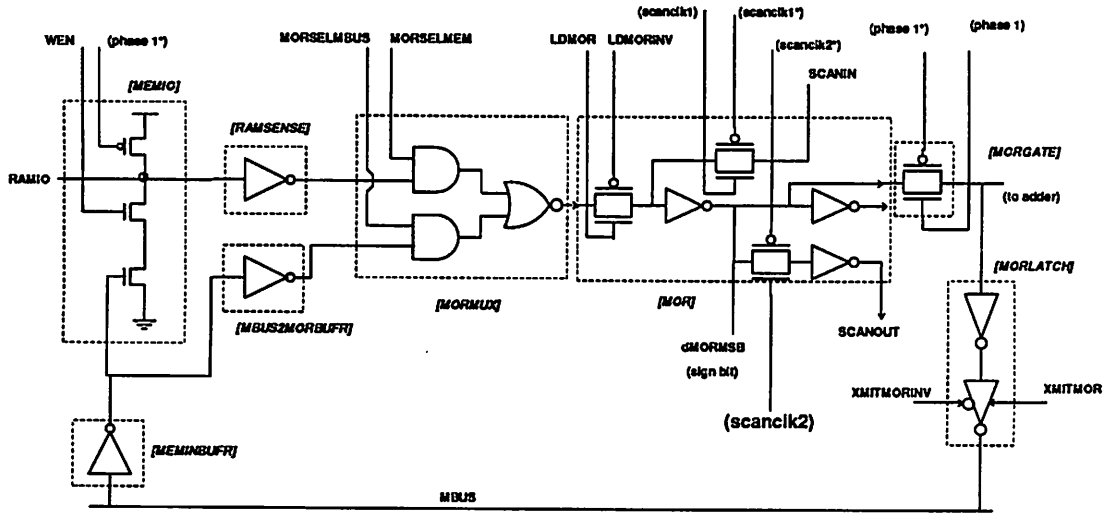


Figure 5.28: Local memory interface and memory output register circuits.

and the condition cc is valid. cc is generated by the logical unit (LGU). The control circuit also generates a WRITE signal which is used in the RAM.

For reading the RAM, the control unit asserts pR which generates READ for use in the RAM. A read operation dumps the data on the RAMIO which is sensed by the RAMSENSE inverter. The PMOS transistor of this inverter has been sized large (10/3) for giving a high inverter threshold in an attempt to improve the sensing speed of any drop in the bit-line's pre-charged voltage. The output of the RAMSENSE is loaded in the MOR (memory output register) latch through a mux. MOR is a pipeline register and its output may be sent either to the MBUS or to the adder (not shown in figure 5.28). The output of MOR is actually enabled via MORGATE on phase 1. The MOR's MSB is brought out for testing the sign of the data in the MOR. As data moves around the loop from MOR to MBUS to memory and then read back into MOR, a consistent polarity is maintained; there is no data inversion in writing the MOR data into memory and reading it back from the memory.

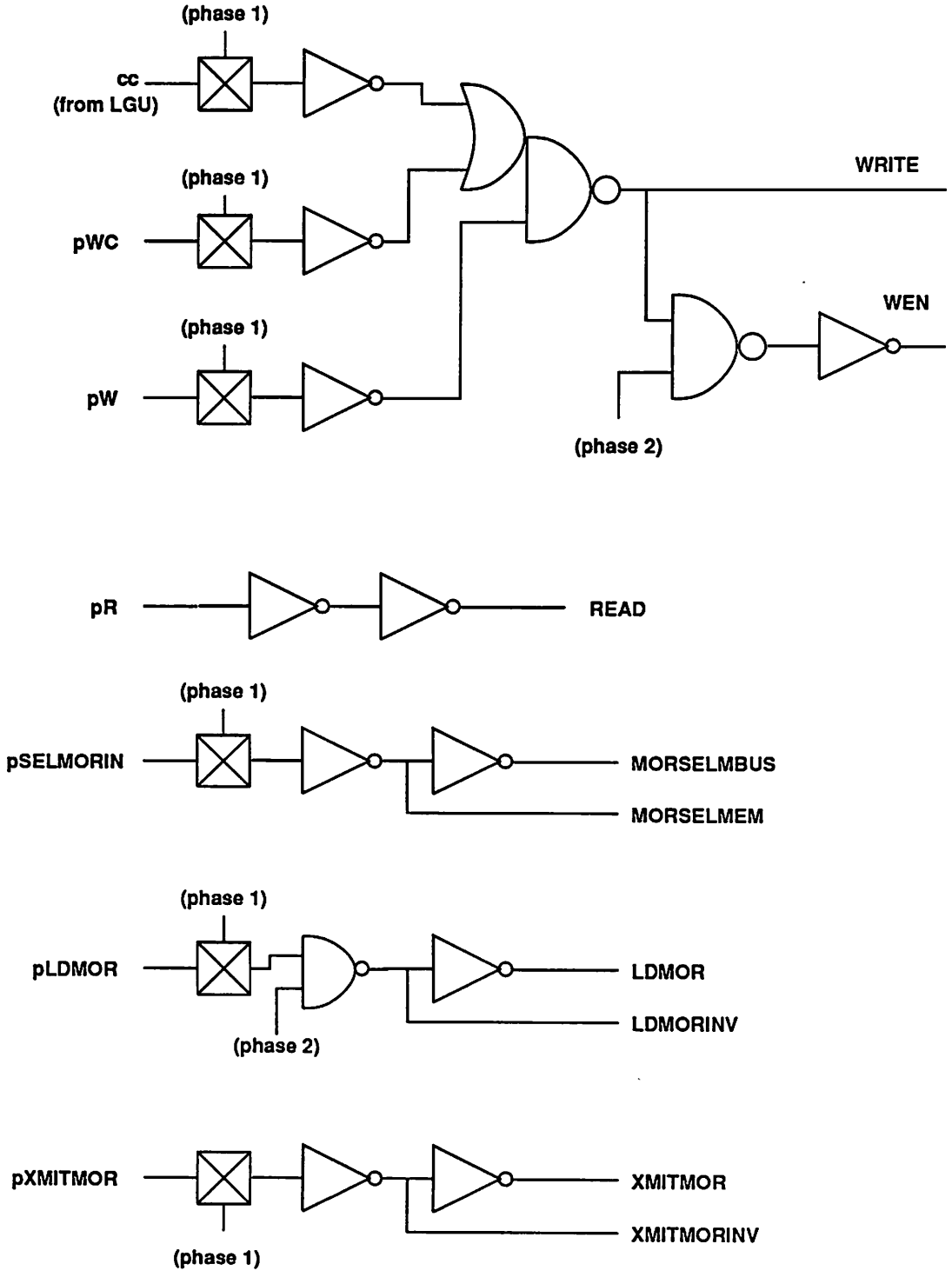


Figure 5.29: Control circuits for the memory interface section of AU.

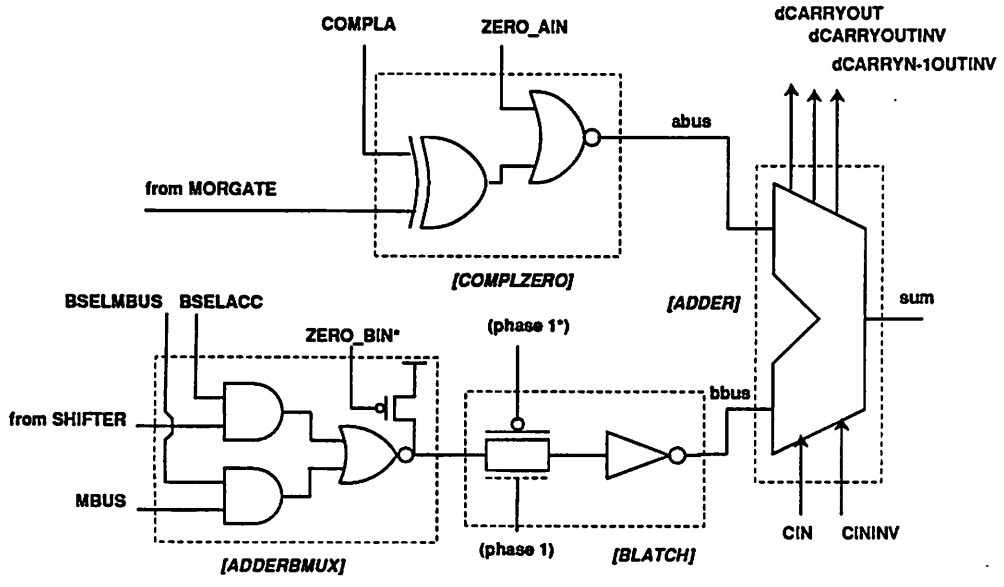


Figure 5.30: Arithmetic section of the data path.

Arithmetic Operations

The arithmetic section of the data path shown in figure 5.30 performs all the arithmetic operations. The bbus input of the adder gets its data from either the shifter's output or the MBUS. The bbus is forced to zero by asserting pZERO_BIN and is incremented by forcing abus to zero and setting CIN high. The BLATCH circuit is necessary for isolating the adder from the barrel shifter while the barrel shifter is being pre-charged during phase 2. This ensures the adder and the subsequent circuits have valid data during phase 2. Note, this is in accordance with the second rule for data path construction listed in section 5.5.1.

The abus gets its data from the MOR after being operated on by the COMPLZERO circuit. Several operations can be performed on the MOR output before feeding it to the abus. These include *negation, absolute value, and negation of absolute value.* Another important use of the COMPLZERO circuit and its associated control circuit (see figure 5.31) is in performing the shift and add multiplication. The multiplicand is loaded in the MOR whereas the multiplier is loaded in the coefficient register (rcoef). The following operation forms the multiplication product in the accumulator.

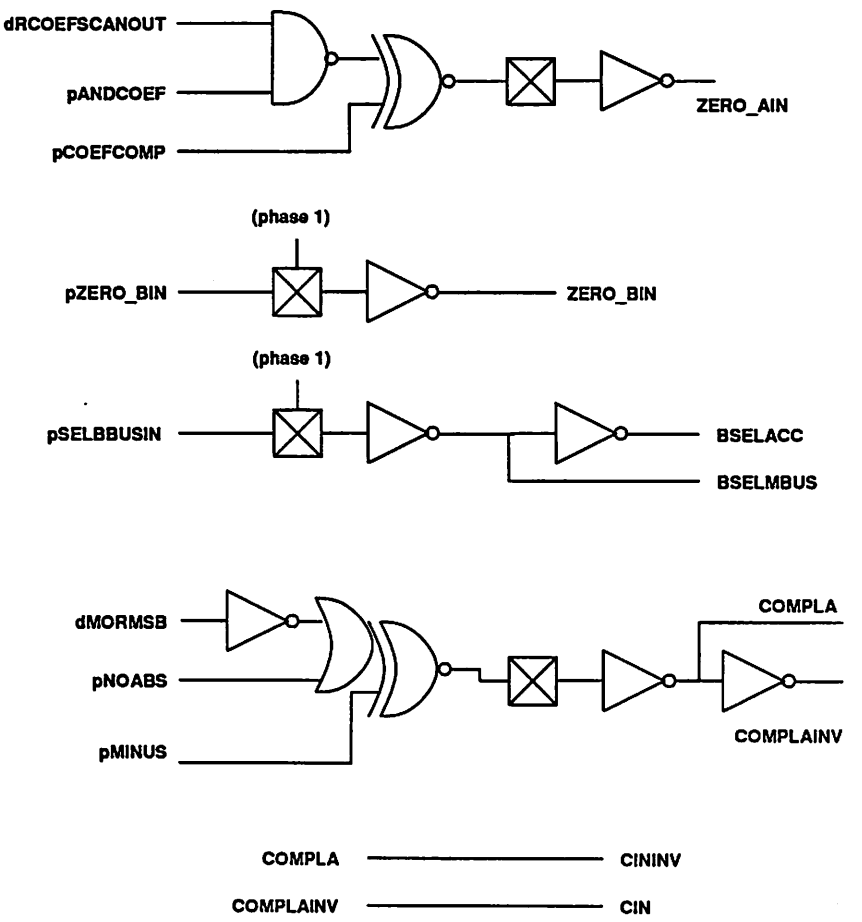


Figure 5.31: Control circuits for the arithmetic section.

abus	pNOABS	pMINUS	pANDCOEF	pCOEFCOMP
0	1	0	0	0
MOR	1	0	0	1
-MOR	1	1	0	1
|MOR|	0	0	0	1
-|MOR|	0	1	0	1
rcoef.MOR	1	0	1	0
rcoef.-MOR	1	1	1	0
rcoef.|MOR|	0	0	1	0
rcoef.-|MOR|	0	1	1	0
rcoef*.MOR	1	0	1	1
rcoef*.-MOR	1	1	1	1
rcoef*.|MOR|	0	0	1	1
rcoef*.-|MOR|	0	1	1	1
INCR. bbus	1	1	0	0

Table 5.1: Truth table for the various operations on MOR output listed in the first column.

$product = MOR.\text{-}rcoef[n\text{-}1] + \sum_{i=1}^{n-1} MOR.rcoef[n-i-1]/2^i$,

where n is the number of bits in the coefficient. The rcoef is shifted right, lsb first, on each iteration of the multiply operation. This serial output of the coefficient register, dRCOEFSCANOUT, is used in the control circuit (figure 5.31) for generating the ZERO_AIN signal. The multiplication (*AND operation* between the MOR data and the serial output of the rcoef is described by the following boolean operations.

COMPLA = (dMORMSB* + pNOABS) $\oplus$ pMINUS $\cdots$ (in control circuit)

xorout = COMPLA $\oplus$ MOR $\cdots$ (in COMPLZERO circuit)

ZERO_AIN = (rcoef[i].pANDCOEF)* $\oplus$ pCOEFCOMP $\cdots$ (in control circuit)

abus = (ZERO_AIN + xorout)* $\cdots$ (in COMPLZERO circuit)

The various operations that can be performed on MOR's output and the corresponding logic values of the control signals are listed in Table 5.1. Figure 5.30 also shows three control outputs from the adder: dCARRYOUT, dCARRYOUTINV, and dCARRYN-1OUTINV. The first two signals are the carry-out and carry-out* from the MSB slice whereas the last signal is the carry-in into the MSB slice and is

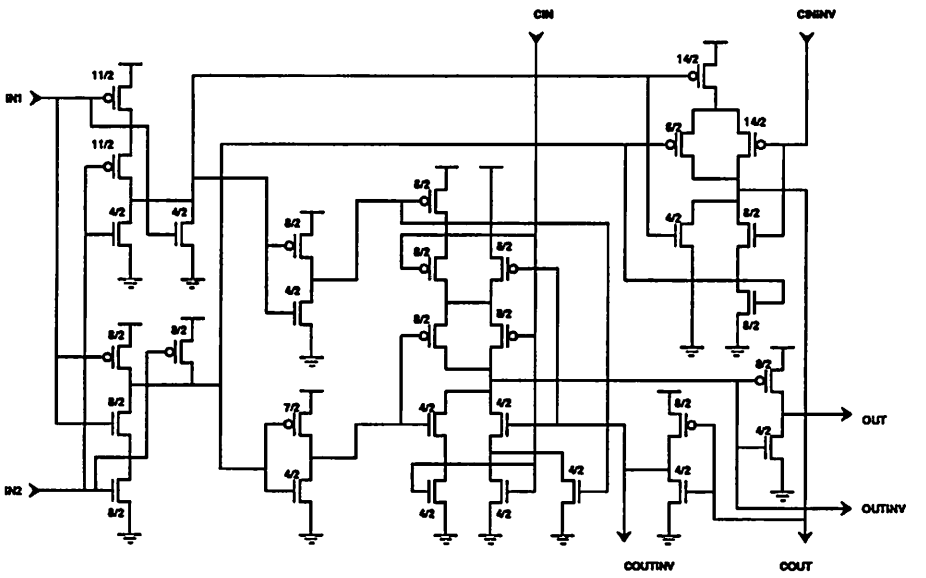


Figure 5.32: Circuit diagram of the even slice of the adder. For the MSB slice, CININV is also brought out as COUTN-1INV.

used for determining overflows in 2's complement arithmetic. The circuit diagram for the adder is shown in figure 5.32.

Coefficient Register

As discussed in the previous section, the coefficient register, rcoef, is used to hold and serially shift out the multiplier. A scan-type latch, circuit diagram shown in figure 5.33, is used as a coefficient register. The scan output of the register is also used as the serial output port of the register. The control circuits (figure 5.34) are designed for serially shifting out the data when either the scan-clocks or pSHIFTCOEF signal is active. When the register is being shifted out during multiplication, the serial input is held low, thus shifting in zeros.

Saturation Logic

The largest 2's complement number the data path can handle is limited to $(2^{n-1}-1)$ to -2^n, where n is the number of bits. Adder operations can, however, lead to overflows giving erroneous results. If no special measures are taken the

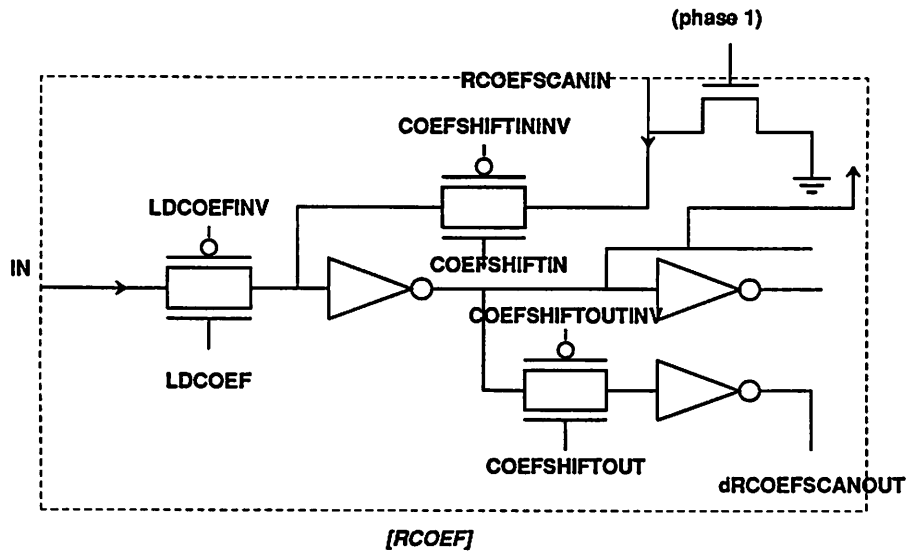


Figure 5.33: A scan-type latch is used as the coefficient register for multiplication.

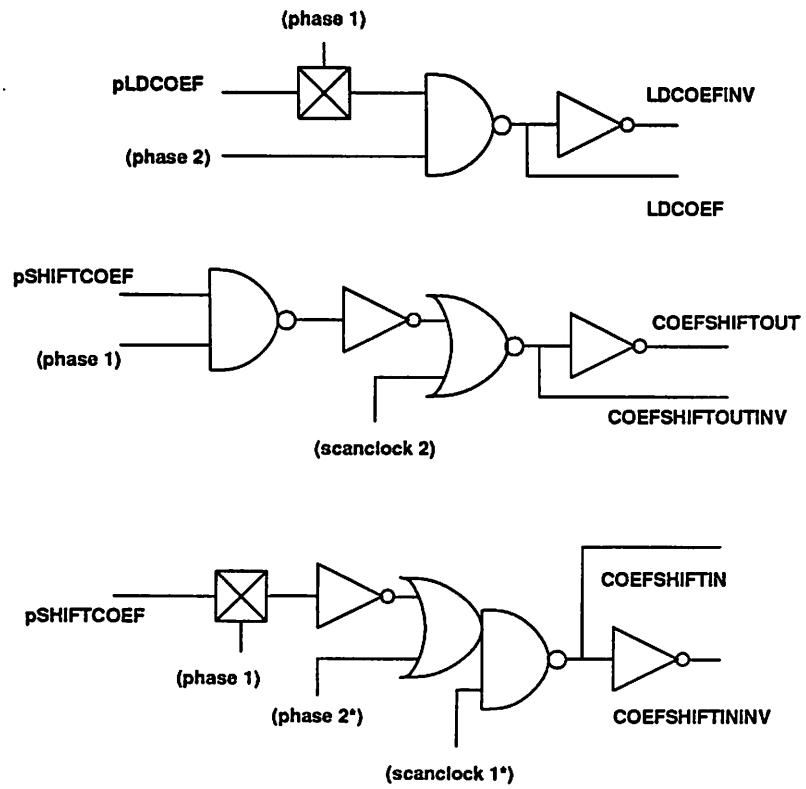


Figure 5.34: Control circuits for the coefficient register.

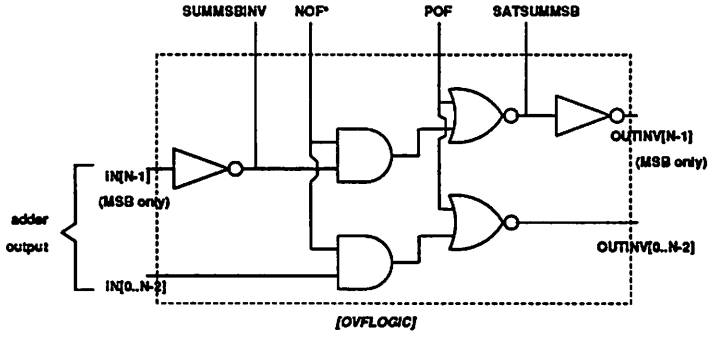


Figure 5.35: The adder outputs are saturated to either positive maximum or negative maximum on overflow.

adder output wraps around, i.e. cyclically goes from positive maximum to negative maximum when incrementing and in the other direction when decrementing. In the AU data path, the adder's output is saturated at the positive maximum or the negative maximum, as the case may be, whenever the sum overflows. This saturation operation is performed by OVFLOGIC circuit shown in figure 5.35. The control circuits are shown in figure 5.36. POF becomes active high during positive overflow and NOF* becomes active low during negative overflow. The saturation operation, however, must be disabled under certain conditions.

During the iterative shift and add multiplication, even though an intermediate partial product may overflow, the number is restored to within the maximum limits after the right shift operation on the next iteration. Therefore, saturation is disabled under program control by the microinstruction *nosat* during multiplication. This asserts the pNOSAT signal which disables POF and NOF*. The use of nosat in effect results in a word size of $n+1$, with carry-out being the sign bit. As discussed later in the section on barrel shifter, POF and NOF* are also used in the barrel shifter's control circuit for generating the MSB during right shift. The MSB before saturation (dSUMMSBINV) and after saturation (dSATSUMMSB) are also brought out from the OVFLOGIC circuit for use in the control circuits.

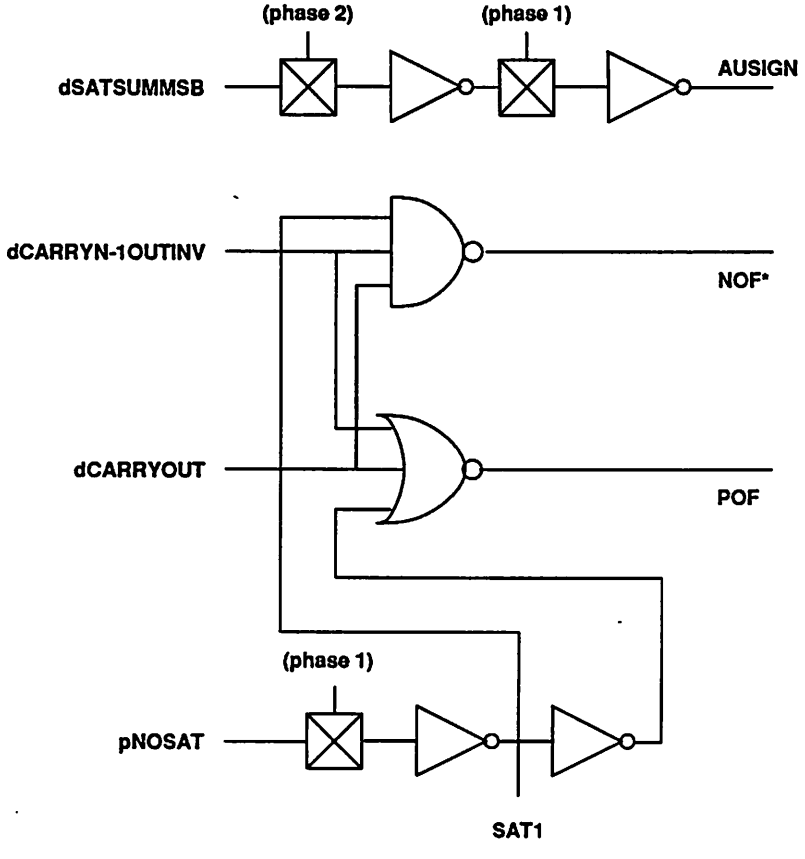


Figure 5.36: Control circuits for the saturation logic.

Accumulator

The output of the adder is stored in the accumulator. The accumulator is a scan-type latch but with the addition of the mux, ACCMUX, has the effect of a register (see figure 5.37). The mux either allows a new output from the adder (for aip microinstruction) to be stored in the accumulator or maintains the previous data. In retrospect, the accumulator and the mux can be replaced by a two-phase register. The output of the accumulator is also connected to the barrel shifter as well as to the MBUS through the ACCUMLATCH.

As shown in the control circuits in figure 5.38, the accumulator may be conditionally loaded under program control using two of the microinstructions: (i) *aip*, accumulate if positive, which asserts the control signal, pAIP; and (ii) *acond-load*, conditionally load accumulator, which asserts pSUMCOND. In the latter case, loading depends on the status of the condition code, cc, from the logical unit (LGU). Note, the two microinstruction should not be specified in the same cycle.

Registers and I/O Ports

The data path has two, two-phase registers which are used as temporary registers and also as I/O registers. The registers can be loaded either from the accumulator or from a parallel I/O port as shown in figure 5.39. On the other hand, the output of the registers are connected to the MBUS.

For writing data to the external port, the MBUS is connected to the input of a bi-directional I/O pad through PORTOUTBUFFER. The direction of the pad's data transfer is controlled by WRPORT. Normally, the I/O pad is always in the input mode. During write operation, however, the control signal, WRPORT, switches the direction of data transfer through the I/O pad and data is transferred from the MBUS to the I/O pad. On the other hand, during read operation a control signal, RDPORT, enables a tristate buffer, IOPORTLATCH, in order to allow data transfer from the pad to the RBUS and eventually into one of the registers. In addition, RDSTRB and WRSTRB signals are sent off-chip and are asserted during port read and write operations, respectively. The desired port address for selecting an

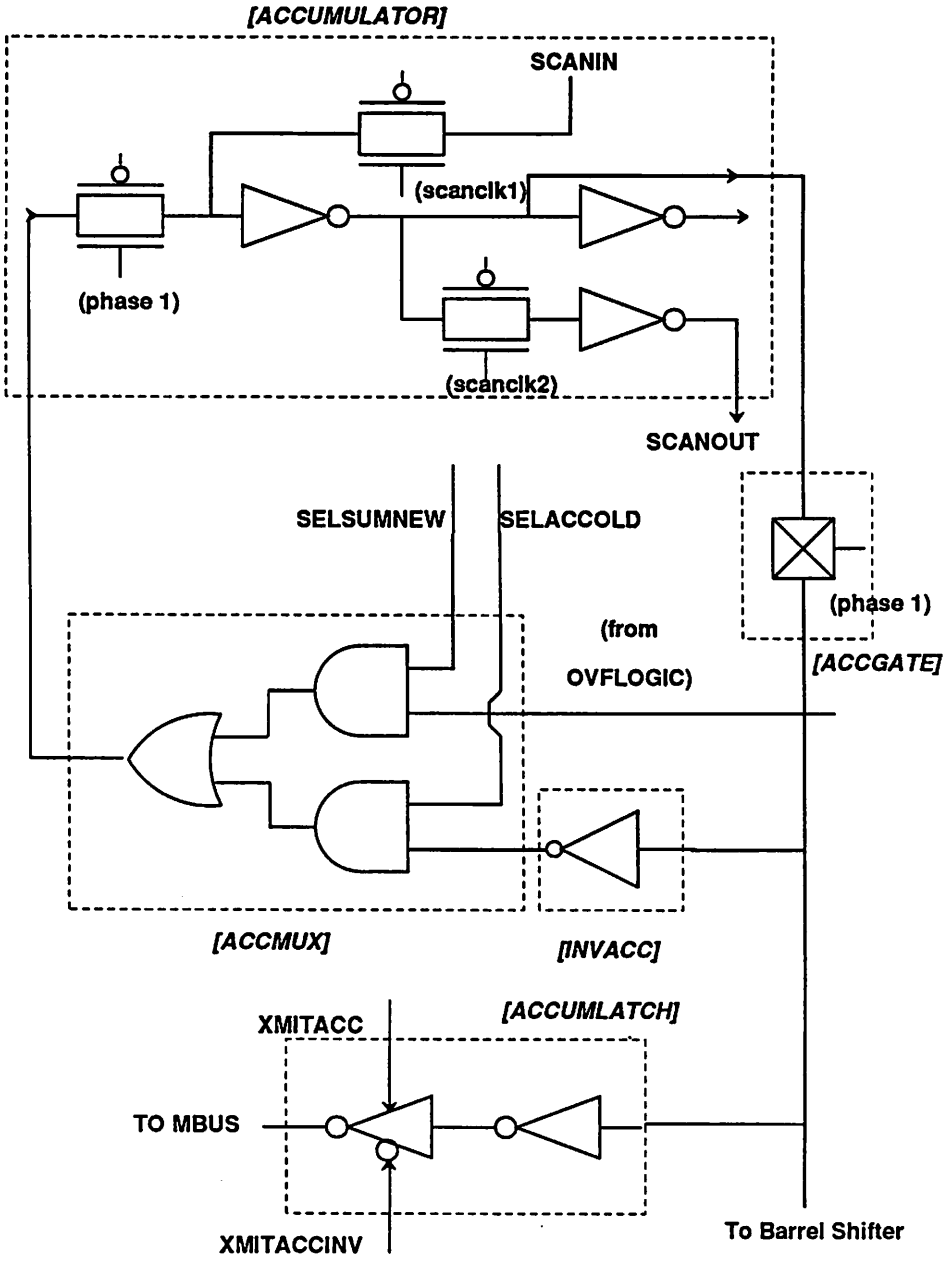


Figure 5.37: Accumulator circuits.

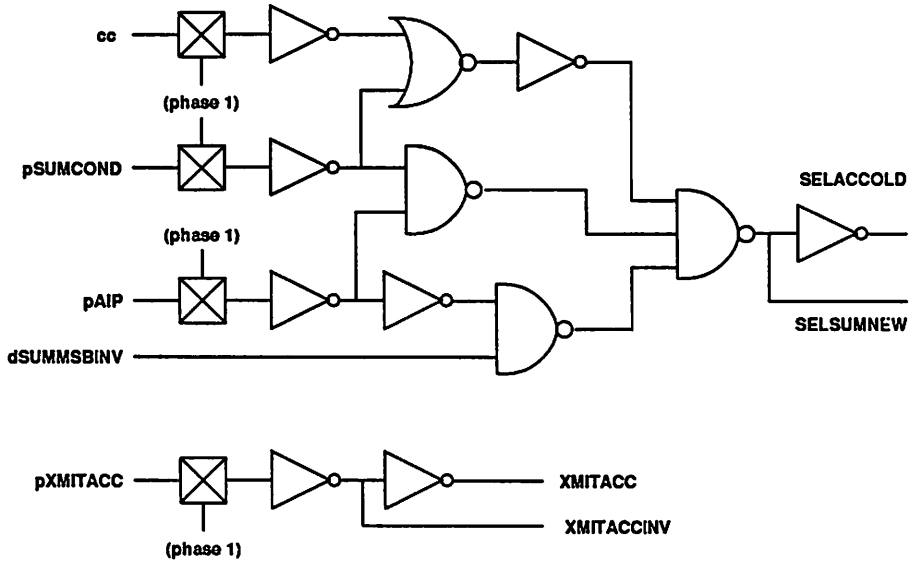


Figure 5.38: Control circuits for accumulator.

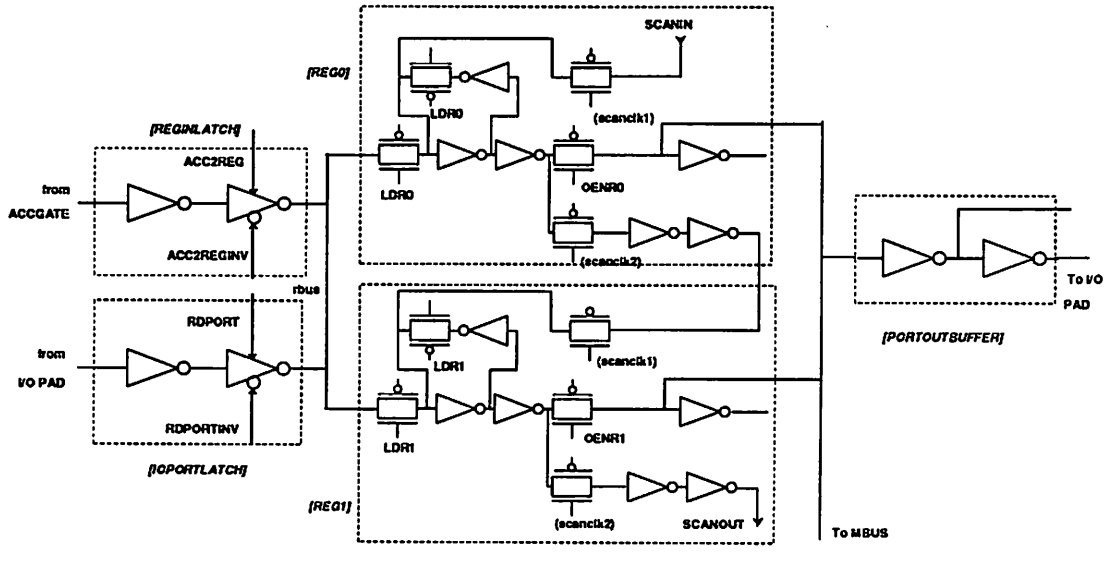


Figure 5.39: Register circuits for the AU data path.

