UIUCDCS-R-75-761

AN ARRAY COMPUTER FOR THE CLASS OF PROBLEMS TYPIFIED BY THE GENERAL CIRCULATION MODEL OF THE ATMOSPHERE

BY

Marvin L. Graham and D. L. Slotnick

December 1975

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN · URBANA, ILLINOIS

Report No. UIUCDCS-R-75-761

AN ARRAY COMPUTER FOR THE CLASS OF PROBLEMS TYPIFIED BY THE GENERAL CIRCULATION MODEL OF THE ATMOSPHERE

by

Marvin L. Graham and D. L. Slotnick

December 1975

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

*This work was supported in part by NASA Goddard Space Flight Center under Grant No. US NASA NAS-5-23334 and was submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science, December 1975, for Marvin L. Graham.

TABLE OF CONTENTS

1. Introduction

The goal of the research described by this paper was the design of a computer suited to the class of problems typified by the general circulation model of the atmosphere. The research was supported in large part by the Goddard Institute for Space Studies (GISS) of the National Aeronautics and Space Administration (NASA). The needs that prompted GISS to support the research imposed several practical constraints on the design which was sought. A fundamental goal was that the machine which resulted from the design was to have roughly 100 times the computing capability of the GISS IBM 360/95 which is now used for research with a general circulation model. Their desire to increase the spatial resolution of that model by refining the grid implied the need for a 100 fold increase in computing capability to stay even in terms of the real time.

A second requirement was that the resulting machine be programmable in a higher level language similar to FORTRAN. The current model is written almost entirely in FORTRAN, and the GISS staff planned to modify an existing compiler for CFD - a FORTRAN-like language - for ILLIAC IV for use with their new machine. Moreover, the new machine was to cooperate in the general circulation experiments on the expanded models with the IBM 360/95; the IBM machine would continue to be used for the pre-processing and post-processing of model data which it now performs for the smaller model which it also now executes. The implication of the FORTRAN and IBM machine constraints is that the machine possess floating point arithmetic capability, and that the floating point format of the machine be close to that of the IBM 360 series.

A third constraint on the design was that the cost of the machine resulting from the design effort was to be significantly less than that of

other extant machines of similar computing capability. Among these are the ILLIAC IV, the Texas Instruments Corporation Advanced Scientific Computer, and the Control Data Corporation STAR.

A final constraint on the design was that it be feasible to fabricate a complete system and put it in operation by early 1978. A clear implication of this and the preceding constraint is that there is neither time nor money for the development of new hardware families, let alone new chips. The design will have to be made in terms of an existing hardware family with components readily available off-the-shelf.

2. The Problem

Several groups in the United States are working on global general circulation models. The three largest efforts are those of Mintz and Arakawa at UCLA (Arakawa, 1972; Mintz, 1974), Smagorinsky and Manabe at the Geophysical Fluid Dynamics Laboratory (GFDL) (Smagorinsky, 1963) and Kasahara and Washington at the National Center for Atmospheric Research (Kasahara, 1967). The UCLA model is of primary interest to this research because the model run by GISS (Tsang, 1973) is a modified form of that model.

2.1 General Circulation Models

A general circulation model simulates the behavior of a three dimensional spherical atmosphere on a digital computer. The bulk of the computing load necessary in the simulation is the time integration of the equations of fluid dynamics of the atmosphere. In the UCLA model, subroutines called COMP1 and COMP2 perform this time integration of the equations of motion. Every six cycles through COMP1-COMP2, the effects of solar radiation in heating the atmosphere and the effects of evaporation, condensation and precipitation are introduced through the execution of the COMP3 and COMP4 subroutines. The process is shown in Figure 2.1.2-1. Every four cycles through the process illustrated by Figure 2.1.2-1, a table look-up process is used to introduce the effects of long-wave infra-red energy absorbtion.in the GISS model.

Table 2.1-1 lists the parameters which define the conditions under which the model operates. Table 2.1-2 lists the variables of the model and gives their spatial dimensions. Figure 2.1-1, which is taken from a GISS

Prescribed parameters.

To use the atmospheric general circulation model, for this or any other planet, the following parameters must be prescribed:

Radius, surface gravity and rotation speed of the planet.

Solar constant, and orbital parameters of the plant.

Total atmospheric mass.

Thermodynamical and radiation constants.

Geographical distributions of open ocean, ice covered ocean, bare land and land covered by glacial ice.

Elevation of the bare land and glacial ice.

Surface roughness.

Thickness of the sea ice.

Ocean surface temperature.

Table 2.1-1. The Parameters of the General Circulation Model

Variables of the Atmospheric Model

Horizontal Velocity	
West to East component	$U(X,Y,Z)$
South to North component	$V(X,Y,Z)$
Temperature	$T(X,Y,Z)$
Water Vapor (specific humidity)	$q(X,Y,Z)$
Surface Atmospheric Pressure	$p_0(X,Y)$
Parameters of the Planetary Boundary Layer (PBL)	
Boundary Layer Depth	(X,Y)
Temperature Discontinuity at the PBL	(X,Y)
Moisture Discontinuity at the PBL	(X,Y)
Parameters of the Earth's Surface	
Ground Temperature	(X,Y)
Ground Water Storage	(X,Y)
Mass of Snow on the Ground	(X,Y)

A Future Variable of the Atmospheric Model

Ozone Concentration	(X,Y,Z)

Table 2.1-2 The Variables of the General Circulation Model and their Dimensionalities

$$\frac{dV}{dt} + fk \times V + \nabla_\sigma \Phi + \sigma\alpha\nabla\pi = F$$

$$\frac{\partial\pi}{\partial t} + \nabla_\sigma \cdot (\pi V) + \frac{\partial}{\partial\sigma}(\pi\dot{\sigma}) = 0$$

$$p\alpha = RT$$

$$\frac{d\theta}{dt} = \frac{1}{c_p}\frac{\theta}{T}Q$$

$$\frac{1}{\pi}\frac{\partial\Phi}{\partial\sigma} = -\alpha$$

$$\frac{dq}{dt} = -C+E$$

Here the notation is

V	horizontal velocity
t	time
f	Coriolis parameter
k	vertical unit vector
∇_σ	two-dimensional gradient operator
σ	the vertical coordinate [= $(p-p_t)/(p_s-p_t)$]
p	pressure
p_t	pressure at top of model atmosphere, constant
p_s	pressure at bottom of model atmosphere
α	specific volume
π	$p_s - p_t$
F	horizontal frictional force
R	gas constant
T	temperature
θ	potential temperature
c_p	specific heat at constant pressure
Q	heating rate per unit mass
Φ	geopotential
q	water vapor mixing ratio
C	rate of condensation
E	rate of evaporation.

Figure 2.1-1 The Primitive Equations and the Variables of the GISS General Circulation Model.

report on the model (Somerville, 1974), shows the basic equations of the model. The remainder of this section will describe the UCLA and GISS models. The emphasis will be on describing the differences between the first UCLA model (Arakawa, 1972), the GISS model which evolved from it (Somerville, 1974; Tsang, 1973) and the second UCLA model (Mintz, 1974) to illustrate the range over which variations of the current GISS model may run in future models.

2.1.1 Vertical Levels

The first UCLA model had only three vertical levels. The current GISS model has nine, and the second UCLA model has twelve. GISS hopes to expand to a fifteen level model. The new UCLA model incorporates a special "sponge layer" as its highest level to damp out spurious numerical wave reflections (Mintz, 1974).

2.1.2 Time

The first UCLA model and the GISS models use the explicit matsuno predictor-corrector method for advancing time. For a variable Q, the scheme uses a forward and a backward step to advance time by one interval in the following way:

$$\text{Forward} \qquad \frac{\hat{Q}(t_{n+1}) - Q(t_n)}{t_{n+1} - t_n} = f'(Q(t_n))$$

$$\text{Backward} \qquad \frac{Q(t_{n+1}) - Q(t_n)}{t_{n+1} - t_n} = f'(Q(t_{n+1})^*)$$

The forward step uses the current values of the variable and the function f', which approximates the derivative, to produce an estimate, $\hat{Q}(t_{n+1})$, for the value of the variable at the next time. The backward step uses the estimated value to compute $Q(t_{n+1})$, the value of the variable at the next time. The process is illustrated by Figure 2.1.2-1.

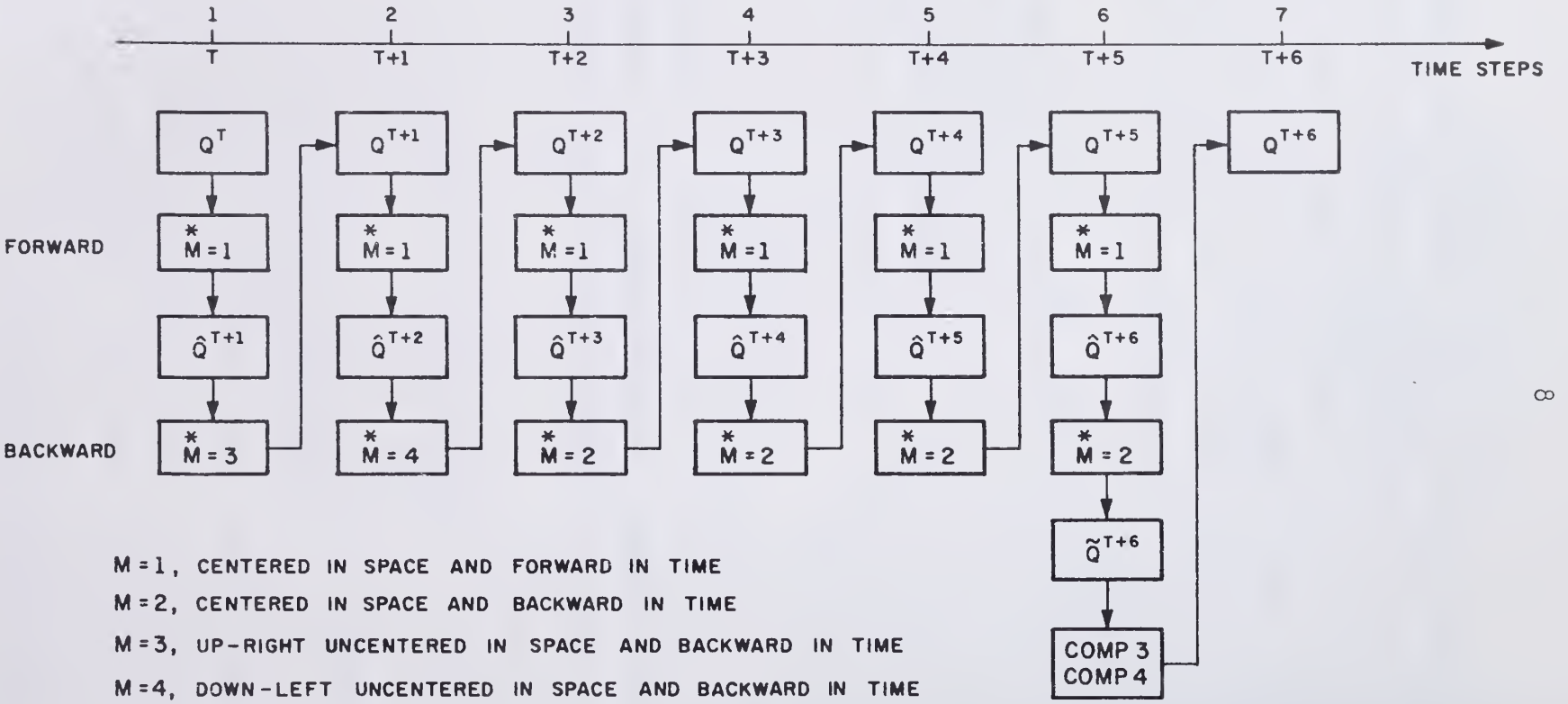


Figure 2.1.2-1 The Sequence of Time Steps and Spatial Differencing in the Time Integration Procedure

The GISS version of the model for the IBM 370/165 takes advantage of the fact that only one complete copy of the variables is needed for this method to reduce the storage requirements of the model by roughly half.

The new UCLA model uses the leapfrog scheme to advance time. This scheme computes a value for the variable A at time t_{n+1} as follows:

$$\frac{A(t_{n+1}) - A(t_{n-1})}{2(t_{n+1}) - t_n} = f'(A(t_n)).$$

This scheme takes half the computer time, but requires twice the space of the Matsuno scheme, since two complete sets of the variables are required to compute a new value. The leapfrog scheme is numerically superior to the Matsuno scheme in that it does not amplify or damp the solution, but it is inferior in that it tends to produce two separate and divergent solutions. The new UCLA model will couple these two solutions by introducing one Matsuno step for every six leapfrog steps.

Figure 2.1.2-1, taken from Tsang (1973), shows the sequence of computation in the current UCLA and GISS models. Each normal time step conisists of a COMP1-COMP2 call for a forward (estimator) time step and another COMP1-COMP2 call for a backward (corrector) step. Every six normal steps, the effects of solar radiation and evaporation are computed by a call on COMP3 and COMP4. The value of the variable M determines which form of the difference algorithm will be used in the COMP1-COMP2 routines. The following section discusses the need for the spatial difference variations.

2.1.3 Horizontal Resolution and Various Differencing Schemes

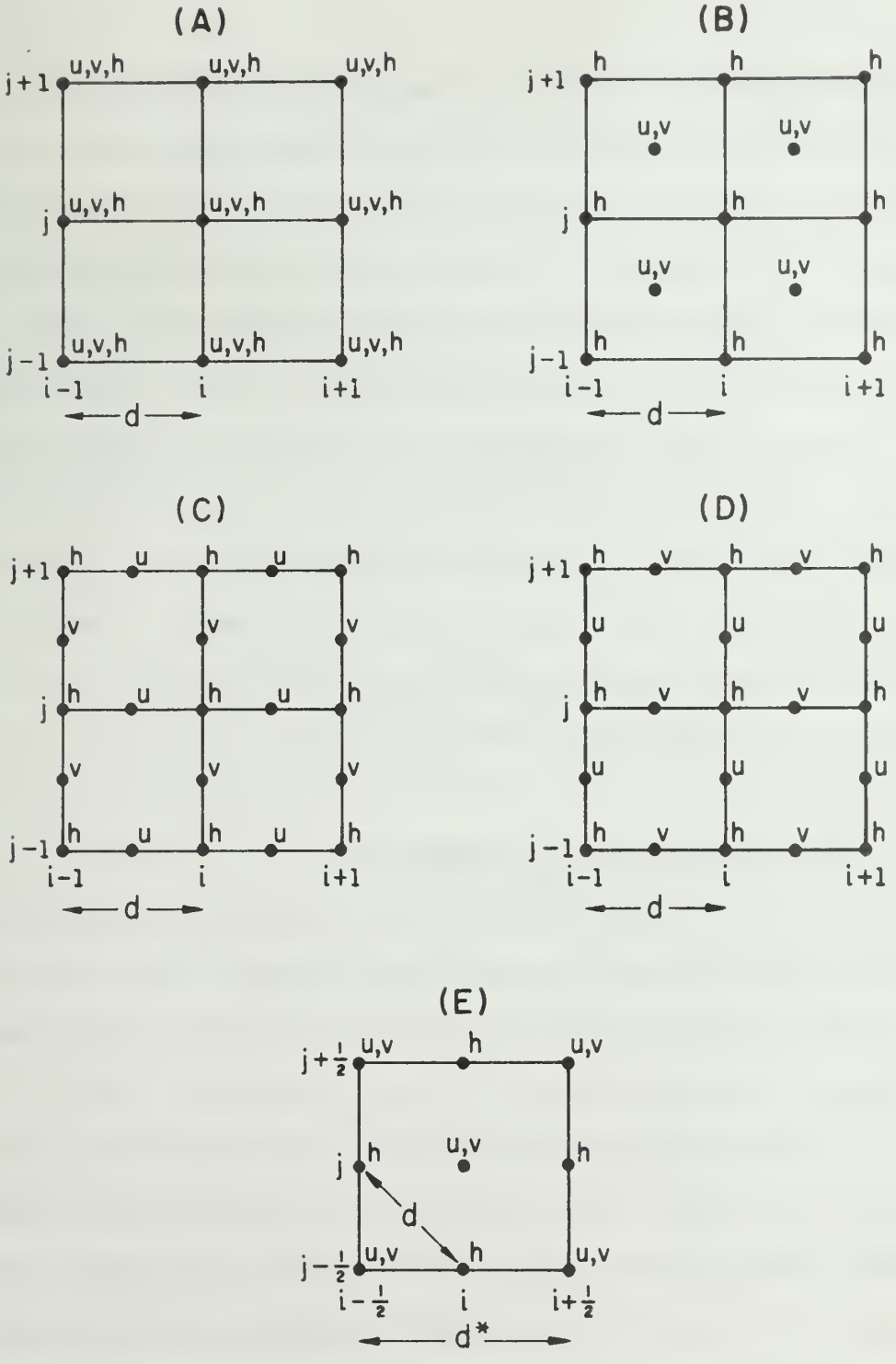
Both UCLA models and the most frequently used version of the GISS model have 72 points around circles of latitude, and 46 circles of latitude

from pole to pole (including the poles). For the next decade GISS is interested in models of two different sizes for the proposed computer (Halem, 1974). Both models will have 15 vertical levels (i.e., 15 spherical shells) and differ only in the number of points around the equator of the model. The two sizes of interest are:

1. A model with 128 points around the equator and with 96 circles of latitude. We will call this the 96 x 128 grid.
2. A model with 256 points around the equator and with 192 circles of latitude. We will call this 192 x 256 grid.

All of the models use a stagered grid system, which stores the values of the primary meteorological variables at different points in space. Figure 2.1.3-1, which is taken from (Mintz, 1974), shows five grid schemes which have been considered. The first UCLA model and the current GISS model use scheme B. Arakawa has decided to use scheme C in the new UCLA model. The basis for this decision, which follows in the next paragraph, illustrates the intricacy of the model.

Convection of moisture from the earth's surface to high altitudes, called cumulus convection , is an important atmospheric phenomenon, especially in the tropics. The scale of this motion is tens of kilometers; the distance between grid points at the equator is 156 kilometers even for the 256 point model. Arakawa found a means to parameterize cumulus cloud convection so that its effects could be felt by the model in spite of the fact that direct simulation - as the model does for winds, temperature and specific humidity - is not possible. The parameterized cumulus convection produces rising and subsiding air motion which frequently occurs in a



u: the west to east component of the horizontal flow
v: the south to north component of the horizontal flow
h: the distance from the surface to the top of the atmosphere in the model

Figure 2.1.3-1 Staggered Grid Schemes

checkerboard pattern. To use scheme B for the grid layout, one must average the values of pressure at the corners of each grid square to compute the effect of pressure on the flow fields. Rising motion at one corner is cancelled by subsidence at another, and the net effect is that the cumulus convection goes unnoticed by the model. Arakawa devised the intricate time and space difference scheme shown in Figure 2.1.2-1 (taken from Tsang, 1973) to counteract this insensitivity. The differencing scheme uses a cycle of space centered and uncentered differences to permit the checkerboard pattern produced by cumulus convection to influence the model. When grid scheme C is used, these elaborate gyrations are unnecessary. Primarily for this reason, Arakawa has decided that scheme C will be used in the next UCLA model. The current model, which is the basis for the GISS work, uses scheme B.

2.2 GISS Modifications to the Model

Several modifications of the UCLA model were made by GISS. Only one of these has a major impact on this research. This is the distinctly different approach to the treatment of high latitude regions which GISS has adopted, and which they call the split grid model.

The meridian lines on a sphere get progressively closer as they approach the poles. The Courant stability criterion (Fox, 1961), $c\,\Delta t < \Delta x$, where c is the highest velocity in the model, requires that a very small time step be used to avoid numerical instability in these regions. The UCLA approach to this problem is to smooth across progressively wider bands of meridional lines as the meridians get closer together. The GISS approach is to progressively reduce the number of meridians by a factor of two as one moves from the equator to the poles. This divides the sphere into several

regions as illustrated in Figure 2.2-1. Within each region, the number of meridians is constant. The region boundaries are chosen to keep the inter-meridian distance roughly constant for all regions. In the split grid model, the need for zonal smoothing is much reduced but not completely eliminated. Table 2.2-1 shows the number of split grid regions for grids with different numbers of points on the equator.

Meridians at the equator	Number of split grid regions
72	5
128	7
256	11
512	15

Table 2.2-1. The Number of Split Grid Regions for Various Model Sizes

The split grid model offers two advantages over the UCLA smoothing approach. The first is that a larger time step can be used throughout the model, since the smallest increment in the "x" direction is larger in a split grid model. Also, there is a potential storage saving for the split grid model. The split grid scheme does have the liability that it is more difficult to program.

Whether a rectangular UCLA-style model or a GISS split grid model is used, some averaging of polar values must be done. Thus, there is a clear inherent parallelism in the processing which strongly suggests parallel computation on circles of constant latitude.