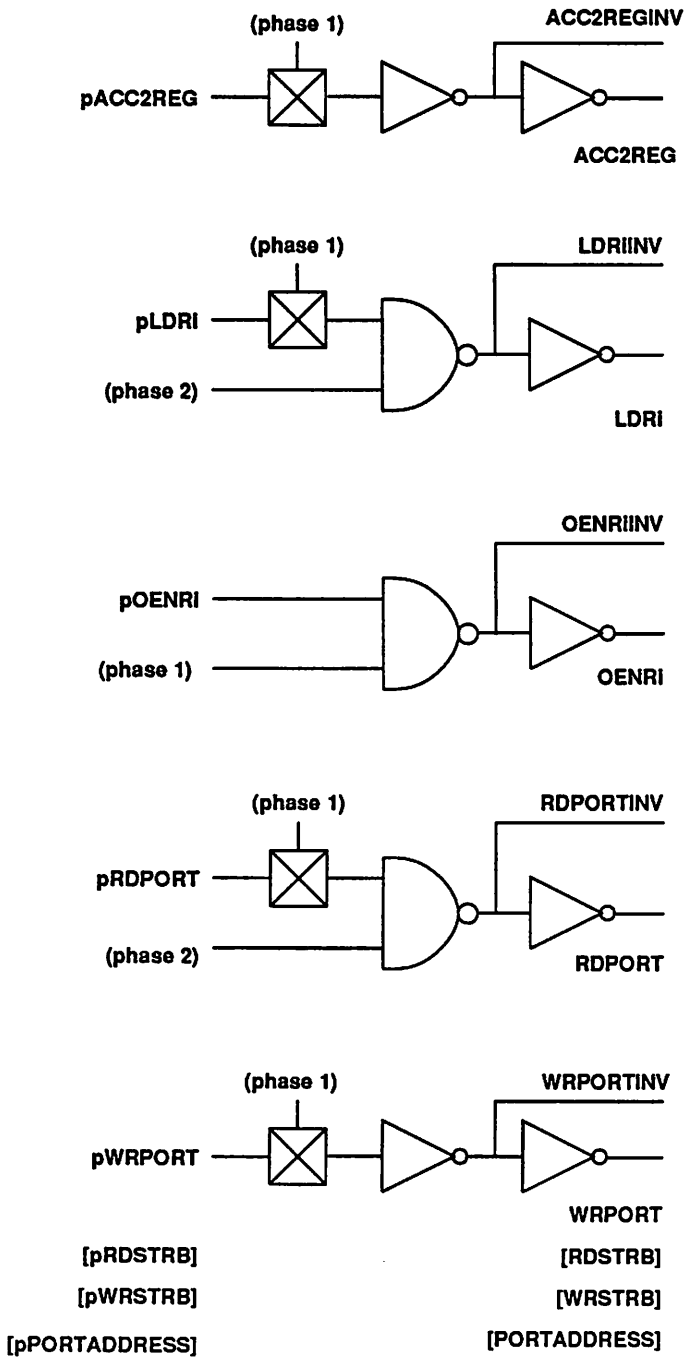


Figure 5.40: Control circuits for registers and I/O port. WRPORT, RDSTRB, WRSTRB, PORTADDRESS, and their complement signals are all generated using the same type of circuits. i in OENRi and LDRi refers to the register number (i = 1 or 2).

external I/O device is put out on the PORTADDRESS bus. For the robot control processor this bus is four bits wide. All the I/O control signals become valid during phase 1 and remain valid for the entire cycle (till the end of phase 2).

Barrel Shifter

The barrel shifter, shown in figure 5.41, uses dynamic circuits and is precharged on phase 2. The barrel shifter design imposes the following restriction on the type of circuit that can be connected at its input. *During the evaluation phase, the circuit driving the barrel shifter's input must have a path to ground in order to be able to pull down the precharged node.* The shifter can shift right (down) the accumulator output from zero to six bit positions and shift left (up) one position. In shifting down, the MSB positions are sign extended. If *nosat* is in effect then the sign is the carry-out from the adder output, as discussed earlier under saturation logic. The control circuit which generates the sign bit is shown in figure 5.42. During shift left the LSB position gets a zero.

The desired amount of shift is encoded in a three-bit wide control field of the control-word. This is decoded in the barrel shifter control circuits as shown in figure 5.42. The decoded control lines going into the barrel shifter must be driven by Type 1 control circuits in order to prevent the shifter's output node from loosing charge during precharge phase. This is necessary because the shifter's pull down path does not have an evaluate transistor that can be turned off during precharge.

MBUS to EABUS Interface

The AU data path is connected with both the APU and the PCU. In the case of APU, data may be transferred in both directions. On the other hand, for PCU, data is transferred only from the AU to the PCU for loading the timer register. On the AU side, all data transfers are done through the MBUS. The MBUS is connected with the EABUS which in turn connects with the APU and the PCU. Interface circuits between the MBUS and the EABUS are shown in figure 5.43. Data transfers from the MBUS to the EABUS take place on phase 1 of the clock

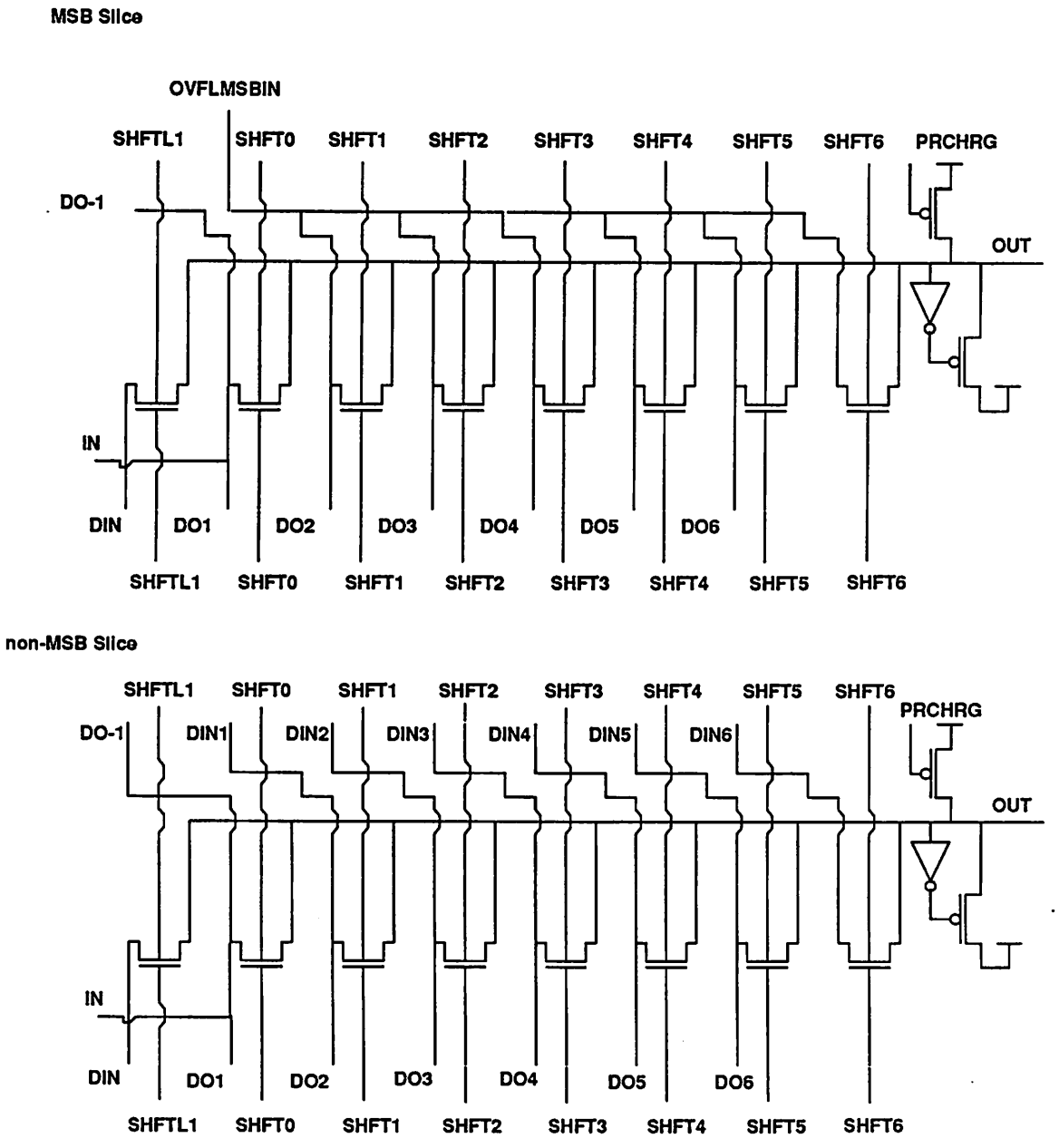


Figure 5.41: The barrel shifter is based on dynamic CMOS circuit design technique with a PMOS pull-up.

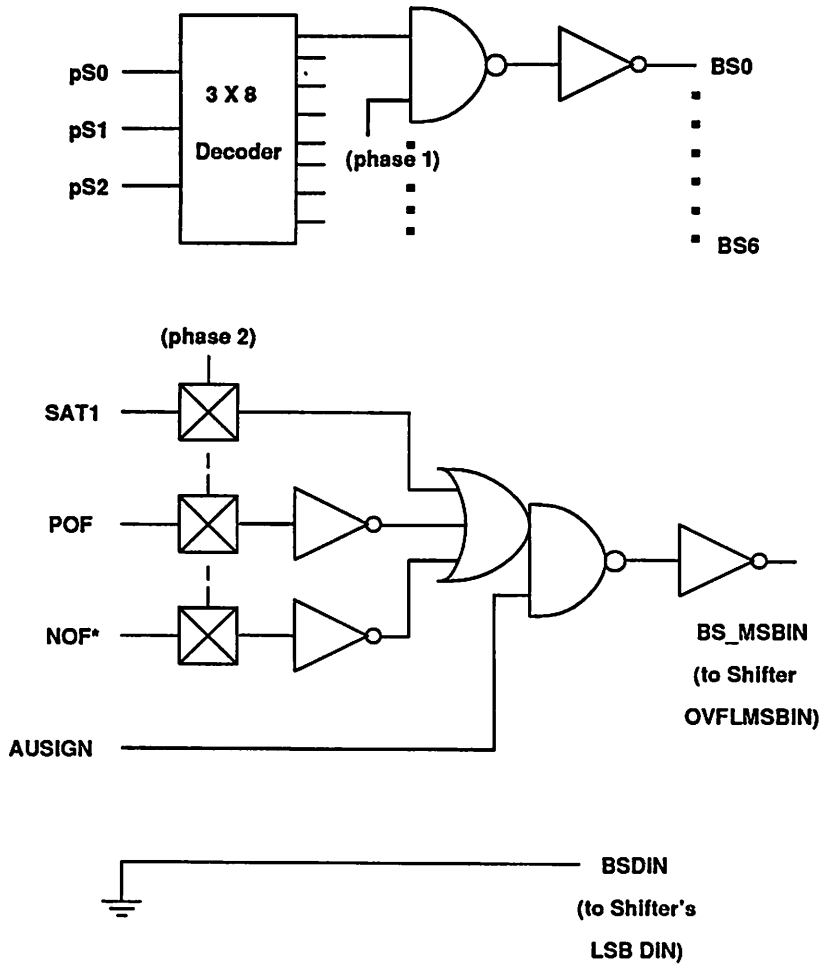


Figure 5.42: Control circuits for the barrel shifter.

cycle leaving phase 2 for loading in the destination register. On the other hand, transfers from the EABUS to the MBUS is first temporarily stored in AREGIN on phase 2 in order to avoid a long critical path. On phase 1 of the next cycle, the data is moved into MBUS and eventually loaded into a destination register (accumulator/rcoef/MOR) or memory on phase 2. This is consistent with the timing scheme of the rest of the data path.

Since the size of the MBUS and the EABUS are parameterized and may be different, the following policy is adopted for interfacing the two on the assumption the MBUS is larger. Data transfer from MBUS to EABUS is truncated on the MSB side since APU and PCU use positive integers only; whereas, data transfer from EABUS to MBUS is sign extended since positive or negative constants from control-word's address field may be transferred to AU via APU. (see figure 5.44). These parameterized interconnections are specified in the sdl files.

5.6 Address Processing Unit (APU)

The address processing unit is a simple data path dedicated for computing addresses of variables in ram and for loop counting. The components of the APU are three scan-type registers, an adder, a pull-down circuit (zero) for one of the adder input, an output latch for the adder, and a clocked inverter. These are shown in figure 5.45. The adder circuit is same as one used in AU. All computations inside the APU are based on unsigned integer arithmetic.

The registers can be conditionally or unconditionally loaded from the EABUS. Conditional loading is based on one of two different conditions: status of condition code from LGU; sign bit from the APU's adder (load if positive). The sign bit is also available for use by the PCU and LGU. The APU circuit allows modification of the register's content and loading it back into the register in the same cycle. On phase 1 the register data is applied to one of the adder's input. The address field in the PCU's control-word, which became stable at the end of the previous cycle, is applied to the other input of the adder. The adder gets evaluated during phase 1 and the output is transferred to the register during phase 2 via the

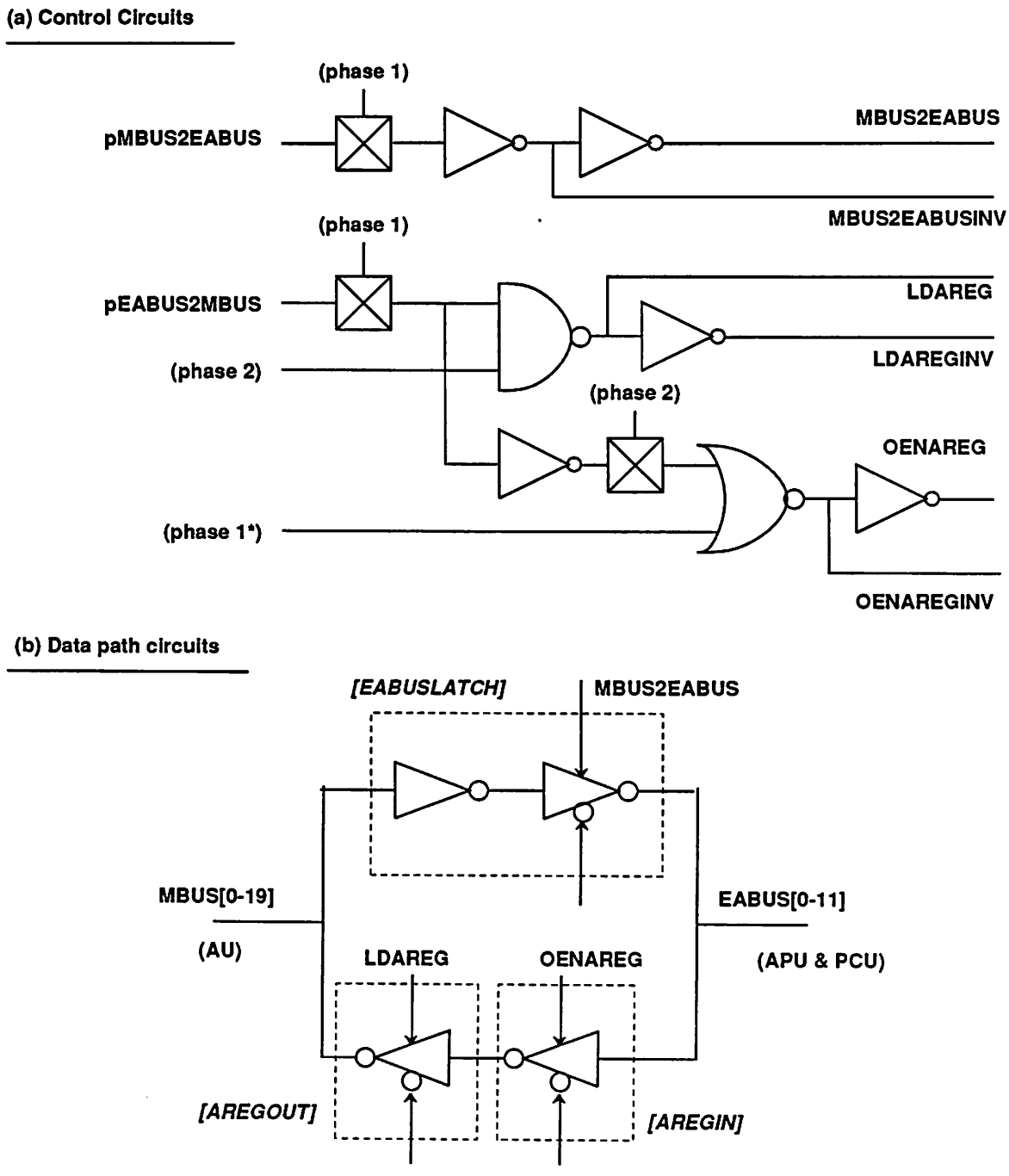


Figure 5.43: MBUS-EABUS interface circuits and their control circuits.

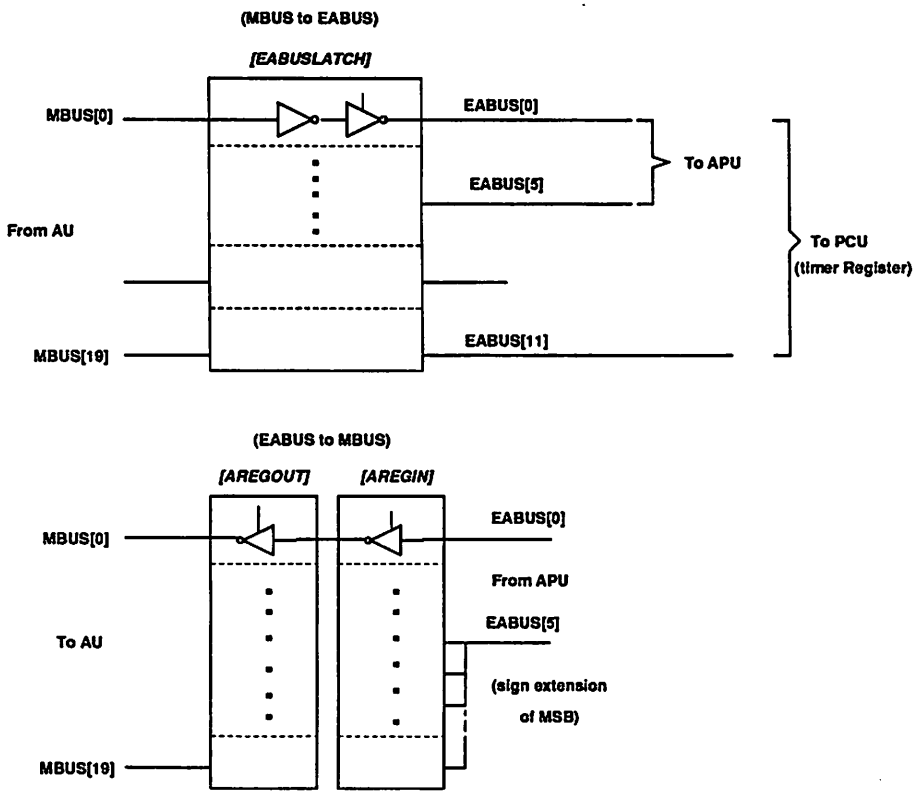


Figure 5.44: Interconnection network between EABUS and MBUS.

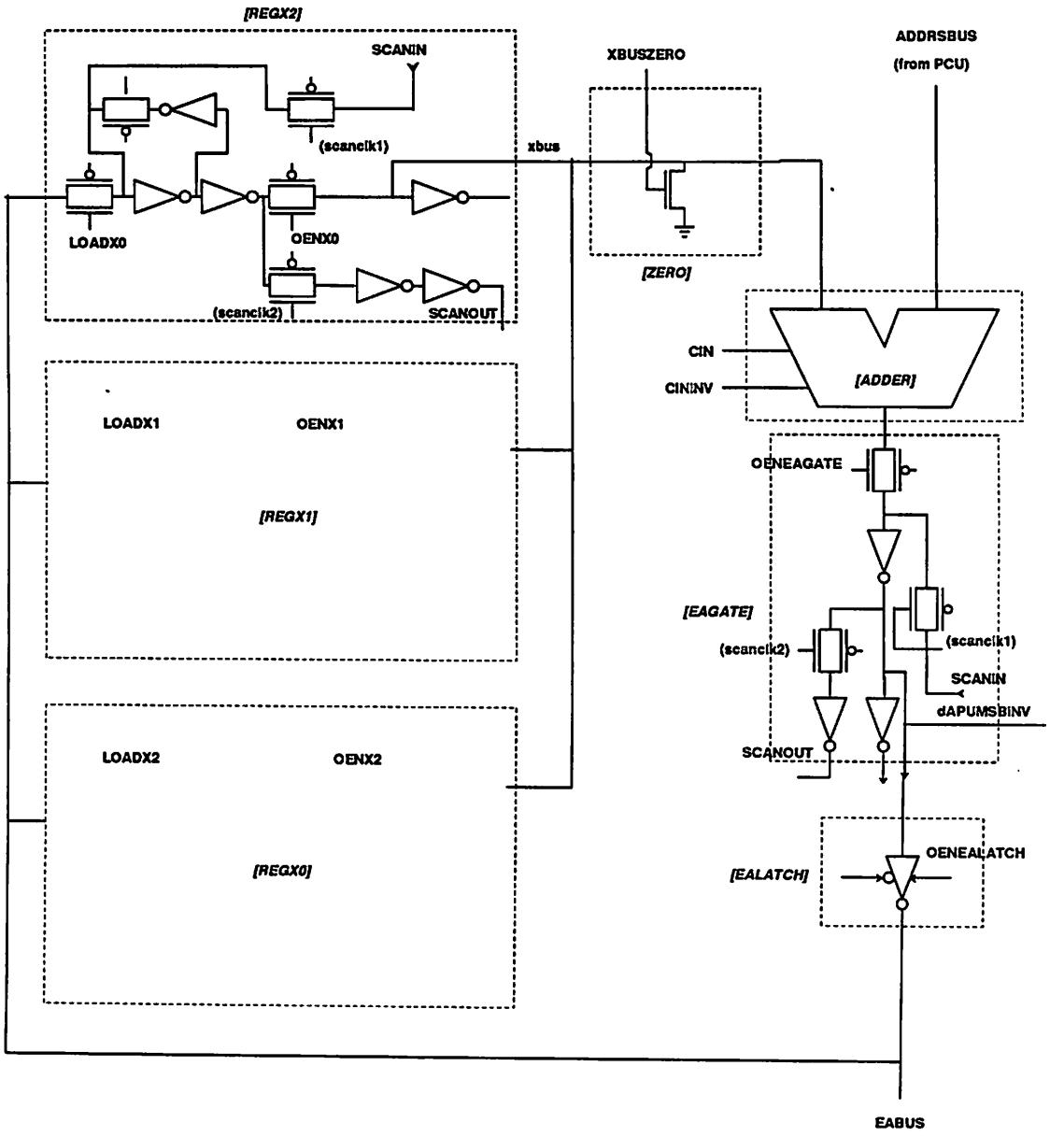


Figure 5.45: Circuit diagram of APU.

EABUS. The adder output may also be temporarily stored in the dynamic latch, EAGATE. The tristatable inverter, EALATCH, allows data from other sources (e.g. AU) to be loaded into the register by isolating the adder output from the EABUS. Another feature of the circuit is to allow constants specified in the address field to be loaded in the registers or even moved to AU through EABUS and eventually MBUS. This is made possible by the pull-down circuit, ZERO, which forces a zero on the xbus. All the control circuits are shown in figure 5.46.

5.7 Local Memory (RAM)

The processor has an on-board local memory for storing variables and constants. The ram is composed of two child macrocells (figure 5.47) each of which is assembled using the TimLager tiler program. The two cells are then put together with Flint. The design of the memory cell is based on a three transistor CMOS ram cell shown in figure 5.48 and described later. These cells are organized in two dimensional arrays with each row representing a memory word. A decoder selects a memory word based on the address input. There is no column decoding for selecting individual bits. A selected memory word may be written or read through the bit-lines; a common bit-line is shared by a cell for both reading and writing data. The number of bit-lines is of course equal to the number of bits in the memory word.

The read and write operation is triggered by READ and WRITE control signals respectively, generated in the control circuits of the AU data path. Inside the ram, READ and WRITE signals are latched in the RAMCTL cell as shown in figure 5.49. The address, on the other hand, is generated by the APU during phase 1 and is held stable during phase 2. Static CMOS nor gates are used as decoder so that the decoding can start as soon as the address becomes valid sometime during phase 1. We have assumed the nor gate output becomes valid before the start of phase 2. The decoder output is fed into two dynamic nand gates: one generates the read-select signal and the other generates the write-select signal as shown in figure 5.50. Meanwhile the bit-line of the 3-T cell is precharged on phase 1; the precharge circuit, MEMIO, being inside the AU data path. On phase 2 when the

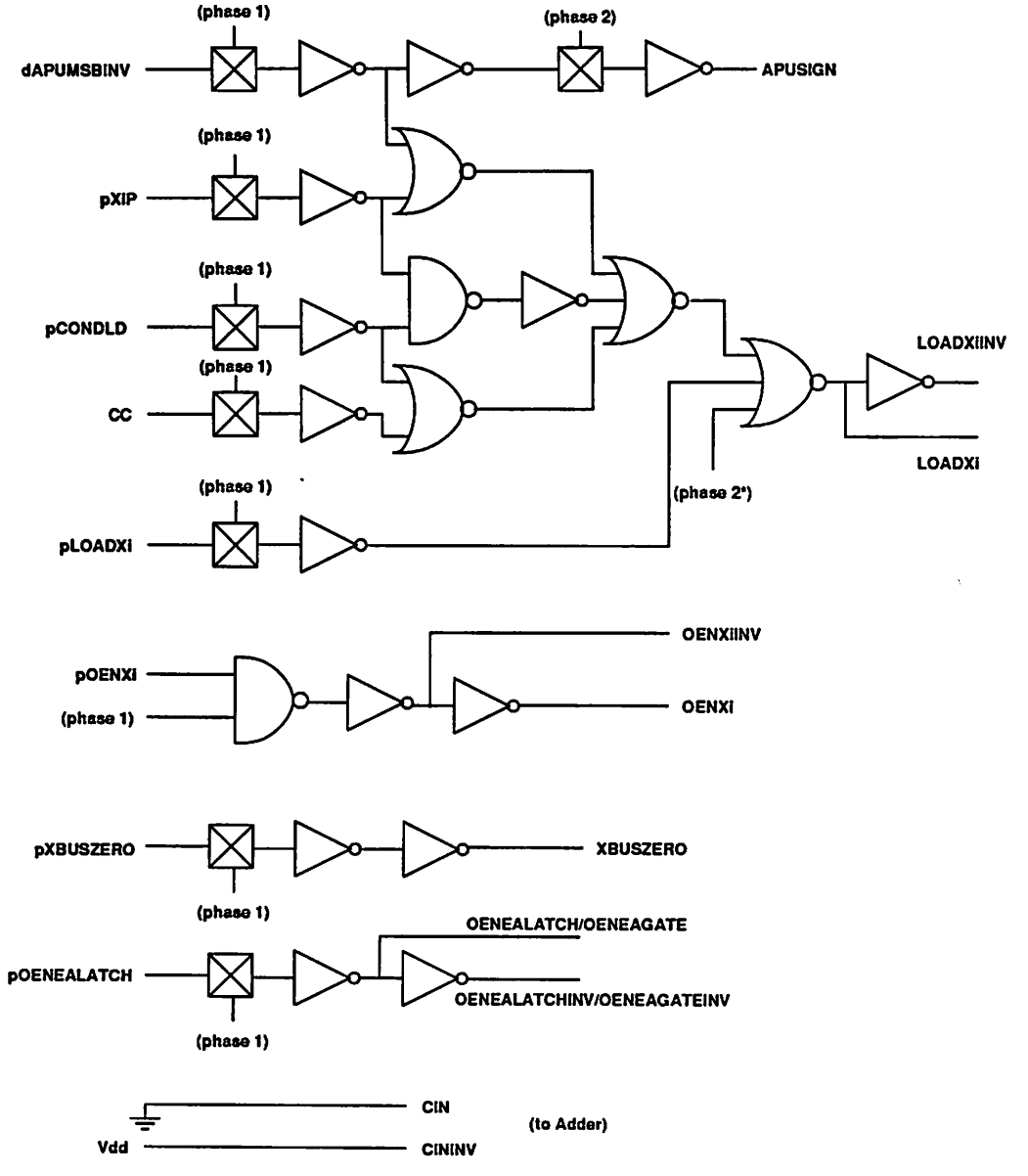


Figure 5.46: Control circuits for the APU. The control circuits for loading and enabling the registers are identical for all the three registers. i indicates register number (as in LOADXi).

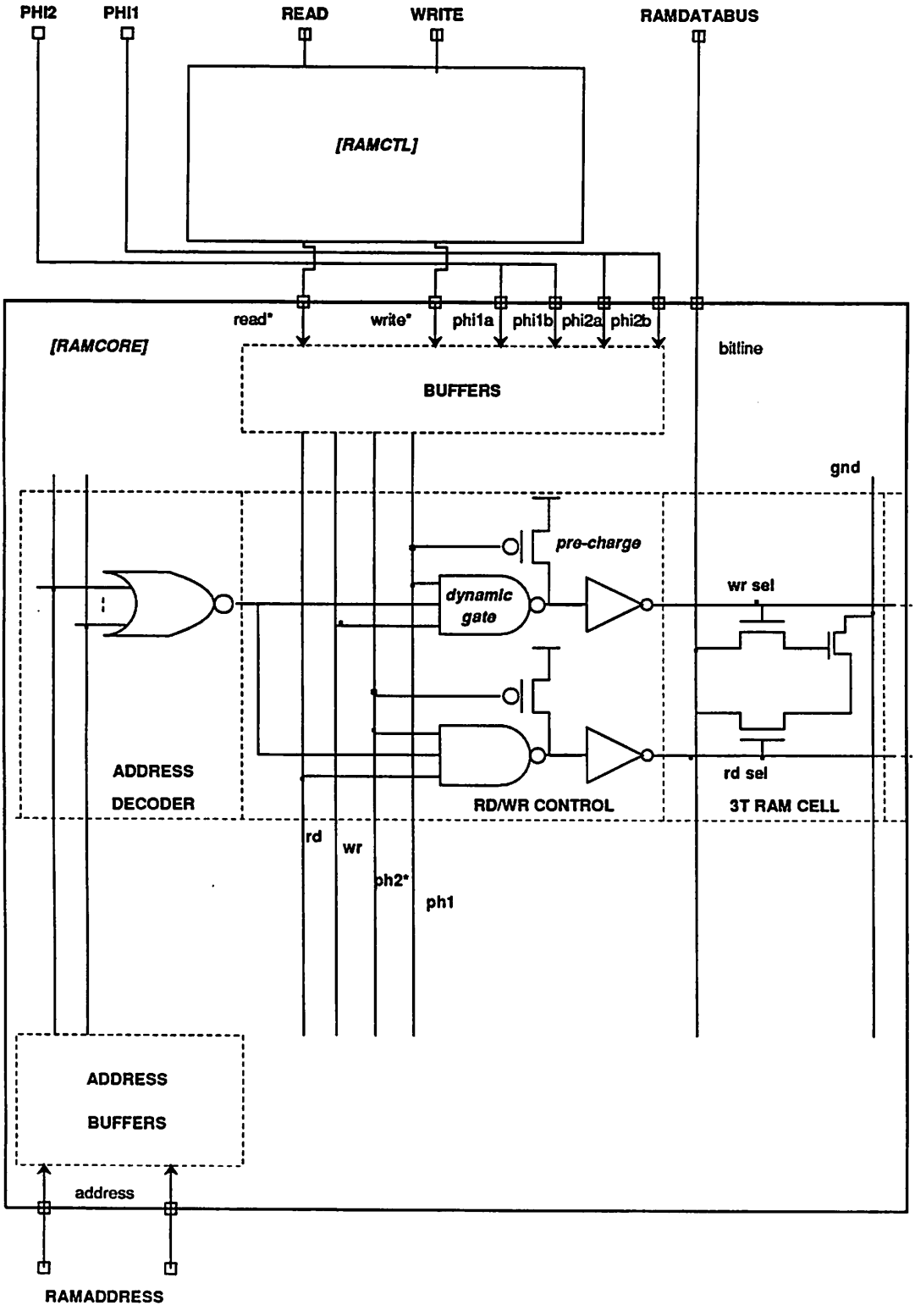


Figure 5.47: The ram macrocell is composed of two child macrocells: RAMCORE and RAMCTL.

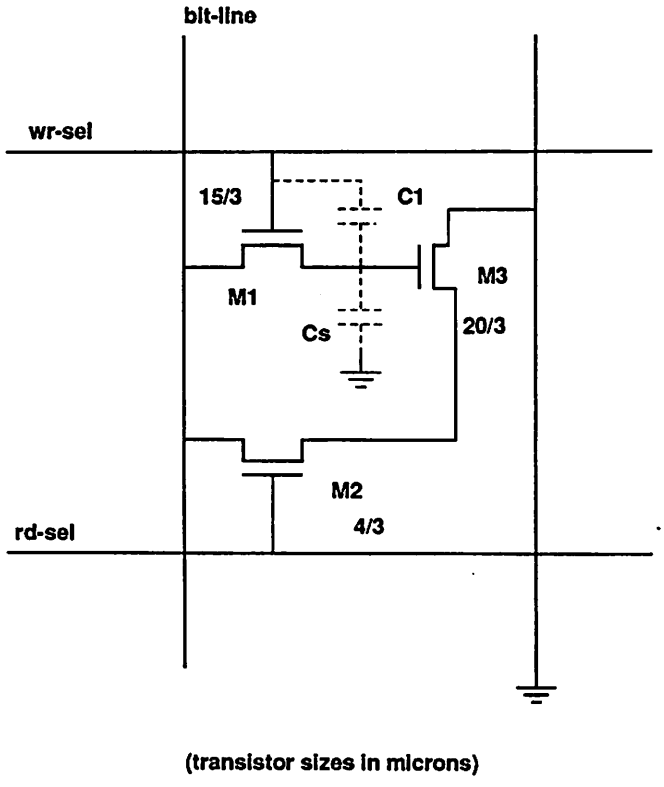


Figure 5.48: Three transistor memory cell used in the ram.

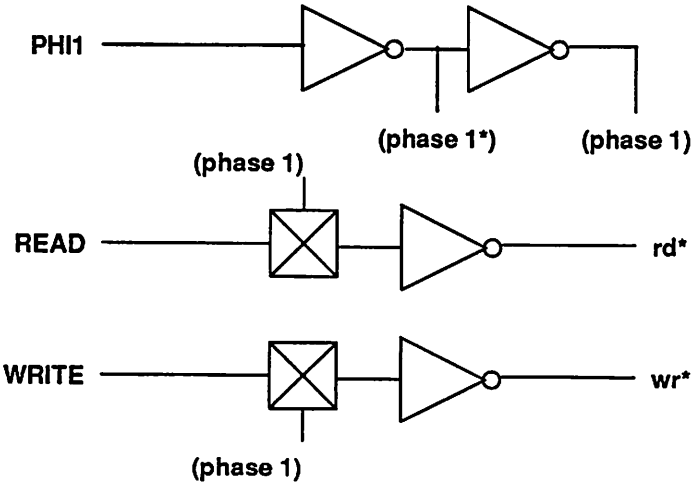


Figure 5.49: Control circuits for the ram.

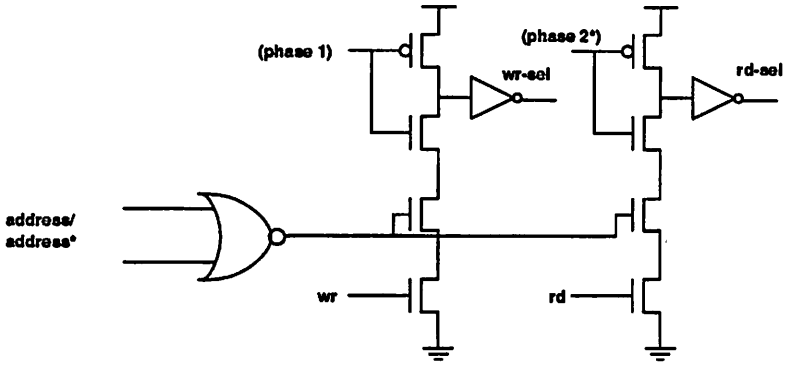


Figure 5.50: The address decoding circuits.

read-select or write-select become valid, data is either dumped on the bit-line for read operation or stored in the cell from the bit-line for write operation. The operation of the ram is described in the timing diagram shown in figure 5.51.

Three transistor ram cell

Since the design of the ram cell is very critical for satisfactory performance, we performed detailed spice analysis of the cell. As shown in figure 5.48, the cell consists of three n-channel transistors. The following considerations went into the selection of the transistor sizes which is based on a P-well CMOS process.

When writing a logic '1' into the cell, the storage node capacitor, C_s, is charged high. Two effects, however, degrade the charge and hence the voltage stored at this node: (a) Due to body-effect the maximum voltage stored on the storage node can rise only up to $V_{g1} - V_t = V_{wr-sel} - V_t$. (b) When wr-sel goes low at the end of write cycle, the capacitive coupling between M_1 gate and the storage node results in further degradation of the voltage. This degradation of the storage node voltage affects the read access time. When reading a cell into which a '1' has been written M_2 and M_3 must discharge the large bit line capacitance which has been precharged high. This discharge time is the major component of the read access time. With a lower M_3 gate voltage the discharge time would be larger. Hence the voltage on the gate of M_3 (storage node) must be kept as high as possible when storing a '1'.

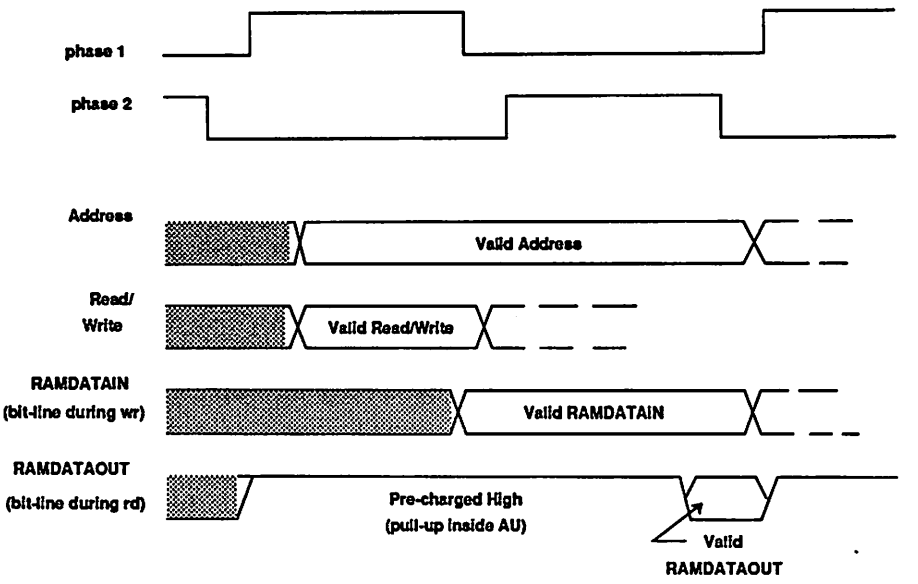


Figure 5.51: Timing diagram for the ram.

The body-effect being a function of the doping levels cannot be changed for a given process. On the other hand, the effect of capacitive coupling can be reduced by minimizing the ratio C_s/C_1. Our spice simulations showed a worst case voltage of 2.5 volts on the storage node with Vdd equal to 4.5 volts.

Minimizing the capacitance on the bit-line is important for quickly discharging it during read operation. In our design we assumed a maximum of 64 cells on each bit-line. Since most of the capacitance comes from the drain diffusions of M_1 and M_2, the drain areas have been kept to minimum.

Sizing the transistors M_2 and M_3 is a trade-off between providing a large W/L for quick discharge of the bit-line and the desire to keep the cell size small. Moreover, width of M_2 must be kept small to minimize the drain diffusions and also to keep the capacitive load small on the rd-select line.

Sizing of M_1 also involves trade-offs between conflicting requirements. A large W/L for M_1 would provide a faster charging of the storage node during write operation. However, a large M_1 would lead to a larger capacitive load on the wr-select line. Since C_s is relatively small, its charging time is not a limiting factor in

the performance of the ram. Hence a minimum sized transistor is chosen for M_1. The sizes of the other transistors are shown in figure 5.48.

5.8 Logical Unit (LGU)

The logical unit (block diagram shown in figure 5.52) is basically a finite state machine and uses very much the same circuits as in the PCU's finite state machine. The mux_latch (CFSMREG) circuit is used only as a latch and the various unused terminals are grounded. The state transitions and the output of the LGU are determined by the external conditional signals (such as APU and AU sign bit) and the state address provided by the control-word in the PCU. For the robot control processor, bits 48 through 50 in the control-word specify the LGU state to be evaluated. The output of the LGU is the condition code, cc, which is used in various conditional operations as discussed earlier in this chapter. Since cc becomes valid during phase 2, all the conditional operations using cc are done following the cycle in which LGU is evaluated. Thus a one cycle delay must be allowed between generating a status signal (such as sign bit) in the data path and executing a conditional operation based on it by setting cc.

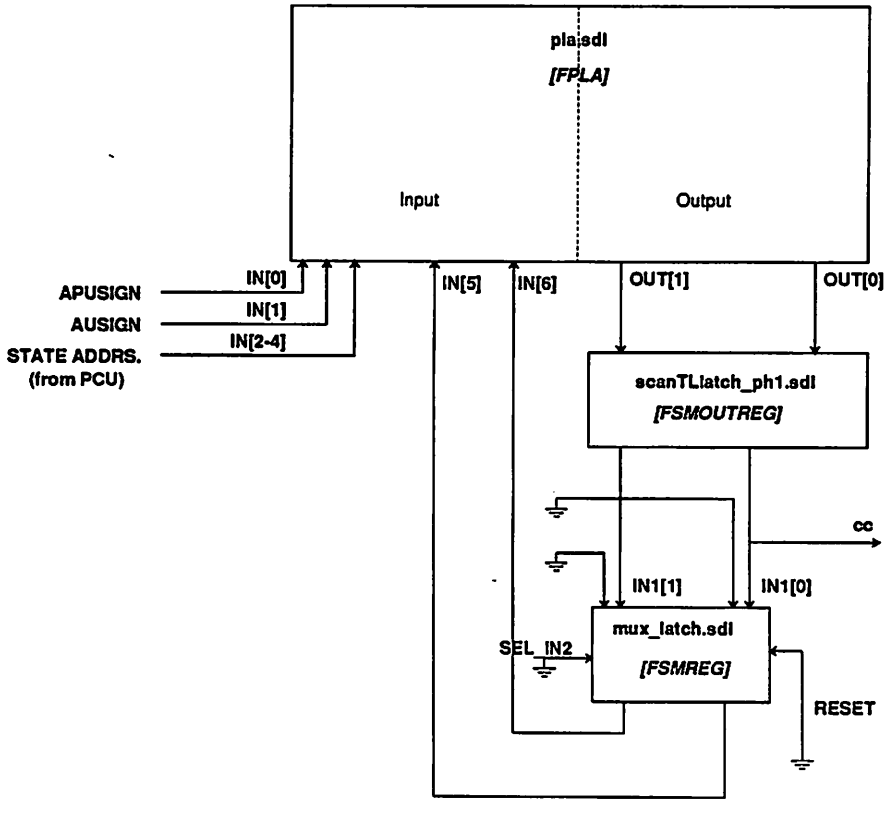


Figure 5.52: The logical unit is essentially a finite state machine.

Chapter 6

Design for Testability

Effective and adequate testing has been a perennial problem for integrated circuits and continues to grow in importance with the increase in size and complexity of circuits. The advent of CAD based IC design techniques has brought even greater attention to the testing problem. Silicon assembly and silicon compiler systems, such as the one described in this thesis, now permit system designers to assemble fairly complex custom circuits without getting involved with low-level circuit design issues. This usually involves connecting together pre-designed leafcells and macrocells which are often parameterized. The consequence of this is a different chip for every application; testing becomes even more important in this case for ensuring the combination or configuration of cells and the customization of the layout done for a particular design do in fact result in circuits that perform satisfactorily. Whereas functional simulations during design phase verifies the functionality, testing of custom chips after fabrication is necessary for several reasons: (i) Fault detection; (ii) Evaluation of performance in terms of maximum achievable speed; (iii) Identification of sections of the circuit that limit speed. On the other hand, the resources and effort usually spent on testing standard parts may not always be justified for testing low volume, custom parts where overall turn around time must be kept short and the cost low.

An effective testing strategy must make testing an integral part of the design process. The following considerations guided the development of our testing

methodology.
(i) Adopt a uniform testing strategy for all custom chips with minimum amount of chip specific external test circuits (boards).
(ii) Provide observability and controlability of internal states.
(iii) Minimize the number of device-under-test (DUT) pins assigned for testing purposes, thus simplifying tester-board design.
(iv) Permit testing of the DUT at full speed but perform communication with the tester at low speed for simplifying tester-board design.

6.1 Testing Strategy

Our testing strategy encompasses both design for testability and tester design for a complete solution. The solution has three components: scan design of all registers; an external test-board; and a generic, tester interface macrocell (Test-logic) which is incorporated in each chip design. Test-logic macrocell allows a single test-board design to test different chips. A high-level view of the test set-up is shown in figure 6.1. In the beginning of the testing procedure, a break-point condition is loaded through the scan-in pin into a break-point register inside the Test-logic macrocell. The processor is then allowed to function at a desired operating speed. When the break-point condition is satisfied, the Test-logic circuit disables the master-clock and asserts a break-point flag. This triggers the test-board circuits to begin a read-out of the scan registers and also simultaneous write-in into the registers. Either the same data may be restored in the registers or new data may be loaded as desired. When all the registers in the scan path have been read and a new break-point is loaded, the master-clock is enabled and normal operation resumes till the next break-point. The data collected by the test-board is eventually transferred to the host for analysis. The host also provides test vectors and break-points to the test-board. The Test-Mode* signal is used for disabling the test circuits when the chip is being used for normal operations. In all, seven pins are required for testing purposes although some of the pin functions may be multiplexed on common

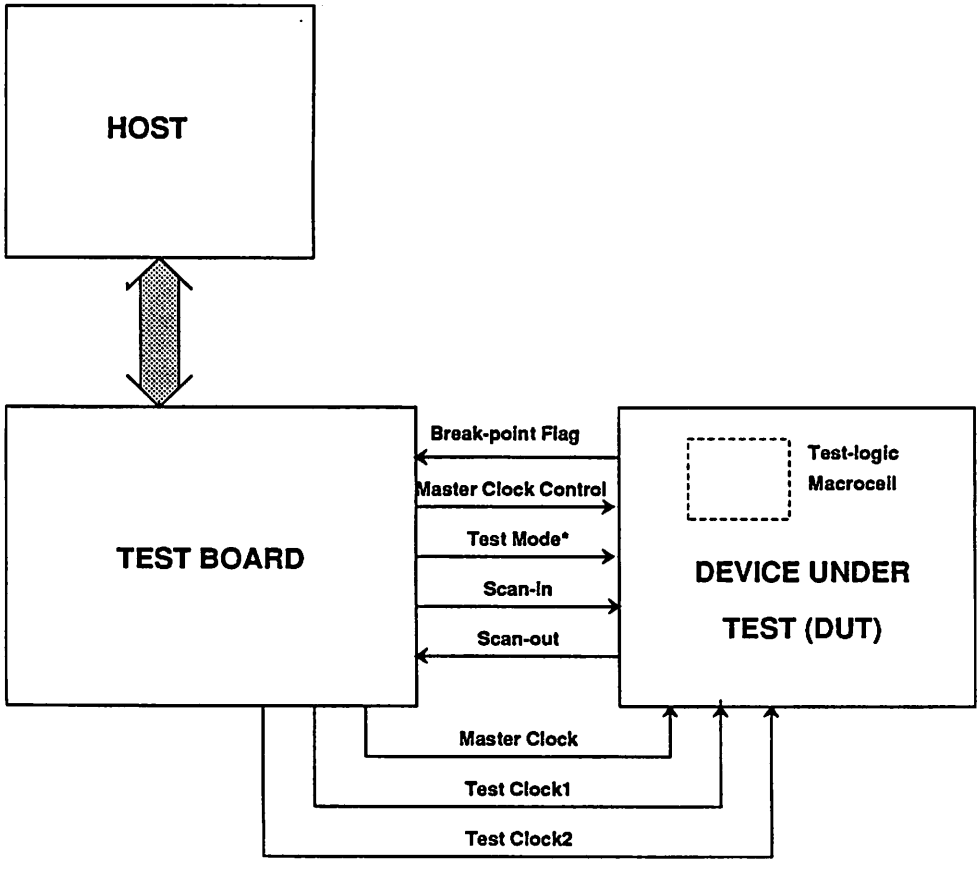


Figure 6.1: Block diagram of the test setup.

pins. The pins are break-point flag, master-clock control (MCC), scan-in, scan-out, Tphi1 (phase 1 of test-clock), Tphi2 (phase 2 of test-clock), and Test-mode*.

6.1.1 Scan Design

Scan path techniques [WiPa83] are very popular for testable design. Using scan registers in a design, allows one access and control of internal nodes in a chip. In our design, separate two phase clocks are used for normal operation and for test operation. Since some of the latches and registers may load on phase 1 and others on phase 2 of the master clock, keeping the test and master clock independent, simplifies the design. All the registers and latches are serially connected together through their scan-input and scan-output terminals (figure 6.2). The scan-output terminal of individual register bits is connected to the input of the next bit by abutment in tiled cells. The ends of the scan path are brought out as the chip's scan-in and scan-out terminals. When starting a serial read-out of the scan path, *shift-out must be performed first before a shift-in is done.* Otherwise, the data in the leading bit position would be lost.

Scan latch and a scan register are the two basic types of scan circuits. Two versions of each type of circuit were designed; one an exact mirror of the other about the horizontal axis. This makes routing of the scan path interconnections easier in bit-slice data paths. Alternate registers in the data path are mirrored so that scan-out of one register can be connected to the scan-in of the next register on the same side of the data path.

Scan Latch

The scan latch circuit, shown in figure 6.3, is a dynamic latch. A single phase of the clock (phidA) controls the data input through sDINt and sDINb terminals. These two terminals are electrically identical but are physically brought out on the top and bottom of the cell respectively. They are named differently to satisfy the layout tools. When phidA is asserted, data from the input terminal is transferred to the output terminals sDOUTt and sDOUTb at the top or bottom of

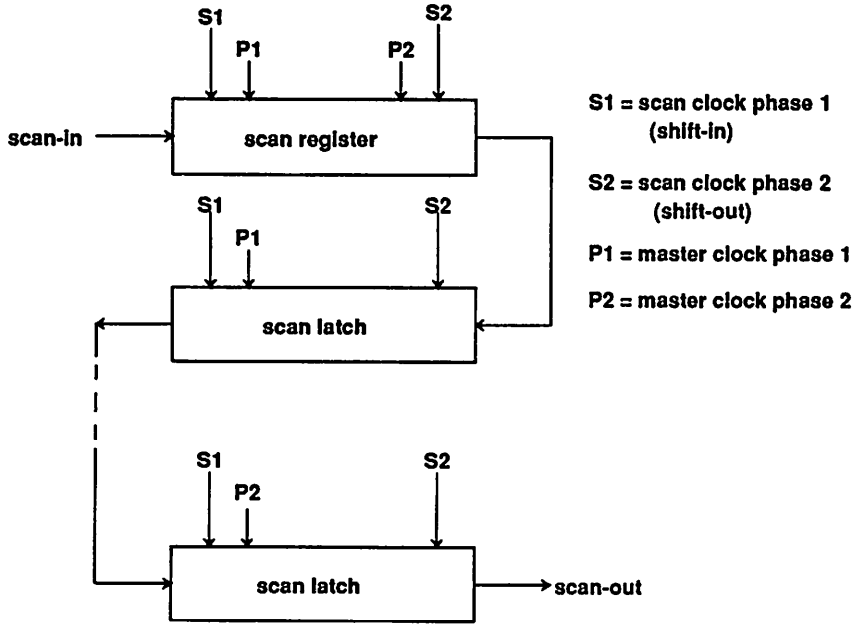


Figure 6.2: Hooking scan registers and latches into a serial scan path. Note the two scan latches in the figure operate on different clock phases during normal operation. By keeping the scan clock independent of the master clock, any timing conflicts that may arise due to serial connection of latches operating in different phases are avoided.

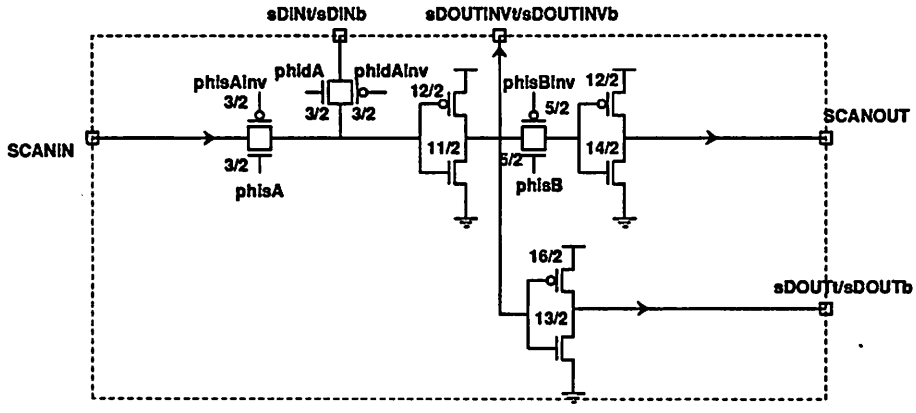


Figure 6.3: Circuit diagram of scan latch. The latch operates on only one phase during normal operation.

the cell respectively. For testing purposes, the SCANIN and SCANOUT terminals are used to serially connect the scan path. During testing, phidA is held low while phisA and phisB serially shift the data. PhisA and phisB are connected to phase 1 and phase 2 respectively of the scan clock. The two phases essentially act as shift-in and shift-out signals for the data.

Scan Register

The scan register circuit, shown in figure 6.4, is a two phase static register. Under normal operation both the input and output transfer gates, T2 and T4, for the scan-in and scan-out terminals are turned off. Data is loaded through the sDINt or sDINb terminal. During load, T1 is turned on and T3 is turned off which disconnects the feedback path. This allows much faster loading and reliable circuit operation compared to some designs which do not have a transfer gate in the feedback path but rely on a stronger driver at the data input for changing the stored state. The addition of T3 results in only a slightly larger cell. On the other hand, during testing, the feedback transfer gate T3 remains closed when data is being shifted in through scan-in terminal. This is done in order to avoid complicated control circuits for generating the LOAD/LOADINV signals for T3. Keeping the feedback path closed does not cause any problem during test mode

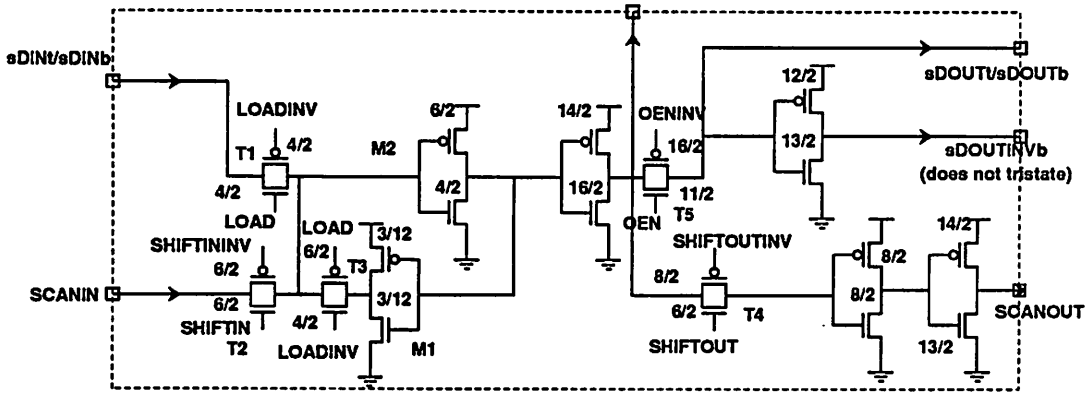


Figure 6.4: A two phase scan register.

since test clocks are operated at around 1 MHz. Ample time is thus available for changing the register state. Furthermore, shifting a new data into the register is helped by having a weak feedback inverter, M1, as shown in figure 6.4.

For reading out the data, transfer gate T5 is enabled. As the figure shows, sDOUTb and sDOUTt are tristateable whereas sDOUTINVb is not. During testing, data is shifted out through T4. The two inverters after T4 are necessary for providing an active drive into the next stage.

6.1.2 Test-logic Macrocell

Test-logic is a generic macrocell which can be easily incorporated into any chip design using a two-phase clocking scheme. The hierarchy of the Test-logic macrocell is shown in figure 6.5. A schematic of the macrocell is given in figure 6.6. The break-point generator circuit, BPGEN, contains a scan register (break-point register) and a comparator. The comparator compares the register's contents with the control unit FSM's state and the control-word address (PC output). When a match is found, a BPMATCH signal goes high setting a R-S flip-flop (cross-coupled nor gates). This results in BPFLAG becoming high and at the same time disables the two-phase master clock going to the rest of the chip's circuits. The processor thus goes into a 'wait' state. The master clock within the Test-logic circuit, however, continues to operate.

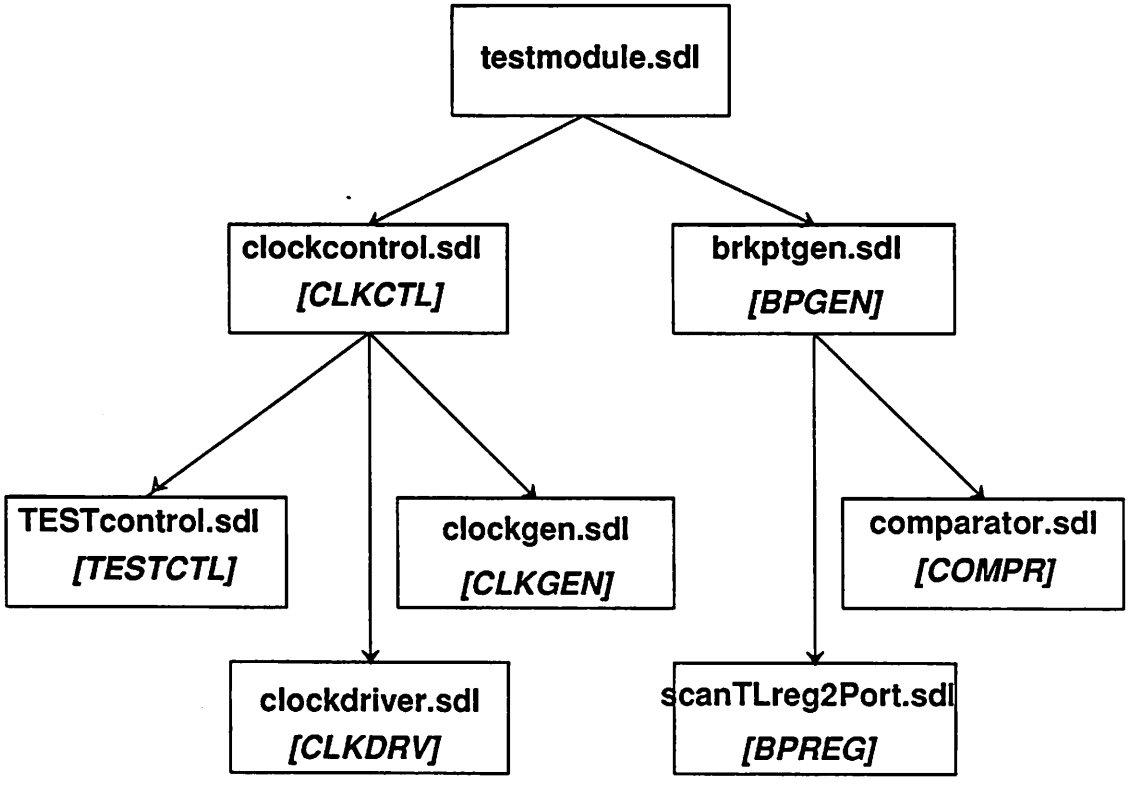


Figure 6.5: Cell hierarchy of the Test-logic macrocell.

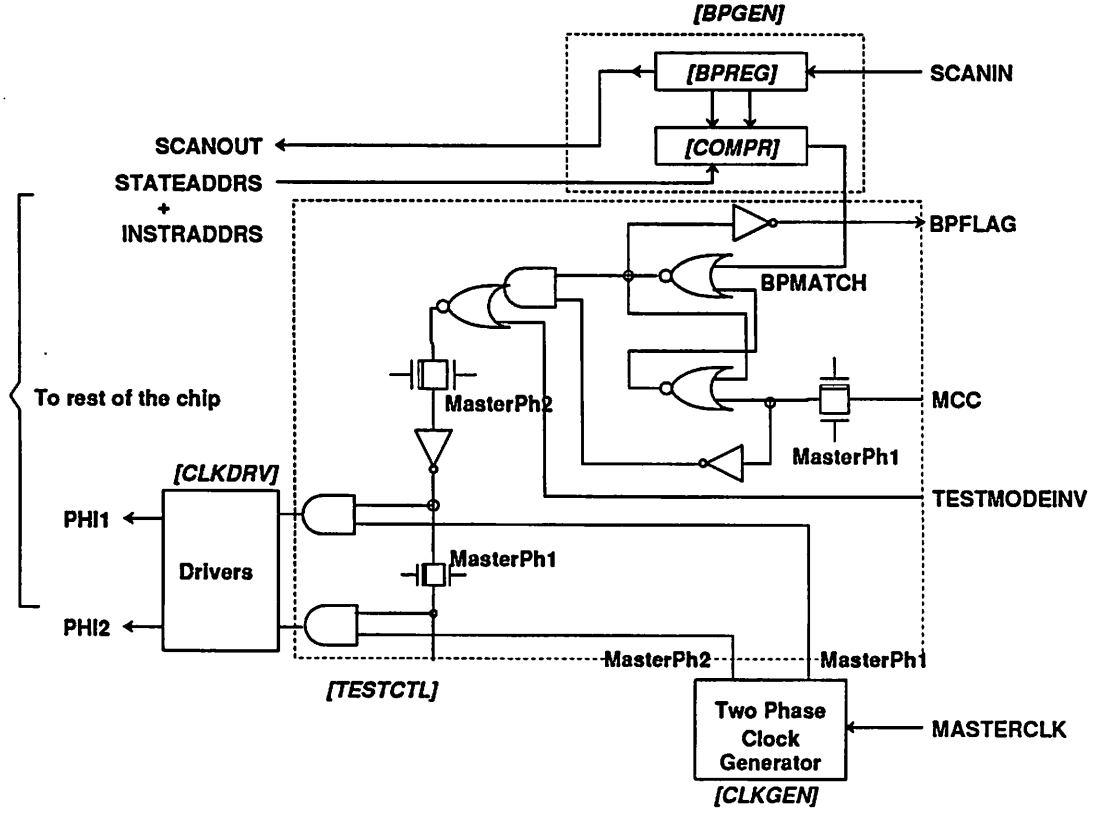


Figure 6.6: Schematic of the Test-logic macrocell.

BPFLAG is constantly monitored by the host. When BPFLAG is asserted, the host responds by applying a master-clock control signal, MCC to the chip. This resets the R-S flip-flop but continues to hold the two-phase clocks low. Meanwhile, the tester reads and writes into the scan register through the SCANOUT and SCANIN pins. This includes loading a new break-point state in the break-point register. On completion of this process, the host releases MCC thus enabling the two-phase master clock, allowing the processor to resume operation. Note, the break-point generator circuit can be modified or replaced to match the needs of a particular processor. On the other hand, the Test-logic macrocell provides a common interface through which the tester can communicate with the chip.

6.1.3 Test-Board

The test-board (figure 6.7) uses a fairly simple design for collecting the scan path data from the chip and loading in new data into the scan registers. The board plugs into one of the expansion slots on an IBM PC-AT and communicates with it through the PC bus. On the test-board are six main registers.

(i) *Cntl-fifo:* PC writable; holds the control-bit vector for a mux.

(ii) *Din-fifo:* PC writable; holds the test vector for the scan path.

(iii) *Dout-fifo* PC readable; stores the output data from the scan path.

(iv) *Counter register* PC writable; holds the length of the scan path.

(v) *Status register* PC readable; contains status signals from the DUT as well as the test-board.

(vi) *Control register* PC writable; holds the control signals for the test-board and the DUT. The signals include RSTDUT for resetting the DUT; RSTBOARD for resetting the flip-flops and registers on the test-board; LDSCAN for enabling the test-clocks in order to begin serial shifting of the scan path; CLRTC for clearing a flip-flop which is set when counter completes its count; ENFIFO3 for enabling Dout-fifo in order to read Dout-fifo from the PC.

In addition, an address decoder generates read/write signals for the various registers. These are listed below:

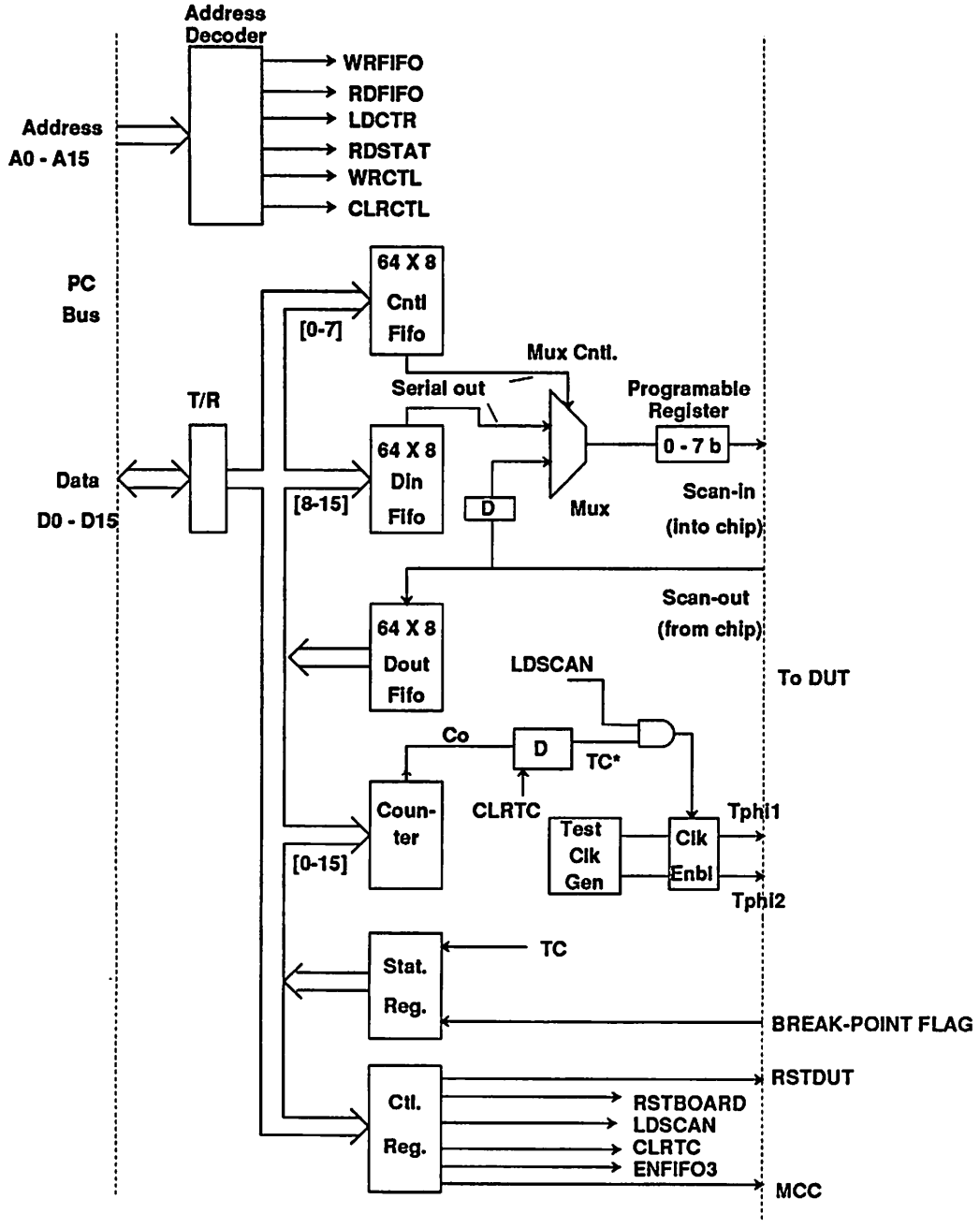


Figure 6.7: Schematic of the test-board. Only the essential components and signals are shown here.