2.3 The Effects of the Oceans on the Atmosphere

Until recently, meteorologists have assumed that the effects of the oceans on heat transfer from the equator to the high latitude regions was negligible. Lately, however, this view has changed, as evidenced by the

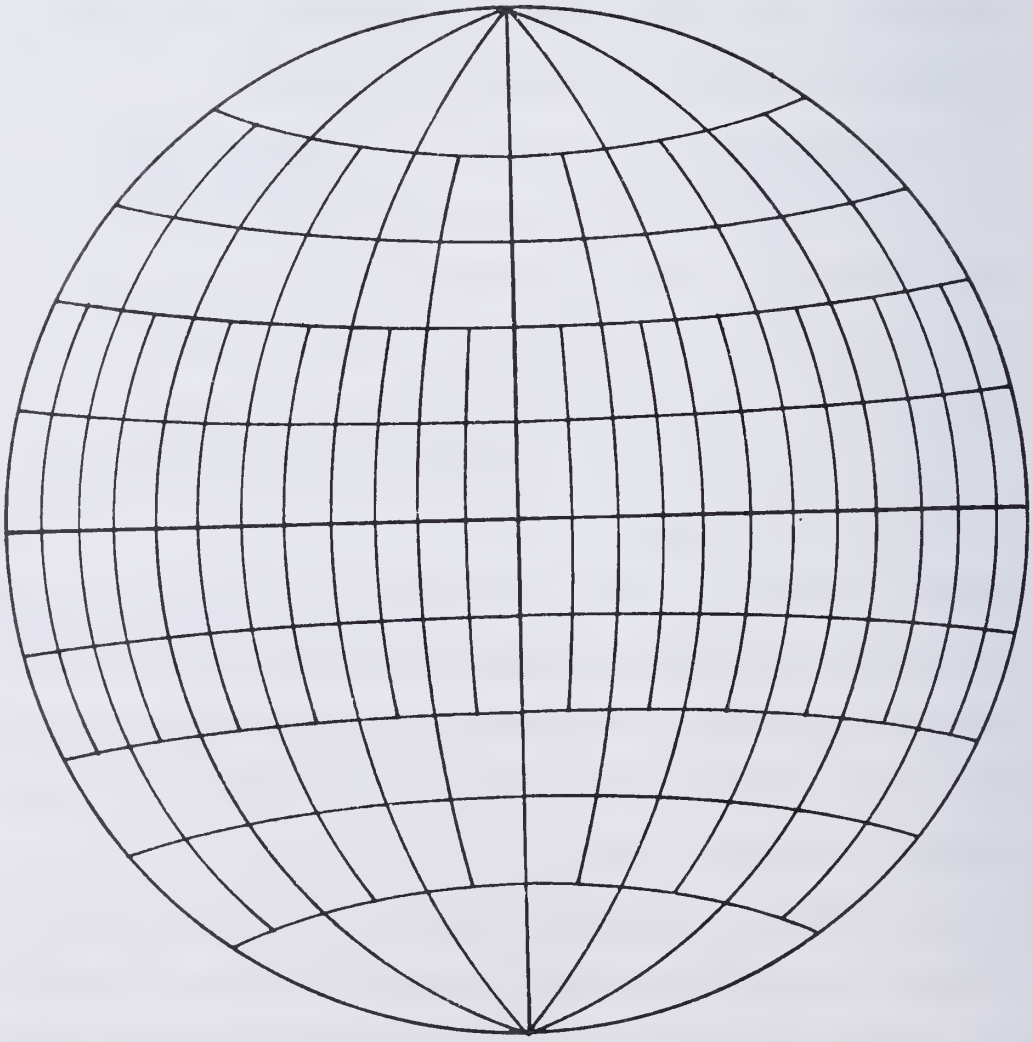


Figure 2.2-1 The GISS Split Grid Model

relatively large emphasis on ocean modelling at the UCLA workshop (Mintz, 1974), and by the decision of the UCLA group to couple an ocean model and an atmospheric model in a future model. Whereas the atmospheric equations are integrated in time by explicit numerical methods, Semptner of the UCLA staff indicated that all known ocean models advance time by successive over-relaxation - an implicit method (Semptner, 1974). He also feels that IBM 360 single precision arithmetic is sufficient for solving the system of equations for a 46 x 72 grid.

Semptner also cited work at GFDL (Manabe, 1969) which indicates that integration of the atmospheric equations consumes 40 times the amount of computer time as does integration of the ocean equations for the same simulated time. This dramatic difference results from the differences in the two fluids, and the fact that the implicit solution scheme permits the use of significantly larger time step than an explicit scheme would.

While it is clear that an ocean model will be required to improve current results, it is not clear what the details of the ocean model must be. Recent observations and numerical work (Mintz, 1974) have shown the existence of small scale (40-50 kilometer) ocean phenomena. Whether these are important, and if so, whether their effects can be parameterized (as was cumulus convection in the atmospheric model) is yet to be shown. The potential need for an ocean model coupled to the atmospheric model will be most explicitly reflected in the size of memory that we recommend.

2.4 Input and Output Requirement of the Model

The proposed mode of operation for the new machine is that it receive its program and initial data from the GISS IBM 360/95 by using an IBM channel with a data handling capacity of $6(10)^6$ bits per second.

A problem thus received would be run in stand-alone fashion by the machine with periodic dumps of model status. The current GISS model writes an output record every two hours of simulated model time for a 46 x 72 x 9 grid. Table 2.4-1 shows the variables which constitute these records, the sizes of the records for a 46 x 72 x 9, 96 x 128 x 15, and a 192 x 25 x 15 grid, the lower bound on the elapsed time to write the record using the channel at its maximum rate, and an estimate of the computing time required for the new machine to compute two hours of model simulation.

	BYTES		
DATA	46 x 72 x 9	96 x 128 x 15	192 x 256 x 15
TAU	4	4	4
C(300)	1,200	1,200	1,200
Q(NS,EW,V,4)	476,928	2,949,120	11,796,480
P (NS,EW) TS (NS,EW) SHS(NS,EW) GT (NS,EW) CW (NS,EW)	13,248	49,152	196,608 each
Total	544,372	3,196,084	12,780,724
Transmission Time	0.726	4.26	17.04 Seconds
Estimated Computation Time	0.031	0.39	3.15 Seconds

Table 2.4-1 Record Sizes and Transmission times for Various Grid Sizes

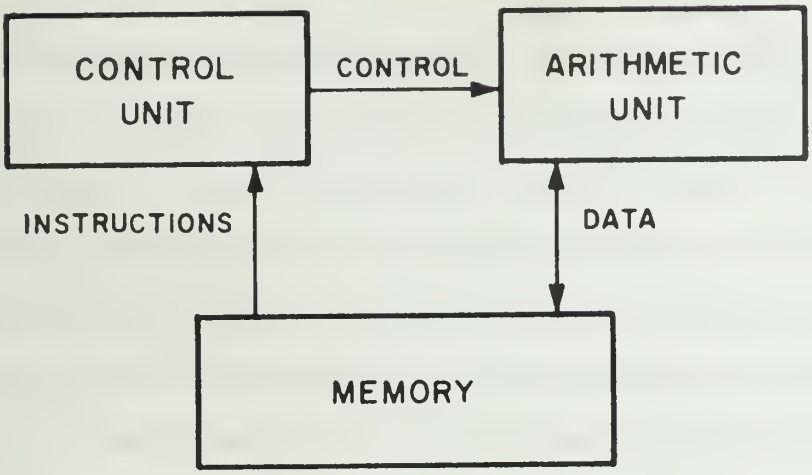
As Table 2.4-1 makes clear, data output from the model will have to come at less than two hour simulated time intervals if the machine is not to become heavily output bound. It is doubtful that channel transmission capacity can be increased nearly enough to reduce to output time significantly with respect to the computation time.

3. The Array Computer

A computing capability improvement by a factor of 100 over the capability of the 360/95 is a big order. In the time span specified for the development of this design, there is no hope of achieving this improvement purely by increased raw hardware speed. Indeed, physical realities such as the bound imposed by the speed of propagation of electromagnetic waves may make this path forever impossible. Clearly, if the capability increase can be achieved, it must be achieved by using a machine organization different from that of the 360/95.

The approach we shall take is to organize the machine as an array processor. Applications research (Carroll, 1967) for an early array processor, the SOLOMON (Slotnick, 1962), has shown that the array processor organization is ideally suited to the class of problems that the general circulation model typifies: solution of partial differential equations on a large grid. Indeed, the GISS general circulation model has been successfully converted for execution on the ILLIAC IV (Slotnick, 1968), the only operational large scale array processor.

Figure 3-1 contrasts the organization of an array processor with that of a conventional computer. In a conventional machine, control hardware (shown in the figure collected into one functional block and labelled the Control Unit) interprets the instruction stream and provides signals which control the operation of the rest of the hardware, collected into the block called the Arithmetic Unit. In most conventional machines, both the instructions and the data are stored in one memory. In most conventional computers, as suggested above, the control and arithmetic - or execution -



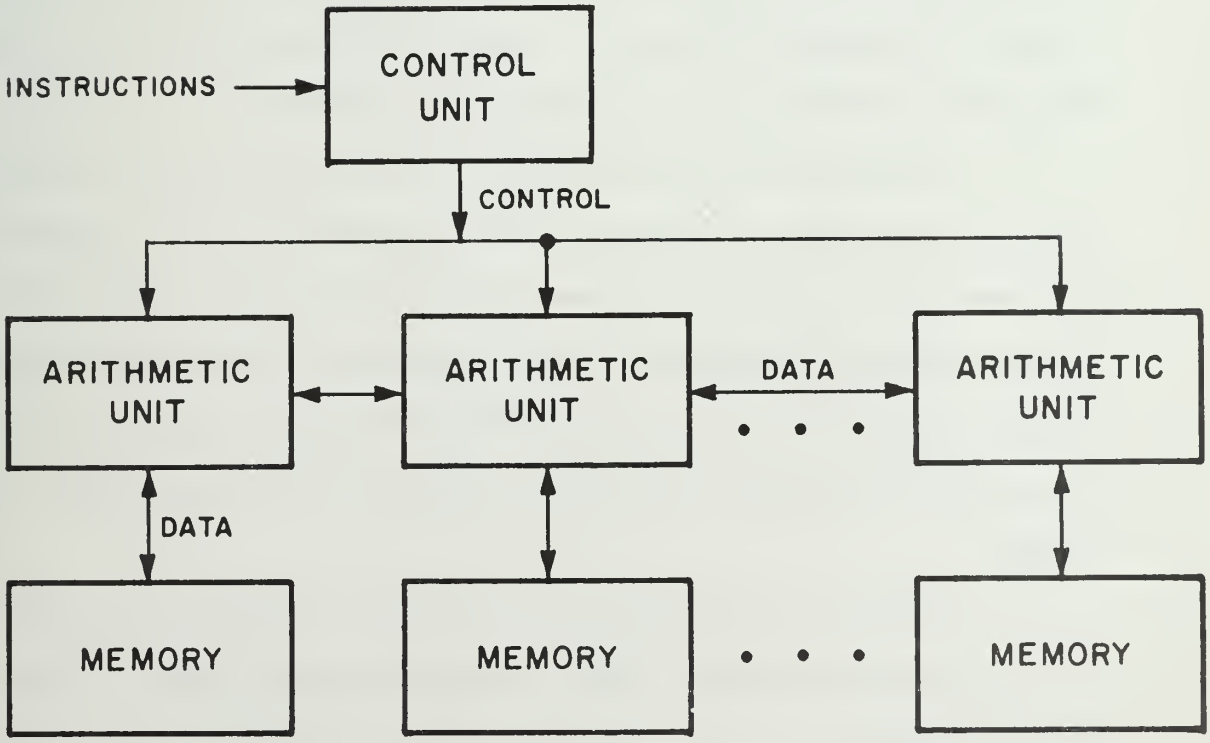


Figure 3-1 The Basic Structures of a Classical Computer and an Array Computer

functions are seldom as clearly separated as the figure suggests. In the array computer, however, the control and execution functions are clearly separated. The arithmetic unit is replicated many times (1024 in the SOLOMON (Slotnick, 1962) and 64 in the ILLIAC IV (Slotnick, 1968)), and the data memory is divided so that each of the arithmetic units operates on its own data stream under the control of one common program. In a conventional computer, conditional tests on data values in the single data stream alter the flow of the single instruction stream. In the array processor, residual local control in the processors of the array permits conditional tests on data to allow individual processors to skip executing instructions. In a standard technique for controlling iterations, the control unit samples the activity status of the processors in the array, and stops the iteration when all of them become inactive.

Application studies reported by Kuck (Kuck, 1968) have shown that another local control feature is a vital element in an array processor. The ability of each processor to index a control unit supplied data address permits much more flexible use of the processors in the array. In the general circulation model, processor level indexing is necessary to support the table look up process used in the radiation calculation phase of the model.

Virtually all problems for which array processors are suited require that the processors in the array exchange data values. In the SOLOMON computer, the 1024 processors were arranged in a square thirty-two processors on a side, and each processor could access the memories of its four nearest neighbors in addition to its own. The sixty-four processors

of the ILLIAC IV are also arranged in a square, and each processor can receive values from its nearest four neighbor processors. In the design described in this paper, we use a separate routing network model after the suggestions of Lawrie (Lawrie, 1973) which permits much more flexible interprocessor communication. Figure 3-2 shows the design described in the remainder of this paper in block form. The machine includes a control unit, 256 array processors and their memories, and a sixteen unit three stage routing network.

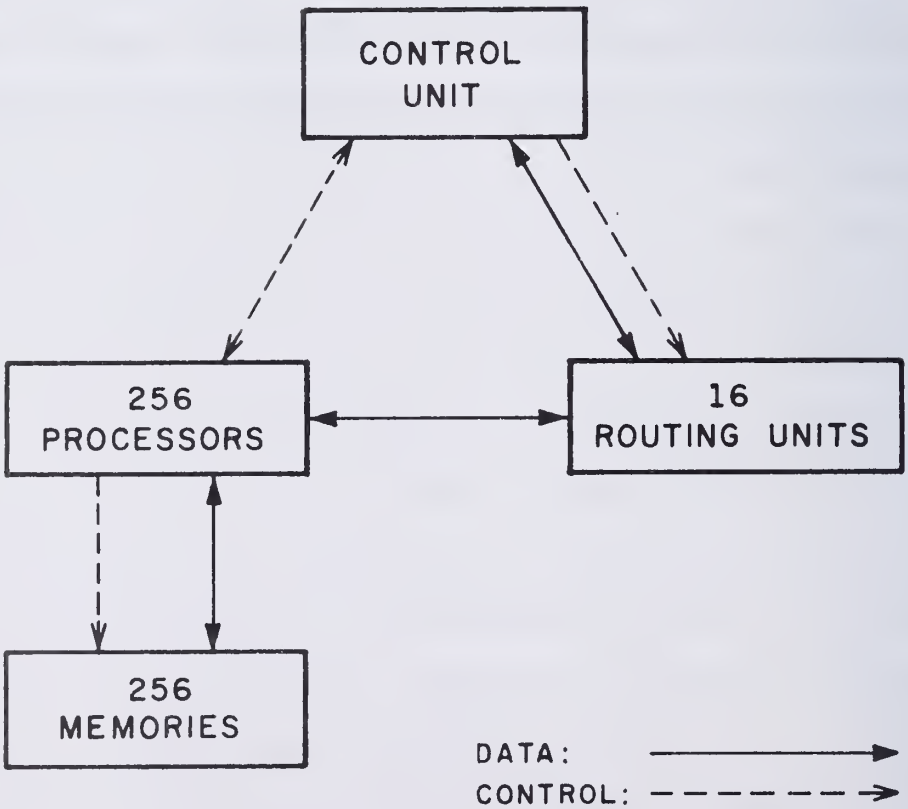


Figure 3-2 Block Diagram of the System

4. The System Design

The following sections will describe the system design. The initial sections will establish the important parameters of the design. Subsequent sections will discuss the arithmetic processor, routing network, and control unit of the system.

4.1 System Parameters

In this group of sections, the basis for the word length, memory size, and other basic system parameters choices are given.

4.1.1 Word Size

The UCLA and GISS models run in single precision of the IBM System/360 (Arakawa, 1972; Tsang, 1973). Williamson and Washington of the National Center for Atmospheric Research (NCAR) performed precision experiments with the NCAR model (Williamson, 1973). Normally, the CDC machines on which that model runs operate on a forty-eight bit fraction. Through software means, they ran twenty-four and twenty-one bit test cases, and compared the result with a forty-eight bit control runs. They concluded that "the lower-precision arithmetic planned for the next generation of computers [that is, twenty-four bit fractions] does not seriously affect the results from the current NCAR [five degree, six layer] global circulation model." Dr. Larry Gates of the Rand Corporation has recently rescinded his decision to run the Rand modification of the UCLA model in double precision (Gates, 1975). He said that difference between single and double precision test runs are well within the so-called "predictability error" for hydrodynamics calculations discussed by Lorentz (Lorentz, 1963).

On the basis of the above information, we have decided that single precision arithmetic is sufficient for the execution of the model.

4.1.2 Word Format

The system was designed to operate in conjunction with IBM series 360 computers at GISS. Data preprocessing steps to prepare input for the system and data post processing steps to analyze the results of experiments will be done on the IBM equipment. Programming for the system is to be in a FORTRAN-like higher level language, so that floating point operation is required. Because of the cooperation required between the system and the 360, it was decided to make the floating point format of the mcahine the same as that of the 360 (IBM, 1970). The floating point format for the design is shown in Figure 4.1.2-1. A floating point word is represented in sign magnitude form by a one bit sign, a seven bit exponent, and a twenty-four bit fraction. A zero sign bit is used for non-negative numbers. The seven bit exponent field contains a biased representation for exponent vlaues between minus sixty-four and plus sixty-three inclusive. The proper representation for an exponent value is found by adding the value to the bias, sixty-four. Thus, for example, an exponent field value of 41 base sixteen represents an exponent value of plus one. The magnitude part ot the number is a proper fraction; that is, the exponent is an implicit binary point at the left of the most significant fraction bit. The exponent field represents the power of sixteen which must multiply the fraction to correctly express the value of the floating point number as a whole. Because the exponent radix is sixteen, a change of one in the exponent value requires a shift of four bit positions in the fraction to represent the same numerical value. Thus, the twenty-four bit fraction can be regarded as a six hexidecimal digit fraction; each hexidecimal digit is represented by four continguous bits of the fraction, and shifts of the fraction are made in multiples of four bit positions.

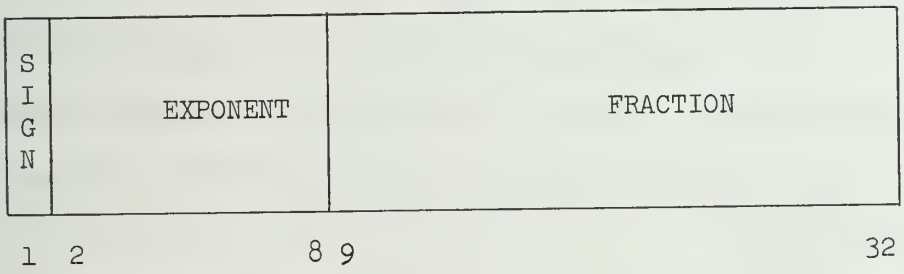


Figure 4.1.2-1 The Floating Point Word Format

4.1.3 Memory Requirements

Based on experience with the cost of development of special high data rate disk systems which we obtained with ILLIAC IV, we decided that the memory of the machine should be large enough to contain all of the data. The memory requirement was estimated by running the COMMON for the 360/95 model through the IBM FORTRAN/H compiler. Space for four three dimensional variables (two velocity components, salinity and temperature) and one two dimensional variable (the vertically averaged stream function) of an eventual ocean model was added for the 96 x 128 and 192 x 256 models.

Because that machine would have a program memory separate from its data memory for the processor array, space for the program is not included in the following estimates. Table 4.1.3-1 displays the amount of memory required for several sizes of the model, including the 96 x 128 and 192 x 256 models with oceans.

	words of memory	
NS x EW x Z	no ocean	7 level ocean
82 x 128 x 15	1,378,411	--
96 x 128 x 15	1,613,289	1,969,641
128 x 200 x 15	3,358,601	--
256 x 401 x 15	13,457,305	--
164 x 256 x 15	5,506,125	--
192 x 256 x 15	6,445,385	7,870,793

Table 4.1.3-1

The machine should be built with 2^{23} words of memory to accommodate the 192 x 256 grid. Each of the 256 processor memories would have 2^{15} (or 32768) words. Each of these words will contain thirty-two information bits

and six Hamming code bits (Hamming, 1950) for detection and correction of single bit errors. The decision to include error detection and correction hardware was taken on the advice of the staff of the University of Illinois Physics department. They have constructed semi-conductor memory for their computer, and found that the error detection and correction bits which they included were well worth while, both in terms of improved system operation and increased maintainability (Downing, 1974).

4.1.4 Measurements of the GISS Model

To discover the relative importance of multiplication and the frequency of double precision operations in the execution of the model, the GISS model was run for one time step on the University of Illinois' 370/158 under the control and observation of a program which computes the frequencies of all instructions executed by the program it observes. A series of runs was made to permit instruction counts for the important parts of the model to be determined. Execution times for these parts of the model were determined by the GISS staff (Karn, 1974) during a one man year effort which produced an ILLIAC IV version of the GISS model. Table 4.1.4-1 shows the number of instructions executed in each of three parts of the model, the 360/95 time for execution of those parts, and the instruction processing rate of the 360/95. Table 4.1.4-2 gives the frequencies for single and double precision floating point multiplications and divisions in the parts of the model.

Approximately half of the instructions executed were floating point instructions. These were nearly equally divided between addition and subtraction on one hand and multiplication and division on the other. The

Part of the Model	Instructions	360/95 Time	360/95 Rate
Initialization	11,891,631		
COMP1-COMP2	69,480,878	10.3 sec.	6.75 MIPS
COMP3	43,505,137	6.54 sec.	6.65 MIPS

Table 4.1.4-1 Measurement Values

360 Instruction	Initialization	COMP1-COMP2	COMP3
MDR	1	423,936	132,480
MD	330	16	103,765
MER	756	2,221,358	823,153
ME	2,134	4,022,947	2,056,291
DDR	3	105,984	33,120
DD	1	0	0
DER	77	359,584	615,025
DE	1,773	440,950	929,372

Table 4.1.4-2 Instruction Counts

ratio of multiplications to divisions (weighting COMP1-COMP2 by six to account for the more frequent use of these routines in normal model execution) is 6.15 multiplications to one division. The vast majority of the double precision floating point operation are performed by one assembly language subroutine which raises a number to a constant power. This routine uses double precision because the speed of single and double precision operations on the IBM 360/95 is the same. An approximation formula with a few more terms can be used without requiring any double precision.