(i) *WRFIFO:* Provides the write signal for loading Cntl-fifo and Din-fifo. Both the fifo registers share half the data word. Din-fifo gets 0-7 bits and Cntl-fifo gets 8-15 bits.

(ii) *RDFIFO:* Provides the read signal for reading Dout-fifo.

(iii) *LDCTR:* Provides the load signal for the counter register.

(iv) *RDSTAT:* Provides the read signal for the status register.

(v) *WRCTL:* Provides the write signal for the control register.

(vi) *CLRCTL:* Provides the clear signal for resetting the control register.

Initially, the PC loads data into Din-fifo and Cntl-fifo. Din-fifo contains the serial test data to be loaded into the scan registers including the break-point register. For each bit of test data, Cntl-fifo contains a corresponding control bit for setting a mux on the test-board. As shown in figure 6.7, the mux is used to select either a new data from the Din-fifo for serially shifting into the scan path or recirculating the scan data back into the scan path. Thus any section of the scan path may be modified, or restored to its original state during the serial shifting operation. The scan data is also loaded serially into the Dout-fifo.

A flow-chart describing the operation of the tester is given in figure 6.8. The two-phase test-clock generated on the test-board is enabled by the host after receiving the break-point flag signal. The same clock is also used for serially shifting data in and out of the fifo registers on the test-board. Once the serial shift operation starts, a counter on the test-board keeps track of the number of bits being shifted. The counter is initialized by the host to count up to a number equal to the number of bits in the scan path. When the exact number of shifts are completed, a carry-out signal from the counter sets a flip-flop called TC. This signal disables the test-clock and also sets a bit in the status register. As a result, the host, which continuously monitors the status register, responds by up loading the scan path data in Dout-fifo and down loading new data into the Din-fifo and Cntl-fifo. In order to simplify transfer of data between host and the tester, the test vectors, which are loaded and read out of the fifo registers, are made integral multiples of eight. This is achieved through a programmable register on the test-board which extends the scan path by a length of 0 to 7 bits. Once the data transfer between the host and the fifo

registers is completed, the host releases MCC which restarts normal operation of the chip.

The entire operation from setting the break-point flag to re-enabling the processor is done in less than 5ms in order to avoid corrupting the data on dynamic nodes of the processor circuits. The tester operating at 1 MHz takes about 0.5 ms to read out a scan path of 512 bits. (In the present tester design, the Dout fifo can accommodate up to 64×8 bits). This leaves enough time for the host to write 128 (64×2) words of data into Din and Control fifo, read 64 words of data from Dout fifo, and perform some control functions such as asserting Master-clock control, enabling test-clocks, etc. Controlling the tester operation from the host avoids the need for a complicated tester design with a dedicated micro-processor on the tester-board itself. Another design issue is the size of the host memory. Since each break-point requires 128×8 bits, with a 2 M bits extended memory on an IBM PC-AT, we can handle 2000 break-points. The desired number of break-points is initially specified by the user to the host. It keeps track of the number of break-points occurring, in a software counter. When the desired number of break-points is completed, the host exits the testing session.

6.2 Implementation of test circuits in the Robot Control Processor

Since the robot control processor operates under program control, the natural way to set break-points is by specifying program states. As discussed in section 6.1.2, the break-points are given in terms of the PCU's fsm state and the instruction address from the program counter. For a custom processor such as the robot controller, the chip needs to be tested only for verifying the program in the control-store works. The PCU design makes feasible the addition of a test-state in which a small test program is executed. Transition to the test-state may be controlled through external signals going into the PCU's fsm.

In the robot control processor, scan registers are used every where except

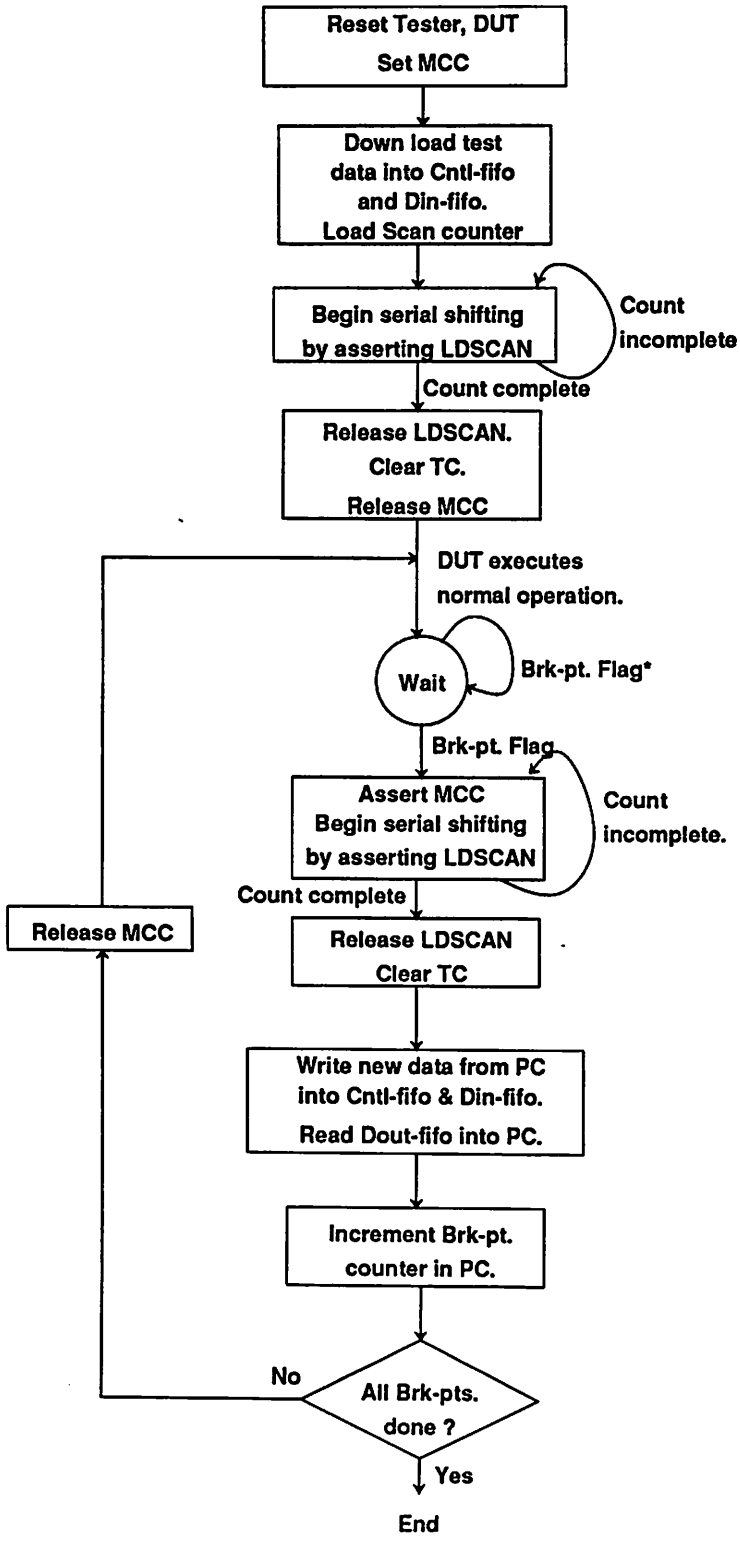


Figure 6.8: Flow chart for the testing procedure.

Macrocell Name	Register Name	Register Type	Register size
TESTLOGIC	BPREG	scanTLreg2Port	11 bits
APU1	REGX0	scanreg2Portmx	6
APU1	REGX1	scanreg2Port	6
APU1	REGX2	scanreg2Portmx	6
APU1	EAGATE	scanlatch_ph1	6
LGU	FSMOUTREG	scanTLlatch_ph1	2
PCU1	CFSMOUTREGI	scanTLlatch_ph1	7
PCU1	CFSMOUTREGII	scanTLlatch_ph1	8
PCU1	PCOUTREG	scanTLlatch_ph1	4
PCU1	LPCOUTREG	scanTLlatch_ph1	7
PCU1	TIMERINREG	scanTLreg2Port	11
PCU1	TIMEROUTREG	scanTLlatch_ph1	12
PCU1	CSTOREOUTREGI	scanTLlatch_ph1	27
PCU1	CSTOREOUTREGII	scanTLlatch_ph1	30
AU1	MOR	scanlatch_ph1mx	20
AU1	ACCUMULATOR	scanlatch_ph1	20
AU1	REG0	scanreg2Portmx	20
AU1	REG1	scanreg2Port	20
AU1	RCOEF	scanlatch_ph1mx	20

Table 6.1: List of scan registers/latches used in the robot control processor. They are listed in the same order in which they are connected. Scan-in of the chip goes into BPREG and scan-out comes out of RCOEF. The total number of bits in the scan path of the chip is 243.

in the ram. The outputs of the control-store and the fsm in the PCU are also latched in scan type latches. This potentially allows the user to introduce new control words and perform operations not in the original program. In order to analyze the data from the scan path, the interconnections of the scan path registers must be known. These interconnections for the robot control processor are given in table 6.1. The registers are listed in the same order in which they are connected.

Chapter 7

Implementation of the Robot Control Algorithm on the Custom Processor

The adaptive control algorithm for a two-axis robot, described in chapter 7, is implemented on the custom digital signal processor described in chapter 3 and 5. Controllers for both the joints are implemented on a single processor which also directly handles all the I/O operations through the parallel prots. The layout was automatically generated using the LagerIII silicon compiler system described in chapter 4. Input to the compiler is provided at the assembly language level. Higher-level input was not used since the RL compiler (for compiling 'C' like programs) was not fully operational at the time the chip was generated. Moreover, implementing the algorithm at the assembly language level provides a much clearer understanding about the capabilities and limitations of the processor. Since the processor performs integer arithmetic only, all the variables at various computational nodes are scaled in order to avoid overflows and underflows. The scaling is described in the next section following which a description is given about the utilization of the various resources of the processor in implementing the robot control algorithm. In section 7.3, some fabricated chips are discussed.

7.1 Scaling

Integer arithmetic makes scaling necessary for avoiding overflows while at the same time attempting to obtain maximum dynamic range at each computational node. Through simulations of the algorithm in 'C' (see appendix G for the basic program), we determined the dynamic range required at each computational node. Based on these, suitable scale factors were specified at each node. All scale factors were constrained to powers of two in order to avoid actual multiplications and divisions. In many cases, the scale factors were combined with the gain for minimizing the number of operations. In general, conversion of a decimal representation of a quantity x into its integer representation is given by,

$$x.2^{b-1} / S$$

where S is a scale factor chosen to ensure x/S is within ± 1 and b is the integer word-size. Thus the integer representation has an effective gain of $2^{b-1} / S$. The gain for the input quantities should also take into account the gain through the transducers.

The position decoder generates 153600 pulses per revolution of the robot joint or 24446 pulses per radian. In order to keep the register size required to accumulate the position count to 16 bits, we chose to divide the count by four, resulting in an effective gain of 6111.5 counts per radian through the decoder. Looking at it in another way, this is equivalent to a scale factor of 5.36 or 8/1.49. Consequently as shown in figure 7.1, the reference input must also be scaled by 8/1:49 before converting into a 16 bit integer representation (assuming a 16 bit data path). Similarly, the velocity input through the 12 bit A/D has a gain of 325.9 based on a maximum speed of 2π radians per second. This is equivalent to a scale factor of 32/5.088 for a 16 bit integer representation.

Going back to the position input, the error between it and the reference input is multiplied with a gain K_p as discussed in chapter 2. The question then is what should be the scaled value of K_p that must be loaded into the processor. Considering a typical value of 40 for the gain, we need a scale factor of 64 for getting the value below unity. It is also multiplied with 1.27/1.49 so that the input to the

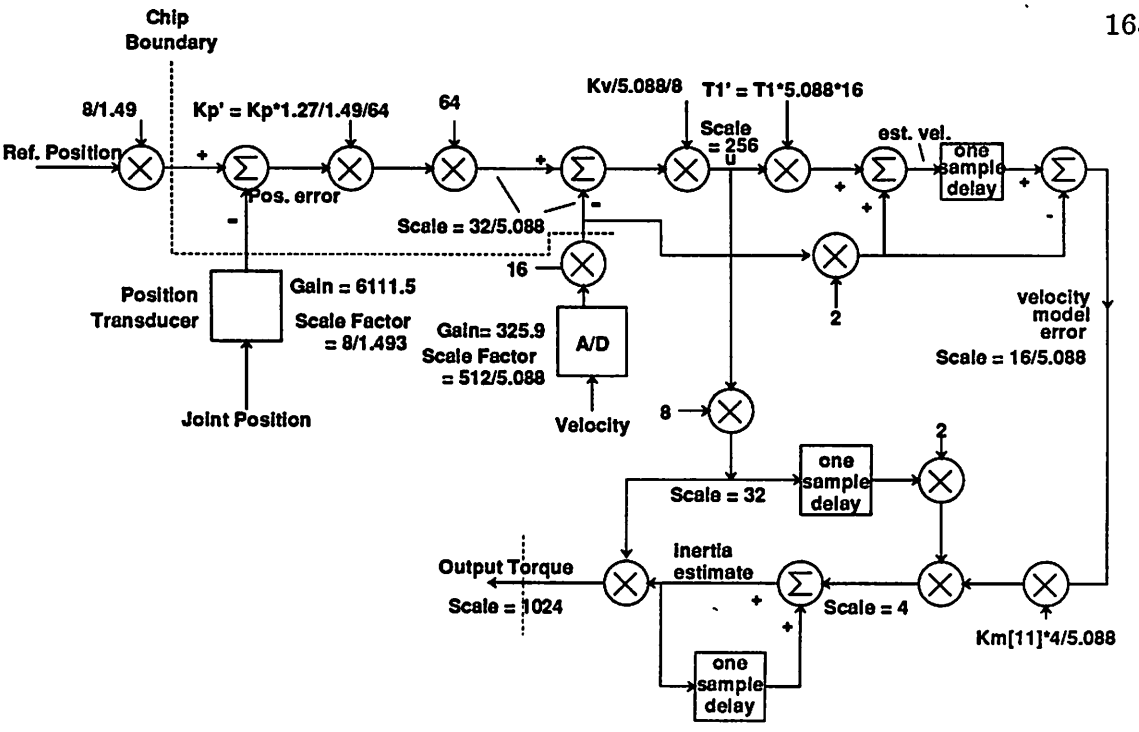


Figure 7.1: Partial flow-diagram showing scale factors at various nodes. Multiplication with constants are done through shifting. All the coefficients are first read from an external source. These coefficients are pre-scaled with the appropriate factors as shown. Position and velocity measurements are also pre-scaled before being read into the chip.

velocity summing node has a scale factor of 32/5.088, which is same as that of the output of A/D. This, along with scale factors for all the other nodes are shown in figure 7.1. Typical values for the coefficients and their scale factors are given in table 7.1.

7.2 Programming the custom processor

The assembly language program written for implementing the adaptive control algorithm is given in appendix F. It has 66 distinct states and 35 blocks of code. One can easily trace the flow of the program by following the state transitions specified in the *cfsm* section of the assembly language program. In the first three states, which are executed only once in the beginning after reset, the program

$Coeff./Variable$	Typical Value	$1/Scale$
K_p	40	1.27/1.49/64
K_v	20	1/5.088/8
T	0.001	$5.088 * 16$
$K_{m[11]}$	1	4/5.088
$K_{m[12]}$	0.1	4/5.088
$K_{m[21]}$	0.1	4/5.088
$K_{m[22]}$	0.1	4/5.088
$\hat{m}_{[11]}$	8.5	1/32
$\hat{m}_{[12]}$	1.25	1/32
$\hat{m}_{[21]}$	1.25	1/32
$\hat{m}_{[22]}$	0.5	1/32

Table 7.1: Coefficients/Variables along with typical values and scale factors.

does some initialization and reads in coefficients and other data. The rest of the states are executed once every sample. The program has two sections: an outer loop which starts with state, PCTL1 and an inner loop which starts with state, ADTRIG. The variable, L, read from outside during initialization determines the number of iterations of the inner loop for one iteration of the outer loop. Note, only one iteration takes place per sample period. By setting some of the coefficients (such as K_p or K_v) in the program to zero, some functions of the program (such as P-control or D-control) can be disabled.

The assembly program was processed by the rassCG program which generated the parameter file (see appendix B) used for layout. The parameter file includes the pla table for the fsm and the cstore. Both the tables were minimized using espresso [Rud86] which resulted in 10-20% reduction in their sizes. The total number of instructions in the program are 172 and the execution time is between 593 to 601 cycles per sample. The exact number of cycles depends on the conditional branches taken. The size of the sample period, however, can be fixed by the timer by waiting in the IDLE state till the timer completes its count. The large ratio between the number of cycles per sample and the total number of instructions in the program indicates good usage of subroutines and loops for minimizing the program size.

Subroutines, Addressing

Since only one level of subroutine was used, we decided not to use a stack; instead, subroutines were implemented by having different states execute the same block of code. For instance multiplication is used a number of times in the program and therefore it is made into a subroutine. Block 5 executes the basic shift and add operation. As seen in the cfsm section of the assembly program, block 5 is addressed by all the states starting with the name MULT. The size of the code space required for multiplication is further reduced by putting the basic shift and add operation into a loop thus having only one instruction in block 5. The loop iterates 18 times; first and last instructions of the multiplication routine are not included in the loop since they have somewhat different operations. As a result of using a subroutine and a loop, the number of instructions for doing the core multiplication is reduced to just one.

An important consideration in executing subroutines is passing data. In the multiplication routine, data is passed to the subroutine via the data path registers rcoef (to hold the multiplier) and mor (to hold the multiplicand). Another way of passing data to the subroutines is through a pointer. For example, block 23, which performs friction compensation is used twice; once for each joint. The address pointer for the variable ul (either for joint 1 or joint 2) is loaded into REGX0 of the APU prior to executing block 23. The first instruction in block 23 reads in the value of ul from memory using the address pointer. Passing address pointers is even more advantageous when several variables have to be passed to a subroutine. Another example of using APU register for addressing is in block 1 where the block is repeated 21 times in order to read in a series of input data from an external port. The data is first loaded into the AU register R0 while in the same cycle the previously loaded data in R0 is moved into memory. The memory address is obtained by incrementing APU register REGX0 on every iteration. Incrementing is done by specifying the address field in the control-word to be 'one', which then gets added to the register. The modified value of REGX0 is also stored back into the register on ever iteration. In addition to addressing, the APU registers are also used as loop

counters.

Looping

In block 32, for example, REGX1 is used as a loop counter for 22 iterations of the block in order to output a series of data. REGX1 is first loaded in block 31 and decremented by one on each iteration of the block. As specified in the cfsm section of the program, when the register value becomes negative, the fsm makes a state transition from OUTPUT state to DELAY state. Note, OUTPUT state is the one in which block 32 was being executed. As seen in the code for block 32, decrementing the loop counter, REGX1, takes one cycle; however, this operation is combined with the data output operation to the port and therefore does not cost any extra instruction cycles. For a single instruction loop, however, using an APU register as a loop counter would cost an extra cycle (100% increase!). Therefore in states RDCONSTS (block 1) and MULT1 (block 5), for example, the PCU loop counter is used. The state transitions following completion of the loops are specified in the cfsm section. Further more, the PCU loop counter does not require any instruction cycle overhead for initializing the counter prior to entering the loop. Next, we examine the implementation of conditional operations in the processor.

Conditional Operations

States FRICCOMP1 (block 21), FRICCOMP2A (block 22), and FRICCOMP3A (block 23) are involved in implementing the friction compensation discussed in chapter 2. Given below are the program blocks for friction compensation. The state transitions and microinstructions are written in a pseudo mnemonic.

```
FRICCOMP1 (block 21): ; this is a state
    (1) read(FRth) into mor ; friction threshold
    (2) move FRth into acc, read(Xv1) into mor ; read joint velocity
    (3) take negative of FRth ; negative of FRth
    (4) acc = |XV1| - FRth ;
```

```
    (5) REGX1 = addrs[f1] ; address of fric. comp.
if AUSIGN is 1 go to FRICCOMP3A ; state transition
else if AUSIGN is 0 go to FRICCOMP2A
FRICCIMP2A (block 22):
    (1) set cc=AUSIGN in LGU, read(f1) into mor ;
    (2) move REG0 into mor ; Kf.ev
    (3) acc = f1- Kf.ev if cc is set   ; accumulate if ef is neg.
    (4) set cc=!cc ; complement cc
    (5) acc = f1+Kf.ev if cc is set       ; accumulate if ef is pos.
    (6) write(f1) to memory ;
go to FRICCOMP4 ; state transition
FRICCOMP2A (block 23):
    (1) read(u1) into mor ;
    (2) acc = -u1    ; take negative
    (3) set PFLAG=AUSIGN in LGU ; set PFLAG if u1 pos
    (4) acc = u1  ;
    (5) set cc = AUSIGN in LGU, read(f1) ; set cc if u1 is neg
    (6) acc = f1, mv REG0 to mor ;
    (7) acc = f1-Kf.ev if cc is set    ; accumulate if u1 is neg
    (8) set cc = PFLAG in LGU    ; set cc if u1 is pos
    (9) acc = f1+Kf.ev if cc is set    ; accumulate if u1 is pos
    (10) write(f1) to memory ;
```

The above program code implements friction compensation by setting condition code, cc, in LGU and using it to execute several conditional accumulate instructions for providing decisions. At the end of state FRICCOMP1 (block 21), a conditional branch operation is also performed. Note, the branch is delayed by one cycle following the subtraction operation done in instruction 4. However, in this case a useful operation is done during the extra cycle (instruction 5). The use of LGU, on the other hand, requires one cycle for setting condition codes before a conditional accumulate can be done. In the above implementation 14 cycles are

needed in the worst case for providing friction compensation. It also requires three states and a total of 20 instructions in three blocks. An alternative implementation without using LGU but using PCU for all conditional operations is described below.

The code for friction compensation using only PCU for conditional operations has four new states, FS1, FS2, FS3, and FS4 which replace FRICCOMP2A and FRICCOMP3A. Associated with them are four new blocks, 51, 52, 53, and 54 which replace blocks 22 and 23.

```
FRICCOMP1 (block21):
{same as before} ; |Xv1| - FRth
if AUSIGN is positive go to FS1
else if AUSIGN is neg go to FS2
FS1 (block51):
    (1) acc = acc, read(f1) into mor ; check sign of Xv1 in acc
    (2) no op ; need dummy cycle for delayed conditional branch
if AUSIGN is positive go to FS3
else if AUSIGN is neg go to FS4
FS2 (block52):
    (1) read(u1) into mor ;
    (2) acc = mor, read(f1) into mor ;
    (3) no op ; dummy cycle for delayed conditional branch.
if AUSIGN is positive go to FS3
else if AUSIGN is neg go to FS4
FS3 (block53):
    (1) acc = mor+r1 ; f1+Kf.ev
    (2) write(f1) to memory ;
FS4 (block54):
    (1) acc= mor-r1 ; f1-Kf.ev
    (2) write to f1 ;
```

The above code achieves friction compensation with only 9 cycles in the worst case. It uses five states and a total of 13 instructions in five blocks. Clearly, for the friction compensation function, solely using PCU's zero over-head conditional branch operations for decision making provides a much more efficient implementation than using logical unit. For comparison, we also give below code [Anwr87] for implementation of the friction compensation on a TMS32010. The code takes 14 cycles in the worst case and takes up 25 instruction-rom words. (Numbers inside parenthesis are number of cycles required for the instruction.)

```
FRIC $MACRO A,B,C,D ; A=Xv, B=u1, C=FRth, D=+1 or -1
LAC :A:,0 (1)
SUB :C: (1)
BLEZ $+6 (2)
LAC ONE,8 (1)
SACL :D: (1)
B $+19 (2)
LAC :A:,0 (1)
ADD :C: (1)
BGEZ $+6 (2)
LAC MINUS,8 (1)
SACL :D: (1)
B $+11 (2)
LAC :B:,0 (1)
BLEZ $+6 (1)
LAC ONE,8 (1)
LAC ONE,8 (1)
SACL :D (1)
B $+4 (2)
LAC MINUS,8 (1)
SACL :D: (1)
```

Port Address	Description	Symbol
0 (RD)	Coefficients/ Constants	
1 (RD)	joint1 vel (A/D)	$\theta_{v[1]}$
2 (RD)	joint2 vel (A/D)	$\theta_{v[2]}$
3 (WR)	joint1 torque (D/A)	$q_{[1]}$
4 (WR)	joint2 torque (D/A)	$q_{[2]}$
5 (RD)	joint1 pos (quad. dec)	$\theta_{[1]}$
6 (RD)	joint2 pos (quad. dec)	$\theta_{[2]}$
7 (RD)	joint1 ref pos	$r_{[1]}$
8 (RD)	joint2 ref pos	$r_{[2]}$
9 (WR)	ram variables	
10 (WR)	A/D trigger signal	

Table 7.2: Description of input and output ports

```
$END
```

I/O operations

All I/O operations are performed through the 20 bit wide parallel I/O bus. Port addresses are provide through the 4 bit wide I/O address bus. In the program nine of the 16 port addresses are used as shown in table 7.2. For reading velocity data through the A/Ds, a begin-conversion signal is written to port 10 during state, ADTRIG, near the beginning of each sample. All the coefficients and constants are read through port 0 whereas the current value of the various ram locations are written to port 9. Table 7.3 gives the list of quantities sequentially (one per cycle) read in through port 0 during state RDCONSTS and table 7.4 gives the list of quantities sequentially (one per cycle) written out to port 9 during state OUTPUT.

Symbol	Description
$K_{p[2]}$	joint2 P-control gain
$K_{p[1]}$	joint1 P-control gain
$K_{v[2]}$	joint2 D-control gain
$K_{v[1]}$	joint1 D-control gain
$K_{i[2]}$	joint2 I-control gain
$K_{i[1]}$	joint1 I-control gain
$K_{f[1]}$	joint1 fric. coeff.
$K_{f[2]}$	joint2 fric. coeff.
$K_{m[11]}$	adaptation gain
$K_{m[12]}$	adaptation gain
$K_{m[21]}$	adaptation gain
$K_{m[22]}$	adaptation gain
$\hat{m}_{[11]}$	inertia estimate
$\hat{m}_{[12]}$	inertia estimate
$\hat{m}_{[21]}$	inertia estimate
$\hat{m}_{[22]}$	inertia estimate
V_{db}	dead band threshold
V_f	friction threshold
T	sample period
L	ratio of inner loop iterations to outer loop iterations
PERIOD	sample period: number of cycles

Table 7.3: List of input quantities sequentially read through port 0

Symbol	Description
$\theta_{[1]}$	joint1 position
$\theta_{[2]}$	joint2 position
$\theta_{v[1]}$	joint1 velocity
$\hat{\theta}_{v[1]}$	joint1 model velocity
$e_{v[1]}$	joint1 model vel error
$u1_{[1]}$	joint1 PI-control output
$e_{u[1]}$	joint1 vel error
$u_{[1]}$	joint1 PID output
$\theta_{v[2]}$	joint2 velocity
$\hat{\theta}_{v[2]}$	joint2 model velocity
$e_{v[2]}$	joint2 model vel error
$u1_{[2]}$	joint2 PI-control output
$e_{u[2]}$	joint2 vel error
$u_{[2]}$	joint2 PID output
$\theta_{v[2]}$	joint2 velocity
$\hat{m}_{[11]}$	inertia estimate
$\hat{m}_{[12]}$	inertia estimate
$\hat{m}_{[21]}$	inertia estimate
$\hat{m}_{[22]}$	inertia estimate
$q_{[1]}$	joint1 torque
$q_{[2]}$	joint2 torque
$\hat{d}_{f[1]}$	joint1 fric. comp.
$\hat{d}_{f[2]}$	joint2 fric. comp.

Table 7.4: List of output quantities written to port 9

7.3 Chip Implementation

A complete customized processor for a two-axis, adaptive robot arm controller was successfully generated using the architecture and design methodology described in this thesis. All layout verifications were done by first extracting the circuit from the layout and then performing switch level simulations using *esim* [Term]. We also wrote and verified behavioral simulation models of the leaf-cells using *dsim* [Shung88]. However, it proved to be too slow to be very useful for simulating the entire processor. In addition to simulation, we also fabricated the major macrocells separately and tested them for proper operation. Copies of the die photos of the 3-T ram and the PCU appear in figures 7.2 and 7.3 respectively.

A copy of the die photo of the complete processor is shown in figure 7.4, the major macrocells are identified in figure 7.5, and the pin-out is given in figure 7.6. The chip was fabricated in 2 micron CMOS process through MOSIS. It contains > 16000 transistors and measures 8406 microns by 7145 microns. Several of the chips were tested on a test set-up built for the purpose. Tests showed the chip can perform at up to 15 MHz with a power consumption of approximately 200 milli-watts. Some of the results are discussed below.

7.3.1 Test Results

The first test was done to check the scan path. The internal master clock was disabled by pulling TESTMODINV and MCC pins high and a sequence of 1's and 0's were serially shifted in through the SCANIN pin while the output on the SCANOUT pin was observed. This output correctly followed the input pattern after a delay of exactly 243 cycles, which is the length of the scan path in this chip. When the entire scan path is serially loaded with either 1's or 0's, the BPFLAG is set as expected since the BPREG as well as PCOUTREG and CFSMOUTREGI have the same data. As soon as a sequence of 1's (or 0's) being serially shifted through the scan path is changed to a sequence of 0's (or 1's) the BPFLAG disappears, since BPREG no longer matches the other two register. After 55 shift-cycles, when

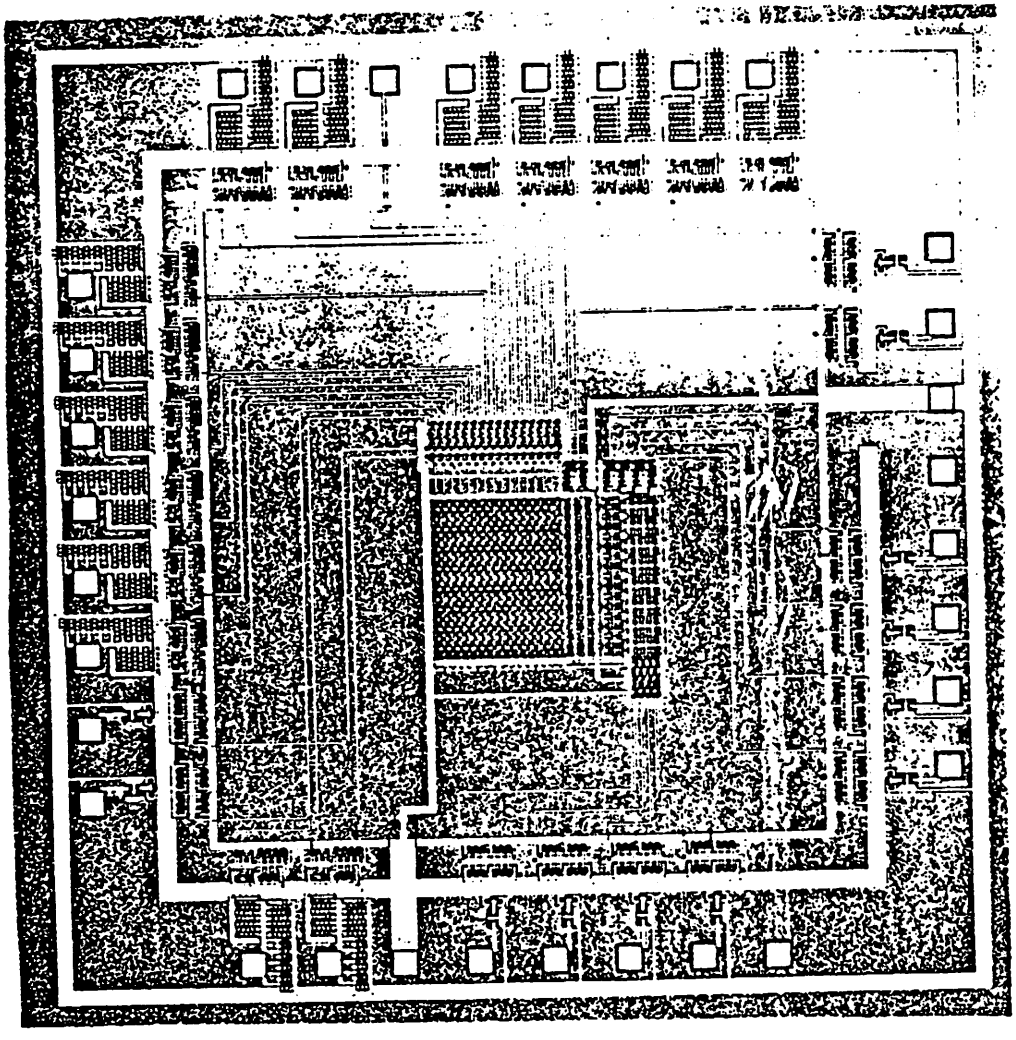


Figure 7.2: Die photo of a 3T ram test chip.