On the basis of the above information, we decided to design a single precision processor whose floating point addition and multiplication times are comparable. Double precision operations will be performed on the single precision hardware of the design relatively slowly since they occur with such low frequency.

4.1.5 Processor Speed Requirements

The system is to have roughly one hundred times the processing capability of the IBM 360/95 for the weather model. As we saw in section 4.1.4, the 360/95 executes approximately $6.7\ (10)^6$ operations per second on the GISS general circulation model. We have already decided that the machine we design will be an array processor with an architecture similar to that of ILLIAC IV. How many processors should the machine have? To achieve $6.7(10)^8$ operations per second, a 256 processor machine must perform one operation in 382 nano-seconds; a 512 processor machine need only perform one operation in 764 nano-seconds. On the other hand, as we will see in section 4.3 - which discusses the routing network - it is important to have the number of processors be a perfect square: 256 is the

square of sixteen, but 512 is not a perfect square. Moreover, a 256 processor machine will be more reliable and have a higher availability than a similar 512 processor machine. Therefore, we will design a machine with 256 processors. We would, therefore, like the operation time for a processor to be on the order of 400 nano-seconds.

4.1.6 The Choice of TTL Technology for the Processor

It was clear from the outset that the time and budget constraints on the design necessitated using an existing integrated circuit technology, and in fact a family which is currently commercially available "off the shelf". The choice must be either TTL, MOS, or ECL (Hnatek, 1973). A higher level of integration (that is, more powerful individual packages is avalable in the TTL family than is available in the ECL family. Moreover, the new Schottky variant of TTL logic is nearly as fast as ECL. The speed of MOS logic is far slower than that of even standard TTL. A floating point processor with a fast multiplier will surely require using several hundred integrated circuits in its design. Fewer high level packages are required than low level packages to achieve the same functions, and package savings pay off in both board and interconnection savings. Therefore, we chose to design the processor in terms of TTL integrated circuits.

Package savings in the processor design result from the use of two different package interconnection properties of two different special forms of TTL logic. These are discussed in the following two sections.

4.1.6.1 Open Collector Logic and the Wire AND

A standard TTL output stage is shown in Figure 4.1.6.1-1. The active pull-up provided by transistor Q1 is that it permits faster operation

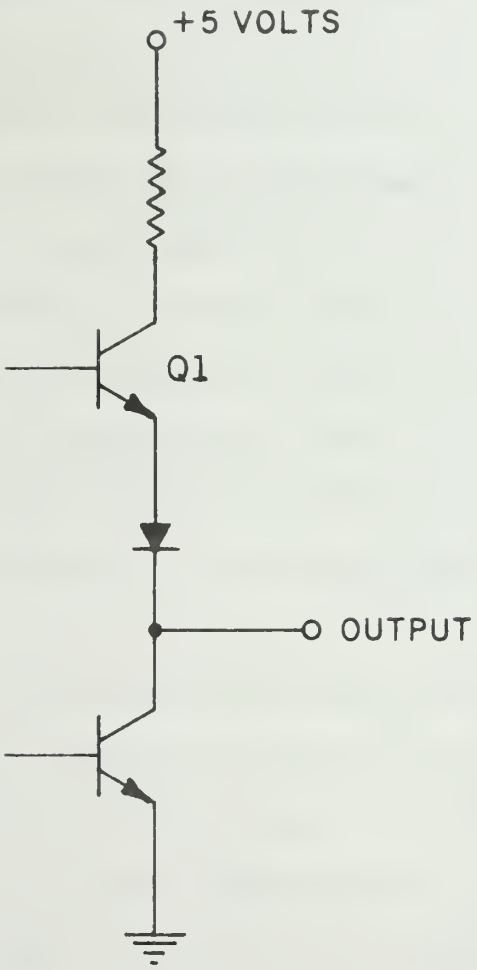


Figure 4.1.6.1-1 The Standard TTL Totem Pole Active Pull-up Output Stage

than that of the resistor-transistor (RTL) or diode-transistor (DTL) families from which the TTL family evolved. The passive output stage of Figure 4.1.6.1-2 of the DTL family is used in some of the slower of the TTL integrated circuits. Deletion of the pull-up resistor of the passive output stage results in the so-called output collector output. Open collector outputs of several packages can be wired together through a common external pull-up resistor. If all of the output signals so wired together are logic ones, each circuit will source less than one milliamp so the resulting current flow for the entire collection of wire ANDed circuits results in a logic one. However, if one or more of the wire ANDed output signals is a logic zero, the corresponding circuits will sink on the order of forty milliamps, so that the resulting voltage level of the ensemble falls to that of a logic zero.

Within the processor, the open collector outputs of the Signetics 8243 eight position scalers used in the right operand alignment shift logic and the normalization left shift logic are wire ANDed together. An enable signal for the device permits forcing all eight output signals to logic ones regardless of the state of the eight input signals. One of the two shift networks is enabled at a time, so that its output bits, ANDed with ones of the disabled device, determine the net output of the ensemble.

4.1.6.2 Tri-state Logic and the Wire OR

The National Semiconductor Corporation holds the patents for another output control technique which they refer to by the registered trademark "tri-state" logic. Standard TTL circuits augmented by the National technique have an enabling input which can be used to force the

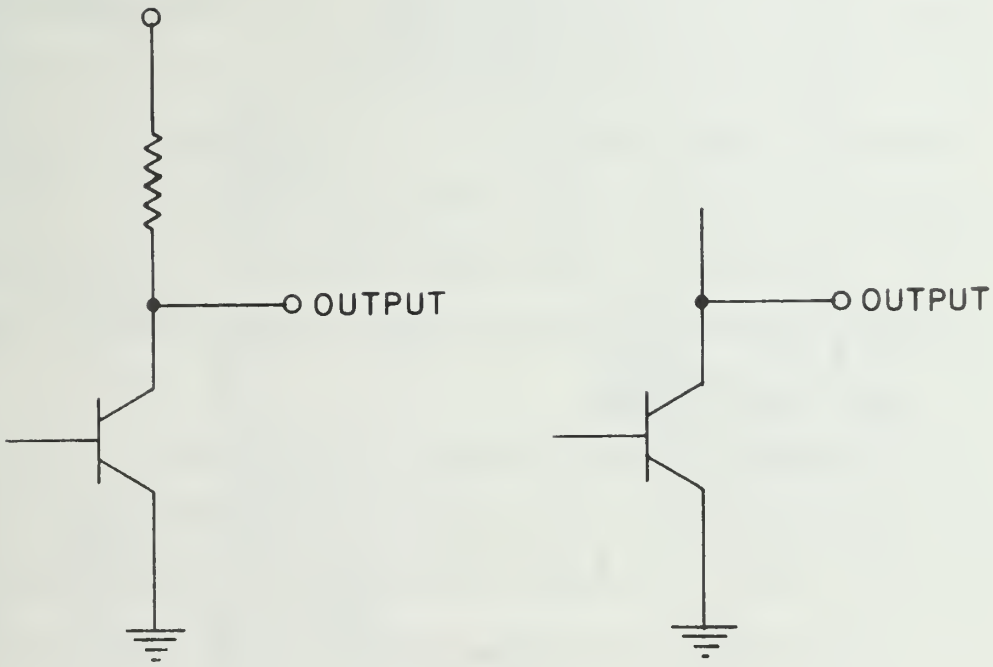


Figure 4.1.6.1-2 TTL Passive Pull-up and Open Collector Output Stage

outputs of the device to a high impedance state (Hnatek, 1973). The output impedance of a standard TTL output is nominally fifty ohms. The output impedance of a disabled tri-state output is nominally 50,000 ohms. Thus, if several tri-state outputs are wired together and all but one of them are disabled, the current into or out of the disabled outputs is negligible compared to that for the one enabled output. Up to one hundred or more tri-state outputs can be wired together on a single bus. The resulting wired connection is usually referred to as a wired OR, and its logic state is determined by the logic state of the enabled output.

The processor design makes extensive use of tri-state devices to reduce the need for selectors between otherwise competing signals.

4.2 The Processor Design

A simplified block diagram of the processor is shown in Figure 4.2-1. The names in the blocks of this figure (with the exception of the 2/1 Selector blocks) are the names of the Figure or Figures which present the logic of that block in more detail. Each of these blocks is described in detail in the following sections.

Multiplication is performed by logic external to that shown in Figure 4.2-1. The two twenty-four bit operands to be multiplied are sent to the multiplier as shown, and both the most and least significant halves of the product are returned. See section 4.2.5.2.4 and (Stenzel, 1975) for a detailed description of the multiplier.

The processor as a whole is a large combinatorial circuit which is conditioned by control signals from the control unit. It operates in steps governed by one clock pulse. A typical cycle begins with operand selection.

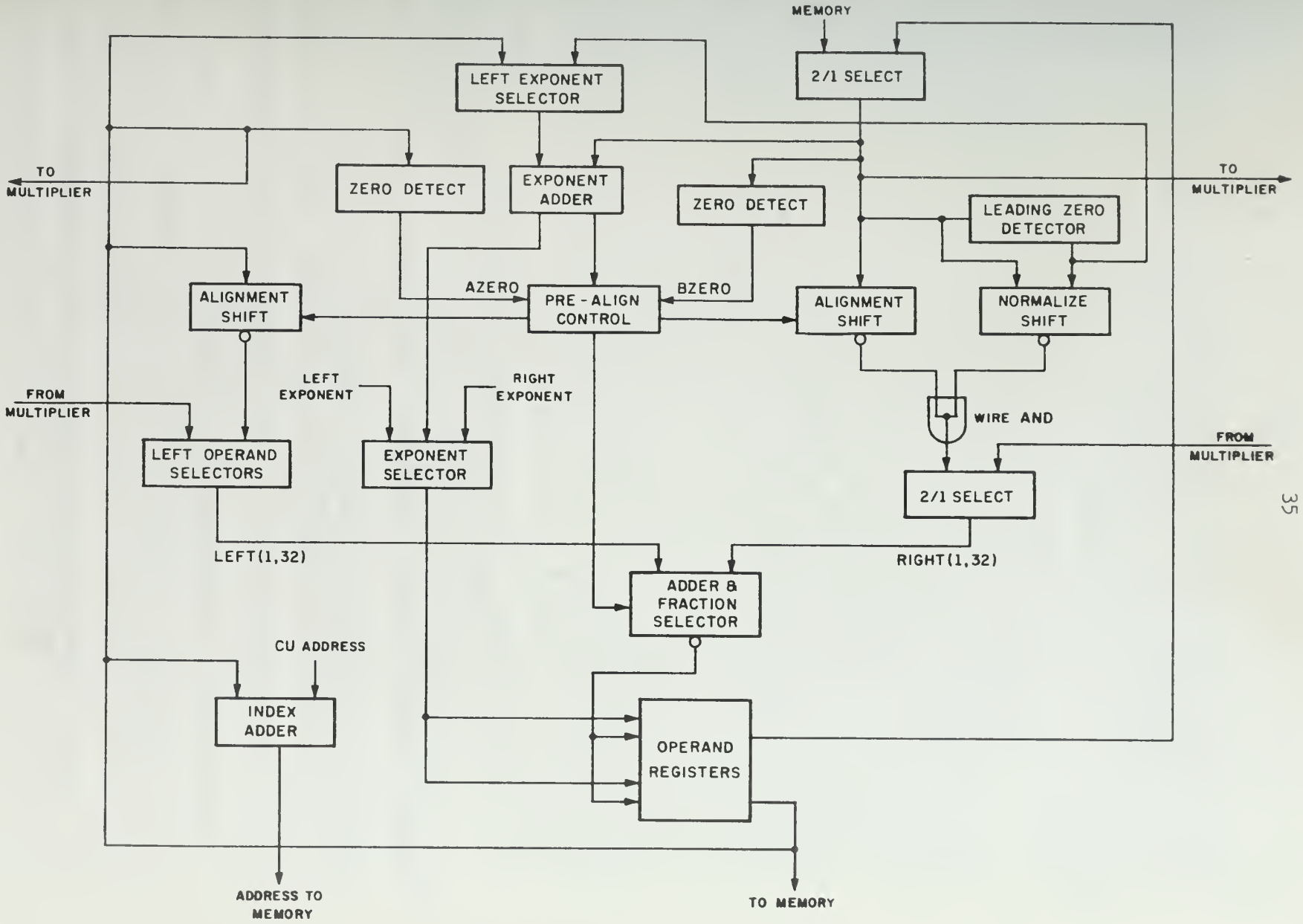


Figure 4.2-1 Block Diagram of the Processor

Two operands, one of which may come from memory, flow through the paths in the logic selected by the set of control signals. At the completion of a cycle, result values are clocked into the registers specified by the set of control signals.

In any logic design, options are available at many stages. The rules governing the choice among options in this design can be qualitively stated as follows: minimize cost and package count, but not at the expense of time in the critical path. Cost is reflected not only in the direct cost of the packages, but also by the amount of board area (and hence the number of boards) which the packages occupy. Minimizing the number of boards can lower overall cost by reducing the need for backplane wiring or mother boards and eliminate the need for inter-board connections. The board area for a package was assumed to be proportional to the number of pins which the package has. Although this assumption is not strictly true, it serves well as an operation rule of thumb when making design choices.

4.2.1 Conventions Used in the Figures Which Describe Logic

Designing computer hardware in terms of existing integrated circuit packages differs from computer design in terms of discrete components. In many cases, the designer working with integrated circuits finds that no existing package exactly suits the need of the moment. What he must then do is make the best compromise he can with the packages which are available, according to the general guidelines which he has adopted.

The simplest example of the above general comment is that it often happens that an N-input gate of some type is needed. A concrete example in this design is that a four input OR gate is needed by the logical demands of the function to be implemented. What are available are two input OR gates

and two, four, and five input NOR gates. Among these gates, only - the five input NOR gate - is available in Schottky form. When the desired logic function is in a time-critical path, the highest speed element should be used. Hence, one finds himself using a five input gate for a four input function. Many instances of such use occur in this design. When they occur in the figures, only the number of inputs which are required for the logic function being implemented are shown. The extra leads which may exist are assumed to be connected to sources of logic ones or zeros as necessary. For example, the extra input of the above five input NOR gate would have to be connected to a constant logic zero source to guarantee the correct operation of the logic in which it is used.

Detailed documentation for the integrated circuits used in this design can be found in four industry data books. In the description which follows, the following notation given in Table 4.2.1-1 was used for naming components.

Form of the Name	Source for Detailed Information
SN74xxxx	The TTL Data Book for Design Engineers, First Edition, Document Number CC-411, Texas Instruments Incorporated, 1973.
	Supplement to the TTL Data Book for Design Engineers, First Edition, Document Number CC-416. Texas Instruments Incorporated, 1974.
SIGxxxx	Signetics Digital, Linear, MOS Data Book, Signetics Corporation, 1974.
AMxxxx	Advanced Micro Devices Data Book, Advanced Micro Devices Incorporated, 1974.
NATxxxx	Digital Integrated Circuits, National Semiconductor Corporation, 1974.

Table 4.2.1-1 The Notation for Package Names in the Logic Design Figures

4.2.2 Signal Name Notation Used in the Design Description

In the description of the design in the follwoing sections, signals will be named by an identifier of eight or less capital letters and digits. The first character of a signal name will be a letter. Multi-bit signals are named by a single identifier to which bit specifications are appended. A bit specification is a list of up to three integers separated by commas and enclosed in parentheses. The bits of multi-bit signals are numbered from one for the most significant to N for the least significant bit of an N bit signal. A bit specification which consists of a single integer specifies the single bit of the multi-bit signal with that integer as its bit number. In a bit specification with two integers, the first specifies the bit number of the most significant bit of the signal and the second specifies the number of contiguous bits in the signal. The third integer of a three integer bit specification is the difference between successive bit numbers in the specified signal. Table 4.2.2-1 gives several examples of signal names.

Signal Name	Meaning
A	the one bit signal A
B(3)	bit three of the multi-bit signal B
B(1,32)	bits one through thirty-two of the multi-bit signal B
B(5,4)	bits five through eight of the multi-bit signal B
C(1,2,4)	bits one and five of the multi-bit signal C

Table 4.2.2-1 Several Examples of Signal Names

This notation for signal names is used consistantly throughout the text and figures which describe the design. It is also used for signal names in the input language for the logic simulation package described in section 5.1. In the truth tables which follow, a lower case "x" signifies that the package described by the truth table operate correctly for any value of the signal represented by the "x".

4.2.3 Inversion in the Logic Figures

When the function of an integrated circuit includes the logical complement of the inputs, this is shown by a small circle external to the rectangle which represents the integrated circuit. The alignment shift blocks of Figure 4.2-1 are an example of an inverting block.

4.2.4 Detailed Description of Two Packages

Two packages, the Texas Instruments SN74S157 and the Signetics 8263, are described in detail in this section. Two reasons motivate these detailed descriptions. First, these packages are typical of most of the integrated circuits which are used in this design. Second, and perhaps more important, these particular packages perform critical functions in the design. All of their features are exercised, so that a full understanding of the design is impossible without a full understanding of these two packages.

4.2.4.1 The Texas Instruments SN74S157

The Texas Instruments SN74S157 is a quadruple two-to-one selector. It accepts two four bit input operands and a one bit selection signal and produces a four bit output. The output is the four bit input designated by the selection signal. There is one more input, however. A one bit strobe signal can be used to force the outputs to zeros without regard to the input

signals. There are several occasions in the design where the strobe signal is used to good advantage. The truth table for the SN74S157 is given in Table 4.2.4.1-1.

Inputs				Output
Data		Selection	Strobe	
x	x	x	1	0
A(1,4)		0	0	A(1,4)
x	B(1,4)	1	0	B(1,4)

Table 4.2.4.1-1 The Truth Table for the Texas Instruments SN74S157

4.2.4.2. The Signetics 8263

The Signetics 8263 is a quadruple three-to-one selector. It accepts three four bit input operands, a two bit selection signal, and a one bit complement signal, and produces a four bit output. The output is the four bit input designated by the selection signal. The two bit selection signal can specify one of four input signals; the fourth state is used to set the output to zero without regard to any of the input signals. The complement signal can be used to specify that the output is to be the logical complement of the selected input. The truth table for the Signetics 8263 is given in Table 4.2.4.2-1.

Inputs					Output
Data			Selection	Complement	
X	X	X	00	0	0000
A(1,4)	X	X	01	0	A(1,4)
X	B(1,4)	X	10	0	B(1,4)
X	X	C(1,4)	11	0	C(1,4)
X	X	X	00	1	1111
A(1,4)	X	X	01	1	$\overline{A(1,4)}$
X	B(1,4)	X	10	1	$\overline{B(1,4)}$
X	X	C(1,4)	11	1	$\overline{C(1,4)}$

Table 4.2.4.2-1 The Truth Table for the Signetics 8263

4.2.5 The Processor Design

In the two sections which follow, the design of the processor is completely described. The first of these sections describes functional logic blocks in their own right without regard to the contributions which those blocks make in the operation of the processor. The second section describes how the processor performs normalization, rounding, floating point addition/subtraction, floating point double precision addition/subtraction, floating point multiplication, and finally floating point division. This section relies on an understanding of the former sections describing the various logic blocks. It describes the control logic which is necessary to integrate the operation of those logic blocks to perform the desired operations.

4.2.5.1 Logic Blocks

The following sections describe several logic elements which perform definite functions in support of larger operations in the processor.

4.2.5.1.1 The Zero Detect Logic

A zero detect logic block produces the logical OR of thirty-two bits. Three instances of the zero detect block occur. In all three cases, the thirty-two input bits constitute a thirty-two bit operand fraction. Figure 4.2.5.1.1-1 depicts the zero detect logic. The packages used are four SN74S260 dual five-input positive NOR gates and one SN74S133 thirteen-input positive NAND gate. Each of the NOR gates is used to produce the NOR of four input fraction bits. The eight results are combined by the NAND gate to yield the desired OR of the thirty-two input bits.

In Figure 4.2.5.1.1-1, the four bit groups shown as inputs to the NOR gates represent four bit digits of a fraction. In only one of the three

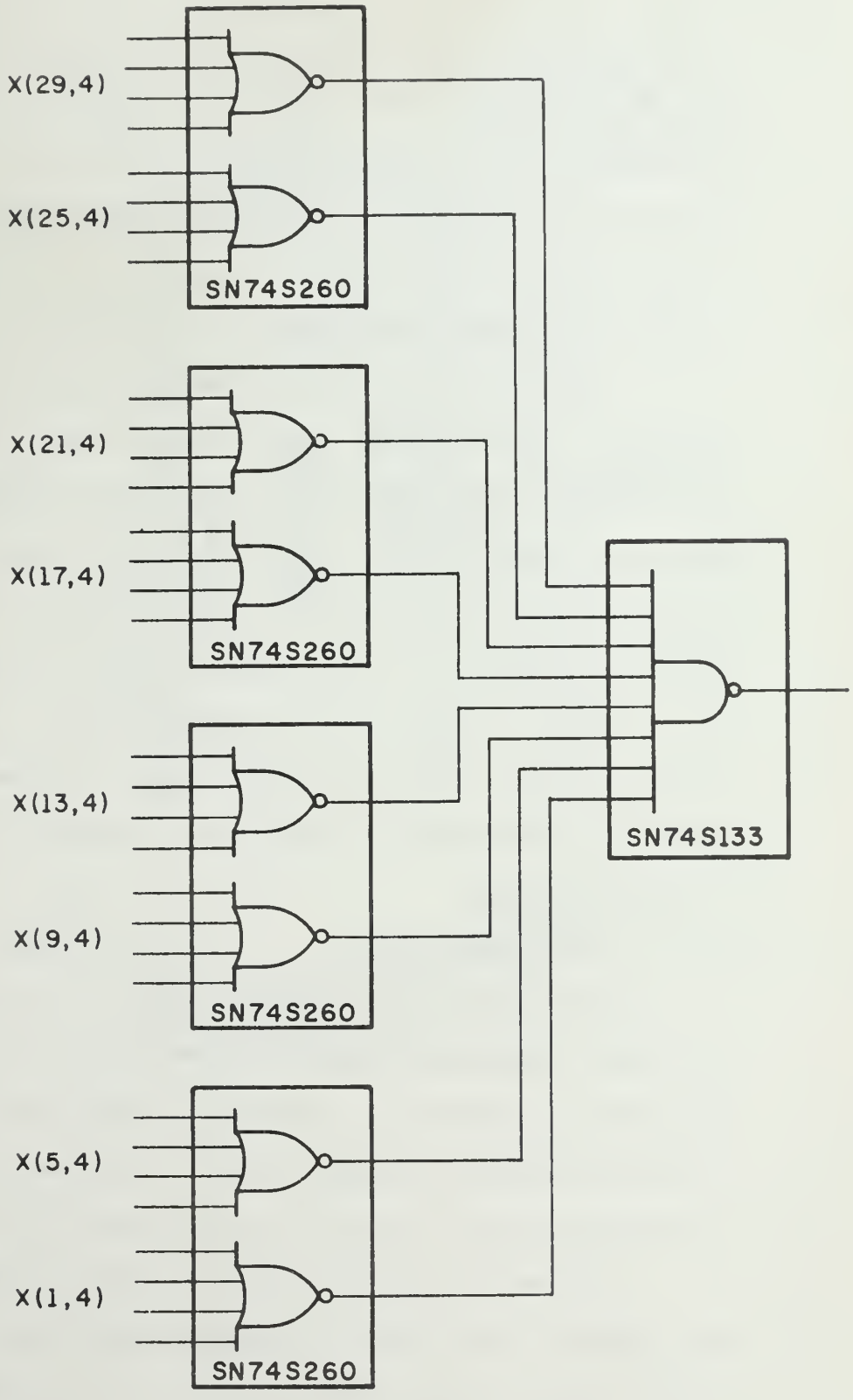


Figure 4.2.5.1.1-1 The Zero Detect Logic

instances of the zero detect logic is this rigid connection scheme necessary. (See section 4.2.5.2.1 Normalization.) In the other two cases, the total of forty NOR gate inputs can be connected in whatever manner is convenient for circuit board routing purposes.

4.2.5.1.2 The Fraction Comparator

This logic block is built entirely with the SN74S85 four bit comparator. This integrated circuit accepts a pair of four bit operands and three signals which permit fabrication of multi-bit comparators and produces three one bit output signals. Figure 4.2.5.1.2-1 shows one SN74S85, and illustrates how it is used in this design. Table 4.2.5.1.2-1 is the truth table for the SN74S85. Figure 4.2.5.1.2-2 shows how eight SN74S85's are used to compare two thirty-two bit fraction values. The output signal AGTR is a logic one if and only if the A(1,32) input signal exceeds the B(1,32) input signal. The ABEQ signal is a logic one if and only if the input signal values are identically equal.

4.2.5.1.3 The Exponent Adder

The exponent adder, shown in Figure 4.2.5.1.3-1, accepts two eight bit exponent quantities, AEXP(1,8) and BEXP(1,8), one three bit function specification, ABFUNC(1,3), and a one bit input carry signal, EXCARRY. The two eight bit exponent inputs consist of a zero bit as most significant bit, followed by the seven bits of the biased exponent for the two operands.

The exponent adder produces the eight bit combination of the two input exponents, EXC1(1,8), as specified by the function, ABFUNC(1,3), the absolute value of the difference of the two input exponents, ABS(1,7), and two one bit control signals, EXC2 and EXC2BAR.

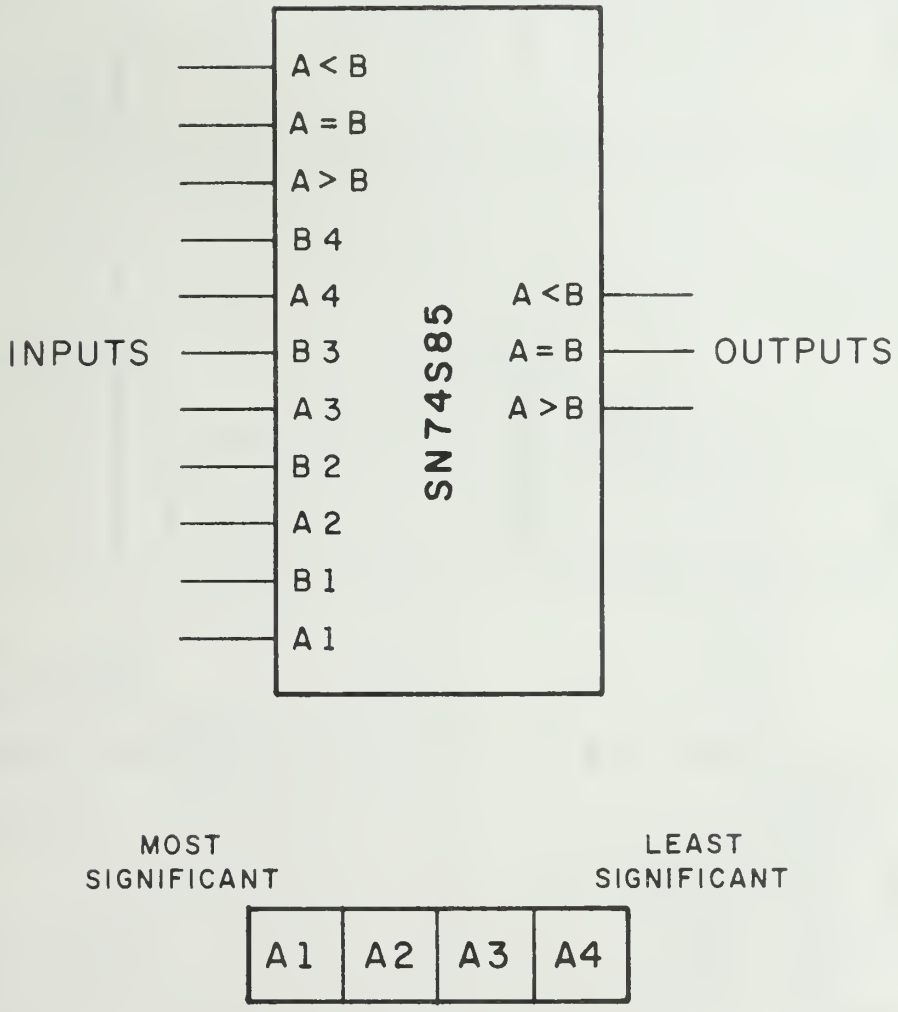


Figure 4.2.5.1.2-1 The SN74S85 Four Bit Comparator

Relation of the 4 bit data inputs	Cascading Inputs			Outputs		
	A = B	A < B	A > B	A = B	A < B	A > B
A > B	X	X	X	0	0	1
A < B	X	X	X	0	1	0
A = B	1	X	X	0	1	0
	0	0	0	0	1	1
	0	0	1	0	0	1
	0	1	0	0	1	0
	0	1	1	0	0	0

Table 4.2.5.1.2-1 The Truth Table of the SN74S85 Four Bit Comparator

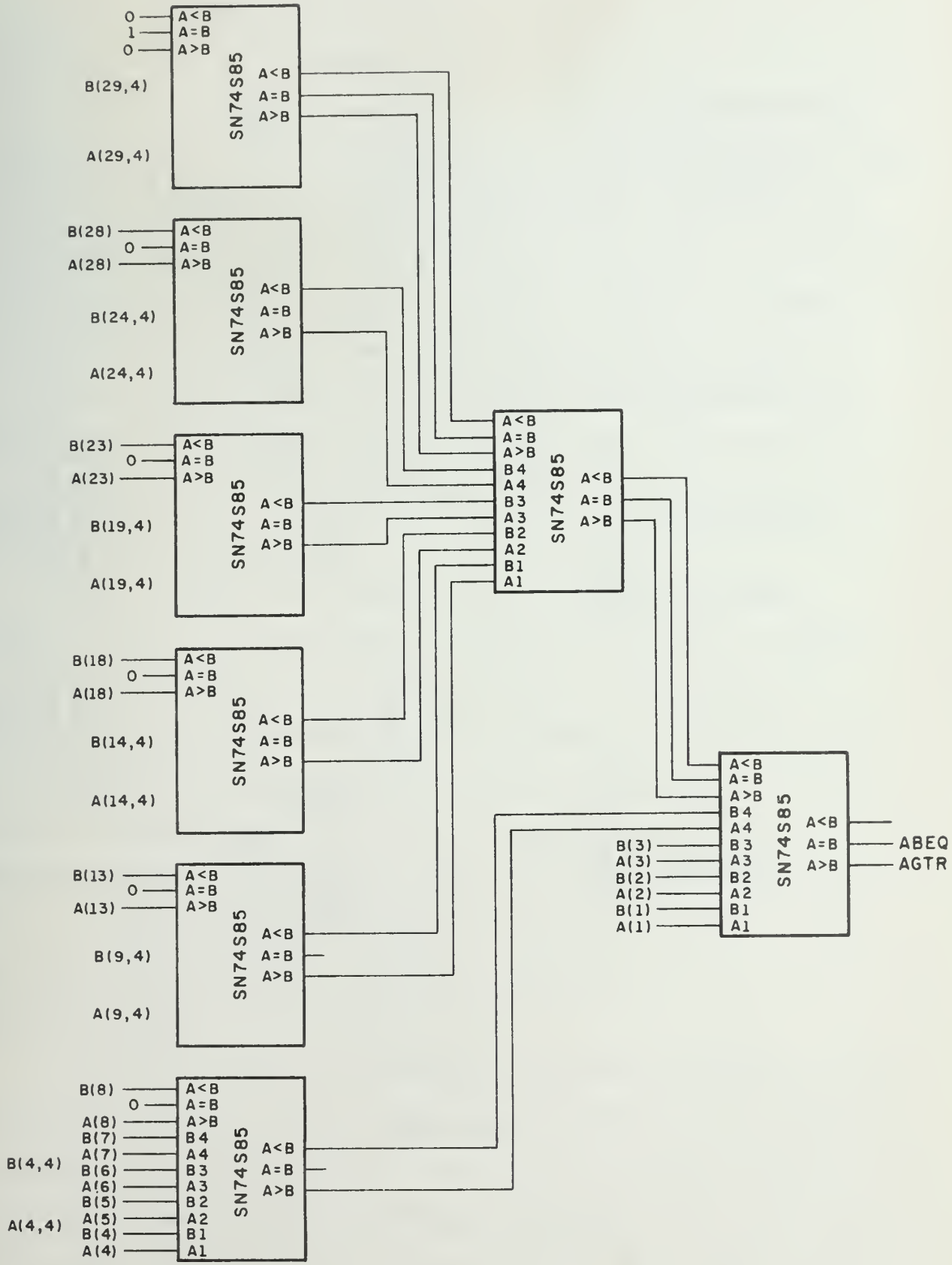


Figure 4.2.5.1.2-2 The Fraction Comparator

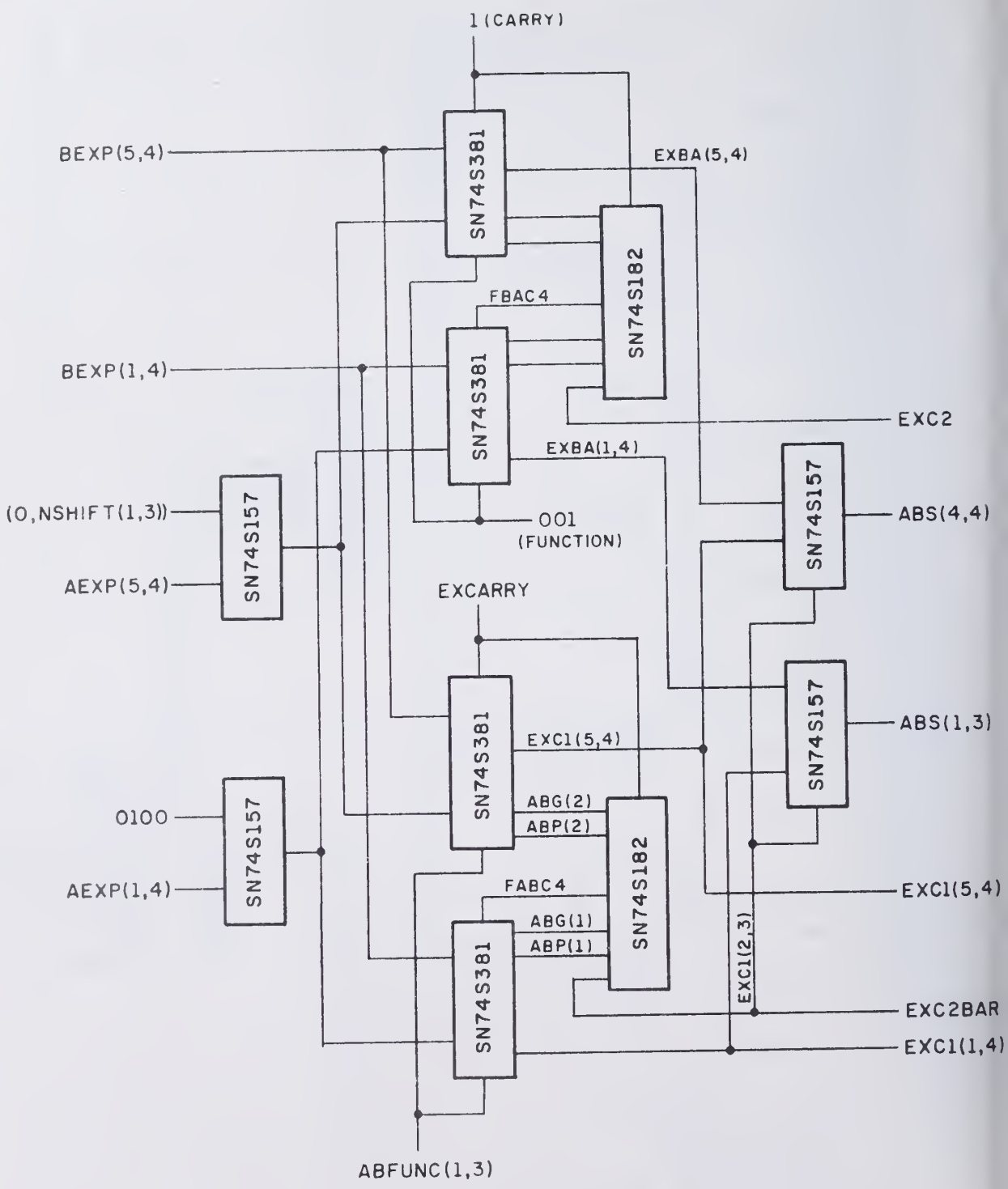


Figure 4.2.5.1.3-1 The Exponent Adder

The main functional component of the exponent adder is the SN74S381 arithmetic-logic unit. The functions performed by the SN74S381, together with the function codes which specify them, are shown in Table 4.2.5.1.3-1 (Texas Instruments Corporation, 1974). The SN74S381 does not produce an output carry signal. Instead, it produces the standard pair of carry look ahead singals for the two four bit operands. One of these signals indicates whether the two input operands will generate a carry; the other signal indicates whether an input carry of one will be propagated (Ledley, 1960). The generate and propagate signals must be used in conjunction with a carry generator such as SN74S182 (Texas Instruments Corporation, 1973).

The exponent adder actually consists of two eight bit adders working in parallel. The one shown at the top of Figure 4.2.5.1.3-1 always computes the difference A(1,8) - B(1,8). The lower adder computes the function specified by the control unit signals ABFUNC(1,3) and EXCARRY. When ABFUNC(1,3)=010, and EXCARRY=1, ABS(1,7) is the absolute value of the exponent difference and EXC2 and EXC2BAR have the meanings given in Table 4.2.5.1.3-2. The absolute value is computed by computing both A(1,8) - B(1,8) and B(1,8) - A(1,8), and selecting the positive result with the pair of SN74S157 two-to-one selectors by using EXC2BAR as the selection signal.

4.2.5.1.4 Shifting

Fraction alignment shifting and the normalization shifting are both accomplished by using the Signetics 8243 eight bit position scaler (Signetics Corporation, 1974, pp. 3.28 through 3.32). This device has open collector outputs so that several can be wire ANDed together. The shifted output bits are the logic complements of their corresponding input bits. When disabled,

Inputs				Output
A(1,4)	B(1,4)	Function	Carry	
X	X	000	X	0000
A(1,4)	B(1,4)	001	0	B(1,4) - A(1,4) - 1
A(1,4)	B(1,4)	001	1	B(1,4) - A(1,4)
A(1,4)	B(1,4)	010	0	A(1,4) - B(1,4) - 1
A(1,4)	B(1,4)	010	1	A(1,4) - B(1,4)
A(1,4)	B(1,4)	011	0	A(1,4) + B(1,4)
A(1,4)	B(1,4)	011	1	A(1,4) + B(1,4) + 1
A(1,4)	B(1,4)	100	X	A(1,4) $\oplus$ B(1,4)
A(1,4)	B(1,4)	101	X	A(1,4) OR B(1,4)
A(1,4)	B(1,4)	110	X	A(1,4) AND B(1,4)
X	X	111	X	1111

Table 4.2.5.1.3-1 Functions of the SN74S381 with Active High Carry and Data

Signal	Value	Meaning
EXC2	0	A(1,8) > B(1,8)
	1	A(1,8) $\leq$ B(1,8)
EXC2BAR	0	A(1,8) $\leq$ B(1,8)
	1	A(1,8) > B(1,8)

Table 4.2.5.1.3-2 The Meanings of EXC2 and EXC2BAR

the device emits logic ones. Output bits which, because of the specified shift, have no corresponding input bits are also logic ones.

Because the exponent base of the floating point system used in this design is sixteen, alignment and normalization shifting always require a shift by a multiple of four bit positions. The alignment shift logic, Figure 4.2.5.1.4-1, and the normalization shift logic, Figure 4.2.5.1.4-2, can therefore be implemented by using only four SIG8243's each. Each of the scalers accepts one bit from the same position within each of the eight digits of the thirty-two bit fraction to be shifted. The shift amount for each is the number of digit positions to shift.

Although the SIG8243 has both an enable and an inhibit input to control the output state, this design uses only the inhibit signal. When the inhibit signal is a logic one, the output bits are all logic ones. Disabled outputs are used to provide zero operands when the shift amount exceeds seven, and also for several other cases in the design where zero operands are needed. The details of alignment shift control are given in section 4.2.5.2.3 which discusses floating point addition and subtraction Normalization shift control is discussed in section 4.2.5.2.7 on double precision addition and subtraction. When the inhibit signal is a logic zero, shifting of the input bits takes place as specified by the three bit shift select signal.

The device performs shifts in only one direction. Both left and right shifts can be implemented by proper use of the scaler as shown in Figure 4.2.5.1.4-1 and Figure 4.2.5.1.4-2 by altering the orientation of the device with respect to the most significant bit of the input signal.