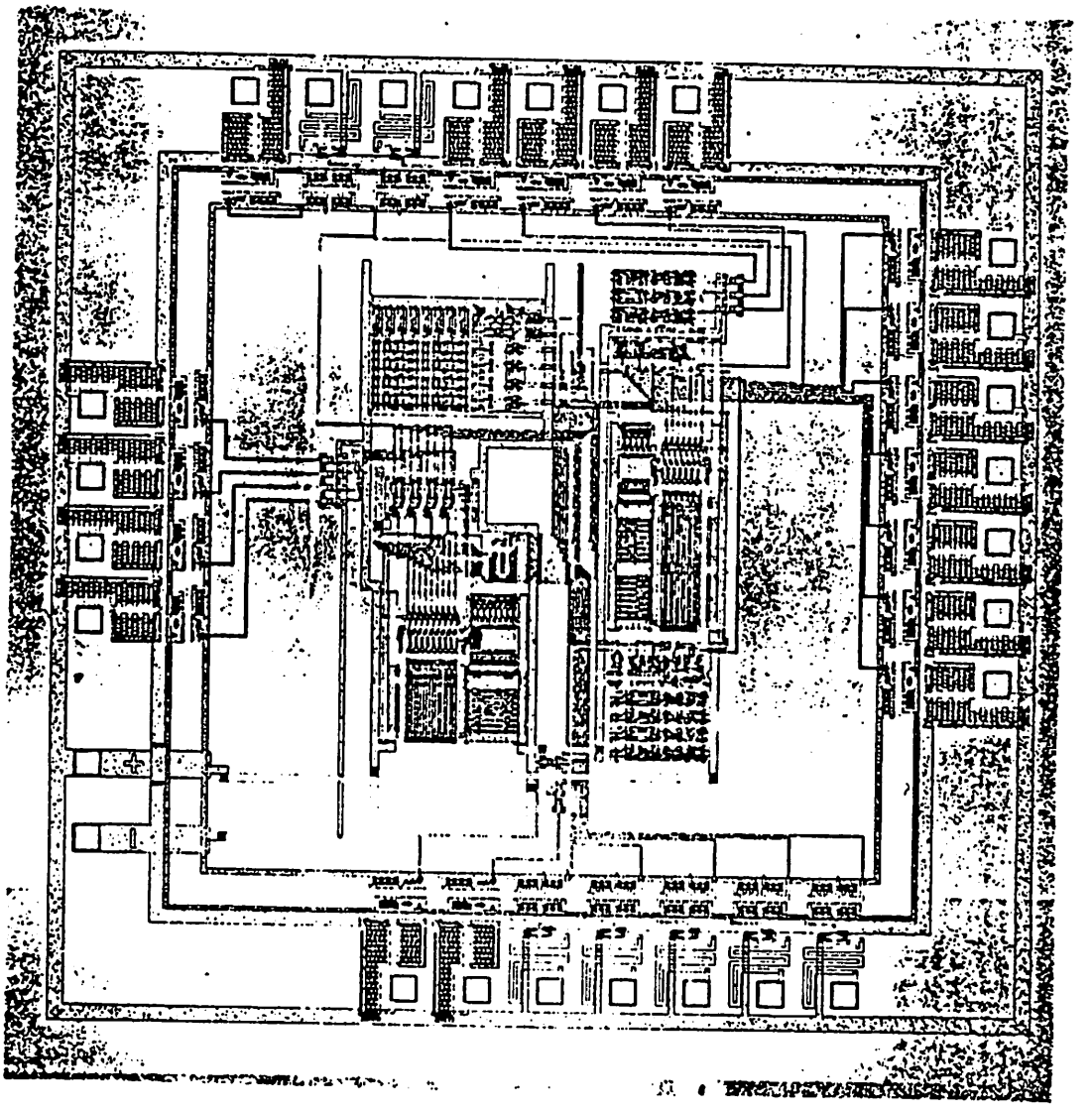


Figure 7.3: Die photo of a control unit (PCU) test chip.

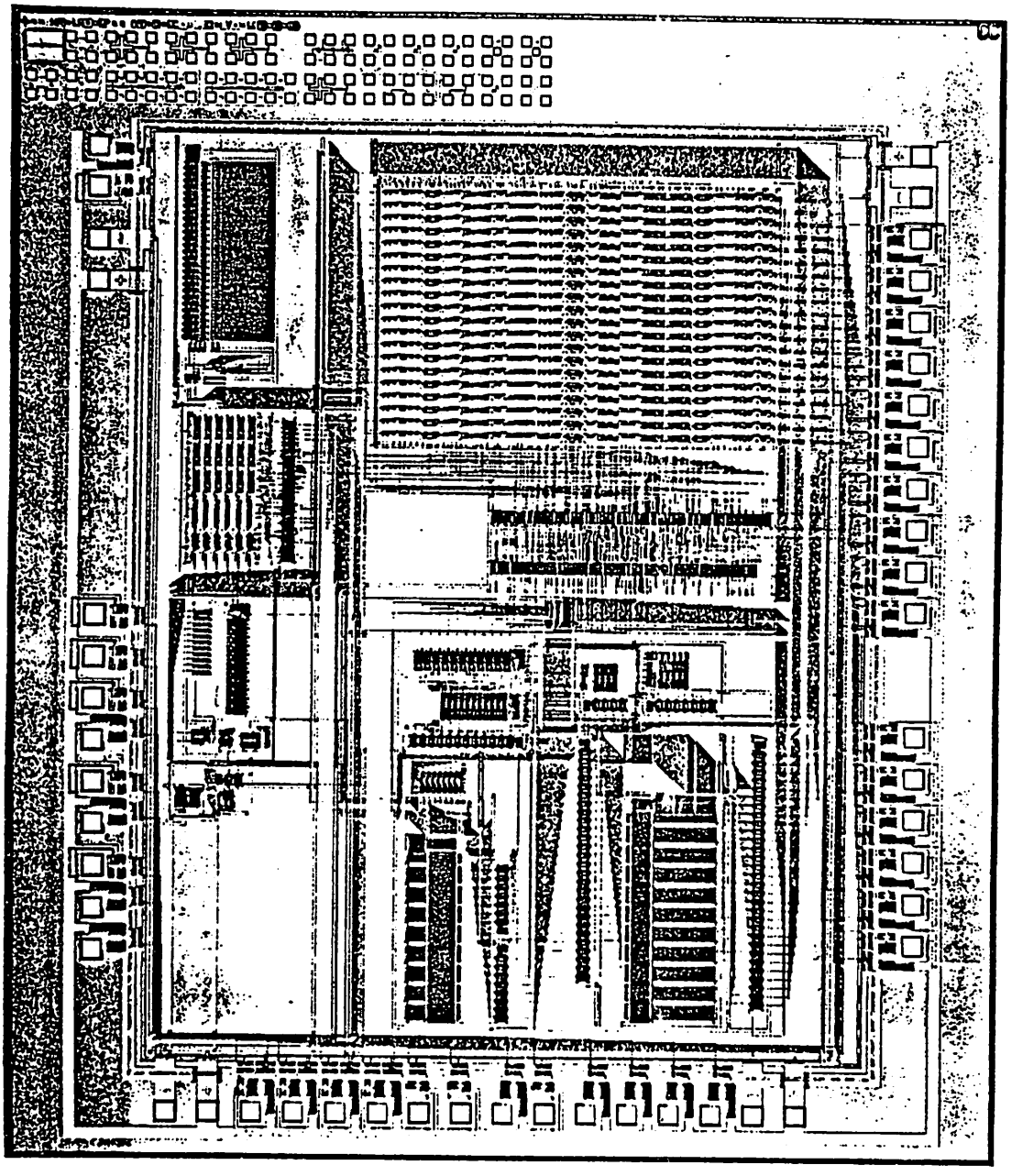


Figure 7.4: Die photo of the robot controller chip.

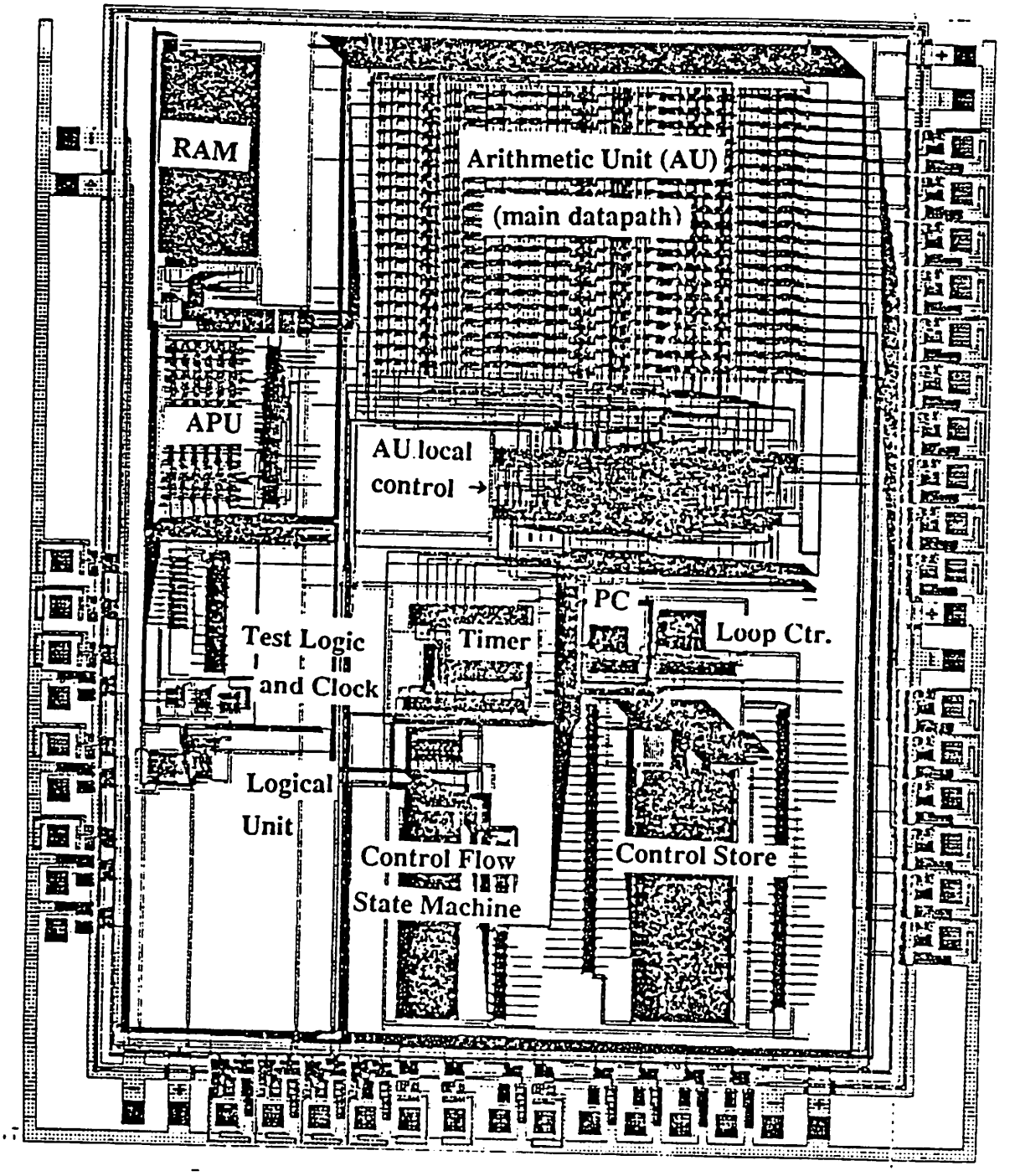


Figure 7.5: A cifplot of the robot controller chip showing the major macrocells.

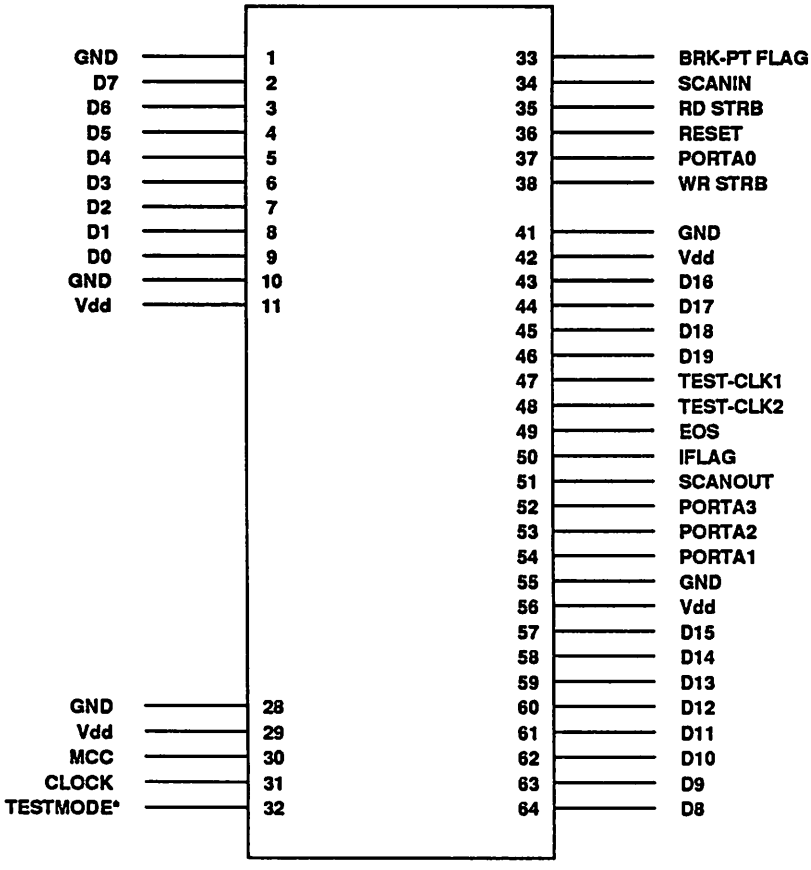


Figure 7.6: Pin out of the robot controller chip.

PCOUTREG and CFSMOUTREGI also get serially loaded with 0's (or 1's) through the scan path, the BPFLAG sets immediately.

In the second experiment the processor was allowed to run in its normal mode and the various output pins were observed. RDSTRB, WRSTRB, and PORTA0-PORTA3 (address port) were verified to appear at the correct program steps. This showed the processor was sequencing properly and correctly setting the read/write strobes as well as the port addresses. Moreover by applying the RESET, we could get the processor to begin the program sequence again.

For testing out the arithmetic operations, we allowed the processor to run in its normal mode for a few cycles and then used the scan path to serially read out the internal registers. However, we found the data coming out of the SCANOUT pin, after the processor has gone through a few normal cycles, is corrupted. Upon some experimentation we arrived at the following explanation. When the processor is stopped in its normal mode, (this is done by tying MCC pin high for this experiment and controlling the internally generated two-phase master clock through TESTMODEINV pin. [see chapter 6]) some of the control signals that get latched in the local control-logic of the data paths continue to hold. This would be true for *Type 1-2* circuit (see section 5.5.1). Consequently, if a register's control signal to load parallel data is generated with a Type 1-2 circuit, the parallel input to the register would remain turned ON during serial scan. This of course would cause contention between the data coming into the register through the parallel input and the data coming through the serial input. We found the only register where such a situation occured is EAGATE in APU. The control signal OENEAGATE is generated with a Type 1-2 circuit as shown in figure 5.46 and hence would have the problem just described.

Our investigations also showed that data is not corrupted if the registers are scanned immediately after reset cycles. Since during reset, OENEAGATE signal is not asserted, a scan operation following reset provides good scan data. We used this fact for getting around the problem of data corruption. After a series of normal cycles, the processor is always reset for three cycles before shifting the scan data. The three cycles give enough time for the reset control-word to propagate into the

data path control logic. This of course destroys the control state of the processor; however, all the register data remain intact and can be read out through the serial scan path. We should point out, however, this problem with scan path does not affect the normal operation of the processor. Using this technique, we verified the APU registers are correctly read and written. Furthermore, the adder operation also takes place correctly. A few other tests were also done to check the data path operations. Thus, all the major functions of the processor have been verified to perform satisfactorily.

Chapter 8

Conclusions and Future Work

The architecture, circuits, cell library, and design methodology described in this dissertation have been successfully used to generate a customized, microprogrammed processor for a two-axis, adaptive robot controller. We have shown that a fairly complex algorithm with complicated control structure can be efficiently implemented on a custom processor from a behavioral description at the microprogram level provided a suitable, pre-defined architectural model is used. The chip has been tested and can work up to 15 MHz. Since approximately 600 cycles are required during each sample period for completing the computations required for two joints, at 15 MHz we can reach a sampling period of 0.04 ms. This is over ten times faster than the target sample period of 0.5 ms. Therefore a single custom chip, such as the one described in this dissertation, can potentially implement control algorithms for several joints.

In order to let system designers easily implement their algorithms on custom processors, support for high-level language input is very important. Current research work makes feasible extension of the user input to higher level languages such as 'C' or Silage. This clearly brings us closer to the exciting possibility of allowing system designers to generate custom IC solutions for their applications without requiring expertise in IC design. Our design approach, based on a predefined architecture model consisting of a generic control unit and application specific data paths with local memory, lends itself readily to generation of customized

processors from high-level descriptions of algorithms. At the same time, creating a hierarchical design with parameterized macrocells provides a high-level of flexibility and modularity. Thus, users willing to work at the structural level can easily modify the hardware to more closely match their requirements. This is evidenced from several projects currently underway that adapt the basic hardware design described in this thesis for such diverse applications as Inverse Kinematics solutions for robot arms [Lars87] and a Decision Feedback Equalizer for digital mobile phones [LaMLR88]. To a great extent users are able to create a new processor by simply redefining the top-level structure of the data paths through structural description files.

One difficulty created by allowing users to redefine data paths at the structural level is ensuring a legal configuration that will function properly. At the same time the user must not be burdened with the task of understanding the detailed circuit designs of the cells in order to properly interconnect them. To deal with this problem, we have defined some basic composition rules for hooking together data path cells. These, however, do not deal with any peculiarities of individual cells. Results from testing the chip (see chapter 7), for example, show that Type 1-2 control circuits should not be used for controlling the input of scan type registers or latches. Clearly, a more comprehensive set of rules is required, possibly integrated inside the CAD environment for automatic checking.

Among the various layout generation tools provided by the silicon assembly environment of LagerIII, TimLager (cell tiler) generates the most compact layout. The place and route tool, Flint, and Data Path Compiler, on the other hand, generate layouts that are much less area efficient. Improvements in this area would certainly be very welcomed by circuit designers. Further more, the layout system does not currently provide any feedback on the performance, nor is the layout generation process guided by any consideration for performance such as power, speed, and area. Current tools such as TimLager, however, do have the ability to support parameterization of the cells for selecting driver sizes and bus widths to match the load requirements. In future, transistor sizing programs [Shy88] may also be incorporated within Lager in order to simplify leafcell designer's task. Another

weakness in the CAD support is the lack of a good behavioral simulator. This problem is currently being addressed through the THOR simulator [Rob].

Whereas the design approach described in this thesis does allow rapid creation of designs, fabrication process still forces an eight to ten week delay. Moreover, after fabrication, there is only very limited flexibility for modifications and changes during testing and system development. This points to the need for providing other implementation techniques such as gate arrays and EPLD (erasable programmable logic) within the design environment for even faster prototyping.

Bibliography

[AlBaFi] J. S. Albus, A. J. Barbera, M. L. Fitzgerald, "Sensor Robotics in the National Bureau of Standards," *SPIE* vol. 360, pp. 14-21.

[Alt87] Altera Corporation, Santa Clara, California, Data Book, 1987.

[Anwr87] George Anwar, M. E. Dept., U. C. Berkeley, private communication.

[Astrm84] Karl J. Astrom and Bjorn Wittenmark, Computer Controlled Systems, pp. 180-187, New-Jersey: Prentice-Hall, Inc.

[AzShBr88] S. K. Azim, C-S Shung, R. W. Brodersen, "Automatic Generation of a Custom Digital Signal Processor for an Adaptive Robot Arm Controller," *ICASSP 88,* vol. 4, pp. 2021-2024, 1988.

[Azim88] S. K. Azim and R. W. Brodersen, "A Custom DSP Chip to Implement a Robot Motion Controller," *Proceedings of CICC,* Rochester, N.Y., May, 1988.

[Bar87] W. Baringer, et. al., "A VLSI Implementation of PPPE for Real-Time Image Processing in Radon Space – Work in Progress," *Proceedings, Workshop on Computer Architectures for pattern Analysis and Machine Intelligence,*, Seattle, Wa., Oct. 1987, pp. 88-93.

[Dav85] David Nitzan, "Development of Intelligent Robots: Achievements and Issues," *IEEE Journal of Robotics and Automation,* vol. RA-1, no. 1, pp. 3-13, Mar. 1985.

[DuDe79] S. Dubowsky and D. T. DesForges, "The Application of Model-Referenced Adaptive Control to Robotic Manipulators," *ASME Journal of Dynamic Systems, Measurements, and Control,* vol. 101, Sep. 1979, pp. 193-200.

[GaDon88] Daniel D. Gajski and Donald E. Thomas, "Introduction to Silicon Compilation," Silicon Compilation, ed. D. Gajski, pp. 1-48, Addison-Wesley, Inc.

[Gaj88] Daniel D. Gajski, Silicon Compilation, ed. D. Gajski, Addison-Wesley, Inc.

[Hil85] P. N. Hilfinger, "Silage: A Language for Signal Processing," *Proceedings of CICC,* May 1985.

[Hor86] R. Horowitz, M-C Tsai, G. Anwar, M. Tomizuka, "Model Reference Adaptive Control of a Two Axis Direct Drive Manipulator Arm," *Proceedings of the 1987 IEEE Int. Conf. on Robotics and Automation,* April 1987, pp. 1216-1222.

[HorTom82] R. Horowitz and M. Tomizuka, "Discrete Time Model Reference Adaptive Control of Mechanical Manipulators," *Computers in Engineering,* vol. 2, Robots and Robotics, ASME, pp. 107-112.

[ILC73] The Engineering Staff, Synchro Conversion Handbook, ILC Data Device Corp., Bohemia, New York.

[Ira84] Ira B. Cushing, "A New High Accuracy Angular Position Transducer," *Motor-Con Proceedings,* April 1984, pp. 283-289.

[Jain86] R. Jain, et. al., "Custom Design of VLSI PLM-FDM Transmultiplexer from System Specifications to Layout using a Cad System," *IEEE Journal of Solid State Circuits,* vol. SC21, no. 1, Feb. 1986, pp. 73-85.

[JaNoHa85] J. R. Jasica, S. E. Noujaim, R. Hartley, M. J. Hartman, " A Bit-Serial Silicon Compiler," *Proceedings of ICCAD,* Nov. 1985, pp. 91-93.

[Koem86] Henriecus Koeman, "Application-Specific IC Design Technologies," *IEEE Micro,* pp. 42-50, Feb. 1985.

[KoiGu81] A. J. Koivo and T. H. Guo, " Adaptive Linear Controller for Robotic Manipulators," *IEEE Trans. on Automatic Control,* vol. AC-28:2, 1983, pp. 162-170.

[Lars87] Lars Thon, "ASICS for Inverse Kinematics," *EECS/ERL Research Summary,* EECS Dept., U.C. Berkeley, 1988.

[LaMLR88] L. Svensson, M. Torkelson, L. Thon, R. Jain, "Implementation Aspects of a Decision Feedback Equalizer ASIC Using an Automatic Layout Generation System," *To be presented in ISCAS,* Finland, June 1988.

[LeeJa84] Johnny K. Lee and Alan Smith, "Branch Prediction Strategies and Branch Target Buffer Design," *IEEE Computer,* Jan. 1984, pp. 6-22.

[LiFrSi87] K-S. Lin, G. Franz, R. Simar, "The TMS320 Family of Digital Signal Processors," *Proceedings of the IEEE,* vol. 75, no. 9, Sep.1987, pp. 1143-1159.

[Ma87] Mani Srivastava, "Automatic Generation of CMOS Data Paths in LAGER Framework," EECS Dept., U.C. Berkeley, M. S. Report, May 1987.

[MdLki82] C. A. Mead and G. Lewicki, "Silicon Compilers and Foundries Will Usher in User-designed VLSI," *Electronics,* pp. 107-111, Aug. 11, 1982.

[Mo86] Motorola, Inc., DSP56000, Digital Signal Processor User's Manual, 1986.

[Nel83] N. Goncalves and H. De Man, "NORA: A Racefree Dynamic CMOS Technique for Pipelined Logic Structures," *IEEE Journal of Solid-State Circuits,* vol. SC-18, no. 3, June 1983, pp. 261-266.

[NSK86] Motornetics Corp., subsidiary of NSK, Elmhurst, Illinois, Megatorque Motor Driver Units, User's Manual, 1986.

[Paul81] Richard P. Paul, Robot Manipulators: Mathematics, Programming, and Control, pp. 158-164, Massachusetts:MIT Press.

[Pope85] Stephen Pope, "Automatic Generation of Signal Processor Integrated Circuits," Memo. no. UCB/ERL M85/11, Electronics Research Laboratory, U.C. Berkeley, Feb. 1985.

[Rab88] J. Rabaey, et. al. "CATHEDRAL-II: A Synthesis System for Multiprocessor DSP Systems," Silicon Compilation, ed. Daniel Gajski, pp. 311-360, Addison-Wesley, Inc.

[RaPoBr85] J. Rabaey, S. Pope, R. Brodersen, "An Integrated Automated Layout Generation System for DSP Circuits," *IEEE Trans. on CAD,* vol. CAD-4, July 1985, pp. 285-296.

[Reu86] P. A. Reutz, "Architectures and Design Techniques for Real-Time Image Processing ICs," Memo. no. UCB/ERL M86/37, Electronics Research Laboratory, U.C. Berkeley, May 1986.

[Rim88] Ken Rimey, "A Compiler for Horizontal-Instruction-Word Signal Processors," Comp. Sci. Div., U. C. Berkeley, Ph. D. Qualifying Proposal, April 1988.

[Rob] Robert Alverson, et. al., "THOR User's Manual: Tutorial and Commands," Technical Report CSL-TR-88-348, Stanford University, Jan. 1988.

[Rud86] R. Rudell, "ESPRESSO - boolean minimization program," *1986 VLSI Tools*, Report no. UCB/CSD 86/272, Comp. Sci. Div., EECS, U.C. Berkeley, Dec. 1985.

[Sil86] Silicon Compilers, Inc., Genesil Users Manual, 1986.

[Sim88] R. Simar in panel discussion, "The DSP VLSI Challenge and System Design Tools," *Digest of Technical Papers,* pp. 217, IEEE ISSCC, San Fransisco, Feb. 17-19, 1988.

[Sh87] C-S Shung, "LagerIII: A Framework for Algorithm-Specific IC Design," *IEEE Solid-State Circuits and Technology Committee fall workshop on Application Specific ICs for DSP,* Napa Valley, California, Sep 2-4, 1987.

[Shung88] C-S Shung, "Integrated CAD System for Algorithm-Specific IC Design," *Ph.D. Dissertation* in preparation, EECS Dept., University of California, Berkeley.

[ShJSB87] C-S Shung, R. Jain, M. Srivastava, R. Brodersen, "LagerIII User's Manual", EECS Dept., University of California, Berkeley, Internal Document, Aug. 1987.

[Shy88] J-M Shy, et. al. "Optimization-Based Transistor Sizing," *IEEE J. Solid-State Circuits*, April 1988.

[Sny85] Wesley E. Snyder, Industrial Robots: Computer Interfacing and Control, pp.206-207, New-Jersey:Prentice-Hall, Inc.

[Snyd85] Wesley
E. Snyder, Industrial Robots: Computer Interfacing and Control, pp. 24-33, New-Jersey:Prentice-Hall, Inc.

[Term] Chris Terman, "Esim - event driven switch level simulator," *1986 VLSI Tools*, Report no. UCB/CSD 86/272, Comp. Sci. Div., EECS, U.C. Berkeley, Dec. 1985.

[TMS83] Texas Instruments, TMS32010 User's Guide, 1983.

[Tom86] M. Tomizuka, R. Horowitz, G. Anwar, "Adaptive Techniques for Motion Control of Robotic Manipulators," *Japan-USA Joint Symp. on Flexible Automation,* July 14-16, 1986, Osaka, Japan.

[Tsai87] M-C Tsai, M. E. Dept., U. C. Berkeley, private communication.

[Vos86] L. De. Vos, R. Jain, H. De. Man, W. Ulbrich, "A Fast Adder-based Multiplication Unit for Customized Digital Signal Processors," *Proceedings of ICASSP,* Tokyo, 1986.

[WiPa83] T. Williams and K. Parker, "Design for Testability – a Survey," *Proceedings of the IEEE,* Jan. 1983, 98-112.

Appendix A

Specification for Assembly Language Input File

The assembly language file has six sections written in a lisp format. Each section is identified with a keyword followed by a list. In the description below bold characters identify a keyword. Expressions enclosed by [] may be optionally repeated any number of times. Expressions in italics must be replaced with appropriate name or number. The format of the assembly language file has been designed by C-S Shung.

Ram Variables:

(**ram** *element* [*elements*])

element := (*array-name n*) – for array variable;
n is the number of array elements.
or *variable-name* – for a single variable.

Logical Unit FSM:

(**dfsm** (*boolean-oper-name (state-name equation*))
[(*boolean-oper-name (state-name equation*))])

boolean-oper-name := name by which the operation
is invoked within a program block.
state-name := fsm's internal or output state.
equation := boolean expression of state and input variables.

Control Unit FSM:

(**cfsm** *(state-name block-number (conditions)* (goto *next-state-name*))
[*(state-name block-number (conditions)* (goto *next-state-name*))])

state-name := name identifying a control state.
block-number := integer identifying a code-block
associated with the state.
conditions := conditions under which state-transition
should occur. Blank means transition occurs
only at the end-of-block. lctestK means when
Kth count in loop count list (see below) is completed.
next-state-name := next-state to which transition takes place.

(Note: If a stack is used, an additional field for return-state)
must be specified

Loop Count:

(**loop_test** *number* [*number*])

number := integer specifying a loop count. The order
K in which the number appears corresponds to the
lctestK signal in cfsm specification above.

Data Path Word-Size:

(**dp_word_size** *N*)

N := integer specifying number of bit-slices in data path.

Timer Reset:

(**reset_timer** *state-name* [*state-name*])

state-name := state in which timer reset is asserted.

Sampling Period:

(**max_sample_intvl** *L*)

L := integer specifying length of sampling period
in terms of number of cycles.

Code-Blocks:

(**rom** (block*P* (*instruction*) [*instruction*])
[(block*Q* (*instruction*) [*instruction*])])

P, Q are integers and correspond to block-number
in cfsm section.
instruction := (*microoperation*) [(*microoperation*)]
microoperation := primitive data transfer operation

Appendix B

Parameters

The parameters listed below are used to parameterize the layout of the processor. They are read by the Design Manager and are passed on to the appropriate layout generator for creating a specific instance of the layout.

```
;
;*****************************************************************
Parameters specifying the type of pad at each pad location and the
name assigned to the pad terminals.
;*****************************************************************
(Enumber  20) ; total number of pads in the East pad-group is 20
(Epad0 (Vdd))
(Epad1 (GND))
(Epad2 (io DATAINPORT\[0\] DATAINPORT\[0\]* EN\[0\] DATAOUTPORT\[0\]))
(Epad3 (io DATAINPORT\[1\] DATAINPORT\[1\]* EN\[1\] DATAOUTPORT\[1\]))
(Epad4 (io DATAINPORT\[2\] DATAINPORT\[2\]* EN\[2\] DATAOUTPORT\[2\]))
(Epad5 (io DATAINPORT\[3\] DATAINPORT\[3\]* EN\[3\] DATAOUTPORT\[3\]))
(Epad6 (io DATAINPORT\[4\] DATAINPORT\[4\]* EN\[4\] DATAOUTPORT\[4\]))
(Epad7 (io DATAINPORT\[5\] DATAINPORT\[5\]* EN\[5\] DATAOUTPORT\[5\]))
(Epad8 (io DATAINPORT\[6\] DATAINPORT\[6\]* EN\[6\] DATAOUTPORT\[6\]))
(Epad9 (io DATAINPORT\[7\] DATAINPORT\[7\]* EN\[7\] DATAOUTPORT\[7\]))
(Epad10 (io DATAINPORT\[8\] DATAINPORT\[8\]* EN\[8\] DATAOUTPORT\[8\]))
(Epad11 (io DATAINPORT\[9\] DATAINPORT\[9\]* EN\[9\] DATAOUTPORT\[9\]))
(Epad12 (io DATAINPORT\[10\] DATAINPORT\[10\]* EN\[10\] DATAOUTPORT\[10\]))
(Epad13 (io DATAINPORT\[11\] DATAINPORT\[11\]* EN\[11\] DATAOUTPORT\[11\]))
(Epad14 (io DATAINPORT\[12\] DATAINPORT\[12\]* EN\[12\] DATAOUTPORT\[12\]))
(Epad15 (io DATAINPORT\[13\] DATAINPORT\[13\]* EN\[13\] DATAOUTPORT\[13\]))
(Epad16 (io DATAINPORT\[14\] DATAINPORT\[14\]* EN\[14\] DATAOUTPORT\[14\]))
(Epad17 (io DATAINPORT\[15\] DATAINPORT\[15\]* EN\[15\] DATAOUTPORT\[15\]))

(Snumber 16) ; total number of pads in the South pad-group is 16
```

```
(Spad0 (GND))
(Spad1 (Vdd))
(Spad2 (io DATAINPORT\[16\] DATAINPORT\[16\]* EN\[16\] DATAOUTPORT\[16\]))
(Spad3 (io DATAINPORT\[17\] DATAINPORT\[17\]* EN\[17\] DATAOUTPORT\[17\]))
(Spad4 (io DATAINPORT\[18\] DATAINPORT\[18\]* EN\[18\] DATAOUTPORT\[18\]))
(Spad5 (io DATAINPORT\[19\] DATAINPORT\[19\]* EN\[19\] DATAOUTPORT\[19\]))
(Spad6 (in TESTCLK1 TESTCLK1*))
(Spad7 (in TESTCLK2 TESTCLK2*))
(Spad8 (out EOS))
(Spad9 (in EXTCOND\[0\] EXTCOND\[0\]*))
(Spad10 (out SCANOUT))
(Spad11 (out PORTADDRESS\[3\]))
(Spad12 (out PORTADDRESS\[2\]))
(Spad13 (out PORTADDRESS\[1\]))
(Spad14 (GND))
(Spad15 (Vdd))

(Wnumber 20) ; total number of pads in the West pad-group is 20
(Wpad0 (out WRITESTRB))
(Wpad1 (out PORTADDRESS\[0\]))
(Wpad2 (in RESET RESET*))
(Wpad3 (out READSTRB))
(Wpad4 (in SCANIN SCANIN*))
(Wpad5 (out BPFLAG))
(Wpad6 (in TESTMODEINV TESTMODEINV*))
(Wpad7 (in CLOCKIN CLOCKIN*))
(Wpad8 (in MCC MCC*))
(Wpad9 (Vdd))
(Wpad10 (GND))
(Wpad11 (space))
(Wpad12 (space))
(Wpad13 (space))
(Wpad14 (space))
(Wpad15 (space))
(Wpad16 (Vdd))
(Wpad17 (GND))
(Wpad18 (space))
(Wpad19 (space))

(Nnumber 16) ; total number of pads in the North pad-group is 16
(Npad0 (space))
(Npad1 (space))
```

```
(Npad2 (space))
(Npad3 (space))
(Npad4 (space))
(Npad5 (space))
(Npad6 (space))
(Npad7 (space))
(Npad8 (space))
(Npad9 (space))
(Npad10 (space))
(Npad11 (space))
(Npad12 (space))
(Npad13 (space))
(Npad14 (space))
(Npad15 (space))
;
;
;*****************************************************************
;The following parameters give specifications for the macrocells
;*****************************************************************
```

AU1wordsize: arithmetic unit word size
num_of_blks: number of code-blocks
max_blk_size: size of the largest code-block
num_of_cstore_mint: number of minterms in the control store (rom)
num_of_cstoreout: number of bits in the cstore output
num_of_states: number of states in the control unit fsm
num_of_cfsm_mint: number of minterms in the control unit fsm
stack_depth: depth of the stack in the control unit
num_of_cond: number of external conditions going into the control unit
num_of_loopslice: number of loops with distinct loop counts
max_loop_size: size of the largest loop
detslice: list of loop-count values
max_sample_intvl: maximum sampling period in number of cycles
cstore-input-plane: input plane bit pattern for control store (rom)
cstore-output-plane: output plane bit pattern for control store
nr_of_lgu_feedback: number of feedback lines in the LGU fsm
nr_of_lgu_out: number of outputs in the LGU fsm
nr_of_lgu_xin: number of external inputs to the LGU fsm
nr_of_lgu_instr: number of boolean operations specified in the LGU fsm
nr_of_lgu_minterm: number of LGU fsm minterms
lgu-input-plane: input plane bit pattern for the LGU fsm

lgu-output-plane: output plane bit pattern for the LGU fsm
fsm-input-plane: control unit fsm input plane bit pattern
fsm-output-plane: control unit fsm output plane bit pattern
ramwords: number of words in the ram
ram-address-plane: ram address plane bit pattern
ram-bit-plane: ram data plane bit pattern; x means rd/wr

Appendix C

Microoperations of the Robot Control Processor

The microoperations listed below are the basic data path operations allowed in the processor described in this thesis. In the context of this processor, an assembly language program is composed of these primitve microoperations. The assembly language program is provided as an input to the *rassCG* program which generates the structural specifications required for producing the layout. In the following list of microoperations, keyword *defun* precedes each microoperation. Keyword *grab* specifies processor resources being used by the microoperation. Keyword *high* identifies all the control signals which are high in order to execute the associated microoperation. Keyword *low* identifies control signals which should be low in order to execute the associated microperation. Comments on the right hand side beginning with % describe the operation.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; AU (arithmetic unit)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; Register load instructions

(defun mor=mem () ;% read memory and load data into mor register.
  (grab mor mem)
  (high pR pLDMOR)
  (low pSELMORIN)) ; R: ram, LDMOR, LDMORINV, MORSELMEM

(defun mem=mbus () ;% write mbus data into memory.
  (grab mem)
  (high pW)) ; W: ram, WEN
```

```
(defun mcondload () ;% write mbus into memory if condition code cc is
true.
  (high pWC)) ; WC

(defun mor=mbus () ;% load mbus data into mor.
  (grab mor)
  (high pSELMORIN pLDMOR)) ; MORSELMBUS

(defun r*=rbus (n) ;% load rbus data into register n.
;% In robot control processor n = 1 or 2.
  (caseq n
 (0 (grab r0) (high pLDR0)) ; LDR0, LDR0INV
 (1 (grab r1) (high pLDR1)) ; LDR1, LDR1INV
 (2 (grab r2) (high pLDR2)) ; LDR2, LDR2INV
 (3 (grab r3) (high pLDR3)) ; LDR3, LDR3INV
 (4 (grab r4) (high pLDR4)))) ; LDR4, LDR4INV

(defun rcoef=mbus () ;% load coefficient register from mbus.
  (grab rcoef)
  (high pLDCOEF)) ; LDCOEF, LDCOEFINV

;;; Move (into a bus) instructions

(defun mbus=mor () ;% move mor register data into mbus.
  (grab mbus)
  (high pXMITMOR)) ; XMITMOR, XMITMORINV

(defun mbus=r* (n) ;% move data from register n into mbus.
   (caseq n
      (0 (grab mbus) (high pOENR0) (low pXMITMOR)) ; ONER0 ONER0INV
      (1 (grab mbus) (high pOENR1) (low pXMITMOR)) ; ONER1 ONER1INV
      (2 (grab mbus) (high pOENR2) (low pXMITMOR)) ; ONER2 ONER2INV
      (3 (grab mbus) (high pOENR3) (low pXMITMOR)) ; ONER3 ONER3INV
      (4 (grab mbus) (high pOENR4) (low pXMITMOR)))) ; ONER4 ONER4INV

(defun mbus=acc () ;% move accumulator data into mbus.
  (grab mbus)
  (high pXMITACC)
  (low pXMITMOR)) ; XMITACC XMITACCINV

(defun rbus=acc () ;% move accumulator data into rbus.
  (grab rbus)
```

```
  (high pACC2REG)) ; ACC2REG ACC2REGINV

(defun rbus=ioport () ;% move data from I/O port into rbus.
  (grab rbus)
  (high pRDPORT))  ; RDPORT, RDPORTINV

(defun ioport=extport (n) ;% read  I/O port where n specifies the port
;% address. Robot control processor has a
;% 4 bit port address.
; RDSTRB
  (grab ioport extport)
  (high pRDSTRB) (low pWRPORT) ; ioport!=mbus
  (caseq n
 ; PORT ADDRESS = 0000,0001,0010, and so on
 (0 (low pPORTADDRESS3 pPORTADDRESS2 pPORTADDRESS1 pPORTADDRESS0))
 (1 (low pPORTADDRESS3 pPORTADDRESS2 pPORTADDRESS1) ; 0001
    (high pPORTADDRESS0))
 (2 (low pPORTADDRESS3 pPORTADDRESS2 pPORTADDRESS0) ; 0010
    (high pPORTADDRESS1))
 (3 (low pPORTADDRESS3 pPORTADDRESS3 pPORTADDRESS2) ; 0011
    (high pPORTADDRESS1 pPORTADDRESS0))
 (4 (high pPORTADDRESS2) ; 0100
    (low pPORTADDRESS3 pPORTADDRESS1 pPORTADDRESS0))
 (5 (high pPORTADDRESS2 pPORTADDRESS0) ; 0101
    (low pPORTADDRESS3 pPORTADDRESS1))
 (6 (high pPORTADDRESS2 pPORTADDRESS1) ; 0110
    (low pPORTADDRESS3 pPORTADDRESS0))
 (7 (high pPORTADDRESS2 pPORTADDRESS1 pPORTADDRESS0) ; 0111
    (low pPORTADDRESS3))
 (8 (low pPORTADDRESS0 pPORTADDRESS1 pPORTADDRESS2  ) ; 1000
    (high pPORTADDRESS3 ))
 (9 (low pPORTADDRESS1 pPORTADDRESS2 ) ; 1001
    (high pPORTADDRESS3 pPORTADDRESS0 ))
 (10 (low pPORTADDRESS0 pPORTADDRESS2 ) ; 1010
     (high pPORTADDRESS3 pPORTADDRESS1 ))
 (11 (low pPORTADDRESS2) ; 1011
     (high pPORTADDRESS3 pPORTADDRESS1 pPORTADDRESS0))
 (12 (low pPORTADDRESS0 pPORTADDRESS1) ; 1100
     (high pPORTADDRESS3 pPORTADDRESS2))
 (13 (low pPORTADDRESS1) ; 1101
     (high pPORTADDRESS3 pPORTADDRESS2 pPORTADDRESS0 ))
 (14 (low pPORTADDRESS0) ; 1110
```

```
    (high pPORTADDRESS3 pPORTADDRESS2 pPORTADDRESS1 ))
 (15 (high pPORTADDRESS3 pPORTADDRESS2 pPORTADDRESS1 pPORTADDRESS0 ))))

(defun ioport=mbus () ;% mbus data send to the I/O port.
  (grab ioport)
  (high pWRPORT)) ; WRPORT, WRPORTINV

(defun extport=ioport (n) ;% write data to external port where n
;% specifies the port address (4 bits).
  (grab extport)
  (high pWRSTRB) ; WRSTRB
  (caseq n
 ; PORT ADDRESS = 0000,0001,0010, and so on
 (0 (low pPORTADDRESS3 pPORTADDRESS2 pPORTADDRESS1 pPORTADDRESS0))
 (1 (low pPORTADDRESS3 pPORTADDRESS2 pPORTADDRESS1) ; 0001
    (high pPORTADDRESS0))
 (2 (low pPORTADDRESS3 pPORTADDRESS2 pPORTADDRESS0) ; 0010
    (high pPORTADDRESS1))
 (3 (low pPORTADDRESS3 pPORTADDRESS3 pPORTADDRESS2) ; 0011
    (high pPORTADDRESS1 pPORTADDRESS0))
 (4 (high pPORTADDRESS2) ; 0100
    (low pPORTADDRESS3 pPORTADDRESS1 pPORTADDRESS0))
 (5 (high pPORTADDRESS2 pPORTADDRESS0) ; 0101
    (low pPORTADDRESS3 pPORTADDRESS1))
 (6 (high pPORTADDRESS2 pPORTADDRESS1) ; 0110
    (low pPORTADDRESS3 pPORTADDRESS0))
 (7 (high pPORTADDRESS2 pPORTADDRESS1 pPORTADDRESS0) ; 0111
    (low pPORTADDRESS3))
 (8 (low pPORTADDRESS0 pPORTADDRESS1 pPORTADDRESS2  ) ; 1000
    (high pPORTADDRESS3 ))
 (9 (low pPORTADDRESS1 pPORTADDRESS2 ) ; 1001
    (high pPORTADDRESS3 pPORTADDRESS0 ))
 (10 (low pPORTADDRESS0 pPORTADDRESS2 ) ; 1010
     (high pPORTADDRESS3 pPORTADDRESS1 ))
 (11 (low pPORTADDRESS2) ; 1011
     (high pPORTADDRESS3 pPORTADDRESS1 pPORTADDRESS0))
 (12 (low pPORTADDRESS0 pPORTADDRESS1) ; 1100
     (high pPORTADDRESS3 pPORTADDRESS2))
 (13 (low pPORTADDRESS1) ; 1101
     (high pPORTADDRESS3 pPORTADDRESS2 pPORTADDRESS0 ))
 (14 (low pPORTADDRESS0) ; 1110
     (high pPORTADDRESS3 pPORTADDRESS2 pPORTADDRESS1 ))
```

```
 (15 (high pPORTADDRESS3 pPORTADDRESS2 pPORTADDRESS1 pPORTADDRESS0 ))))

(defun acc=0 () ;% set abus=bbus=0 resulting in acc=0.
  (grab abus bbus acc)
  (high pNOABS pZERO_BIN) ; ZERO_BIN, ZERO_AIN
  (low pANDCOEF pMINUS pCOEFCOMP))

(defun acc=sum () ;% load adder output into accumulator.
  (grab acc))

(defun acc=abus () ;% load abus into accumulator by setting
;% bbus=0
  (grab acc bbus)
  (high pZERO_BIN)) ; ZERO_BIN

(defun acc=bbus () ;% load bbus into accumulator by seting
;% abus=0
  (grab acc abus)
  (high pNOABS) ; ZERO_AIN
  (low pMINUS pANDCOEF pCOEFCOMP))

(defun abus=1 () ;% increment bbus,
;% actually abus=0 and cin=1
  (grab abus)
  (high pNOABS pMINUS)
  (low pANDCOEF pCOEFCOMP)) ; COMPLA, COMPLAINV

(defun abus=mor () ;% move mor data into abus.
  (grab abus)
  (high pNOABS pCOEFCOMP)
  (low pMINUS pANDCOEF))

(defun abus=-mor () ;% move negative mor into abus.
  (grab abus)
  (high pNOABS pMINUS pCOEFCOMP)
  (low pANDCOEF))

(defun abus=absmor () ;% move absolute mor into abus.
  (grab abus)
  (high pCOEFCOMP)
  (low pNOABS pMINUS pANDCOEF))
```

```
(defun abus=-absmor () ;% move negative of absolute mor into
;% abus.
  (grab abus)
  (low pNOABS pANDCOEF)
  (high pMINUS pCOEFCOMP))

(defun abus=coef.mor () ;% move product of mor and right most
;% bit of the coefficient register into
;% abus. Used for shift/add
;% multiplication.
  (grab abus)
  (high pNOABS pANDCOEF)
  (low pMINUS pCOEFCOMP))

(defun abus=coef.-mor () ;% same as above except use -mor.
  (grab abus)
  (high pNOABS pMINUS pANDCOEF)
  (low pCOEFCOMP))

(defun abus=coef.absmor () ;% same as above except use absol. mor.
  (grab abus)
  (high pANDCOEF)
  (low pNOABS pMINUS pCOEFCOMP))

(defun abus=coef.-absmor () ;% same as above except use -absol. mor.
  (grab abus)
  (high pMINUS pANDCOEF)
  (low pNOABS pCOEFCOMP))

(defun abus=~coef.mor ()        ;% same as above except use compl. coeff.
  (grab abus)
  (high pANDCOEF pCOEFCOMP pNOABS)
  (low pMINUS))

(defun abus=~coef.-mor () ;% same as above except use compl.
;% coeff. and negative mor.
  (grab abus)
  (high pANDCOEF pCOEFCOMP pNOABS pMINUS))

(defun abus=~coef.absmor () ;% same as above except use compl.
;% coeff. and absol. mor.
  (grab abus)
```

```
  (high pANDCOEF pCOEFCOMP)
  (low pNOABS pMINUS))

(defun abus=~coef.-absmor () ;% same as above except use compl.
;% coeff. and neg. absol. mor.
  (grab abus)
  (high pANDCOEF pCOEFCOMP pMINUS)
  (low pNOABS))

(defun bbus=mbus () ;% move mbus data to bbus.
  (grab bbus)
  (low pZERO_BIN)
  (low pSELBBUSIN))

(defun bbus=acc>* (n) ;% shift accumulator right n bits
;% and move data to bbus.
   (grab bbus) ;% n is 0 to 6
   (low pZERO_BIN)
   (high pSELBBUSIN)
   (caseq n
      (0 (low pS2 pS1) (high pS0))
      (1 (low pS2 pS0) (high pS1))
      (2 (low pS2) (high pS1 pS0))
      (3 (high pS2) (low pS1 pS0))
      (4 (high pS2 pS0) (low pS1))
      (5 (high pS2 pS1) (low pS0))
      (6 (high pS2 pS1 pS0))
      (t (format t "Illegal instruction: right-shift (~a) out of bound~%"
n))))

(defun bbus=acc<* (n) ;% shift accumulator left n bits
;% n can be  only 1.
   (grab bbus)
   (high pSELBBUSIN)
   (low pZERO_BIN)
   (caseq n
      (1 (low pS2 pS1 pS0))
      (t (format t "Illegal instruction: left-shift (~a) out of bound~%"
n))))

(defun acondload () ;% load accumulator if condition code
;% cc is true.
```

```
  (high pSUMCOND))

(defun shrcoef () ;% serially shift coefficient register
;% one bit to the right. Used for
;% shift/add multiplication.
  (high pSHIFTCOEF))

;;; Miscellaneous "instructions"

(defun nosat () (high pNOSAT)) ;% do not saturate adder output.
(defun aip   () (high pAIP))       ;% accumulate if adder output is positive.

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; APU (address processing unit)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; Regsiter load instructions

(defun x*=eabus (n) ;% load eabus data into apu register n,
;% where n is 0 to 2.
   (caseq n
      (0 (grab x0) (high pLOADX0)) ; LOADX0, LOADX0INV
      (1 (grab x1) (high pLOADX1)) ; LOADX1, LOADX1INV
      (2 (grab x2) (high pLOADX2)) ; LOADX2, LOADX2INV
      (3 (grab x3) (high pLOADX3)) ; LOADX3, LOADX3INV
      (4 (grab x4) (high pLOADX4)))) ; LOADX4, LOADX4INV

(defun xcondload () ;% load apu register only if condition
;% code is true. Used together with
;% the previous instruction.
  (high pCONDLD))

;;;Move (into a bus) instructions

(defun addr fexpr (l) ;% the argument of addr symbolically
;% identifies a ram variable.
   (grab dbus)
   (ramdecodebase (car l)))

(defun offset (l) ;% the argument of offset is added
;% to the address constant during
;% compile time.
```

```
   (ramdecodeoffset 1))

(defun xip () ;% load if apu adder output is pos.
  (high pXIP))

(defun xbus=x* (n) ;% move data from apu register n to
;% xbus, where n is 0 to 2.
   (caseq n
      (0 (grab xbus) (low pXBUSZERO) (high pOENX0)) ; OENX0, OENX0INV
      (1 (grab xbus) (low pXBUSZERO) (high pOENX1)) ; OENX1, OENX1INV
      (2 (grab xbus) (low pXBUSZERO) (high pOENX2)) ; OENX2, OENX2INV
      (3 (grab xbus) (low pXBUSZERO) (high pOENX3)) ; OENX3, OENX3INV
      (4 (grab xbus) (low pXBUSZERO) (high pOENX4)))) ; OENX4, OENX4INV

(defun xbus=0 () ;% set xbus to zero.
  (grab xbus)
  (high pXBUSZERO)) ; XBUSZERO

(defun eabus=sum ()  ;% dump apu's adder output on eabus.
  (grab eabus)
  (high pOENEALATCH)) ; OENEALATCH, OENEALATCHINV

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Communication between AU and APU
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun eabus=mbus () ;% move mbus data to eabus.
    (grab eabus)
    (high pMBUS2EABUS))

(defun areg=eabus () ;% load areg from eabus.
    (grab areg)
    (high pEABUS2MBUS))

(defun mbus=areg () ;% move areg data to mbus,
;% this must always  be asserted
    (grab mbus) ;% following areg=eabus
  (low pXMITMOR))

(defun timereg=eabus () ;% load timer register in PCU from eabus.
    (grab timerinreg)
    (high pLDTIMER)) ; LDTIMER, LDTIMERINV
```

```
;;; No operation (nop). Handles all the defaults.

(defun nop ()
   (high pXBUSZERO) (ramdecodebase 0) ; xbus=0, dbus=0
   (high pSELMORIN) ; mor=mbus
   (high pXMITMOR) ; mbus=mor
   (high pNOABS) (low pMINUS pANDCOEF pCOEFCOMP) ; abus=0
   (high pZERO_BIN) ; bbus=0
   (low pAIP pSUMCOND) ; acc=sum
   (low pS2 pS1) (high pS0) ; shifterout=acc>0
   (high pWRPORT)) ; ioport=mbus

(defun fsm (n) n) ;% define boolean expressions for
;% LGU.

(defun immed (n) (grab dbus) (ramdecodeoffset n)) ;% specify n as the
address
;% data in the control-word.
;% This is sent to the dbus
;% input of the apu's adder.
```

Appendix D

Bit Assignments of the Control Signals in the Rom

In the list below, the name identifies the control signal and the number following it refers to the bit position in the control-word.

```
;;
;; Control signals used internal to the PCU
;;
EOB 0
pLDTIMER 1
;;; if stack is used then need EOB2
;;; EOB2 2

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Arithmetic Unit microcode bits
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

pWC 2
pAIP 3
pNOSAT 4
pLDMOR 5
pLDR0 6
pLDR1 7
pLDCOEF 8
pXMITMOR 9
pXMITACC 10
pSELMORIN 11
pANDCOEF 12
pCOEFCOMP 13
pNOABS 14
```

```
pMINUS 15
pSELBBUSIN 16
pZERO_BIN 17
pS0 18
pS1 19
pS2 20
pOENR0 21
pOENR1 22
pACC2REG 23
pSUMCOND 24
pSHIFTCOEF 25
pRDPORT 26
pWRPORT 27

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Address Processing Unit microcode bits
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

pLOADX0 28
pLOADX1 29
pLOADX2 30
pCONDLD 31
pOENX0 32
pOENX1 33
pOENX2 34
pXBUSZERO 35
pEABUS2MBUS 36
pMBUS2EABUS 37
pOENEALATCH 38
pXIP 39

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; I/O related microcode bits
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

pRDSTRB 40
pWRSTRB 41
pPORTADDRESS0 42
pPORTADDRESS1 43
pPORTADDRESS2 44
pPORTADDRESS3 45
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; RAM control microcode bits
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

pR 46
pW 47

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; "Other fields "
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

LGUINSTR 48-50 ; Addresses the desired boolean function in the LGU's fsm
ADDRFIELD 51-56 ; Input to the APU for computing ram address
```

Appendix E

Structural Description Files

This appendix contains the top-level structural description (sdl) files of the robot control processor. These files describe the structure of a cell, called *parent-cell*, in terms of types of sub-cells used, list of parameters whose values form the structural specifications for the layout, and the net list. Each net is a single electrical node and lists all the terminals belonging to the net. Each terminal appears as a pair with its cell-name, and enclosed in parenthesis. A cell-name (which always appears first) given as *parent*, signifies that the terminal is on the boundary of the parent-cell. Detailed specification of the sdl files are given in Lager manuals.

The sdl files start with the root sdl file, proc_chip.sdl, and progressively move down the chip hierarchy, which is described in chapter 5. Only the top two-levels of the sdl files are given. The remaining files are available in LagerIII directory, lager/LagerIII/processor/sdl

```
;*********************************************************************
; sdl file for assembling the entire chip from the core processor
; and the pads.
;*********************************************************************
(layout-generator Padroute)
(parent-cell proc_chip
  (parameters
AU1wordsize
side_RAMIO
side_EABUSIN
side_EABUSOUT
side_IOPORT
side_PORTOUT
side_RBUS
side_addrsfield
side_EABUS
```

```
num_of_blks
max_blk_size
num_of_cstore_mint
num_of_cstoreout
num_of_states
num_of_cfsm_mint
stack_depth       ; set to zero - no stack
num_of_cond       ; excluding eob and loop det
num_of_loopslice
max_loop_size
detslice
max_sample_intvl  ; in terms of num of cycles
fsm-input-plane
fsm-output-plane
cstore-input-plane
cstore-output-plane
ramwords
ram-address-plane
ram-bit-plane
nr_of_lgu_feedback
nr_of_lgu_out ; including feedbacks
nr_of_lgu_xin ; external inputs to lgu excl signs.
nr_of_lgu_instr ; num of instr addr bit from cstore.
nr_of_lgu_minterm
lgu-input-plane
lgu-output-plane

  Epad0 Epad1 Epad2 Epad3  Epad4  Epad5  Epad6
  Epad7  Epad8  Epad9 Epad10 Epad11 Epad12 Epad13
  Epad14 Epad15 Epad16 Epad17 Epad18 Epad19 Enumber
  Wpad0 Wpad1 Wpad2 Wpad3  Wpad4  Wpad5  Wpad6
  Wpad7  Wpad8  Wpad9 Wpad10 Wpad11 Wpad12 Wpad13
  Wpad14 Wpad15 Wpad16 Wpad17 Wpad18 Wpad19 Wnumber
  Spad0 Spad1 Spad2 Spad3  Spad4 Spad5
  Spad6 Spad7  Spad8  Spad9 Spad10 Spad11
  Spad12 Spad13 Spad14 Spad15 Snumber
  Npad0 Npad1 Npad2 Npad3  Npad4 Npad5
  Npad6 Npad7  Npad8  Npad9 Npad10 Npad11
  Npad12 Npad13 Npad14 Npad15 Nnumber)
)
(sub-cells
  (proc LGR3
```

```
      (parameters (fplan 'middle)
    (AU1wordsize AU1wordsize)
    (side_RAMIO side_RAMIO)
    (side_EABUSIN side_EABUSIN)
    (side_EABUSOUT side_EABUSOUT)
    (side_IOPORT side_IOPORT)
    (side_PORTOUT side_PORTOUT)
    (side_RBUS side_RBUS)
    (side_addrsfield side_addrsfield)
    (side_EABUS side_EABUS)
    (num_of_blks num_of_blks)
    (max_blk_size max_blk_size)
    (num_of_cstore_mint num_of_cstore_mint)
    (num_of_cstoreout num_of_cstoreout)
    (num_of_states num_of_states)
    (num_of_cfsm_mint num_of_cfsm_mint)
    (stack_depth stack_depth)
    (num_of_cond num_of_cond)
    (num_of_loopslice num_of_loopslice)
    (max_loop_size max_loop_size)
    (detslice detslice)
    (max_sample_intvl max_sample_intvl)
    (fsm-input-plane fsm-input-plane)
    (fsm-output-plane fsm-output-plane)
    (cstore-input-plane cstore-input-plane)
    (cstore-output-plane cstore-output-plane)
    (ramwords ramwords)
    (ram-address-plane ram-address-plane)
    (ram-bit-plane ram-bit-plane)
    (nr_of_lgu_feedback nr_of_lgu_feedback)
    (nr_of_lgu_out nr_of_lgu_out)
    (nr_of_lgu_xin nr_of_lgu_xin)
    (nr_of_lgu_instr nr_of_lgu_instr)
    (nr_of_lgu_minterm nr_of_lgu_minterm)
    (lgu-input-plane lgu-input-plane)
    (lgu-output-plane lgu-output-plane)
    ))

  (scpads1_25 n_group
(parameters  (fplan 'top) (number Nnumber)
     (pad0 Npad0) (pad1 Npad1) (pad2 Npad2) (pad3 Npad3)
     (pad4 Npad4) (pad5 Npad5) (pad6 Npad6) (pad7 Npad7)
```

```
      (pad8 Npad8) (pad9 Npad9) (pad10 Npad10) (pad11 Npad11)
      (pad12 Npad12) (pad13 Npad13) (pad14 Npad14) (pad15 Npad15)
      ))

  (scpads1_25 e_group
(parameters  (fplan 'right) (number Enumber)
      (pad0 Epad0) (pad1 Epad1) (pad2 Epad2) (pad3 Epad3)
      (pad4 Epad4) (pad5 Epad5) (pad6 Epad6) (pad7 Epad7)
      (pad8 Epad8) (pad9 Epad9) (pad10 Epad10) (pad11 Epad11)
      (pad12 Epad12) (pad13 Epad13) (pad14 Epad14)
      (pad15 Epad15) (pad16 Epad16) (pad17 Epad17)
      (pad18 Epad18) (pad19 Epad19)
      ))

  (scpads1_25 w_group
(parameters  (fplan 'left) (number Wnumber)
      (pad0 Wpad0) (pad1 Wpad1) (pad2 Wpad2) (pad3 Wpad3)
      (pad4 Wpad4) (pad5 Wpad5) (pad6 Wpad6) (pad7 Wpad7)
      (pad8 Wpad8) (pad9 Wpad9) (pad10 Wpad10) (pad11 Wpad11)
      (pad12 Wpad12) (pad13 Wpad13) (pad14 Wpad14)
      (pad15 Wpad15) (pad16 Wpad16) (pad17 Wpad17)
      (pad18 Wpad18) (pad19 Wpad19)
      ))

  (scpads1_25 s_group
(parameters  (fplan 'bottom) (number Snumber)
      (pad0 Spad0) (pad1 Spad1) (pad2 Spad2) (pad3 Spad3)
      (pad4 Spad4) (pad5 Spad5) (pad6 Spad6) (pad7 Spad7)
      (pad8 Spad8) (pad9 Spad9) (pad10 Spad10) (pad11 Spad11)
      (pad12 Spad12) (pad13 Spad13) (pad14 Spad14) (pad15 Spad15)
      ))
)

(net datainporte 16 ((LGR3 DATAINPORT 0) (e_group DATAINPORT 0)))
(net dataoutporte 16 ((LGR3 DATAOUTPORT 0) (e_group DATAOUTPORT 0)))
(net datainports 4 ((LGR3 DATAINPORT 16) (s_group DATAINPORT 16)))
(net dataoutports 4 ((LGR3 DATAOUTPORT 16) (s_group DATAOUTPORT 16)))
(net wrport ((LGR3 WRPORT)
(s_group EN 16) (e_group EN 0)
(s_group EN 17) (e_group EN 1)
(s_group EN 18) (e_group EN 2)
(s_group EN 19) (e_group EN 3)
```

```
(e_group EN 4)
(e_group EN 5)
(e_group EN 6)
(e_group EN 7)
       (e_group EN 8)
       (e_group EN 9)
       (e_group EN 10)
       (e_group EN 11)
       (e_group EN 12)
       (e_group EN 13)
       (e_group EN 14)
       (e_group EN 15)))

(net rdstrb ((LGR3 READSTRB) (w_group READSTRB)))
(net writestrb ((LGR3 WRITESTRB) (w_group WRITESTRB)))
(net portaddress_w 1 ((LGR3 PORTADDRESS) (w_group PORTADDRESS)))
(net portaddress_s 3 ((LGR3 PORTADDRESS 1) (s_group PORTADDRESS 1)))
(net mcc ((LGR3 MCC) (w_group MCC)))
(net bpflag ((LGR3 BPFLAG) (w_group BPFLAG)))
(net clockin ((LGR3 CLOCKIN) (w_group CLOCKIN)))
(net eos ((LGR3 EOS) (s_group EOS)))
(net reset ((LGR3 RESET) (w_group RESET)))
(net testmodeinv ((LGR3 TESTMODEINV) (w_group TESTMODEINV)))
(net scanin ((LGR3 SCANIN) (w_group SCANIN)))
(net scanout ((LGR3 SCANOUT) (s_group SCANOUT)))
(net testclk1 ((LGR3 TESTCLK1) (s_group TESTCLK1)))
(net testclk2 ((LGR3 TESTCLK2) (s_group TESTCLK2)))
(net extcond 1 ((LGR3 EXTCOND) (s_group EXTCOND)))
;
;*************************************************************************
; sdl file for assembling the core processor
;*************************************************************************
(layout-generator Flint)
(parent-cell procH (parameters
   ; dummy parm for pad route
   fplan

   ; AU
   AU1wordsize
   side_RAMIO
   side_EABUSIN
   side_EABUSOUT
```

```
    side_IOPORT
    side_PORTOUT
    side_RBUS

    ; APU
    side_addrsfield
    side_EABUS

    ; PCU
    num_of_blks
    max_blk_size
    num_of_cstore_mint
    num_of_cstoreout
    num_of_states
    num_of_cfsm_mint
    stack_depth              ; set to zero - no stack
    num_of_cond              ; excluding eob and loop det
          ; ext cond & apusign, ausign
    num_of_loopslice
    max_loop_size
    detslice
    max_sample_intvl         ; in terms of num of cycles
    fsm-input-plane
    fsm-output-plane
    cstore-input-plane
    cstore-output-plane

    ; RAM
    ramwords
    ram-address-plane
    ram-bit-plane

    ; LGU
    nr_of_lgu_feedback
    nr_of_lgu_out            ; including feedbacks
    nr_of_lgu_xin    ; external inputs to lgu excl signs.
    nr_of_lgu_instr ; num of instr addr bit from cstore.
    nr_of_lgu_minterm
    lgu-input-plane
    lgu-output-plane))

(sub-cells
```

```
  (pcu-au DP
(parameters
 (ramwords  ramwords)
 (nr_of_lgu_instr  nr_of_lgu_instr)
 (AU1wordsize            AU1wordsize)
 (side_RAMIO             side_RAMIO)
 (side_EABUSIN           side_EABUSIN)
 (side_EABUSOUT          side_EABUSOUT)
 (side_IOPORT            side_IOPORT)
 (side_PORTOUT           side_PORTOUT)
 (side_RBUS              side_RBUS)
 (num_of_blks            num_of_blks)
 (max_blk_size           max_blk_size)
 (num_of_cstore_mint     num_of_cstore_mint)
 (num_of_cstoreout       num_of_cstoreout)
 (num_of_states          num_of_states)
 (num_of_cfsm_mint       num_of_cfsm_mint)
 (stack_depth            stack_depth)
 (num_of_cond            num_of_cond)
 (num_of_loopslice       num_of_loopslice)
 (max_loop_size          max_loop_size)
 (detslice               detslice)
 (max_sample_intvl       max_sample_intvl)
 (fsm-input-plane        fsm-input-plane)
 (fsm-output-plane       fsm-output-plane)
 (cstore-input-plane     cstore-input-plane)
 (cstore-output-plane    cstore-output-plane)))

  (ram-apu  LM
 (parameters
  (ramwords               ramwords)
  (num_of_states   num_of_states)
  (max_blk_size   max_blk_size)
  (side_addrsfield        side_addrsfield)
  (side_EABUS             side_EABUS)
  (AU1wordsize            AU1wordsize)
  (ram-address-plane      ram-address-plane)
  (ram-bit-plane          ram-bit-plane)
  (nr_of_lgu_feedback   nr_of_lgu_feedback)
  (nr_of_lgu_xin   nr_of_lgu_xin)
  (nr_of_lgu_instr   nr_of_lgu_instr)
  (nr_of_lgu_out          nr_of_lgu_out)
```

```
  (nr_of_lgu_minterm      nr_of_lgu_minterm)
  (lgu-input-plane        lgu-input-plane)
  (lgu-output-plane       lgu-output-plane))))


(net WRPORT ((parent WRPORT) (DP WRPORT)))
(net LOADX0 ((DP LOADX0) (LM LOADX0)))
(net LOADX1 ((DP LOADX1) (LM LOADX1)))
(net LOADX2 ((DP LOADX2) (LM LOADX2)))
(net CONDLD ((DP CONDLD) (LM CONDLD)))
(net OENX0 ((DP OENX0) (LM OENX0)))
(net OENX1 ((DP OENX1) (LM OENX1)))
(net OENX2 ((DP OENX2) (LM OENX2)))
(net XBUSZERO ((DP XBUSZERO) (LM XBUSZERO)))
(net OENEALATCH ((DP OENEALATCH) (LM OENEALATCH)))
(net XIP ((DP XIP) (LM XIP)))
(net RAMBUS AU1wordsize ((DP RAMBUS) (LM RAMBUS)))
(net RBUS AU1wordsize ((parent RBUS) (DP RBUS)))
(net RDSTRB ((parent READSTRB) (DP RDSTRB)))
(net WRSTRB ((parent WRITESTRB) (DP WRSTRB)))
(net PORTADDRS 4 ((parent PORTADDRESSS) (DP PORTADDRS)))
(net INPORT AU1wordsize ((parent DATAINPORT) (DP INPORT)))
(net OUTPORT AU1wordsize ((parent DATAOUTPORT) (DP OUTPORT)))
(net EABUS1 (- (integer-length (- ramwords 1)) 1) ((DP EABUS1) (LM EABUS1)))
(net EABUS3 1 ((LM EABUS3) (DP EABUS3)))
(net EOS ((parent EOS) (DP EOS)))
(net PGMSTATE (integer-length (- num_of_states 1)) ((LM PGMSTATE) (DP
PGMSTATE)))
(net LGUINPUTS nr_of_lgu_xin ((parent EXT_LGUIN) (LM LGUINPUTS)))
(net EXTCOND (- num_of_cond 2) ((parent EXTCOND) (DP EXTCOND)))
(net APUSIGN ((DP APUSIGN) (LM APUSIGN)))
(net AUSIGN ((DP AUSIGN) (LM AUSIGN) ))
(net LGUINSTRADDR nr_of_lgu_instr ((DP LGUINSTRADDR) (LM LGUINSTRADDR)))
(net CC ((DP CC)  (LM CC)))
(net RESET ((parent RESET) (DP RESET)))
(net INSTRADDRS (integer-length (- max_blk_size 1)) ((DP INSTRADDRS) (LM
INSTRADDRS)))
(net CSTOREADDRSFIELD (integer-length (- ramwords 1)) ((DP CSTOREADDRSFIELD)
(LM CSTOREADDRSFIELD)))
(net MCC ((parent MCC) (LM MCC)))
(net BPFLAG ((parent BPFLAG) (LM BPFLAG)))
(net MASTERCLOCK ((parent CLOCKIN) (LM MASTERCLOCK)))
```