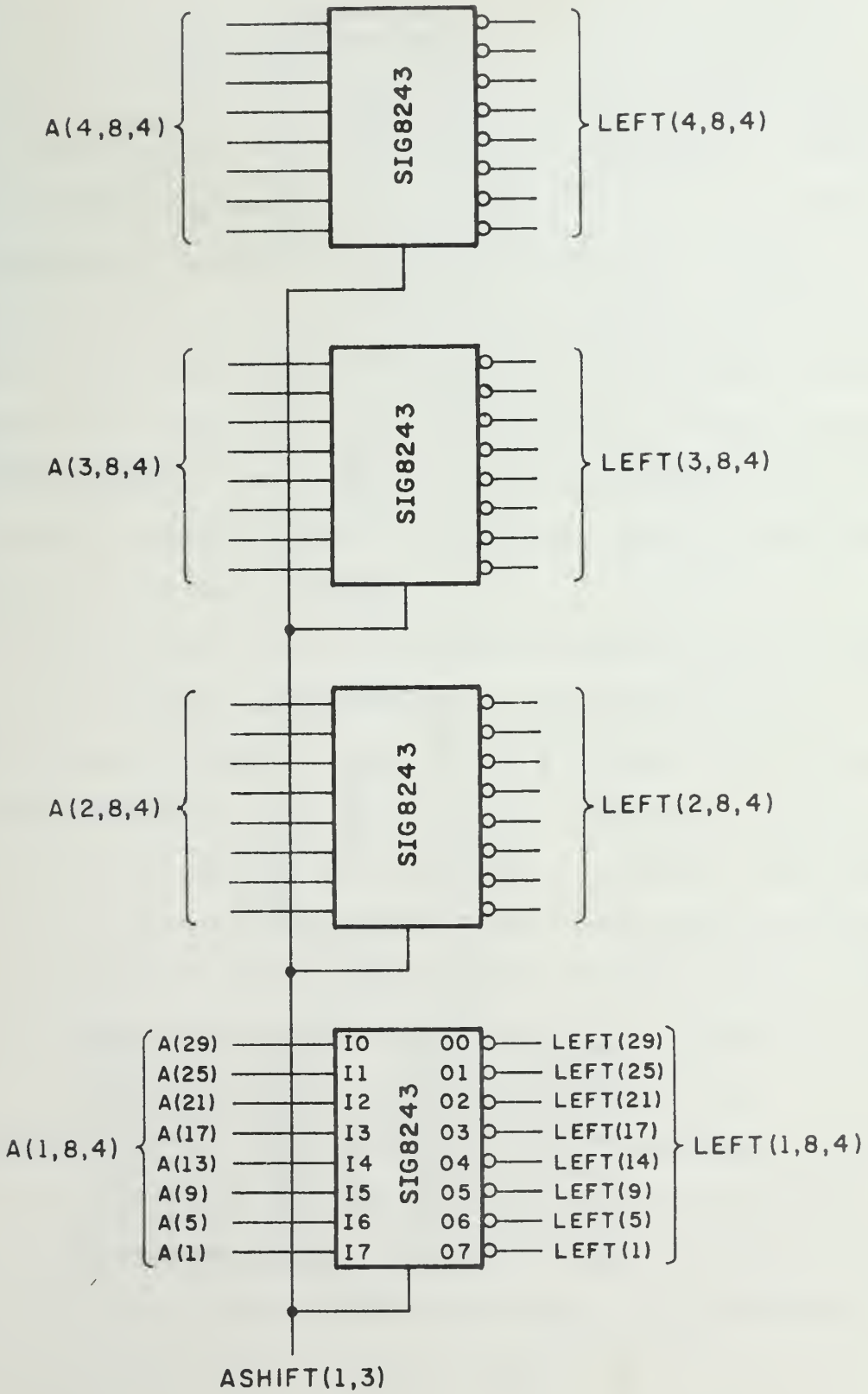


Figure 4.2.5.1.4-1 The Alignment Shifter

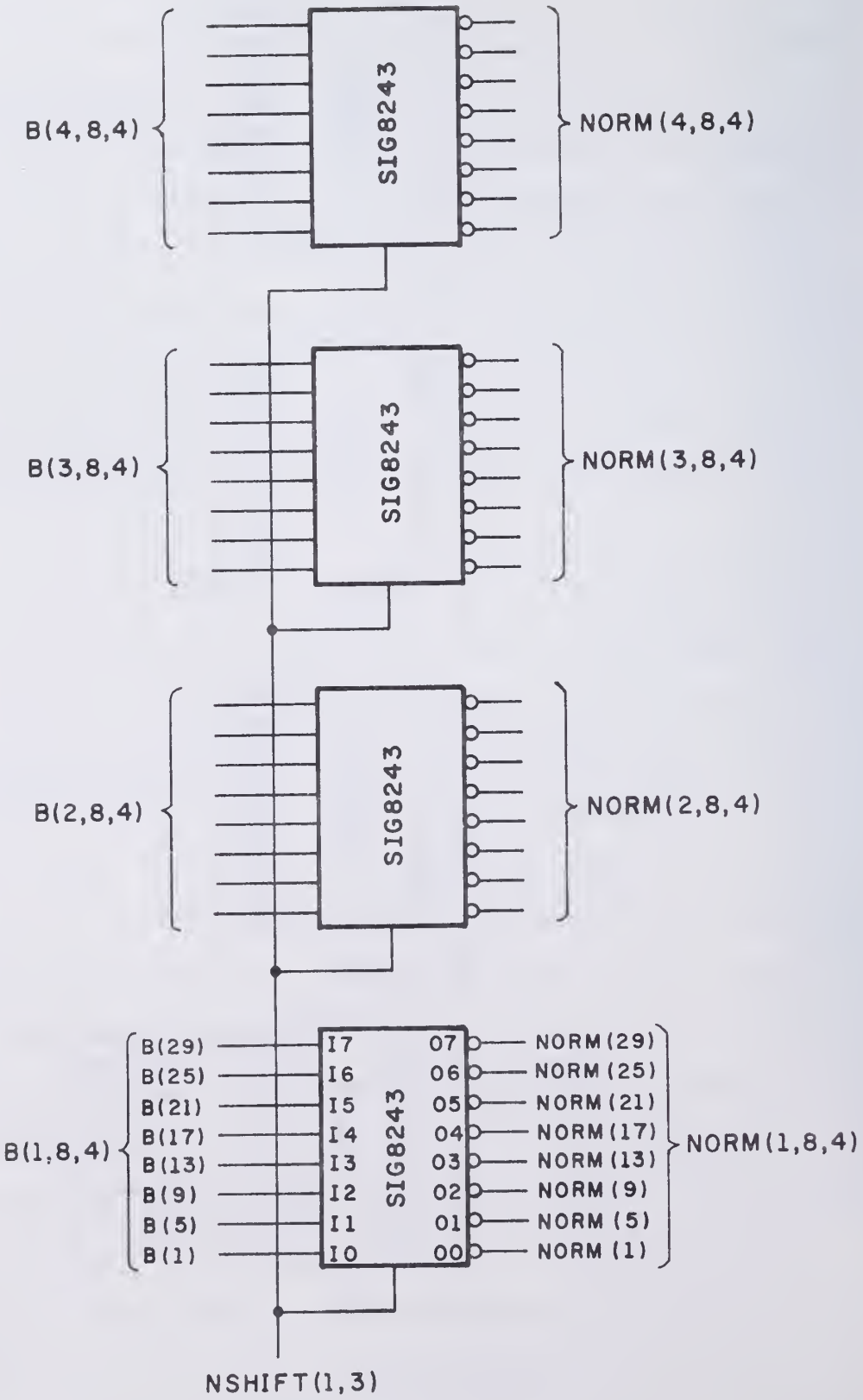


Figure 4.2.5.1.4-2 The Normalization Shifter

4.2.5.1.5 The Left Operand Selection Logic

The left operand selector logic block supplies the left operand to the adder. Two different integrated circuits are used in the left operand selector: the SN74S157 quadruple two-to-one data selector and the SN74S153 dual four-to-one data selector. For clarity of description, the blocks in Figure 4.2.5.1.5-1 do not correspond to the above integrated circuit packages, but rather to the selection functions they perform. They are labelled S157 for the two-to-one function, and S153 for the four-to-one function. Whereas the SN74S153 operates on pairs of four bits, the S153 at the bottom of the figure is shown operating on a single four bit group; the S153 next to the bottom operates on ten four bit groups.

The left operand selector supplies six different operands. They are

1. the fraction output of the left alignment shift logic
2. the twelve high order bits of the first approximation to the reciprocal for division. The other twenty bits of the fraction are forced to one by disabling the left alignment shift logic. As noted above, the alignment shift logic produces complemented outputs, so that the adder operates on active low data. Thus, the ROM which supplies the initial reciprocal approximation must be programmed to supply active low data also.
3. the constant fraction one-half (in active low data form) for use in the division algorithm. The high order bit, LEFT(1), is forced to zero by the bottom S153 of Figure 4.2.5.1.5-1, and the other thirty-one bits are forced to one by a disabled alignment shift network.

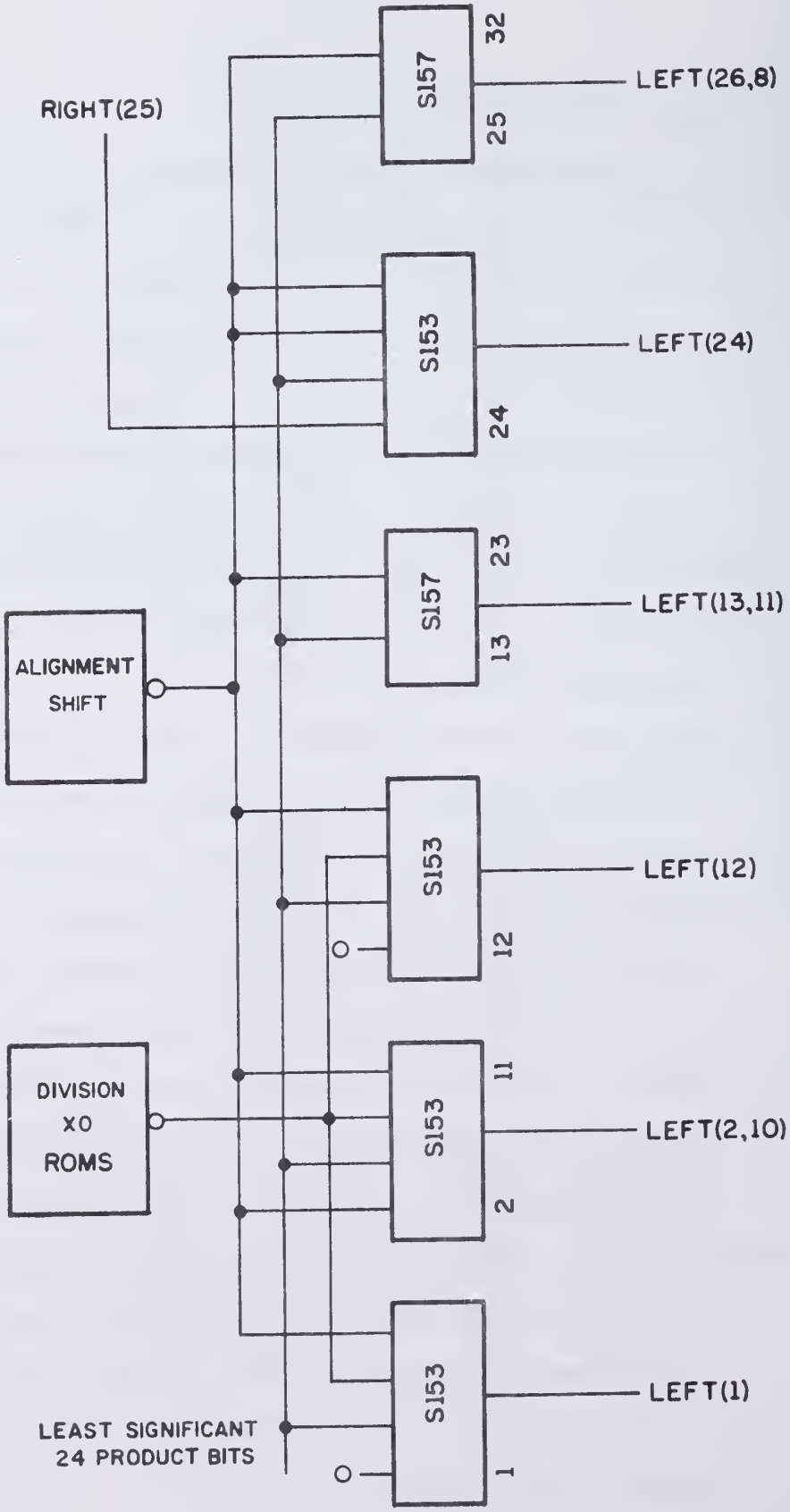


Figure 4.2.5.1.5-1 The Left Operand Selection Logic

4. the constant fraction 2^{-12} for use in the division algorithm. The bit LEFT(12) is forced to zero by the corresponding S153, and the other thirty-one bits are forced to one by a disabled alignment shift network.
5. a value for rounding data values to memory length (twenty-four fraction bits). All bits of this constant are ones from a disabled alignment shift network, except for LEFT(24), which is equal to bit twenty-five of the fraction being rounded.
6. the twenty-four least significant bits of a product. The adder normally operates on active low data, and a logic complement follows the adder. A product return in active high data form. If the least significant part of the product is sought, it is complemented by the adder by using the exclusive OR function with ones forming the disabled right alighnment shift logic.

Since the logic for the left operand selector requires the S153 function on a total of thirteen bits and the S157 function on nineteen bits, seven SN74S153 and five SN74S157 integrated circuits are required to implement it. No control local to the processor is necessary for its operation.

4.2.5.1.6 The Adder

The adder, shown in Figure 4.2.5.1.6-1, accepts two thirty-two bit fractions, LEFT(1,32) and RIGHT(1,32), a function specification, AFUNC(1,24), and an input carry AC. It produces a thirty-two bit output, SUM(1,32), which depends on the input operands, the carry, and the function specification. The SN74S381 arithmetic-logic unit and the SN74S182 look-ahead carry generator.

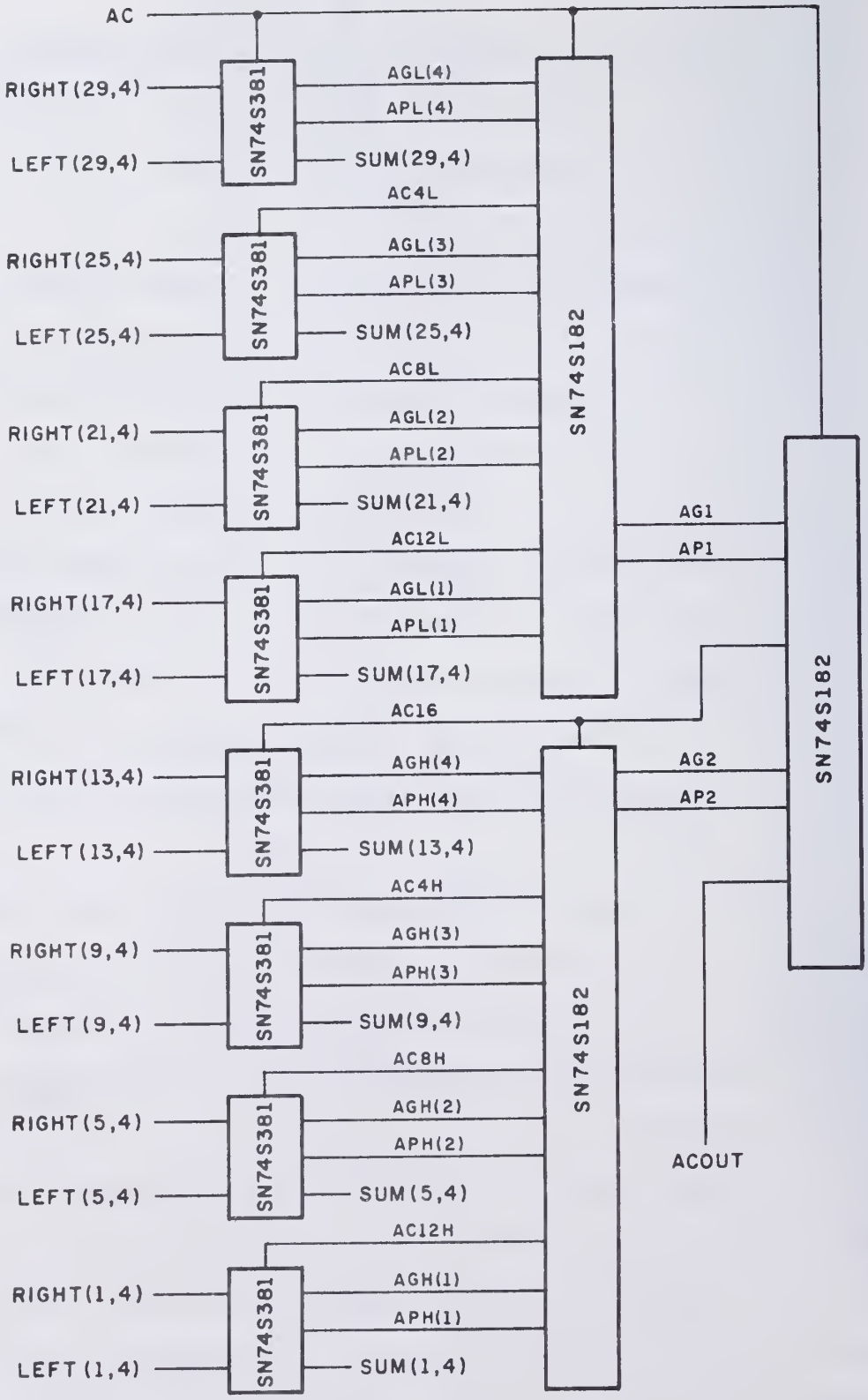


Figure 4.2.5.1.6-1 The Adder

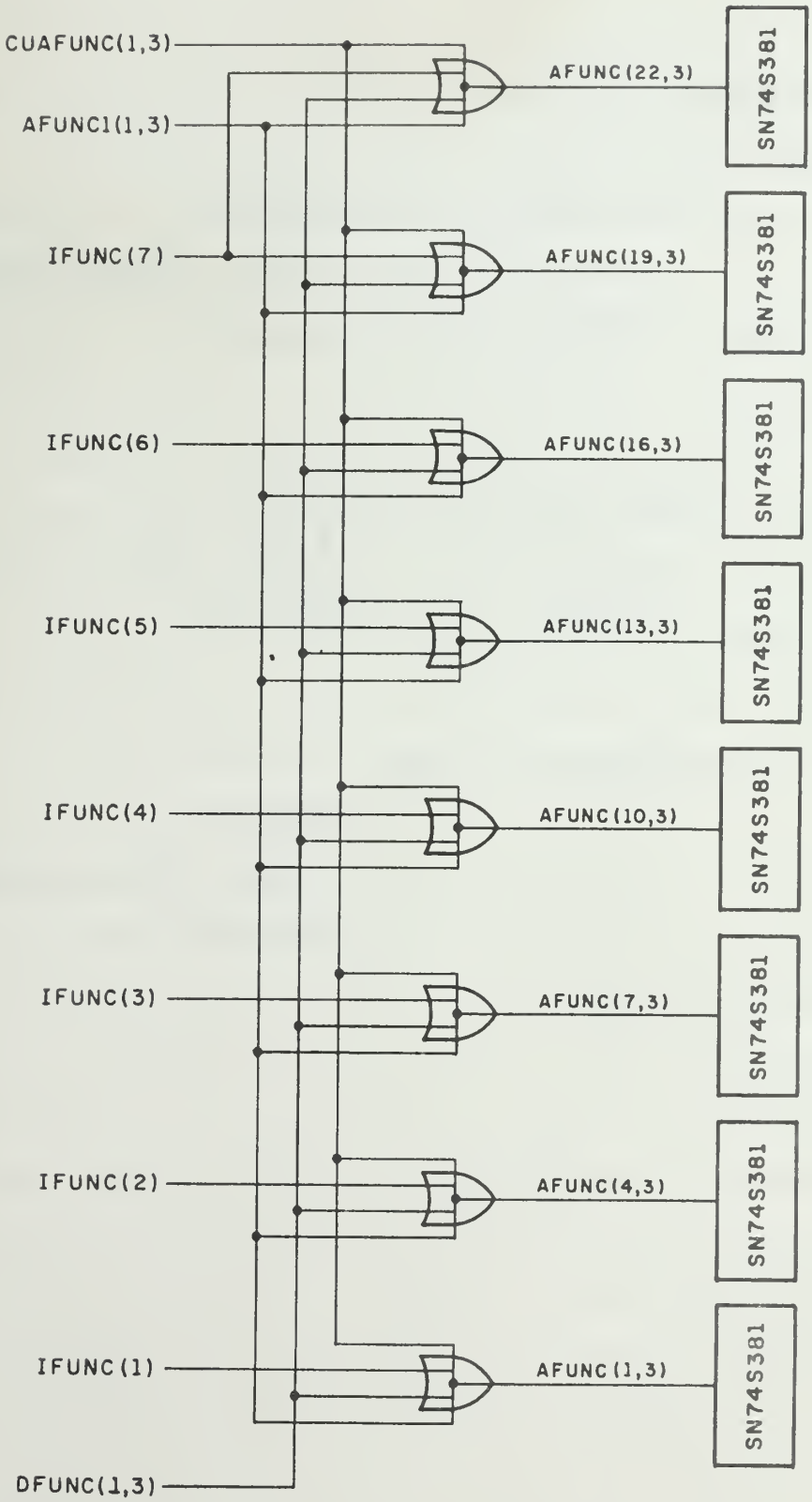


Figure 4.2.5.1.6-2 The Logic for the Signal AFUNC(1,24)

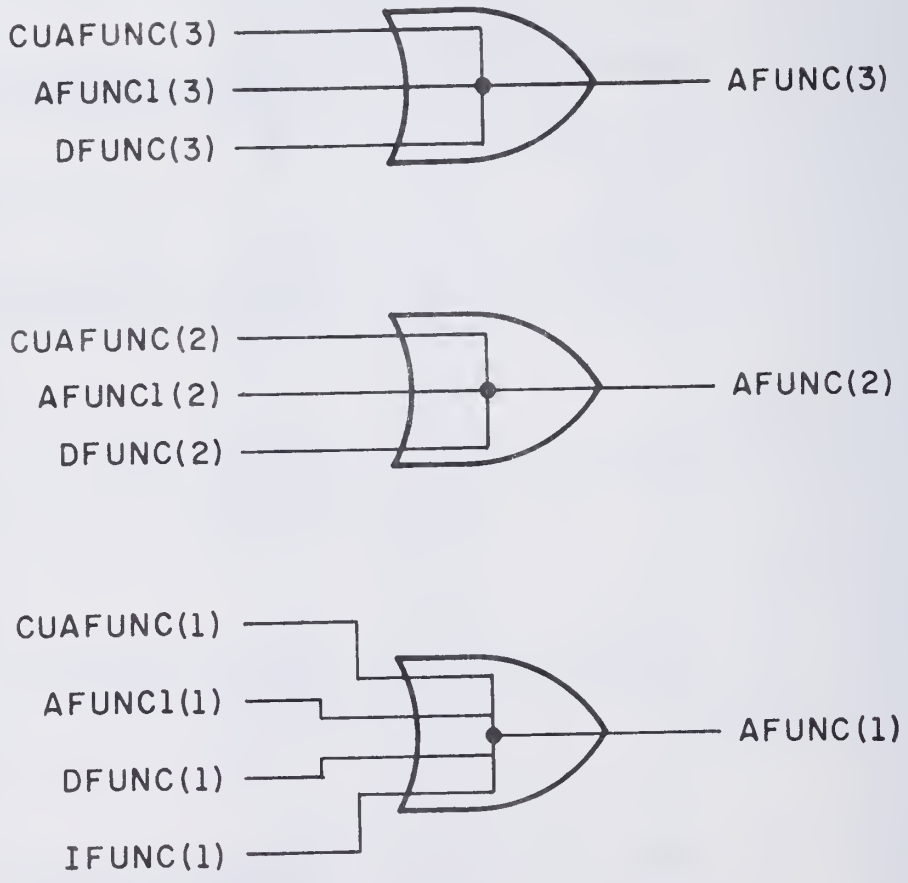


Figure 4.2.5.1.6-3 The Logic for the Signal AFUNC(1,3)

Except in the case of the integerize function, which is described in section 4.2.5.2.6, each SN74S381 performs the same function, so that AFUNC(1,3)=AFUNC(4,3)=. . .=AFUNC(22,3). The functions which can be specified are listed in Table 4.2.5.1.3-1.

The output of the adder is the thirty-two bit result, SUM(1,32), and the carry out, ACOUT. The function input to the SN74S381's is the result of a wire-OR of four separate tri-state sources. Figures 4.2.5.1.6-2 and 4.2.5.1.6-3 show successively more detail about these wire-ORed signals. Figure 4.2.5.1.6-2 shows eight wire-OR's, each of which produces a three bit function specification. Each of these three bit wire-OR's actually consists of three separate wire-OR's like the three shown in Figure 4.2.5.1.6-3. The details of the signals AFUNC1(1,3), IFUNC(1,8), and CUAFUNC(1,3) will be given in sections 4.2.5.2.1 through 4.2.5.2.6.

4.2.5.1.7 Fraction Selection Logic

The adder operates on active low data primarily because the Signetics 8243 eight position scaler, which is used to perform alignment and normalization shifting, has complemented outputs. Therefore, besides selecting one of five possible fraction sources, the fraction selection logic also performs a logical complement. The logic is shown in Figure 4.2.5.1.7-1, and consists of Signetics 8263 quadruple three-to-one selectors and Advanced Micro Devices AM9309 dual four-to-one selectors. The SIG8263's were used where possible to reduce the package count, and the AM9309's were used because no other four-to-one selector which provides complemented outputs is available.

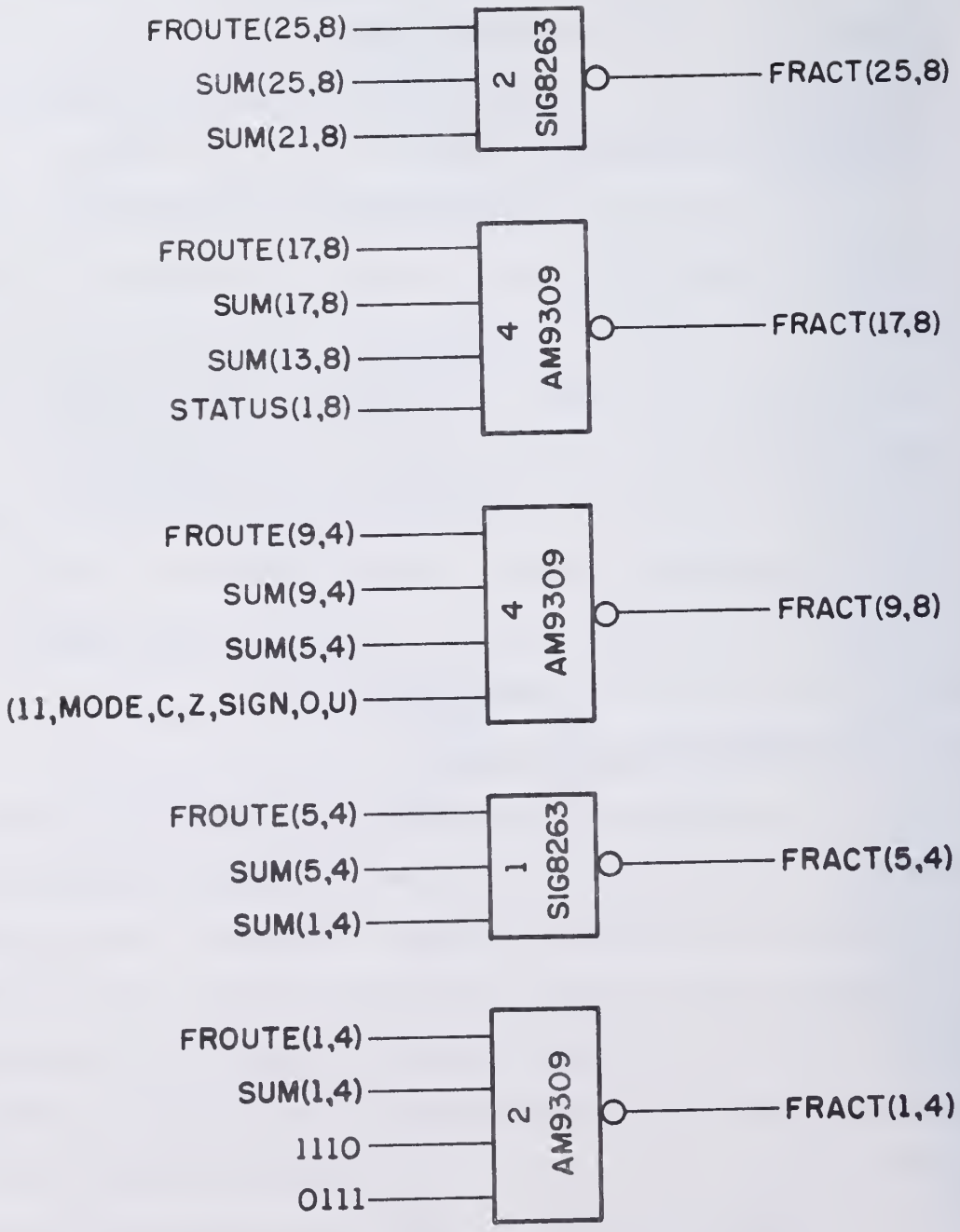


Figure 4.2.5.1.7-1 The Fraction Selection Logic

The five signals which the fraction selection logic accepts as input are:

1. the unmodified output of the Adder, (SUM(1,32).
2. The output of the adder shifted right one digit position (four bit positions) by appropriate selection. The control for deciding between this input and input (1) above depends on whether fraction overflow occurs during fraction addition. The details of this control are given in section 4.2.5.2.3. If the shifted input is selected, the high order digit is forced to 1110, complemented to 0001.
3. The fraction output from the routing logic reassembly register, FROUTE(1,32). The routing logic is the subject of section 4.3.
4. The outputs of the mode flip-flop of section 4.2.5.1.9 and five condition flip-flops (MODE C, Z, SIGN, O, U) which are described in section 4.2.5.1.12, and the output of the status register of the mode logic, STATUS(1,8), which is described in section 4.2.5.1.9. These thirteen bits are supplemented by nineteen bits of ones (complemented to zeros) forced from the SN74S381 arithmetic-logic units (see Table 4.2.5.1.3-1).
5. The special fraction overflow shift of one bit position which uses the high order digit value of 0111, complemented to 1000. This case is fully discussed in section 4.2.5.2.5.

As shown in Figure 4.2.5.1.7-2, the fraction selection logic is in every path which leads to the operand registers. Therefore, one would like it

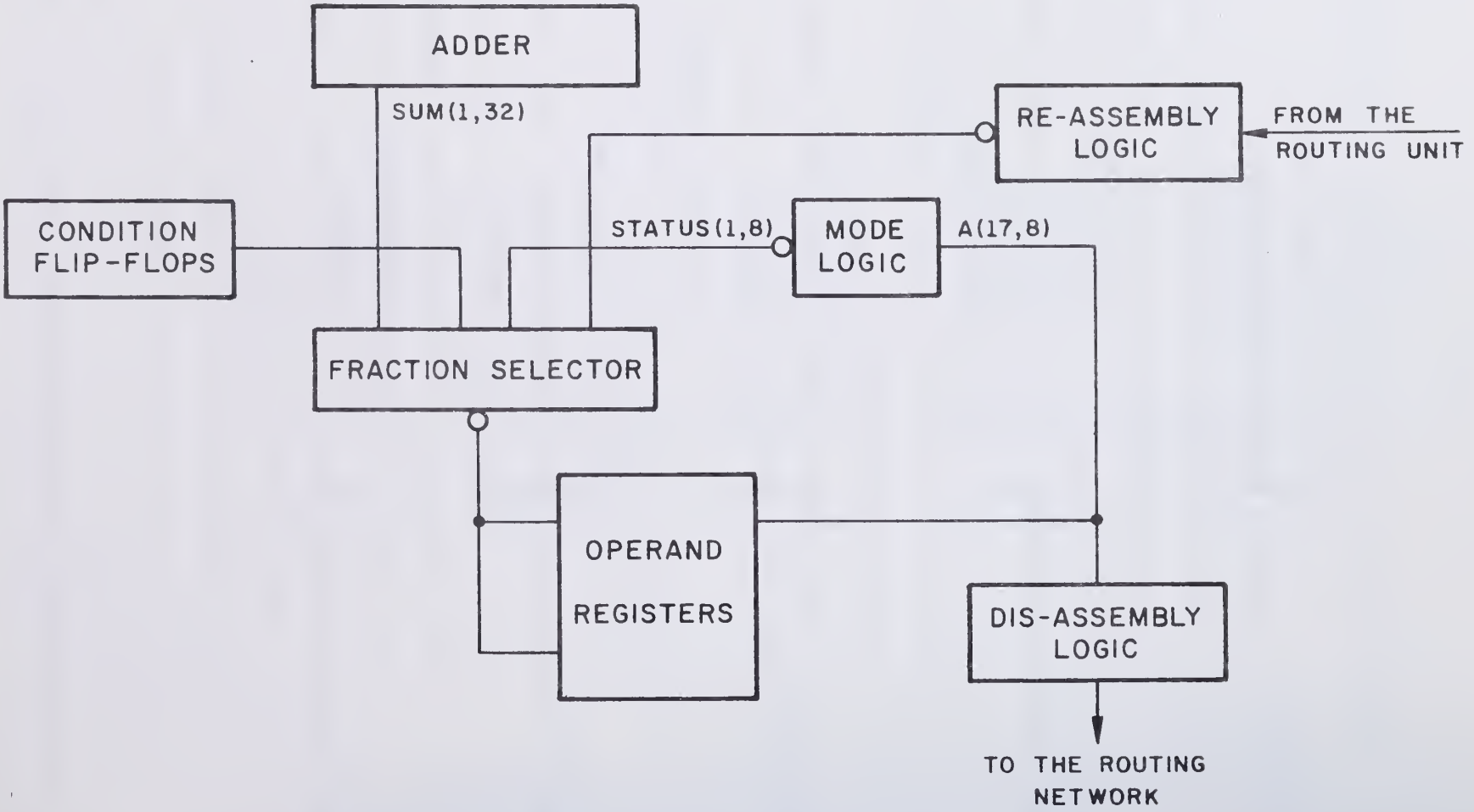


Figure 4.2.5.1.7-2 The Relationship of the Fraction Selector to the Rest of the Processor

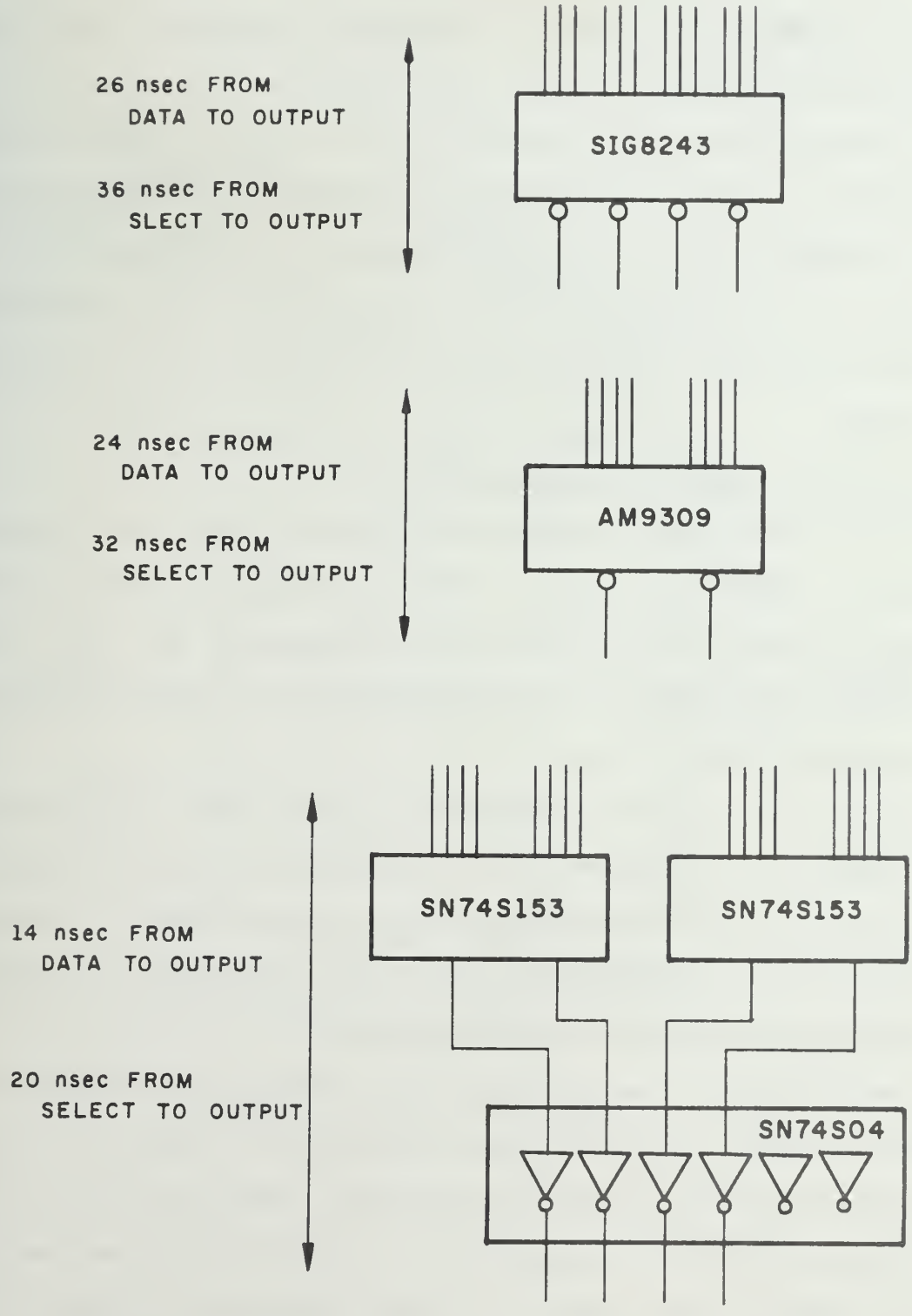


Figure 4.2.5.1.7-3 A Faster Alternative to the Fraction Selection Logic

it to be as fast as possible. Unfortunately, neither the SIG8263 nor the AM9309 is available in Schottky form. Figure 4.2.5.1.7-3 shows how the thirteen package logic of Figure 4.2.5.1.7-1 could be replaced by twenty-two packages: sixteen SN74S153 dual non-complementing four-to-one selectors and six SN74S04 inverters. The gain in time is twelve nano-seconds per operation when the timing depends on the data arrival time at the selectors, and sixteen nano-seconds when the timing depends on the arrival time of the selection signals.

4.2.5.1.8 Exponent Correction Adder

The exponent produced by the exponent adder is not correct in all cases. When fraction overflow occurs, the fraction is shifted right one digit position and the exponent must be increased by one. This case and several others discussed in section 4.2.5.2.5 are handled by the exponent correction adder.

The logic for the exponent correction adder is shown in Figure 4.2.5.1.8-1. It includes two SIG8263 three-to-one selectors which are used to select either the exponent of the left operand, AEXP(1,7), the exponent of the right operand, BEXP(1,7), or the result exponent from the exponent adder, EX1(2,7). Bit EXC1(2) is complemented because it is the bias bit in the biased exponent. When an exponent sum or difference is computed by the exponent adder, the bias bit must be complemented in order for the resulting exponent value to be correctly represented. (See section 4.2.5.1.12.4 or section 4.2.5.1.12.5 for more details.) The logic which produces the selection signal for this selection is shown in Figure 4.2.5.1.8-2. The SN74S151 eight-to-one selector is controlled according to the truth table in

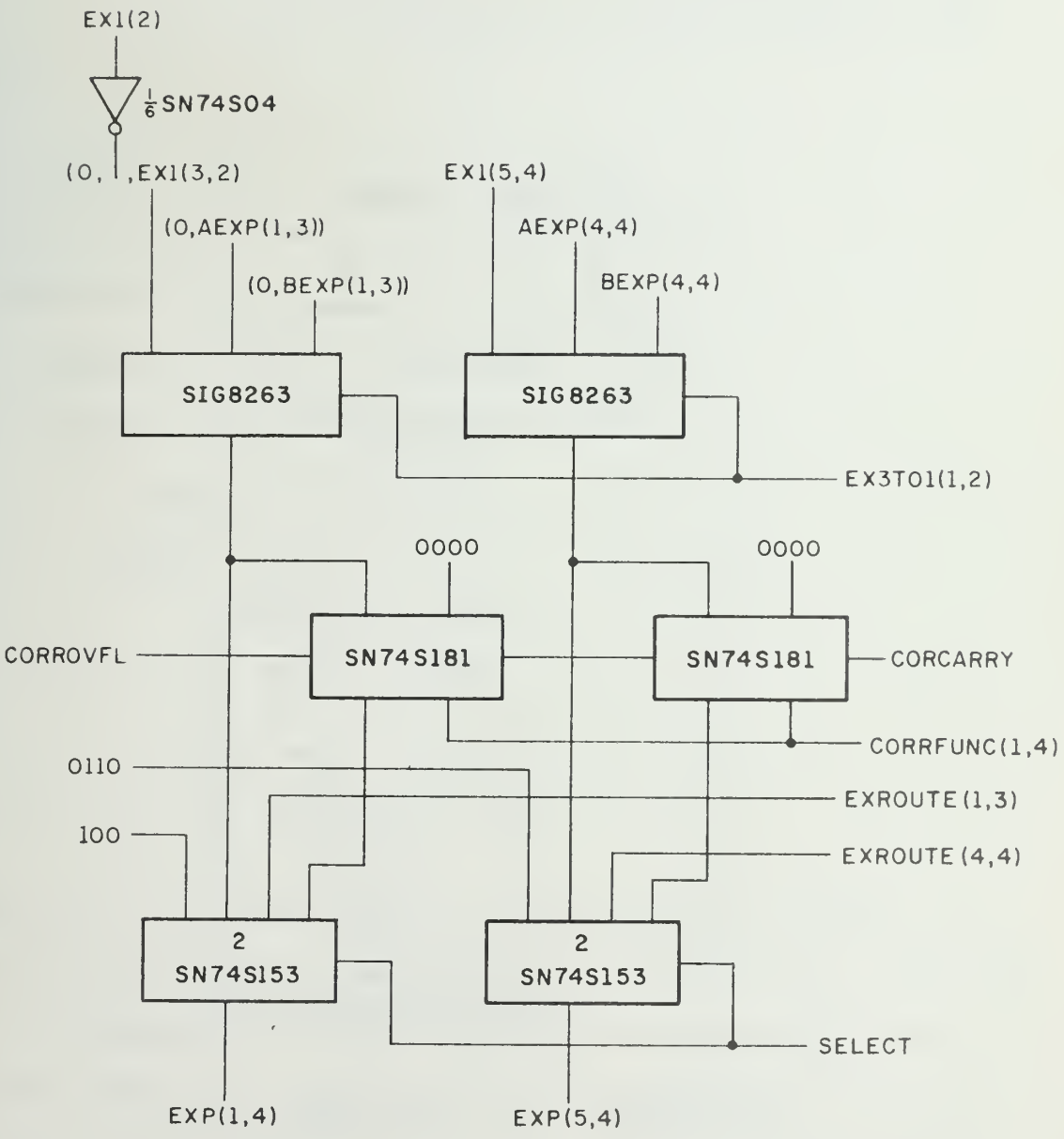


Figure 4.2.5.1.8-1 The Exponent Correction Adder

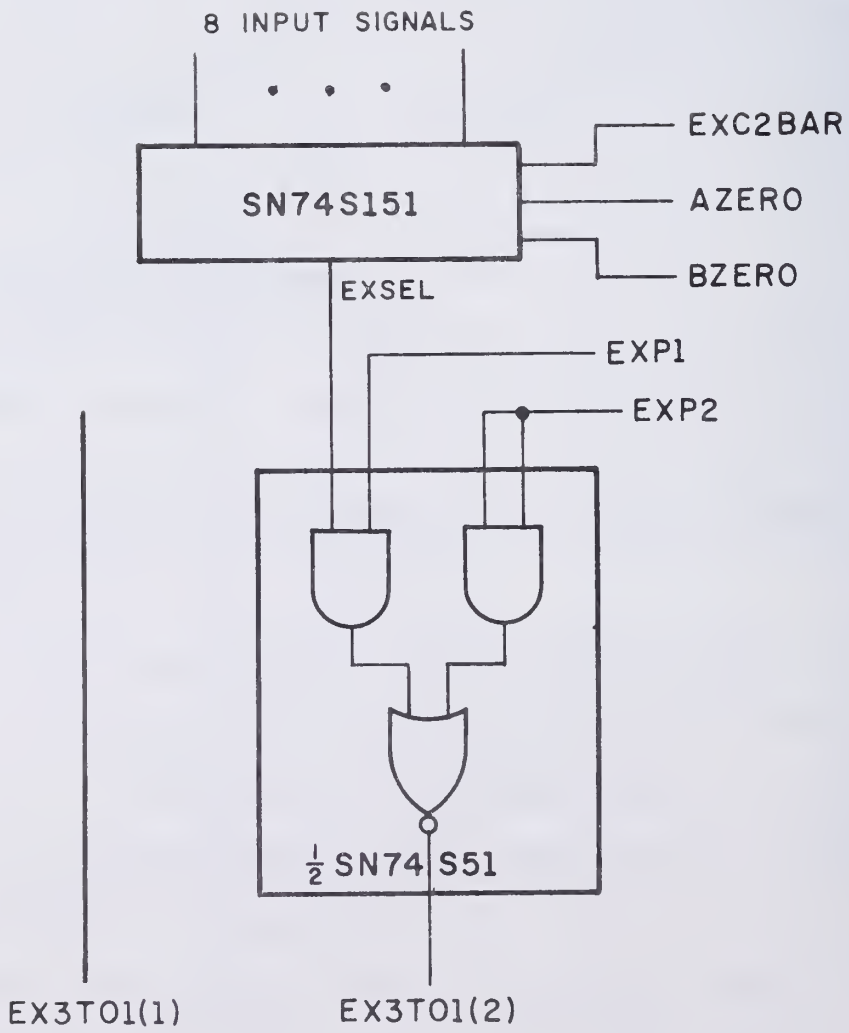


Figure 4.2.5.1.8-2 The Control Signal for Input Selection for the Exponent Correction Adder

Table 4.2.5.1.8-2; its inputs are wired to the logic constants indicated by Table 4.2.5.1.8-1. EXP1, EXP2, and EX3T01(1) are control signals from the control unit.