```
(net TESTMODEINV ((parent TESTMODEINV) (LM TESTMODEINV)))
(net READ ((DP READ) (LM READ)))
(net WRITE ((DP WRITE) (LM WRITE)))
(net SCANIN ((parent SCANIN) (LM SCANIN)))
(net SCANLINK.1 ((LM SCANOUT) (DP SCANIN)))
(net SCANOUT ((DP SCANOUT) (parent SCANOUT)))
(net PHI1 ((LM PHI1) (DP PHI1)))
(net PHI2 ((LM PHI2) (DP PHI2)))
(net TESTCLK1 ((parent TESTCLK1) (DP TESTCLK1) (LM TESTCLK1)))
(net TESTCLK2 ((parent TESTCLK2) (DP TESTCLK2) (LM TESTCLK2)))
;
;
;*****************************************************************************
; sdl file for generating macrocell pcu-au
;*****************************************************************************
;
(layout-generator Flint)
(parent-cell  pcu-au
   (parameters num_of_blks
max_blk_size
num_of_cstore_mint
num_of_cstoreout
num_of_states
num_of_cfsm_mint
stack_depth              ; set to zero - no stack
num_of_cond              ; excluding eob and loop det
  ; ext cond & apusign, ausign
num_of_loopslice
max_loop_size
detslice
max_sample_intvl         ; in terms of num of cycles
fsm-input-plane
fsm-output-plane
cstore-input-plane
cstore-output-plane
nr_of_lgu_instr
ramwords
AU1wordsize
side_RAMIO
side_EABUSIN
side_EABUSOUT
side_IOPORT
```

```
side_PORTOUT
side_RBUS))

(sub-cells
  (pcuH PCU1
    (parameters  (num_of_blks            num_of_blks)
(max_blk_size           max_blk_size)
(num_of_cstore_mint     num_of_cstore_mint)
(num_of_cstoreout       num_of_cstoreout)
(num_of_states          num_of_states)
(num_of_cfsm_mint       num_of_cfsm_mint)
(stack_depth            stack_depth)
(num_of_cond            num_of_cond)
(num_of_loopslice       num_of_loopslice)
(max_loop_size          max_loop_size)
(detslice               detslice)
(max_sample_intvl       max_sample_intvl)
(fsm-input-plane        fsm-input-plane)
(fsm-output-plane       fsm-output-plane)
(cstore-input-plane     cstore-input-plane)
(cstore-output-plane    cstore-output-plane)))

  (au AU1
     (parameters  (N                        AU1wordsize)
(side_RAMIO             side_RAMIO)
(side_EABUSIN           side_EABUSIN)
(side_EABUSOUT          side_EABUSOUT)
(side_IOPORT            side_IOPORT)
(side_PORTOUT           side_PORTOUT)
(side_RBUS              side_RBUS))))


(net WC ((PCU1 CTLWORD 2) (AU1 pWC)))
(net AIP ((PCU1 CTLWORD 3) (AU1 pAIP)))
(net NOSAT ((PCU1 CTLWORD 4) (AU1 pNOSAT)))
(net LDMOR ((PCU1 CTLWORD 5) (AU1 pLDMOR)))
(net LDRO ((PCU1 CTLWORD 6) (AU1 pLDRO)))
(net LDR1 ((PCU1 CTLWORD 7) (AU1 pLDR1)))
(net LDCOEF ((PCU1 CTLWORD 8) (AU1 pLDCOEF)))
(net XMITMOR ((PCU1 CTLWORD 9) (AU1 pXMITMOR)))
(net XMITACC ((PCU1 CTLWORD 10) (AU1 pXMITACC)))
(net SELMORIN ((PCU1 CTLWORD 11) (AU1 pSELMORIN)))
```

```
(net ANDCOEF ((PCU1 CTLWORD 12) (AU1 pANDCOEF)))
(net COEFCOMP ((PCU1 CTLWORD 13) (AU1 pCOEFCOMP)))
(net NOABS ((PCU1 CTLWORD 14) (AU1 pNOABS)))
(net MINUS ((PCU1 CTLWORD 15) (AU1 pMINUS)))
(net SELBBUS ((PCU1 CTLWORD 16) (AU1 pSELBBUSIN)))
(net ZERO_BIN ((PCU1 CTLWORD 17) (AU1 pZERO_BIN)))
(net S0 ((PCU1 CTLWORD 18) (AU1 pS0)))
(net S1 ((PCU1 CTLWORD 19) (AU1 pS1)))
(net S2 ((PCU1 CTLWORD 20) (AU1 pS2)))
(net OENR0 ((PCU1 CTLWORD 21) (AU1 pOENR0)))
(net OENR1 ((PCU1 CTLWORD 22) (AU1 pOENR1)))
(net ACC2REG ((PCU1 CTLWORD 23) (AU1 pACC2REG)))
(net SUMCOND ((PCU1 CTLWORD 24) (AU1 pSUMCOND)))
(net SHIFTCOEF ((PCU1 CTLWORD 25) (AU1 pSHIFTCOEF)))
(net pRDPORT ((PCU1 CTLWORD 26) (AU1 pRDPORT)))
(net pWRPORT ((PCU1 CTLWORD 27) (AU1 pWRPORT)))
(net WRPORT ((AU1 WRPORT) (parent WRPORT)))
(net LOADX0 ((PCU1 CTLWORD 28) (parent LOADX0)))
(net LOADX1 ((PCU1 CTLWORD 29) (parent LOADX1)))
(net LOADX2 ((PCU1 CTLWORD 30) (parent LOADX2)))
(net CONDLD ((PCU1 CTLWORD 31) (parent CONDLD)))
(net OENX0 ((PCU1 CTLWORD 32) (parent OENX0)))
(net OENX1 ((PCU1 CTLWORD 33) (parent OENX1)))
(net OENX2 ((PCU1 CTLWORD 34) (parent OENX2)))
(net XBUSZERO ((PCU1 CTLWORD 35) (parent XBUSZERO)))
(net EABUS2MBUS ((PCU1 CTLWORD 36) (AU1 pEABUS2MBUS)))
(net MBUS2EABUS ((PCU1 CTLWORD 37) (AU1 pMBUS2EABUS)))
(net OENEALATCH ((PCU1 CTLWORD 38) (parent OENEALATCH)))
(net XIP ((PCU1 CTLWORD 39) (parent XIP)))
(net MEMRD ((PCU1 CTLWORD 46) (AU1 pR)))
(net MEMWR ((PCU1 CTLWORD 47) (AU1 pW)))
(net RAMBUS AU1wordsize ((AU1 RAMIO) (parent RAMBUS)))
(net RBUS AU1wordsize ((AU1 RBUS) (parent RBUS)))
(net pRDSTRB ((PCU1 CTLWORD 40) (AU1 pRDSTRB)))
(net pWRSTRB ((PCU1 CTLWORD 41) (AU1 pWRSTRB)))
(net pPORTADDRS 4 ((PCU1 CTLWORD 42) (AU1 pPORTADDRESS)))
(net RDSTRB ((AU1 RDSTRB) (parent RDSTRB)))
(net WRSTRB ((AU1 WRSTRB) (parent WRSTRB)))
(net PORTADDRS 4 ((AU1 PORTADDRESS) (parent PORTADDRS)))
(net INPORT AU1wordsize ((AU1 IOPORT) (parent INPORT)))
(net OUTPORT AU1wordsize ((AU1 PORTOUT) (parent OUTPORT)))
(net EABUS1 (- (integer-length (- ramwords 1)) 1) ((AU1 EABUSIN) (AU1
```

```
EABUSOUT) (PCU1 TIMERINBUS) (parent EABUS1)))
(net EABUS2 (- (integer-length (- max_sample_intvl 1)) (integer-length
(- ramwords 1))) ((AU1 EABUSIN (integer-length (- ramwords 1))) (PCU1
TIMERINBUS (integer-length (- ramwords 1)))))
(net EABUS3 mergeNet
  ((parent EABUS3 0)
   (AU1 EABUSOUT (- (integer-length (- ramwords 1)) 1) (- AU1wordsize
1) 1)
   (PCU1 TIMERINBUS (- (integer-length (- ramwords 1)) 1) (- (integer-length
(- ramwords 1)) 1))
   (AU1 EABUSIN (- (integer-length (- ramwords 1)) 1) (- (integer-length
(- ramwords 1)) 1))))
(net EOS ((PCU1 EOS) (parent EOS)))
(net PGMSTATE (integer-length (- num_of_states 1)) ((PCU1 PGMSTATE) (parent
PGMSTATE)))
(net EXTCOND (- num_of_cond 2) ((PCU1 CFSMCONDIN 2) (parent EXTCOND)))
(net APUSIGN ((PCU1 CFSMCONDIN 0) (parent APUSIGN)))
(net AUSIGN ((PCU1 CFSMCONDIN 1) (AU1 AUSIGN) (parent AUSIGN)))
(net LGUINSTRADDR nr_of_lgu_instr ((PCU1 CTLWORD 48) (parent LGUINSTRADDR)))
(net CC ((AU1 CC) (parent CC)))
(net RESET ((PCU1 PCURESET) (parent RESET)))
(net INSTRADDRS (integer-length (- max_blk_size 1)) ((PCU1 INSTRNUM) (parent
INSTRADDRS)))
(net CSTOREADDRSFIELD (integer-length (- ramwords 1)) ((PCU1 CTLWORD (-
num_of_cstoreout 1) -1) (parent CSTOREADDRSFIELD)))
(net READ ((AU1 READ) (parent READ)))
(net WRITE ((AU1 WRITE) (parent WRITE)))
(net SCANIN ((PCU1 PCUSCANIN) (parent SCANIN)))
(net SCANLINK.1 ((PCU1 PCUSCANOUT) (AU1 SCANIN)))
(net SCANLINK.2 ((AU1 SCANOUT) (parent SCANOUT)))
(net PHI1 ((AU1 PHI1) (PCU1 PHI1) (parent PHI1)))
(net PHI2 ((AU1 PHI2) (PCU1 PHI2) (parent PHI2)))
(net TESTCLK1 ((AU1 TPHI1) (PCU1 TESTCLOCK1) (parent TESTCLK1)))
(net TESTCLK2 ((AU1 TPHI2) (PCU1 TESTCLOCK2) (parent TESTCLK2)))
;
;*************************************************************************
; sdl file for ram-apu macrocell
;*************************************************************************
;
(layout-generator Flint)
(parent-cell ram-apu
(parameters ramwords
```

```
ram-address-plane
ram-bit-plane
nr_of_lgu_feedback
nr_of_lgu_out            ; including feedbacks
nr_of_lgu_xin    ; external inputs to lgu excl signs.
nr_of_lgu_instr ; num of instr addr bit from cstore.
nr_of_lgu_minterm
lgu-input-plane
lgu-output-plane
side_addrsfield
side_EABUS
num_of_states
max_blk_size
AU1wordsize))

(sub-cells
  (apu APU1
     (parameters (side_addrsfield side_addrsfield)
(side_EABUS side_EABUS)
(N (integer-length
 (- ramwords 1)))))

  (ram3T RAM1
     (parameters (width                    AU1wordsize)
(words                  ramwords)
(ram-address-plane      ram-address-plane)
(ram-bit-plane          ram-bit-plane)))

  (fsm LGU
     (parameters (inwidth  (add  nr_of_lgu_feedback  2 nr_of_lgu_xin
nr_of_lgu_instr))
; feedbacks + 2(au,apu sign) + xin + nr_of_lgu_instr
(outwidth               nr_of_lgu_out)
(minterm                nr_of_lgu_minterm)
(nr_of_feedback         nr_of_lgu_feedback)
(input-plane            lgu-input-plane)
(output-plane           lgu-output-plane)))

  (testmodule TESTLOGIC
     (parameters  (num_of_states num_of_states)
(max_blk_size max_blk_size))))
```

```
(net LOADX0 ((APU1 pLOADX0) (parent LOADX0)))
(net LOADX1 ((APU1 pLOADX1) (parent LOADX1)))
(net LOADX2 ((APU1 pLOADX2) (parent LOADX2)))
(net CONDLD ((APU1 pCONDLD) (parent CONDLD)))
(net OENX0 ((APU1 pOENX0) (parent OENX0)))
(net OENX1 ((APU1 pOENX1) (parent OENX1)))
(net OENX2 ((APU1 pOENX2) (parent OENX2)))
(net XBUSZERO ((APU1 pXBUSZERO) (parent XBUSZERO)))
(net OENEALATCH ((APU1 pOENEALATCH) (parent OENEALATCH)))
(net XIP ((APU1 pXIP) (parent XIP)))
(net RAMBUS AU1wordsize ((RAM1 RAMDATABUS) (parent RAMBUS)))
(net EABUS1 (- (integer-length (- ramwords 1)) 1) ((APU1 EADDRESS) (RAM1
RAMADDRESS) (parent EABUS1)))

(net EABUS3 1 ((APU1 EADDRESS (- (integer-length (- ramwords 1)) 1)) (RAM1
RAMADDRESS (- (integer-length (- ramwords 1)) 1)) (parent EABUS3)))
(net PGMSTATE (integer-length (- num_of_states 1)) ((TESTLOGIC STATEADDRS)
(parent PGMSTATE)))
(net LGUINPUTS nr_of_lgu_xin ((LGU IN) (parent LGUINPUTS)))
(net APUSIGN ((LGU IN nr_of_lgu_xin) (APU1 APUSIGN) (parent APUSIGN)))
(net AUSIGN ((LGU IN (add nr_of_lgu_xin 1)) (parent AUSIGN)))
(net LGUINSTRADDR nr_of_lgu_instr ((LGU IN (add nr_of_lgu_xin 2)) (parent
LGUINSTRADDR)))
(net CC ((APU1 CC) (LGU OUT 0) (parent CC)))
(net INSTRADDRS (integer-length (- max_blk_size 1)) ((TESTLOGIC INSTRADDRS)
(parent INSTRADDRS)))
(net CSTOREADDRSFIELD (integer-length (- ramwords 1)) ((APU1 ADDRSBUS
(- (integer-length (- ramwords 1)) 1) -1) (parent CSTOREADDRSFIELD)))
(net MCC ((TESTLOGIC MCC) (parent MCC)))
(net BPFLAG ((TESTLOGIC BPFLAG) (parent BPFLAG)))
(net MASTERCLOCK ((TESTLOGIC EXTMSTRCLK) (parent MASTERCLOCK)))
(net TESTMODEINV ((TESTLOGIC TESTMODEINV) (parent TESTMODEINV)))
(net READ ((RAM1 READ) (parent READ)))
(net WRITE ((RAM1 WRITE) (parent WRITE)))
(net SCANIN ((TESTLOGIC SCANIN) (parent SCANIN)))
(net SCANLINK.1 ((TESTLOGIC SCANOUT) (APU1 SCANIN)))
(net SCANLINK.2 ((APU1 SCANOUT) (LGU SCANIN)))
(net SCANOUT ((LGU SCANOUT) (parent SCANOUT)))
(net PHI1 ((TESTLOGIC PHI1) (APU1 PHI1) (RAM1 PHI1) (LGU PHI1) (parent
PHI1)))
(net PHI2 ((TESTLOGIC PHI2) (APU1 PHI2) (RAM1 PHI2) (LGU PHI2) (parent
```

```
PHI2)))
(net TESTCLK1 ((APU1 TPHI1) (TESTLOGIC SHIFTIN) (LGU SHIFTIN) (parent
TESTCLK1)))
(net TESTCLK2 ((APU1 TPHI2) (TESTLOGIC SHIFTOUT) (LGU SHIFTOUT) (parent
TESTCLK2)))
```

Appendix F

Assembly Program for the Robot Controller

Following is the assembly language program for implementing the robot control algorithm on the custom processor.

```
;-------------------------------------------------------------------------
; LagerIII assemble language program to implement a two axes adaptive
; robot controller. It includes a PID controller and and adaptive control
; algorithm.
; The PI controller can operate at a slower sampling rate than the D and
; the adaptive controller. The I-controller can be optionally skipped.
; -Khalid Azim 6-24-87
;-------------------------------------------------------------------------
;

(ram
(const 21) ; kp2(0),  kp1(1),  kv2(2),  kv1(3),  ki2(4),  ki1(5),

; kf1(6),  kf2(7),  km11(8), km12(9), km21(10),km22(11),

; mh11[1](12),      mh12[1](13),      mh21[1](14)

; mh22[1](15),      DBAND(16),        FRth(17),

; T1(18),  L(19)    PERIOD(20)

(dataptr 22) ; xp1(0),     xp2(1)

             ; (refmod0)
```

```
; xv1(2),     xvh1(3),    ev1(4),     u1(5),
; eu1(6),     u[0](7)
; eu1 is actually not used

; (refmod1)
; xv2(8),     xvh2(9),    ev2(10),     u2(11),
; eu2(12),     u[0](13)
; eu2 is actually not used

; (mhptr)
; mh11[0](14), mh12[0](15),
; mh21[0](16), mh22[0](17)
;
; q1(18) q2(19)
; f1(20) f2(21)

   ep1
ep2
u1T1
u2T1
(w1 2)
(w2 2)
(temp 4) ; extra locations for unforeseen needs.
 )


(dfsm
  (DBANDTEST (cc (not AU1SIGN))) ; if DBAND>eu1

  (XVNEG (cc AU1SIGN))

  (XVPOS (cc (not cc)))

  (UPOSFLAG (PFLAG AU1SIGN))

  (UNEG (cc AU1SIGN))

  (UPOS (cc PFLAG))
 )

(cfsm
 (INIT 0 () (goto RDCONSTS))
```

```
(RDCONSTS 1 (not lctest0) (goto RDCONSTS))
(RDCONSTS 1 lctest0 (goto SETIMER))
(SETIMER 2 () (goto PCTL1))

; start outer loop - P & I control
(PCTL1 3 () (goto MULT1))
(MULT1 5 (not lctest1) (goto MULT1))
(MULT1 5 lctest1 (goto PCTL2))
(PCTL2 4 () (goto MULT20))
(MULT20 5 (not lctest1) (goto MULT20))
(MULT20 5 () (goto PCTL3))
(PCTL3 6 Iflag (goto MULT2))
(PCTL3 6 (not Iflag) (goto ADTRIG))
(MULT2 5 (not lctest1) (goto MULT2))
(MULT2 5 lctest1 (goto ICTL1))
(ICTL1 7 () (goto MULT3))
(MULT3  5   (not lctest1) (goto MULT3))
(MULT3         5       lctest1                          (goto ICTL2))
(ICTL2 8 () (goto ADTRIG))

; start inner loop
; D control
(ADTRIG 9  ()  (goto DCTL))
(DCTL 10  () (goto MULT4))
(MULT4          5   (not lctest1) (goto MULT4))
(MULT4 5 lctest1 (goto REFMDL1))
(REFMDL1 11  () (goto MULT5))
(MULT5 5   (not lctest1) (goto MULT5))
(MULT5 5 lctest1 (goto REFMDL2))
(REFMDL2 12  (not APU1SIGN) (goto DCTL))
(REFMDL2 12 APU1SIGN (goto INRTSETUPA))

; Adaptive control
(INRTSETUPA 13  () (goto MULT6))
(MULT6 5   (not lctest1) (goto MULT6))
(MULT6 5 lctest1 (goto INRTEST1A))
(INRTEST1A 14  () (goto MULT7))
(MULT7 5   (not lctest1) (goto MULT7))
(MULT7 5 lctest1 (goto INRTEST2A))
(INRTEST2A 15  () (goto INRTSETUPB))
(INRTSETUPB 16  () (goto MULT8))
(MULT8 5 (not lctest1) (goto MULT8))
```

```
(MULT8 5  lctest1 (goto INRTEST1B))
(INRTEST1B 14 () (goto MULT9))
(MULT9 5 (not lctest1) (goto MULT9))
(MULT9 5 lctest1 (goto INRTEST2B))
(INRTEST2B 15  () (goto INRTSETUPC))
(INRTSETUPC    17 ()  (goto MULT10))
(MULT10        5   (not lctest1) (goto MULT10))
(MULT10 5 lctest1 (goto INRTEST1C))
(INRTEST1C     14  ()                          (goto MULT11))
(MULT11        5   (not lctest1) (goto MULT11))
(MULT11 5 lctest1 (goto INRTEST2C))
(INRTEST2C     15  () (goto INRTSETUPD))
(INRTSETUPD    18  ()                              (goto MULT12))
(MULT12        5   (not lctest1) (goto MULT12))
(MULT12 5 lctest1 (goto INRTEST1D))
(INRTEST1D     14  ()                              (goto MULT13))
(MULT13        5   (not lctest1) (goto MULT13))
(MULT13 5 lctest1 (goto INRTEST2D))
(INRTEST2D     15  () (goto FRICEST1))

; Friction compensation
(FRICEST1 19 () (goto MULT14))
(MULT14 5 (not lctest1) (goto MULT14))
(MULT14 5 lctest1 (goto FRICEST2))
(FRICEST2 20 () (goto MULT15))
(MULT15 5 (not lctest1) (goto MULT15))
(MULT15 5 lctest1 (goto FRICCOMP1))
(FRICCOMP1 21 (not AU1SIGN) (goto FRICCOMP2A))
(FRICCOMP1 21 AU1SIGN (goto FRICCOMP3A))
(FRICCOMP2A 22 () (goto FRICCOMP4))
(FRICCOMP3A 23 () (goto FRICCOMP4))
(FRICCOMP4 24 (not AU1SIGN) (goto FRICCOMP2B))
(FRICCOMP4 24 AU1SIGN (goto FRICCOMP3B))
(FRICCOMP2B 22 () (goto TORQSETUP1))
(FRICCOMP3B 23 () (goto TORQSETUP1))

; Torque computation
(TORQSETUP1 25 () (goto TORQCOMP1A))
(TORQCOMP1A 26 () (goto MULT16))
(MULT16 5 (not lctest1) (goto MULT16))
(MULT16 5 lctest1 (goto TORQSETUP2))
(TORQSETUP2 27 () (goto TORQCOMP1B))
```

```
 (TORQCOMP1B 26 () (goto MULT17))
 (MULT17 5 (not lctest1) (goto MULT17))
 (MULT17 5 lctest1 (goto TORQCOMP2))
 (TORQCOMP2 28 () (goto TORQCOMP1C))
 (TORQCOMP1C 26 () (goto MULT18))
 (MULT18 5 (not lctest1) (goto MULT18))
 (MULT18 5 lctest1 (goto TORQSETUP3))
 (TORQSETUP3 29 () (goto TORQCOMP1D))
 (TORQCOMP1D 26 () (goto MULT19))
 (MULT19 5 (not lctest1) (goto MULT19))
 (MULT19 5 lctest1 (goto TORQCOMP3))
 (TORQCOMP3 30 () (goto OUTPUTSETUP))

; Output states, delay, idle, read position
 (OUTPUTSETUP 31 () (goto OUTPUT))
 (OUTPUT 32 (not APU1SIGN) (goto OUTPUT))
 (OUTPUT 32 APU1SIGN (goto DELAY))
 (DELAY 33  () (goto IDLE))
 (IDLE 35  (not EOS) (goto IDLE))
 (IDLE 35 EOS (goto RDVEL))
 (RDVEL 34 (not APU1SIGN) (goto ADTRIG))
 (RDVEL 34 APU1SIGN (goto PCTL1))
 )

(loop_test 21 18)

(dp_word_size 20)

(reset_timer INIT RDCONSTS SETIMER)

(max_sample_intvl 2048)


(rom

; initialize states
  (block0
    ((acc=0))
    ((mbus=acc) (mem=mbus) (addr ep1) (xbus=0) (eabus=sum)) ; ep1=0
    ((mbus=acc) (mem=mbus) (addr ep2) (xbus=0) (eabus=sum)) ; ep2=0
    ((mbus=acc) (mem=mbus) (addr dataptr) (offset 3)
(xbus=0) (eabus=sum)) ; xvh1=0
```

```
    ((mbus=acc) (mem=mbus) (addr dataptr) (offset 9)
(xbus=0) (eabus=sum)) ; xvh2=0
    ((mbus=acc) (mem=mbus) (addr dataptr) (offset 20)
(xbus=0) (eabus=sum)) ; f1=0
    ((mbus=acc) (mem=mbus) (addr dataptr) (offset 21)
(xbus=0) (eabus=sum)) ; f2=0
    ((mbus=acc) (mem=mbus) (addr u1T1) (xbus=0) (eabus=sum)) ;u1[1]=0
    ((mbus=acc) (mem=mbus) (addr u2T1) (xbus=0) (eabus=sum)) ;u2[1]=0
    ((mbus=acc) (mem=mbus) (addr dataptr) (offset  2)
(xbus=0) (eabus=sum)) ; xv1=0
    ((mbus=acc) (mem=mbus) (addr dataptr) (offset  8)
(xbus=0) (eabus=sum)) ; xv2=0
    ((mbus=acc) (mem=mbus) (addr w1) (offset 1) (xbus=0)
     (eabus=sum)) ; w(w1[1])=0
    ((mbus=acc) (mem=mbus) (addr w2) (offset 1) (xbus=0)
     (eabus=sum) ; w(w2[1])=0
     (ioport=extport 0) (rbus=ioport) (r*=rbus 0))   ;r0=in() - input
1st const.
    ((addr const) (offset -1) (xbus=0) (eabus=sum) (x*=eabus 0))
; x0=const.base_addr-1
  )

; routine to input constants. 21 iterations required.
; During the last iteration dummy data would be input,
; needed to store the last constant.
; Iteration counting is done by the loop counter.
; Addressing is done by x0.
  (block1
    ((ioport=extport 0) (rbus=ioport) (r*=rbus 0)  ; r0=in()
     (mbus=r* 0) (mem=mbus)  ; w(ea)=r0
     (xbus=x* 0) (addr 1) (eabus=sum) (x*=eabus 0))
; x0 = ea = (x0 + 1)
  )

; Set the sample interval timer
; Write zero to the D/A ports to remove any torque.
; Reset quad decoder counter.
  (block2
    ((mor=mem) (addr const) (offset 20) (xbus=0) (eabus=sum)) ; r(ea)
[period]
    ((mbus=mor) (eabus=mbus) (timereg=eabus) ; set timer & rst quad decoder
     (acc=0)) ; same ctl sig for timer & quad reset.
```

```
    ((mbus=acc) (ioport=mbus) (extport=ioport 3));  write zero to D/A
arm1
    ((mbus=acc) (ioport=mbus) (extport=ioport 4));  write zero to D/A
arm2
  )

; P-control (i)
; Read in position and ref data.
; Compute position error.
  (block3
    ((rbus=ioport) (ioport=extport 5) (r*=rbus 0))  ; read quad1 [pos1]
    ((rbus=ioport) (ioport=extport 7) (r*=rbus 0) ; read ref1
     (mbus=r* 0) (mor=mbus) ; mor = pos1
     (mem=mbus) (addr dataptr) (offset 0) (xbus=0) (eabus=sum)) ; w(xp1)
   ((abus=-mor) (bbus=mbus) (mbus=r* 0) (acc=sum) ; acc=ref1-pos1
    (mor=mem) (addr const) (offset 1) (xbus=0) (eabus=sum)) ; r(ea=kp1)=>mor
   ((mbus=mor) (rcoef=mbus)) ; rcoef=kp1
   ((shrcoef) (abus=coef.mor) (acc=abus) (nosat) ; kp1*ep1 [lsb multiply]
    (mbus=acc) (mem=mbus) (addr ep1) (xbus=0) (eabus=sum)) ; w(ep1)
  )

; P-control (ii)
  (block4
   ((shrcoef) (abus=coef.-mor) (bbus=acc>* 1) (acc=sum)) ; msb mult kp1*ep1
   ((bbus=acc<* 1) (acc=bbus)) ; left shift [scale by 64]
   ((bbus=acc<* 1) (acc=bbus)) ; left shift
   ((bbus=acc<* 1) (acc=bbus) ; left shift
    (mor=mem) (addr const) (offset 0) (xbus=0) (eabus=sum)) ; r(ea=kp2)=>mor
    ((bbus=acc<* 1) (acc=bbus) ; left shift
     (mbus=mor) (rcoef=mbus)) ; rcoef=kp2
   ((bbus=acc<* 1) (acc=bbus) ; left shift
     (rbus=ioport) (ioport=extport 6) (r*=rbus 0))       ; read quad2 [pos2]
   ((bbus=acc<* 1) (acc=bbus) ; left shift
     (rbus=ioport) (ioport=extport 8) (r*=rbus 0)        ; read ref2
     (mbus=r* 0) (mor=mbus)                              ; mor = pos2
     (mem=mbus) (addr dataptr) (offset 1) (xbus=0) (eabus=sum)) ; w(xp2)
   ((mbus=acc) (mem=mbus) (addr dataptr) (offset 5)
(xbus=0) (eabus=sum))  ; w(u1)
   ((abus=-mor) (bbus=mbus) (mbus=r* 0) (acc=sum))      ; acc=ref2-pos2
   ((shrcoef) (abus=coef.mor) (acc=abus) (nosat)       ; lsb multiply
     (mbus=acc) (mem=mbus) (addr ep2) (xbus=0) (eabus=sum)) ; w(ep2)
   )
```

```
; P-control (iii)
; Scale up kp2*ep2 and store in uv2
; Also perform ki1*ep1 lsb.
; This result is not used if I-operation is not selected
  (block6
   ((shrcoef) (abus=coef.-mor) (bbus=acc>* 1) (acc=sum)) ; msb mult kp2*ep2
   ((bbus=acc<* 1) (acc=bbus)                ; left shift [scale by
64]
    (mor=mem) (addr const) (offset 19) (xbus=0) (eabus=sum)) ; rd(L)
   ((bbus=acc<* 1) (acc=bbus)                ; left shift
    (mbus=mor) (eabus=mbus) (x*=eabus 2))    ; x2=mem [L],for inner
loop iter.
   ((bbus=acc<* 1) (acc=bbus)) ; left shift
   ((bbus=acc<* 1) (acc=bbus) ; left shift
    (mor=mem) (addr const) (offset 5) (xbus=0) (eabus=sum)) ; rd(ki1)
   ((bbus=acc<* 1) (acc=bbus) ; left shift
    (mbus=mor) (rcoef=mbus)) ; rcoef=mor  [ki1]
   ((bbus=acc<* 1) (acc=bbus)                ; left shift
    (mor=mem) (addr ep1) (xbus=0) (eabus=sum))  ; rd(ep1)
   ((mbus=acc) (mem=mbus) (addr dataptr) (offset 11)
(xbus=0) (eabus=sum)  ; w(uv2)
    (abus=coef.mor) (acc=abus) (nosat))      ; acc=coef.mor [lsb]
   )

; I-control section (i)
  (block7
   ((shrcoef) (abus=coef.-mor) (bbus=acc>* 1) (acc=sum) ; acc=coef.-mor
+ acc>1
    (mor=mem) (addr w1) (offset 1) (xbus=0) (eabus=sum))      ; rd(w1[1])
   ((abus=mor) (bbus=acc>* 0) (acc=sum)                ; acc=ki1*ep1+
w1[1]
    (mor=mem) (addr dataptr) (offset 5) (xbus=0) (eabus=sum)) ; rd(u1)
   ((abus=mor) (bbus=acc>* 0) (acc=sum) ; acc=u1+w1[0]
    (mbus=acc) (mem=mbus) (addr w1) (offset 0) (xbus=0) (eabus=sum)) ;
w(w1[0])
   ((mor=mem) (addr const) (offset 4) (xbus=0) (eabus=sum) ; rd(ki2)
    (rbus=acc) (r*=rbus 1)) ; r1=u1, store later.
   ((mbus=mor) (rcoef=mbus) ; rcoef=mor [ki2]
    (mor=mem) (addr ep2) (xbus=0) (eabus=sum)) ; rd(ep2)
   ((shrcoef) (abus=coef.mor) (acc=abus) (nosat) ; acc=coef.mor (lsb)
    (mbus=r* 1) (mem=mbus) (addr dataptr) (offset 5)
```

```
(xbus=0) (eabus=sum))    ; w(u1)
  )

; I-control section (ii)
  (block8
    ((shrcoef) (abus=coef.-mor) (bbus=acc>* 1) (acc=sum) ; acc=coef.-mor
+ acc>1
     (mor=mem) (addr w2) (offset 1) (xbus=0) (eabus=sum)) ; rd(w2[1])
    ((abus=mor) (bbus=acc>* 0) (acc=sum) ; acc=ki2*ep2+w2[1]
     (mor=mem) (addr dataptr) (offset 11) (xbus=0) (eabus=sum))    ; rd(uv2)
    ((abus=mor) (bbus=acc>* 0) (acc=sum)                          ; acc=uv2+w2[0]
     (mbus=acc) (mem=mbus) (addr w2) (offset 0)
(xbus=0) (eabus=sum))  ; w(w2[0])
    ((mbus=acc) (mem=mbus) (addr dataptr) (offset 11)
(xbus=0) (eabus=sum))  ; w(uv2)
   )

; A/D-Trig : START OF INNER LOOP
; X2 is used for inner loop iteration count - active throughout the inner
loop
; Output A/D trigger
  (block9
    ((addr 1) (xbus=0) (eabus=sum) (areg=eabus)) ; addr=A/D conv. trig.
    ((mbus=areg) (ioport=mbus) (extport=ioport 10) ; wr A/D trig to port10.
     (addr dataptr) (offset 2) (xbus=0) (eabus=sum)
(x*=eabus 0))    ; x0=refmod0
    ((addr 1) (xbus=0) (eabus=sum) (x*=eabus 1)) ; x1=1 (for two iters.
; of next block)
   )