EXC2BAR	AZERO	BZERO	EXSEL OUPTUT
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0

Table 4.2.5.1.8-1 The Low-order Bit of Exponent Selection Control

An EXC2BAR value of one means that the left operand has been shifted, so that the correct exponent for a sum or difference is the exponent of the right operand. An AZERO value of zero means that the left operand fraction was zero; a BZERO value of zero means that the right operand fraction was zero. Control signals from the control unit determine the control signal for the exponent selection process according to the truth table in Table 4.2.5.1.8-2.

Input Signals EXSEL	Output Selection Signal EX3TO1(1,2)	Exponent Selected
x	01	exponent adder value
1	10	left operand exponent
0	11	right operand exponent

Table 4.2.5.1.8-2 Exponent Selection Control

The SN74S181 arithmetic-logic units are used to either add or subtract one from the selected exponent. The values of CORCARRY and CORRFUNC(1,4) necessary to accomplish this are given in Table 4.2.5.1.8-3 which is based on the operating details of the SN74S181 (Texas Instruments Corporation, 1973, p. 383).

Inputs		SN74S181 Output
CORRFUNC(1,4)	CORCARRY	
0000	0	exponent + 1
1111	1	exponent - 1

Table 4.2.5.1.8-3 Control of Exponent Correction Add

The control logic shown in Figure 4.2.5.1.8-3 supplies the CORCARRY and CORRFUNC(1,4) signals. The signal from the division control ROM is explained in section 4.2.5.2.5. The final stage of the exponent correction adder

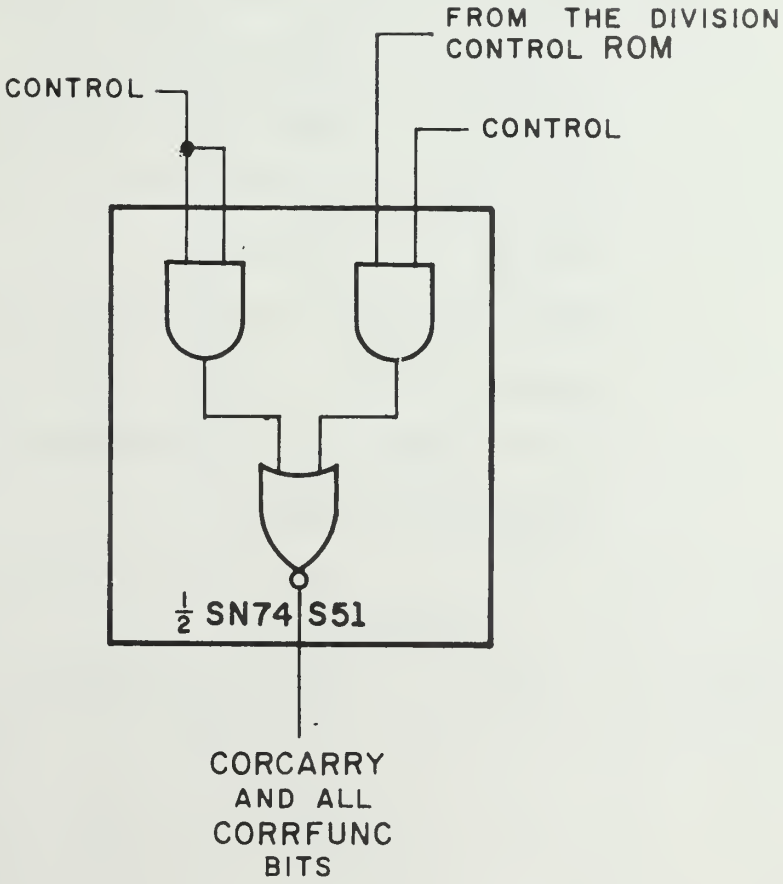


Figure 4.2.5.1.8-3 The CORCARRY and CORRFUNC(1,4) Bits for Exponent Correction Adder Control

performs a selection function for the result exponent similar to that performed for the result fraction by the fraction selection logic described in section 4.2.5.1.7. The selection is performed by four SN74S153 four-to-one selectors according to the logic shown in Figure 4.2.5.1.8-4 and the truth table given in Table 4.2.5.1.8-4. The four final exponent values which can be selected are:

1. The constant 46_{16}, which is the correct biased exponent value for the status register value.
2. The exponent of the value received from the routing unit.
3. The exponent selected by the input selection logic of the exponent correction adder.
4. The above exponent modified by the SN74S181's of the exponent correction adder. This last choice is governed by the OVFLSEL bit whose derivation is explained in detail in section 4.2.5.2.3.

Inputs				Selection Signal	Exponent Selected
control 1	control 2	control 3	OVFLSEL		
0	1	x	x	00	46_{16}
0	0	0	x	01	routing exponent
1	0	1	1	10	selected exponent
1	0	1	0	11	modified exponent

Table 4.2.5.1.8-4 Final Exponent Selection Control Signal

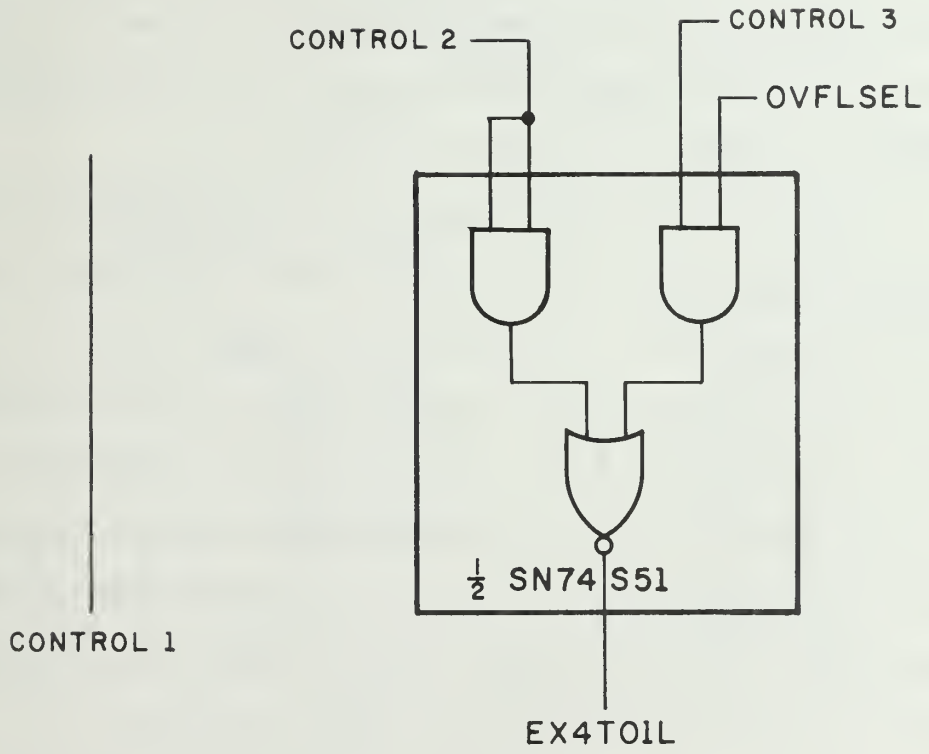


Figure 4.2.5.1.8-4 Control Signal Logic for Final Exponent Selection

4.2.5.1.9 The Mode Logic

The mode logic is shown in Figure 4.2.5.1.9-1. It includes the mode flip-flop register (the SN74S175) and an eight bit status register (the AM9334). The contents of the mode register provides the most important local control function in the processor. When the mode bit is zero, modification of operand register and condition flip-flops (see section 4.2.5.1.12) is not permitted. The status register can be used to store mode register states. Its use is illustrated in sections 6.4 and 6.5.

The mode logic permits combining the current mode state with any one of fifteen bit values local to the processor or with one bit from the control unit MODEIN. The selected bit can be combined with the mode bit using any of the sixteen possible Boolean functions of two variables; the SN74S181 can compute all of these Boolean functions. The resulting bit value can be stored in the mode flip-flop and/or any one of the eight bit positions of the status register. The status bits, STATUS(1,8), the mode flip-flip state, and the condition flip-flop states can all be saved or restored from a processor register (see section 4.2.5.1.7).

The fifteen possible local operand bits for Boolean combination with the mode bit include:

1. the eight processor status register bits, STATUS(1) through STATUS(8)
2. the five condition flip-flop contents, C, Z, SIGN, O, and U,
3. two combinations of conditions flip-flop contents, namely
   a. ZBAR NAND SIGNBAR
   B. OBAR NAND UBAR

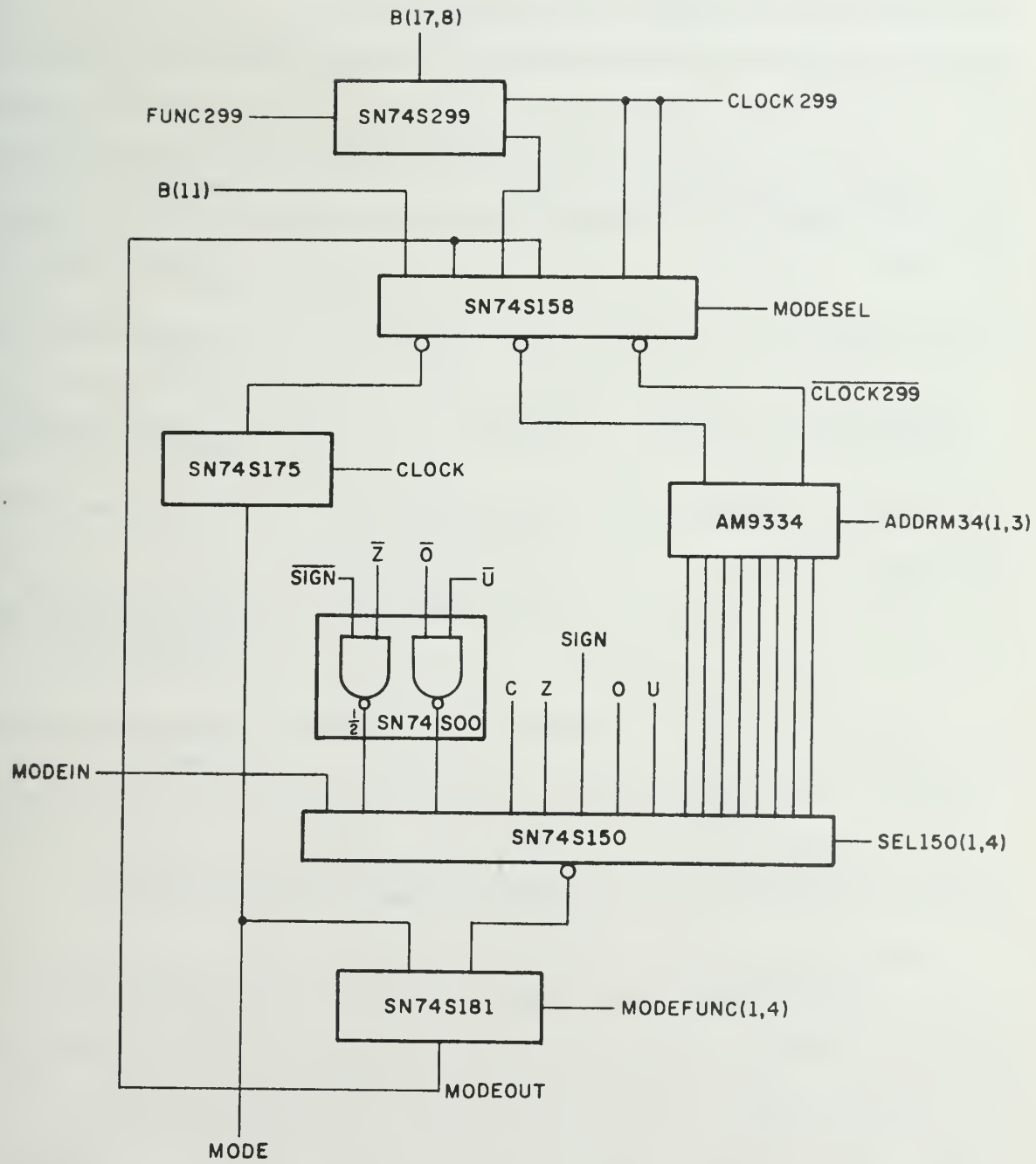


Figure 4.2.5.1.9-1 The Mode Logic

The bits of parts (2) and (3) above permit testing for any of the six possible relations between two numerical values as shown in Table 4.2.5.1.9-1.

Relation of Two Values	Bit	Comments
Equal	Z	A result fraction was zero
Not equal	ZBAR	A result fraction was not zero
Greater than or equal	SIGNBAR	A result sign was positive
Less than or equal	SIGNBAR NAND ZBAR = SIGN OR Z	A result was positive or zero
Greater than	SIGNBAR AND ZBAR	Complement of the above by appropriate SN74S181 Boolean function selection
Less than	SIGN	A result sign was negative

Table 4.2.5.1.9-1 Testing for Any Possible Relation Between Arithmetic Values

The SN74S299 is an eight bit parallel-in parallel-out shift register which can operate at rates up to 50 MHz. It can shift left and right and has a serial bit output. A subset of its facilities is used. Signal FUNC299 is used to select either the parallel load or shift function. It receives eight bits from the processor registers for restoration to the AM9334 status register.

The mode logic can accomplish its operations is significantly less time than can the full processor. If it is desired, this fact can be used to advantage by permitting the control unit to use several different inter-clock pulse intervals for array control. Mode operations, and in particular the serial shift of the eight bits from the SN74S299 to the AM9334, are among the best candidates for this approach.

The status bits, STATUS(1,8), can be saved in a processor register with an assigned exponent value of 46_{16} (a biased exponent of plus six) by appropriate use of the fraction selector, section 4.2.5.1.7, and the final exponent selection part of the exponent correction adder, section 4.2.5.1.8. The fraction selection logic complements its input; there, an inverting two-to-one selector (the SN74S158) is used to reinvert the data.

The AM9334 is an eight bit latch which accepts one input bit and a three bit latch address, ADDRM34(1,3). It stores the input bit in the addressed latch when an input enable signal goes to a logic zero. (See Advanced Micro Devices Incorporated, 1974, pp. 2-149 through 2-154.)

The SN74S150 is an inverting sixteen-to-one selector, controlled by SEL150(1,4). It provides one input to an SN74S181 arithmetic-logic unit which operates in logic mode. The other input to the SN74S181 is the current Mode value. Any of the sixteen possible Boolean combinations of two variables can be specified by MODEFUNC(1,4). (See Texas Instruments Incorporated, 1973, pp. 382-391,)

The SN74S175 is a quadruple flip-flop package which has both MODE and MODEBAR outputs available.

4.2.5.1.10 The Operand Registers

Although memory values have only thirty-two bits, intermediate results within the processor have forty bits. The extra eight bits extended the fraction to thirty-two bits within the r processor. Each processor has sixteen operand registers. They are implemented by using SN74S172 register files. The SN74S172 stores sixteen bits organized as eight two bit words. Figure 4.2.5.1.10-1 illustrates how two SN74S172 packages are used in this design x to form a sixteen word file of two bit words. Twenty such combinations, or a total of forty SN74S172 packages, are required to implement the sixteen forty bit registers of the processor. The top SN74S172 package of each pair is used to store zero through seven, and the bottom packages store words eight through sixteen.

The SN74S172 permits two data words to be read and two data words to be written simultaneously. However, only three addresses are permitted. One address specifies a word to be read, another specifies a word to be written, and the third specifies a word to be read and/or written. The outputs are tri-state; two enabling signals control the two read ports. Two more enabling signals control the two write ports. When a given enabling signal is a logic zero, the port to which it corresponds is permitted to function.

A four bit address is required to select one of sixteen words. Three four bit addresses and four control signals are used to control the registers. The three low order bits of each address are sent to the proper port of each of the forty SN74S172 packages. The high order bits of AADDRESS and BADDRESS are combined with two of the control signals to form the selection inputs of a pair of SN74S153 four-to-one selectors for each enable signal.

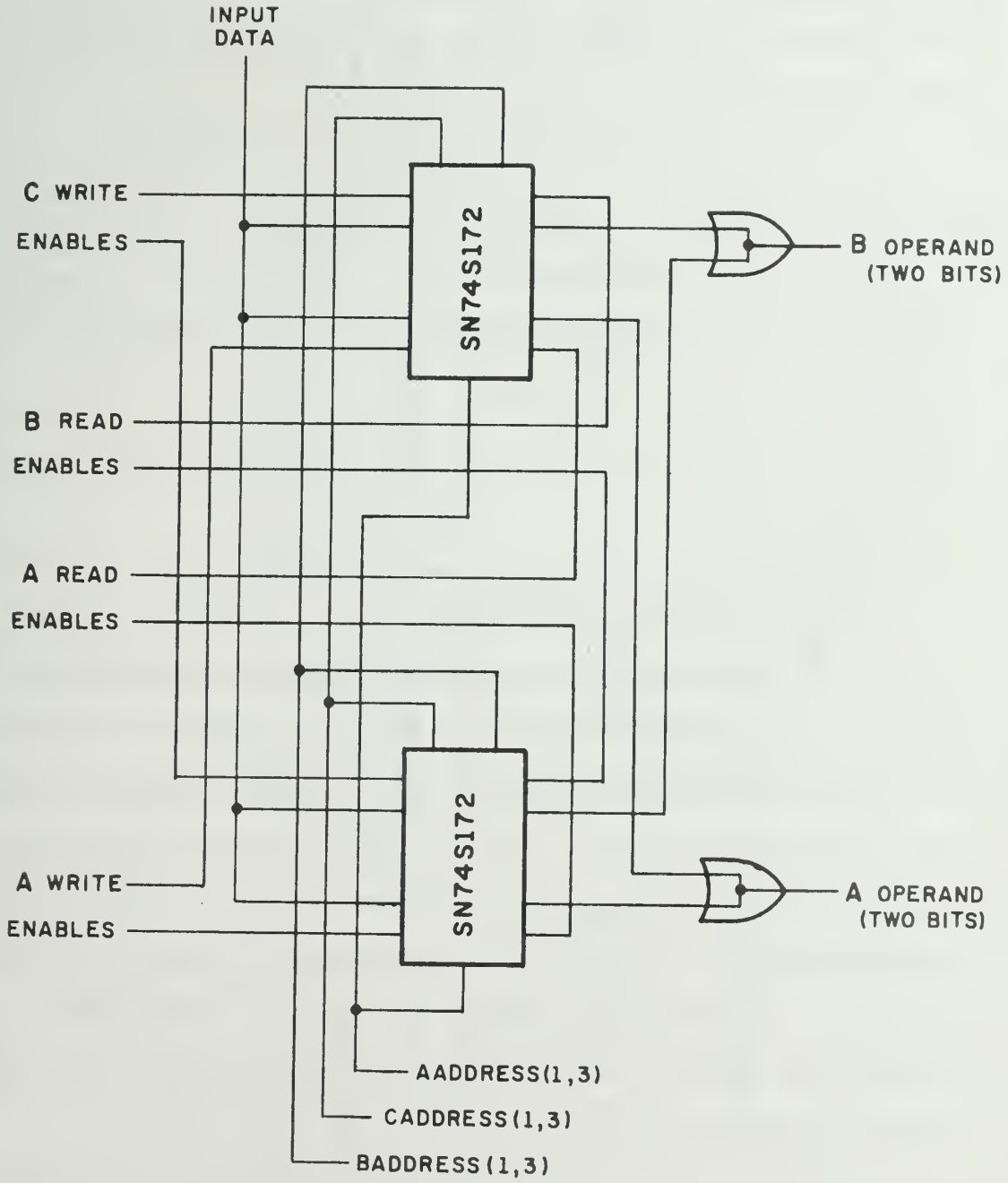


Figure 4.2.5.1.10-1 Sixteen Two Bit Words Implemented with SN74S172 Register Files

One enable signal of each pair controls registers zero through seven, the other registers eight through sixteen. The truth tables for the read enable signals are given in Table 4.2.5.1.10-1.

SELECTION BITS		ENABLE SIGNALS	
High Order Address Bit	Control Bit	Registers Zero through Seven	Registers Eight through Sixteen
0	0	1	1
0	1	1	0
1	0	1	1
1	1	0	1

Table 4.2.5.1.10-1 Truth Table for the Read Enable Signals

The high order bits of AADRESS and CADRESS are combined with the other two control signals to yield the selection signals for two more pairs of SN74S153 four-to-one selectors. These two pairs of selectors supply the A and C write enable signals. The truth table for these selectors is also given by Table 4.2.5.1.10-1, except that the zero logic input is supplied by the MODEBAR output of the MODE flip-flop in each processor. This prohibits any writing into registers of disabled processors. A clock pulse is required to clock input signals into the SN74S172 through an enabled write port.