; D-control
  (block10
   ((mor=mem) (addr 3) (offset 0) (xbus=x* 0) (eabus=sum)) ; rd(u1)
   ((abus=mor) (acc=abus) ; acc=u1
    (mor=mem) (addr 0) (xbus=x* 0) (eabus=sum)) ; rd(xv1)
   ((abus=-mor) (bbus=acc>* 0) (acc=sum)                        ; acc=u1-xv1
[eu1]
    (mor=mem) (addr const) (offset 2) (xbus=x* 1) (eabus=sum)); rd(kv1)
; const + "kv2" + (x1=1) = address(kv1)
   ((mbus=mor) (rcoef=mbus))                                    ; rcoef=kv1
   ((mbus=acc) (mor=mbus))                                      ; mor=eu1
```

```
   ((shrcoef) (abus=coef.mor) (acc=abus) (nosat))        ; acc=coef.mor
(lsb)
  )

; Ref-model (i)
; compute u[0] = kv1*eu1 msb
; compute ev1, compare with DBAND
; compute u[0]*T1 for xvh1
  (block11
   ((shrcoef) (abus=coef.-mor) (bbus=acc>* 1) (acc=sum)) ; acc=coef.-mor
+ acc>1
   ((mem=mbus) (mbus=acc) (addr 5) (xbus=x* 0)) ; w(u[0])
   ((mor=mem) (addr 0) (xbus=x* 0) (eabus=sum) ; rd(xv1) (addr=x0)
    (acc=0)) ; acc=0
   ((mor=mem) (addr 1) (xbus=x* 0) (eabus=sum) ; rd(xvh1)
    (abus=-mor) (acc=abus) ; acc=-mor [-xv1]
    (rbus=acc) (r*=rbus 0)) ; r0=acc[=0]
   ((abus=mor) (bbus=acc<* 1) (acc=sum) ; acc=xvh1-2*xv1 [ev1]

    (mor=mem) (addr const) (offset 16) (xbus=0) (eabus=sum))  ; rd(DBAND)
   ((mbus=acc) (mem=mbus) (addr 2) (xbus=x* 0) (eabus=sum)  ; w(ev1)
    (mor=mbus) ; mor=acc  [ev1]
    (abus=mor) (acc=sum)) ; acc=mor  (DBAND)
   ((abus=-absmor) (bbus=acc>* 0) (acc=sum) ; acc=DBAND-|ev1|
    (mor=mem) (addr const) (offset 18) (xbus=0) (eabus=sum)) ; rd(T1)
   ( DBANDTEST  ; set dfsm
    (mbus=mor) (rcoef=mbus) ; rcoef=T1
    (mor=mem) (addr 5) (xbus=x* 0) (eabus=sum)) ; rd(u[0])
   ((mbus=r* 0) (mem=mbus) (mcondload) (addr 2) (xbus=x* 0) ; wc(ev1=0)
      (shrcoef) (abus=coef.mor) (acc=abus) (nosat)) ; u[0]*T1 lsb
   )

; Ref-model (ii).
; Compute xvh1.
; This is the last block of the loop containing A/D-Trig,D-control,
;    Ref-model(i), and Ref-model(ii).
; Update pointer x0 for the second loop iteration.
; Decrement loop counter x1
  (block12
   ((shrcoef) (abus=coef.-mor) (bbus=acc>* 1) (acc=sum) ; msb mult fo
T1*u1[0]
    (mor=mem) (addr 0) (xbus=x* 0) (eabus=sum)) ; rd(xv1)
```

```
   ((abus=mor) (acc=abus) ; acc=xv1
    (mbus=acc) (mor=mbus)  ; mor=T1*u[0]
    (xbus=x* 0) (addr 6) (eabus=sum) (x*=eabus 0)) ; x0=x0+6,update ptr.
   ((abus=mor) (bbus=acc<* 1) (acc=sum) ; acc=T1*u+2*xv1 [xvh1]
    (xbus=x* 1) (addr -1) (eabus=sum) (x*=eabus 1)) ; x1=x1-1, loop cnt
   ((mbus=acc) (mem=mbus) (addr -5) (xbus=x* 0)) ; w(xvh1)
   )

; Inertia-est setup (A)
; Set up for call to inertia est subroutine to compute mh11[0]
; mh11[1]=>r0, u[1]=>r1, km11=>rcoef, ev1=>mor
;
  (block13
   ((mor=mem) (addr const) (offset 12) (xbus=0) (eabus=sum)) ;rd(mh11[1])
   ((abus=mor) (acc=abus)  ; acc=mh11[1]
    (mor=mem) (addr u1T1) (xbus=0) (eabus=sum)) ; rd(u[1])
   ((mor=mem) (addr const) (offset 8) (xbus=0) (eabus=sum) ; rd(km11)
    (rbus=acc) (r*=rbus 0)  ; r0=mh11[1]
    (abus=mor) (acc=abus)) ; acc=u[1]
   ((mor=mem) (addr dataptr) (offset 4) (xbus=0) (eabus=sum)  ;rd(ev1)
    (rbus=acc) (r*=rbus 1)                                    ; r1=u[1]
    (mbus=mor) (rcoef=mbus)) ; rcoef=km11
   ((shrcoef) (abus=coef.mor) (acc=abus) (nosat))          ; km11*ev1 [lsb
mult]
  )

; Inertia-est  setup(B)
; Set up for call to inertia est subroutine to compute mh12[0]
; mh12[1]=>r0, u[1](2nd joint) =>r1, km12=>rcoef, ev1=>mor
;
  (block16
   ((mbus=acc) (mem=mbus) (addr dataptr) (offset 14) (xbus=0) (eabus=sum))
; store acc=>mh11[0]
   ((mor=mem) (addr const) (offset 13) (xbus=0) (eabus=sum)) ;rd(mh12[1])
   ((abus=mor) (acc=abus)                                     ; acc=mh12[1]
    (mor=mem) (addr u2T1) (xbus=0) (eabus=sum))             ; rd(u[1]) - 2nd
joint
   ((mor=mem) (addr const) (offset 9) (xbus=0) (eabus=sum) ; rd(km12)
    (rbus=acc) (r*=rbus 0)                                  ; r0=mh12[1]
    (abus=mor) (acc=abus))                                  ; acc=u[1] (2nd
joint)
   ((mor=mem) (addr dataptr) (offset 4) (xbus=0) (eabus=sum)  ;rd(ev1)
```

```
    (rbus=acc) (r*=rbus 1)                                  ; r1=u[1]  (2nd
joint)
    (mbus=mor) (rcoef=mbus))                                ; rcoef=km12
   ((shrcoef) (abus=coef.mor) (acc=abus) (nosat))           ; km12*ev1 [lsb
mult]
  )


; Inertia-est  setup(C)
; Set up for call to inertia est subroutine to compute mh21[0]
; mh21[1]=>r0, u1[1] (2nd joint)=>r1, km21=>rcoef, ev2=>mor
;
  (block17
   ((mbus=acc) (mem=mbus) (addr dataptr) (offset 15) (xbus=0) (eabus=sum))
    ; store acc=>mh12[0]
   ((mor=mem) (addr const) (offset 14) (xbus=0) (eabus=sum)) ;rd(mh21[1])
   ((abus=mor) (acc=abus)                                   ; acc=mh21[1]
    (mor=mem) (addr u1T1) (xbus=0) (eabus=sum))            ; rd(u[1]) (1st
joint)
   ((mor=mem) (addr const) (offset 10) (xbus=0) (eabus=sum) ; rd(km21)
    (rbus=acc) (r*=rbus 0)                                  ; r0=mh21[1]
    (abus=mor) (acc=abus))                                  ; acc=u[1] 1st
joint
   ((mor=mem) (addr dataptr) (offset 10) (xbus=0) (eabus=sum)  ;rd(ev2)
    (rbus=acc) (r*=rbus 1)                                  ; r1=u[1] 1st
joint
    (mbus=mor) (rcoef=mbus))                                ; rcoef=km21
   ((shrcoef) (abus=coef.mor) (acc=abus) (nosat))           ; km21*ev2 [lsb
mult]
  )
;
; Inertia-est  setup(D)
; Set up for call to inertia est subroutine to compute mh22[0]
; mh22[1]=>r0, u[1] 2nd joint =>r1, km22=>rcoef, ev2=>mor
;
  (block18
   ((mbus=acc) (mem=mbus) (addr dataptr) (offset 16) (xbus=0) (eabus=sum))
; store acc=>mh21[0]
   ((mor=mem) (addr const) (offset 15) (xbus=0) (eabus=sum)) ;rd(mh22[1])
   ((abus=mor) (acc=abus)                                   ; acc=mh22[1]
    (mor=mem) (addr u2T1) (xbus=0) (eabus=sum))            ; rd(u[1]) 2nd
joint
```

```
   ((mor=mem) (addr const) (offset 11) (xbus=0) (eabus=sum) ; rd(km22)
    (rbus=acc) (r*=rbus 0)                                  ; r0=mh22[1]
    (abus=mor) (acc=abus))                                  ; acc=u[1] 2nd
joint
   ((mor=mem) (addr dataptr) (offset 10) (xbus=0) (eabus=sum)  ;rd(ev2)
    (rbus=acc) (r*=rbus 1)                                  ; r1=u[1]  2nd
joint
    (mbus=mor) (rcoef=mbus))                                ; rcoef=km22
   ((shrcoef) (abus=coef.mor) (acc=abus) (nosat))           ; km22*ev2 [lsb
mult]
  )

;
; Inertia-est (i)
; Complete km*em and setup to do (km*em) * u
;
  (block14
   ((shrcoef) (abus=coef.-mor) (bbus=acc>* 1) (acc=sum) ; km*em msb
    (mbus=r* 1) (rcoef=mbus)) ; rcoef=u
   ((mbus=acc) (mor=mbus)) ; mor=(km*em)
   ((shrcoef) (abus=coef.mor) (acc=abus) (nosat))       ; (km*em)*u [lsb
mult]
  )

;
; Inertia-est (ii)
; Compute mhij[0]
;
  (block15
   ((shrcoef) (abus=coef.-mor) (bbus=acc>* 1) (acc=sum)) ; (km*em)*u msb
   ((mbus=r* 0) (mor=mbus)) ; mor=r0 [mhij[1]]
   ((abus=mor) (bbus=acc>* 0) (acc=sum)) ; mhij[0]
   )

;
; Friction-est (i)
; compute kf1*ev1
;
  (block19
   ((mor=mem) (addr const) (offset 6) (xbus=0) (eabus=sum)) ; rd(kf1)
   ((mbus=acc) (mem=mbus) (addr dataptr) (offset 17) (xbus=0) (eabus=sum))
; w(mh22[0]), left over from inertia comp.
```

```
   ((mor=mem) (addr dataptr) (offset 4) (xbus=0) (eabus=sum) ; rd(ev1)
    (mbus=mor) (rcoef=mbus)) ; rcoef=kf1
   ((shrcoef) (abus=coef.-mor) (acc=abus) (nosat)) ; kf1*ev1
 )

;
; Friction-est (ii)
;
  (block20
   ((shrcoef) (abus=coef.mor) (acc=abus) (nosat) ; kf1*ev1 msb
    (mor=mem) (addr const) (offset 7) (xbus=0) (eabus=sum)) ; rd(kf2)
   ((rbus=acc) (r*=rbus 0) ; r0=kf1.ev1
    (mbus=acc) (rcoef=mbus) ; rcoef=kf2
    (mor=mem) (addr const) (offset 10) (xbus=0) (eabus=sum)) ; rd(ev2)
   ((shrcoef) (abus=coef.-mor) (acc=abus) (nosat)) ; kf2*ev2
  )


;
; Friction-compensation (i)
; compute |xv1|-Frth
;
  (block21
   ((mor=mem) (addr const) (offset 17) (xbus=0) (eabus=sum) ; rd(FRth)
    (rbus=acc) (r*=rbus 1)) ; r1=kf2.ev2
   ((mor=mem) (addr dataptr) (offset 2) (xbus=0) (eabus=sum)  ; rd(xv1)
    (abus=-mor) (acc=abus))  ; acc=-FRth
   ((abus=absmor) (bbus=acc>* 0)  (acc=sum) ; acc=|xv1|-FRth
; test sign in fsm for branch
    (addr dataptr) (offset 7) (xbus=0) (eabus=sum) (x*=eabus 0))
; x0=refmod0, for block23
   ((abus=mor) (acc=abus) ; acc=xv1 reqd by blk22
    (addr dataptr) (offset 20) (xbus=0) (eabus=sum) (x*=eabus 1))
   )
; x1=f1, for block23
;
; Friction-compensation (ii)
; This block is executed on the results of block21 (block24 for joint2)
ausign.
;
  (block22
   ( XVNEG
```

```
    (mor=mem) (addr 0) (xbus=x* 1) (eabus=sum)) ; rd(f1)
   ((abus=mor) (acc=abus)
    (mbus=r* 0) (mor=mbus)) ; mor=r0 [kf.ev1]
   ((abus=-mor) (bbus=acc>* 0) (acc=sum) (acondload))   ;acc=f1-kf.ev1
(condacc)
   ( XVPOS )
   ((abus=mor) (bbus=acc>* 0) (acc=sum) (acondload))    ;acc=f1+kf.ev1
(condacc)
   ((mbus=acc) (mem=mbus) (addr 0) (xbus=x* 1) (eabus=sum)) ; w(f1)
   )


;
; Friction-compensation (iii)
; This block is executed on the results of block21 (block24 for joint2)
ausign.
;
  (block23
   ((mor=mem) (addr 0) (xbus=x* 0) (eabus=sum)) ; rd(u1[0])  x0=refmod0+5
<- u1
   ((abus=-mor) (acc=abus)) ; acc=-u1
   ( UPOSFLAG)
   ((abus=mor) (acc=abus)) ; acc=u1
   ( UNEG
    (mor=mem) (addr 0) (xbus=x* 1) (eabus=sum)) ; rd(f1), x1=f1
   ((abus=mor) (acc=abus) ; acc=f1
    (mbus=r* 0) (mor=mbus)) ; mor=r0  [kf1.ev1]
   ((abus=-mor) (bbus=acc>* 0) (acc=sum) (acondload)) ;acc=f1-kf.ev1 (condacc
   ( UPOS)
   ((abus=mor) (bbus=acc>* 0) (acc=sum) (acondload)) ;acc=f1+kf.ev1 (condacc)
   ((mbus=acc) (mem=mbus) (addr 0) (xbus=x* 1) (eabus=sum)) ; w(f1)
  )

;
; Friction-compensation (iv)
; compute |xv2|-Frth
; branch based on the result
; kfev2=>r0, addr(u2[0])=>x0, addr(f2)=>x1,preparation for block23 to
do joint2
; acc=xv2 for block22
;
  (block24
```

```
   ((mor=mem) (addr const) (offset 17) (xbus=0) (eabus=sum) ; rd(FRth)
    (mbus=r* 1) (bbus=mbus) (acc=sum)) ; acc=r1,[kfev2]
   ((mor=mem) (addr dataptr) (offset 8) (xbus=0) (eabus=sum)  ; rd(xv2)
    (abus=-mor) (acc=abus)                                    ; acc=-FRth
    (rbus=acc) (r*=rbus 0)) ; r0=acc [kfev2] for blk23
   ((abus=absmor) (bbus=acc>* 0)  (acc=sum) ; acc=|xv2|-FRth
     ; test sign in fsm for branch
    (addr dataptr) (offset 13) (xbus=0) (eabus=sum) (x*=eabus 0))
; x0=refmod1, for block23
   ((abus=mor) (acc=abus)                          ; acc=xv2 reqd by blk22
    (addr dataptr) (offset 21) (xbus=0) (eabus=sum) (x*=eabus 1))
; x1=f2 for block23
  )

;
; Torque-setup (i)
; Set up to compute u1[0]*mh11[0]
; mor=mh11[0], acc=u1[0]
;

  (block25
   ((mor=mem) (addr dataptr) (offset 7) (xbus=0) (eabus=sum)); rd(u1[0])
   ((mor=mem) (addr dataptr) (offset 14) (xbus=0) (eabus=sum)
; rd(mh11[0])
    (abus=mor) (acc=abus)) ; acc=u1[0]
  )

; Torque-computation (i)
; shift left u[0] by 3 (X 8) followed by u*mhij
;

  (block26
   ((bbus=acc<* 1) (acc=bbus)) ; acc=u1<1
   ((bbus=acc<* 1) (acc=bbus)) ; acc=u1<1
   ((bbus=acc<* 1) (acc=bbus) ; acc=u1<1
    (mbus=mor) (rcoef=mbus)) ; rcoef=mh11[0]
   ((shrcoef) (abus=coef.mor) (acc=abus) (nosat))      ; u1*mh11 [lsb
multiply]
  )

; Torque-setup (ii)
; Set up to compute u2[0]*mh12[0]
```

```
; mor=mh12, acc=u2, save mh11*u1 => r0
;

  (block27
   ((shrcoef) (abus=coef.-mor) (bbus=acc>* 1) (acc=sum) ; u1*mh11 msb
    (mor=mem) (addr dataptr) (offset 13) (xbus=0) (eabus=sum)); rd(u2[0])
   ((mor=mem) (addr dataptr) (offset 15) (xbus=0) (eabus=sum)
; rd(mh12[0])
    (abus=mor) (acc=abus)                                      ; acc=u2[0]
    (rbus=acc) (r*=rbus 0)) ; r0=u1*mh11
   )

; Torque-computation (ii)
; compute q1 and output it

  (block28
   ((shrcoef) (abus=coef.-mor) (bbus=acc>* 1) (acc=sum) ; u2*mh12 msb
    (mbus=r* 0) (mor=mbus)) ; mor=r0 [u1.mh11]
   ((abus=mor) (bbus=acc>* 0) (acc=sum) ; acc=u2.mh12+u1.mh11
    (mor=mem) (addr dataptr) (offset 20) (xbus=0) (eabus=sum)) ; rd(f1)
   ((abus=mor) (bbus=acc>* 0) (acc=sum) ; acc=acc + f1, [q1]
    (mor=mem) (addr dataptr) (offset 7) (xbus=0) (eabus=sum)) ; rd(u1[0])
   ((mbus=acc) (ioport=mbus) (extport=ioport 3) ; output q1
    (mem=mbus) (addr dataptr) (offset 18) (xbus=0) (eabus=sum)) ;w(q1)
   ((mor=mem) (addr dataptr) (offset 16) (xbus=0) (eabus=sum) ; rd(mh21[0])
    (abus=mor) (acc=abus)) ; acc=u1[0]
  )

; Torque-setup (iii)
; u1*mh21 (msb) => r0
; setup for mh22*u2
;
  (block29
   ((shrcoef) (abus=coef.-mor) (bbus=acc>* 1) (acc=sum)        ; u1*mh21
msb
    (mor=mem) (addr dataptr) (offset 13) (xbus=0) (eabus=sum)); rd(u2[0])
   ((mor=mem) (addr dataptr) (offset 17) (xbus=0) (eabus=sum)
; rd(mh22[0])
    (abus=mor) (acc=abus)                                      ; acc=u2[0]
    (rbus=acc) (r*=rbus 0))                                    ; r0=u1*mh21
   )
```

```
; Torque-compute (iii)
; q2 = u2.mh22 + u1.mh21 + f2
; output q2
;

  (block30
   ((shrcoef) (abus=coef.-mor) (bbus=acc>* 1) (acc=sum) ; u2*mh22 msb
    (mbus=r* 0) (mor=mbus))                             ; mor=r0 [u1.mh21]
   ((abus=mor) (bbus=acc>* 0) (acc=sum)                 ; acc=u2.mh22+u1.mh21
    (mor=mem) (addr dataptr) (offset 21) (xbus=0) (eabus=sum)) ; rd(f2)
   ((abus=mor) (bbus=acc>* 0) (acc=sum)                 ; acc=acc + f2,
[q2]
    (mbus=acc) (ioport=mbus) (extport=ioport 4)         ; output q2
    (mem=mbus) (addr dataptr) (offset 19) (xbus=0) (eabus=sum)) ; w(q2)
  )

;
; Output-data setup
; x0 gets the ptr address for the state variables
; x1 gets the iteration count - number of variables.
;
 (block31
  ((addr dataptr) (xbus=0) (eabus=sum) (x*=eabus 0))  ; x0=dataptr
  ((addr 22) (xbus=0) (eabus=sum) (x*=eabus 1)) ; x1=22 (loop cnt)
  ((mor=mem) (addr dataptr) (xbus=0) (eabus=sum)) ; rd(1st var)
 )
;
; Output-data
; Routine to output state variables
; This is put in a loop to send out all the variables.
; In this case the apu counter is being used instead of the pcu lpc.
; This requires a two cycle block.
;
  (block32
   ((mbus=mor) (ioport=mbus) (extport=ioport 9)  ; output mor
    (addr -1) (xbus=x* 1) (eabus=sum)  (x*=eabus 1)) ; decr x1
   ((mor=mem) (addr 1) (xbus=x* 0) (eabus=sum) (x*=eabus 0))  ; rd(next
var)
  )

;
; Delay variables
```

```
; Implements delay of variables
;
  (block33
   ((mor=mem) (addr dataptr) (offset 7) (xbus=0) (eabus=sum)) ; rd(u1[0])
   ((mor=mem) (addr dataptr) (offset 14) (xbus=0) (eabus=sum) ; rd(mh11[0])
    (abus=mor) (acc=abus)) ; acc=u1[0]
   ((mbus=mor) (mem=mbus) (addr const) (offset 12) (xbus=0) (eabus=sum)
; w(mh11[1])
    (bbus=acc<* 1) (acc=bbus)) ; u1[0] << 1
   ((mor=mem) (addr dataptr) (offset 13) (xbus=0) (eabus=sum)) ; rd(u2[0])
   ((abus=mor) (acc=abus) ; acc=u2[0]
    (mbus=acc) (mem=mbus) (addr u1T1) (xbus=0) (eabus=sum)) ; w(u1[1])
   ((mor=mem) (addr dataptr) (offset 15) (xbus=0) (eabus=sum)
; rd(mh12[0])
    (bbus=acc<* 1) (acc=bbus)) ; u2[0] << 1
   ((mbus=acc) (mem=mbus) (addr u2T1) (xbus=0) (eabus=sum)) ; w(u2[1])
   ((mbus=mor) (mem=mbus) (addr const) (offset 13) (xbus=0) (eabus=sum))
; w(mh12[1])
   ((mor=mem) (addr dataptr) (offset 16) (xbus=0) (eabus=sum))
; rd(mh21[0])
   ((mbus=mor) (mem=mbus) (addr const) (offset 14) (xbus=0) (eabus=sum))
; w(mh21[1])
   ((mor=mem) (addr dataptr) (offset 17) (xbus=0) (eabus=sum))
; rd(mh22[0])
   ((mbus=mor) (mem=mbus) (addr dataptr) (offset 15)
(xbus=0) (eabus=sum))  ; w(mh22[1])
  )

;
; Read velocity
; Read in A/D ports and store the velocity data
; Also check for inner loop count, L
;
  (block34
   ((ioport=extport 1) (rbus=ioport) (r*=rbus 0)) ; r0=xv1
   ((ioport=extport 2) (rbus=ioport) (r*=rbus 0) ; r0=xv2
    (mbus=r* 0) (mem=mbus) (addr dataptr) (offset 2) (xbus=0) (eabus=sum))
; w(xv1)
   ((xbus=x* 2) (addr -1) (eabus=sum) (x*=eabus 2)) ; decr x2
   ((mbus=r* 0) (mem=mbus) (addr dataptr) (offset 8) (xbus=0) (eabus=sum))
   )
; w(xv2)
```

```
;
; Idle block
; Used for doing noops while timer completes its count
;
  (block35
   ((nop))
  )
;
; multiplication routine
; loop counter will count shift-accumulate iterations.
; lsb accum is done in previous block and msb accum is done in following
block
  (block5
    ((shrcoef) (abus=coef.mor) (bbus=acc>* 1)  ; acc=coef.mor + acc>1
     (acc=sum) (nosat))
   )

)
```

Appendix G

Controller Simulation Programs

Two C programs are given in this appendix. The first one implements the adaptive controller and the second one implements a robot arm model. The controller calls the arm model as a subroutine. These programs were used to simulate the control algorithms.

```
;*************************************************************************
; C program for simulating the adaptive controller.
;*************************************************************************
;
#include <stdio.h>
#include <math.h>
#include "controller.h"

FILE *fclose(), *fopen(), *farm1, *farm2, *fothers, *fref1, *fref2;
main ()
{

int    i,j;
int .  K, L, N;
int    CI;
double step;
double evthresh;
double T1; /* inner loop sample period */
double u1[2], u2[2];
double uv1, uv2;
double ev1, ev2;
double em1, em2; /* velocity error wrt to ref. model */
double *xp1, *xp2, *xv1, *xv2;
double q1, q2; /* computed torque */
```

```
double ref1, ref2; /* reference input to PID controller */
double ep1, ep2; /* position error */
double w1[2], w2[2];
double ki1, kp1, kv1; /* pid coeffs. */
double ki2, kp2, kv2;
double km11, km12, km21, km22;
double xvh1, xvh2;
double mh11[2], mh12[2], mh21[2], mh22[2];

xp1 = (double *) calloc(1, sizeof(double)); /* measured position */
xp2 = (double *) calloc(1, sizeof(double)); /* measured position */
xv1 = (double *) calloc(1, sizeof(double)); /* measured velocity */
xv2 = (double *) calloc(1, sizeof(double)); /* measured velocity */

Cntrl_files(); /* open input and output files */

/* Input parameters */
printf("Enter sample period, T1 :  ");
scanf("%lf", &T1);
printf("Enter number of inner loop sample points per T1, L : ");
scanf("%d", &L);
printf("Enter number of integration steps per T1, N :  ");
scanf("%d", &N);
printf("Enter total number of outer sample points desired, K :  ");
scanf("%d", &K);

step = T1/N;
kp1 = KPa*KPb; kp2 = KPa*KPb;
ki1 = KI; ki2 = KI;
kv1 = KVa*KVb; kv2 = KVa*KVb;
km11 = KM11; km12 = KM12; km21 = KM21; km22 = KM22;
evthresh = DEADBAND;
CI = CIVALUE; /* I-controller option */

/*#################################################################*/
/*   FRICTION COMPENSATION NOT INCLUDED IN THIS IMPLEMENTATION */
/*#################################################################*/

/* initialize states */
*xp1 = 0; *xp2 = 0; *xv1 = 0; *xv2 = 0;
mh11[1] = MH11; mh12[1] = MH12; mh21[1] = MH21; mh22[1] = MH22;
w1[1] = 0; w2[1] = 0;
```

```
/* BEGIN_OUTERLOOP  */
for (j=0; j < K; j++) {

/* ## */
if (j==55) {
printf("debug stop \n");
}

fscanf(fref1,"%lf", &ref1);
fscanf(fref2,"%lf", &ref2);

/* P-control */
         ep1 = ref1 - *xp1;
         ep2 = ref2 - *xp2;
uv1 = kp1*ep1;
uv2 = kp2*ep2;

/* I-control */
if (CI == SET) {
w1[0] = ep1 + w1[1];
w2[0] = ep2 + w2[1];
uv1 = ki1 * w1[0] + uv1;
uv2 = ki2 * w2[0] + uv2;
}
/* BEGIN_INNERLOOP */

for (i=0; i < L; i++) {
/* D-control */
ev1 = uv1 - *xv1;
ev2 = uv2 - *xv2;
u1[0] = kv1 * ev1;
u2[0] = kv2 * ev2;

/* vel error wrt ref model and check for dead zone */
em1 = xvh1 - *xv1;
em2 = xvh2 - *xv2;
if (abs(em1) < evthresh) em1 = 0;
if (abs(em2) < evthresh) em2 = 0;

/* ref. model */
xvh1  = u1[0] * T1 + *xv1;
```

```
xvh2  = u2[0] * T1 + *xv2;

/* inertia estimation */
mh11[0] = km11 * em1 * u1[1] + mh11[1];
mh12[0] = km12 * em1 * u2[1] + mh12[1];
mh21[0] = km21 * em2 * u1[1] + mh21[1];
mh22[0] = km22 * em2 * u2[1] + mh22[1];

/* torque computation */
q1 = mh11[0]*u1[0] + mh12[0]*u2[0];
q2 = mh21[0]*u1[0] + mh22[0]*u2[0];

/* robot arm solution */
robot(N,step,q1,q2,xp1,xp2,xv1,xv2);

/* output data to files */
fprintf(farm1,"%lf  %lf  %lf  %g  %lf  %lf \n",ref1,*xp1,*xv1, q1, ep1,
uv1);
fprintf(farm2,"%lf  %lf  %lf  %g  %lf  %lf \n",ref2,*xp2,*xv2, q2, ep2,
uv2);
/* fprintf(fothers,"%lf  %lf  %lf  %lf  %lf  %lf \n",ev1, u1[0], xvh1,
em1, mh11[0], mh12[0]); */
fprintf(fothers,"%lf  %lf  %lf  %lf  %lf  %lf \n",mh21[0], mh22[0], ev2,
u2[0], xvh2, em2);

/* update states and refresh constants in ram */
w1[1] = w1[0]; w2[1] = w2[0];
mh11[1] = mh11[0]; mh12[1] = mh12[0];
mh21[1] = mh21[0]; mh22[1] = mh22[0];
u1[1] = u1[0]; u2[1] = u2[0];
}
}

Close_files();

}
;
;*************************************************************************
; C program for implementing the robot arm
;*************************************************************************
;
#include <stdio.h>
```

```
#include <math.h>
#include "nskparm.h"

/*************************************************/
/* Solves the differential equations describing */
/* a two axis robot arm. Uses the 4th order     */
/* Rungekutta method.          */
/*             -Khalid Azim 5-7-87 */
/*************************************************/

double Cv1, Cv2, m11, m12, m21, m22, mm;
extern FILE *fothers;


robot(N,delh,q1,q2,lxp1,lxp2,lxv1,lxv2)
int N;
double delh, q1, q2, *lxp1, *lxp2, *lxv1, *lxv2;

{
int i;
double Fv1(), Fv2();
double Cm11a, Cm11b, Cm12a, Cm12b, Cm22;
double xp1, xp2, xv1, xv2, pxp1, pxp2, pxv1, pxv2;
double g1,g2,g3,g4,j1,j2,j3,j4,k1,k2,k3,k4,r1,r2,r3,r4;
double m2, m3, m4, m5, l1, l2, I1, I2, I3, I4;

m2 = M2; m3 = M3; m4 = M4; m5 = M5;
l1 = L1; l2 = L2; I1 = II1; I2 = II2; I3 = II3; I4 = II4;

Cv1 = -1*(m4/2 + m5)*l1*l2; /* constant for v1 expr. */
Cv2 = (m4/2 + m5)*l1*l2; /* constant for v2 expr. */

/* constants used in inertia expr.*/
Cm11a = I1 + I2 + I4 + l1*l1*(m2/4 + m3 + m4 + m5) + l2*l2*(m4/4 + m5);
Cm11b = l1*l2*(m4 + 2*m5);
Cm12a = I4 + l2*l2*(m4/4 + m5);
Cm12b = l1*l2*(m4/2 + m5);
Cm22 = I3 + I4 + l2*l2*(m4/4 + m5);

xp1 = *lxp1;
xp2 = *lxp2;
xv1 = *lxv1;
```

```
xv2 = *lxv2;

for (i=0; i < N; i++) {

/* arm inertia */
m11 = Cm11a + Cm11b*cos(xp2);
m12 = Cm12a + Cm12b*cos(xp2);
m21 = m12;
m22 = Cm22;
mm = m11 * m22 - m12 * m12;
/*
fprintf(fothers, "%lf  %lf  %lf  \n", m11, m12, m22);
*/

/* Rungekutta computation */
pxv1 = xv1;
pxv2 = xv2;
pxp2 = xp2;
g1 = delh*pxv1;
j1 = delh*pxv2;
k1 = delh*Fv1(q1, q2,pxv1,pxv2,pxp2);
r1 = delh*Fv2(q1,q2,pxv1,pxv2,pxp2);

pxv1 = xv1 + k1/2;
pxv2 = xv2 + r1/2;
pxp2 = xp2 + j1/2;
g2 = delh*pxv1;
j2 = delh*pxv2;
k2 = delh*Fv1(q1, q2,pxv1,pxv2,pxp2);
r2 = delh*Fv2(q1, q2,pxv1,pxv2,pxp2);

pxv1 = xv1 + k2/2;
pxv2 = xv2 + r2/2;
pxp2 = xp2 + j2/2;
g3 = delh*pxv1;
j3 = delh*pxv2;
k3 = delh*Fv1(q1, q2,pxv1,pxv2,pxp2);
r3 = delh*Fv2(q1, q2,pxv1,pxv2,pxp2);

pxv1 = xv1 + k3;
pxv2 = xv2 + r3;
pxp2 = xp2 + j3;
```

```
g4 = delh*pxv1;
j4 = delh*pxv2;
k4 = delh*Fv1(q1, q2,pxv1,pxv2,pxp2);
r4 = delh*Fv2(q1, q2,pxv1,pxv2,pxp2);

/* evaluate xp1, xp2, xv1, xv2 for this iteration */
xp1 = xp1 + (g1 + 2*g2 + 2*g3 + g4)/6;
xp2 = xp2 + (j1 + 2*j2 + 2*j3 + j4)/6;
xv1 = xv1 + (k1 + 2*k2 + 2*k3 + k4)/6;
xv2 = xv2 + (r1 + 2*r2 + 2*r3 + r4)/6;
}

/* save states */
*lxp1 = xp1;
*lxp2 = xp2;
*lxv1 = xv1;
*lxv2 = xv2;
return;
}

double Fv1(aq1, aq2, axv1,axv2,axp2) /* computes dxv1/dt  */
double aq1, aq2, axv1, axv2, axp2;
{
double  v1, v2, fv1;
v1 = Cv1*(2*axv1 + axv2)*axv2*sin(axp2);
v2 = Cv2*(axv1*axv1)*sin(axp2);
fv1 = (m22*(aq1 - v1) - m12*(aq2 - v2))/mm;
return(fv1);
}

double Fv2(aq1, aq2, axv1,axv2,axp2) /* computes dxv2/dt  */
double aq1, aq2, axv1, axv2, axp2;
{
double v1, v2, fv2;
v1 = Cv1*(2*axv1 + axv2)*axv2*sin(axp2);
v2 = Cv2*(axv1*axv1)*sin(axp2);
fv2 = (m11*(aq2 - v2) - m21*(aq1 - v1))/mm;
return(fv2);
}
```