4.2.5.1.11 The Index Adder

We saw in section 3 that address indexing capability within the processors is an important capability in an array processor. Figure 4.2.5.1.11-1 shows the logic of the index adder which computes a sixteen bit effective address, EADDRE(1,16), within each processor. The adder is implemented with

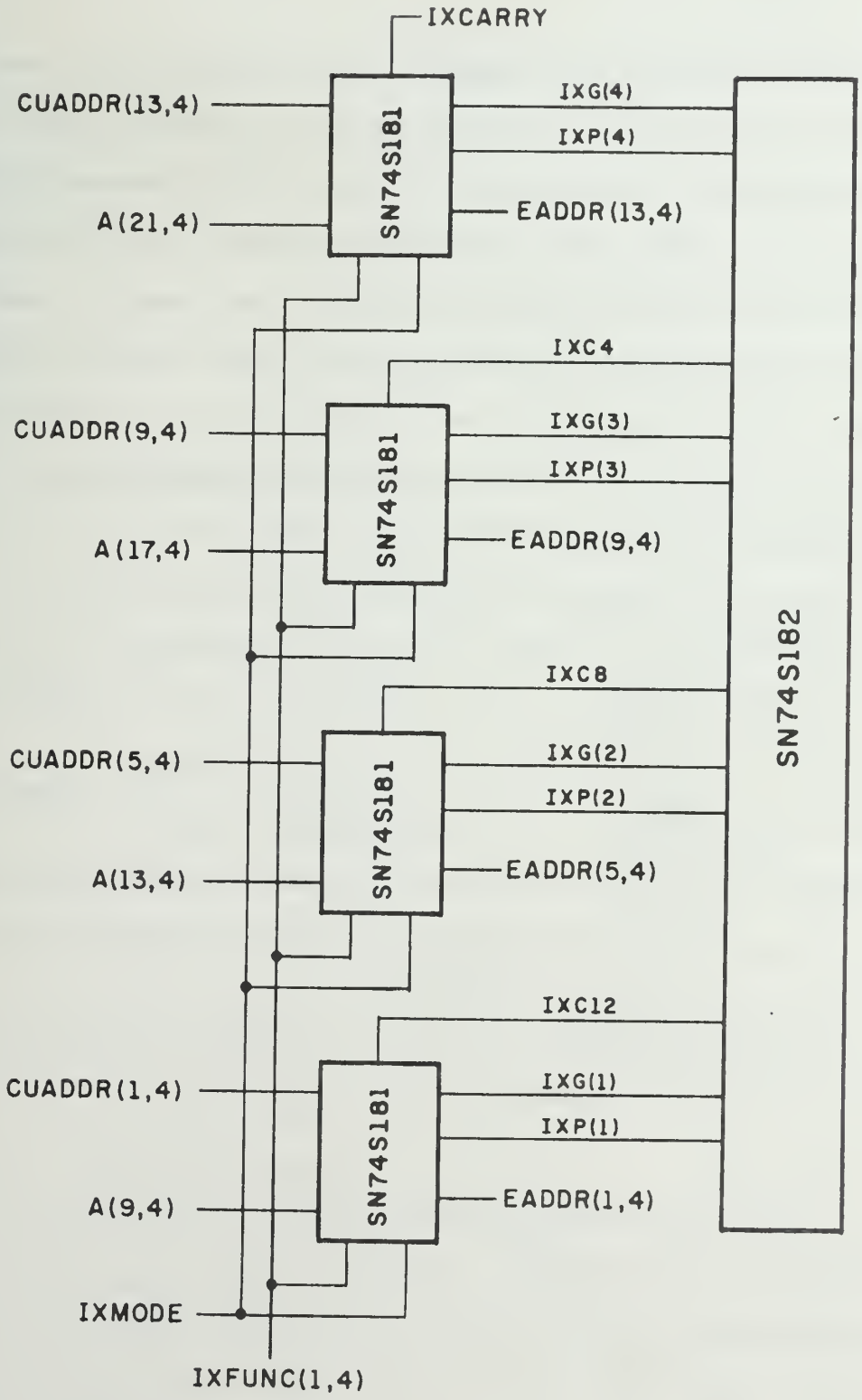


Figure 4.2.5.1.11-1 The Index Adder Logic

SN74S181 arithmetic-logic units augmented with an SN74S182 look-ahead carry generator. It is controlled by a function input, IXFUNC(1,4), and a carry input, IXCARRY, from the control unit. The address from the control unit, CUADDR(1,16), is combined with A(9,16) by the adder. The "A" bits, which come from the operand registers, are the low order sixteen bits of a twenty-four bit memory-length fraction. A twos-complement integer can be produced for use in indexing from a floating point value by performing an unnormalized addition with the value with fraction 80000000_{16} and biased exponent 46_{16}. Two examples of this operation are given in Table 4.2.5.1.11-1.

Initial Operands	Aligned Operands	Sum
46 80000000	46 80000000	46 80000100
41 10000000	46 00000100	
46 80000000	46 80000000	46 7FFFFF00
-41 10000000	-46 00000100	

Table 4.2.5.1.11-1 Two Examples of Processor Index Value Computation

The hexidecimal digits which are underlined in the Sum column of Table 4.2.5.1.11-1 are the part of the "A" operand which is one of the inputs to the index adder.

Indexing of centrally supplied addresses might also be performed by the main adder of the processor. To accomplish this, the control unit supplied address value must be gated to the adder. The least costly way to provide this gating is to replace four of the quadruple two-to-one selectors in the right operand selection logic of Figure 4.2-1 with eight dual four-to-one selectors.

This results in a net package count increase of four packages. The logic described here requires four packages if ripple carry operation is used with the SN74S181 arithmetic-logic units, and five packages - as shown in Figure 2.4.5.1.11-1 if carry look ahead operation is used. Even the ripple carry scheme is faster than requiring the operands and the result to pass through the alignment shifters and fraction selector which use of the main adder requires.

4.2.5.1.12 The Condition Flip-flops

This set of sections describes the five flip-flop which hold information about the results of operations in the processor. The state of each of these flip-flops is protected from being changed when the processor is disabled by having its mode value equal to zero. This control is provided by using the lower of the two CLOCK gating methods of Figure 4.2.5.1.12-1. These gates are not shown in the subsequent figures which illustrate the individual flip-flops. A control signal unique to each and the MODE value are used to produce a mode controlled clock pulse for each of the condition flip-flops.

All of the condition flip-flops are implemented with one-half of an SN74S74 dual flip-flop package. Both the true and complemented states are supplied for use by this package.

4.2.5.1.12.1 The Carry Flip-flop

Figure 4.2.5.1.12.1-1 shows the carry flip-flop and its associated control logic. Its state can be stored in a processor register (see section 4.2.5.1.7), and can be restored from a processor register by selecting the path which includes B(12). The carry out of the adder, ACOUT, can be used to set the state of the carry flip-flop, or it can be ORed with the previous

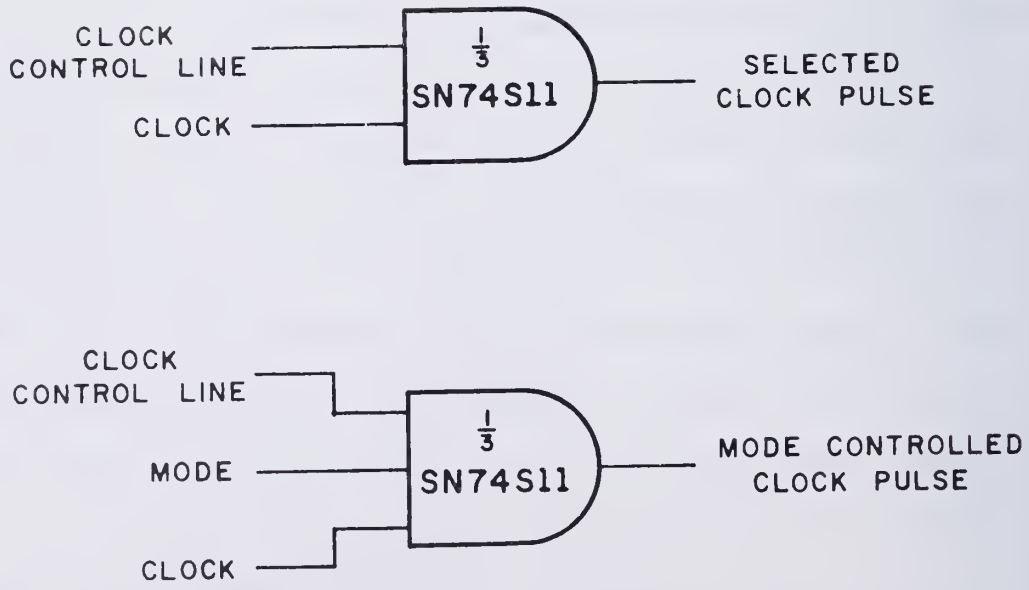


Figure 4.2.5.1.12-1 CLOCK Selection Logic

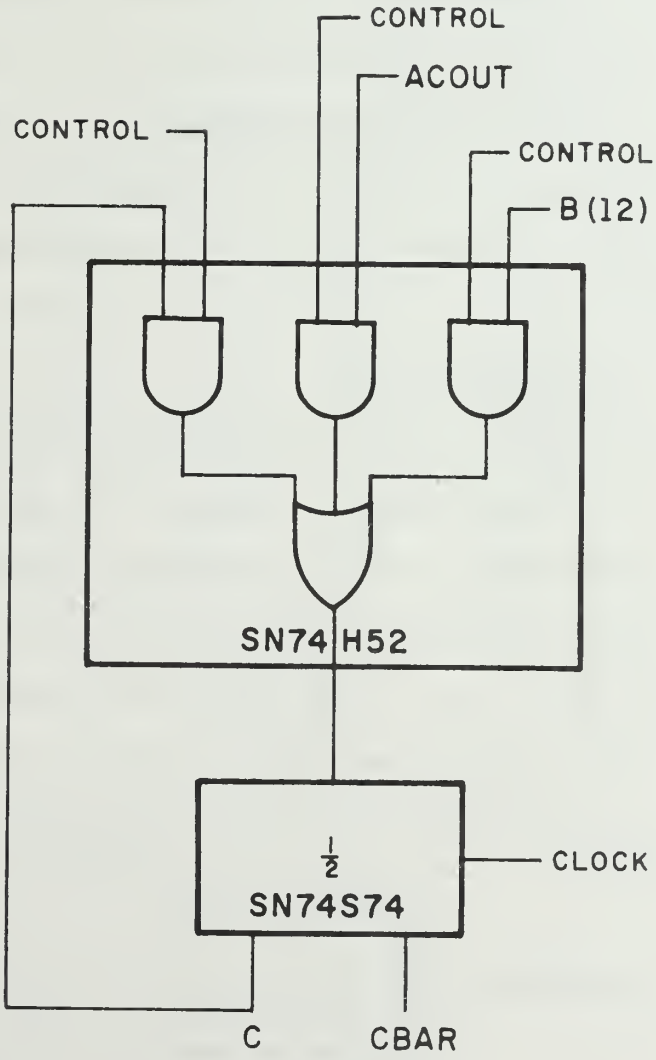


Figure 4.2.5.1.12.1-1 The Cary Flip-flop Logic

state by using the appropriate control signal values.

4.2.5.1.12.2 The Zero Flip-flop

Figure 4.2.5.1.12.2-1 shows the zero flip-flop and its associated control logic. Its state can be stored in a processor register (see section 4.2.5.1.7), and can be restored from a processor register by selecting the path which includes B(13). The primary input to the zero flip-flop is the output of a zero detect block (see section 4.2.5.1.1) which operates on the output of the fraction selection logic (of section 4.2.5.1.7). Previous states can be ORed or ANDed with a current state by using the appropriate signal values.

4.2.5.1.12.3 The Sign Flip-flop

Figure 4.2.5.1.12.3-1 shows the flip-flop and its associated control logic. Its state can be stored in a processor register (see section 4.2.5.1.7), and can be restored from a processor register by using the proper selection signals for the SN74S151 eight-to-one selector and the SN74S153 four-to-one selector shown in the figure. The control logic permits the sign flip-flop to be set to any of the values listed in Table 4.2.5.1.12.3-1.

S I G N A L	M E A N I N G
B(14)	A state presumably previously stored in a processor register
* & ÷	The exclusive OR of the operand signs
S670(4) wire-OR AFUNC(4)	The sign of a sum of difference (see section
EXPA(1)	The sign of the left operand
EXPB(1)	The sign of the right operand
RTESIGN	The sign of an operand from the routing unit
0	A forced positive sign; absolute value
1	A forced negative sign; minus the absolute value

Table 4.2.5.1.12.3-1 Possible Signs for a Result

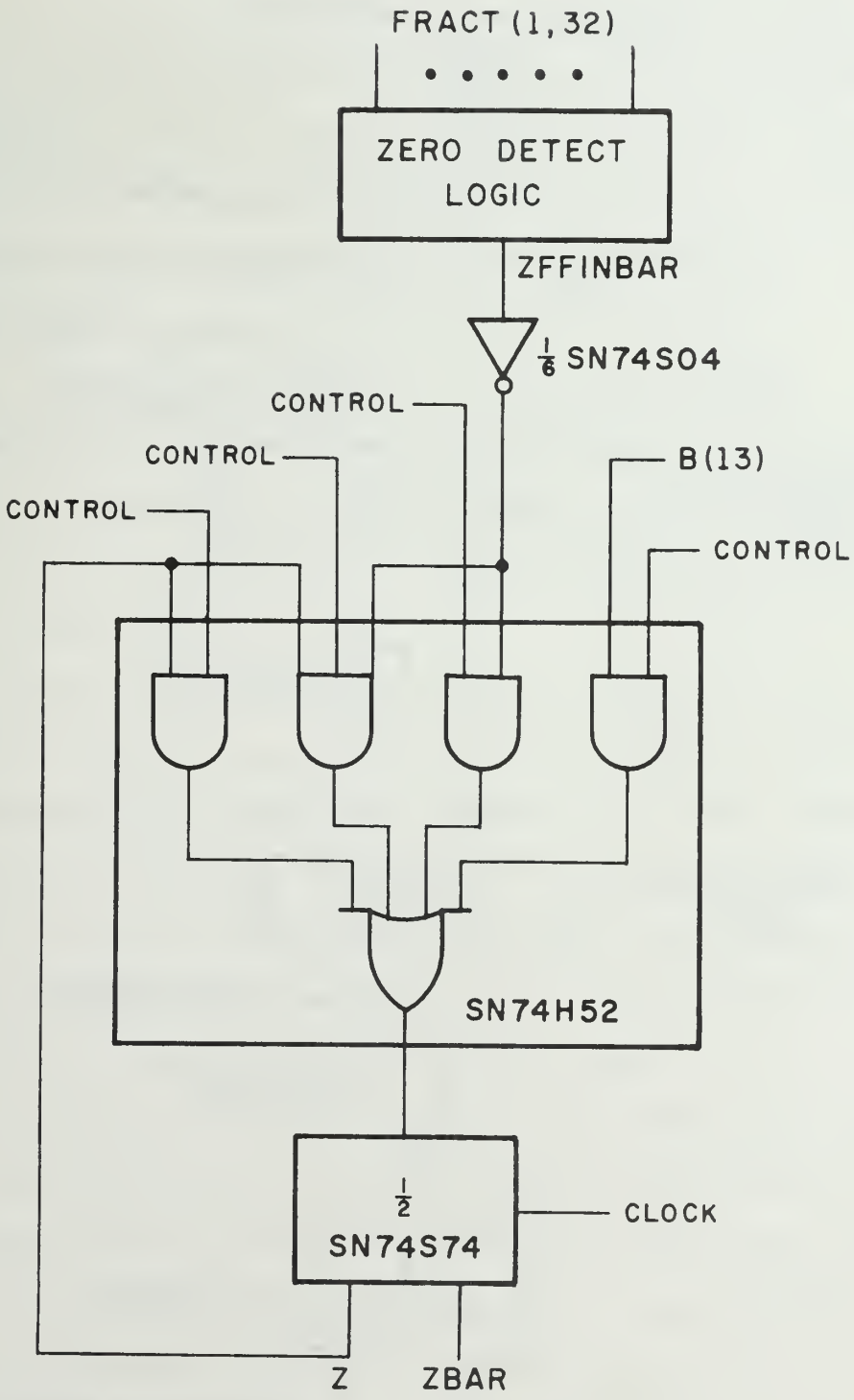


Figure 4.2.5.1.12.2-1 The Zero Flip-flop Logic

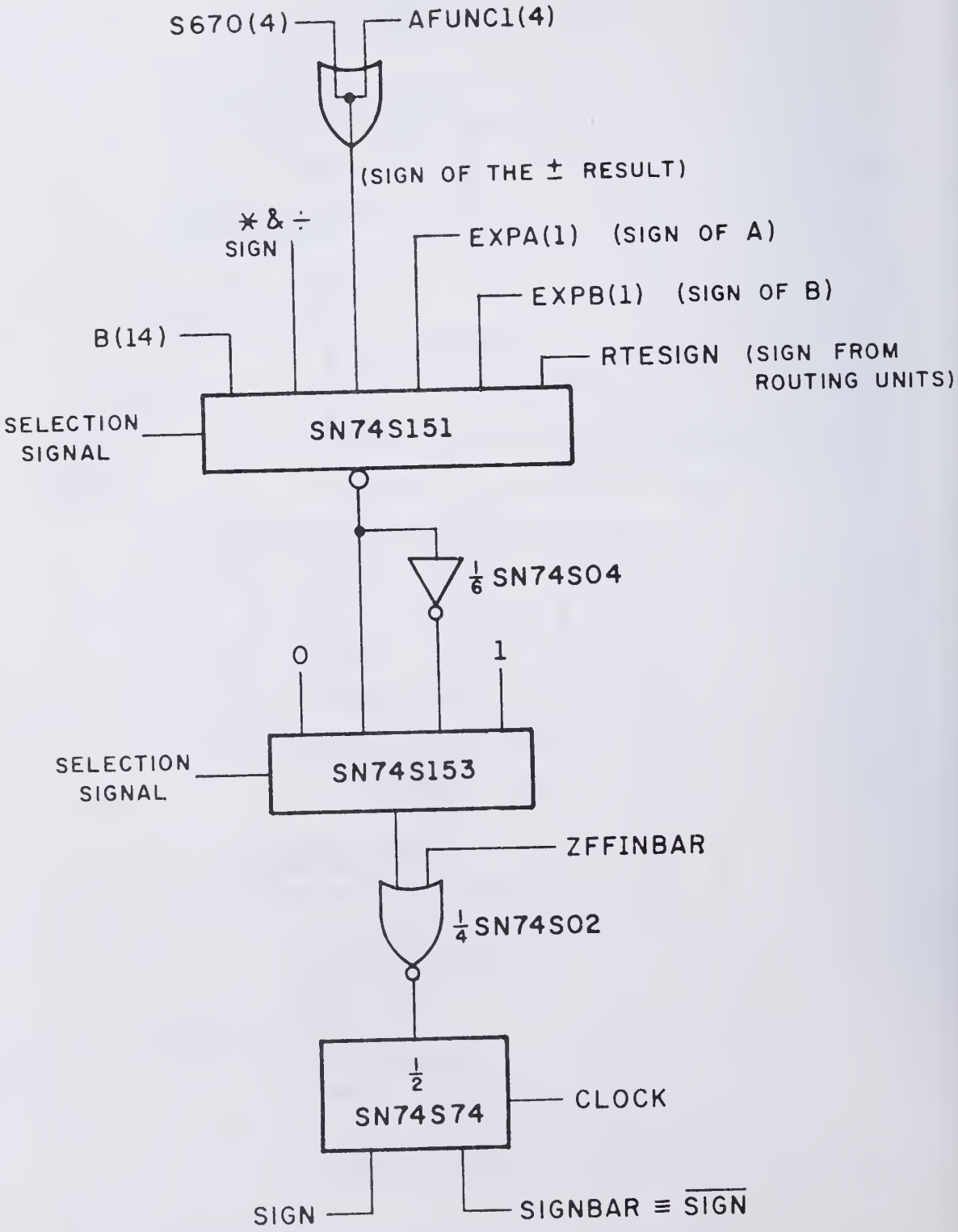


Figure 4.2.5.1.12.3-1 The Sign Logic and the Sign Flip-flop

The complement of any of the first six signs show in Table 4.2.5.1.12.3-1. The NOR gate between the SN74S153 and the flip-flop uses signal ZFFINBAR of the zero flip-flop logic (see section 4.2.5.1.12.3) to insure that the sign of a zero result is always a logic zero, or a positive sign. The NOR gate is used together with appropriate selection by the SN74S153 since no Schottky AND gate is available.

4.2.5.1.12.4 The Overflow Flip-flop

Figure 4.2.5.1.12.4-1 shows the overflow flip-flop and its associated control logic. Its state can be stored in a processor register (see section 4.2.5.1.7), and can be restored from a processor register by selecting the path which includes B(15).

In this design, an overflow condition exists when:

1. an exponent value which exceeds sixty-three is computed. This can occur in the Exponent Adder during the computation of the result exponent for multiplication or division; the signal EXO, described by the truth table in Table 4.2.5.1.12.4-1, is a logic one for this case. Fraction overflow necessitates increasing the exponent by one in the exponent correction adder; signals CORROVFL and EXP(7) cover this case.
2. a division by a zero fraction is attempted. The AZERO signal form the zero detect logic for the left operand fraction covers this case.
3. an attempt is made to integerize a floating point value whose integer part requires more than six hexidecimal digits. Signal INTRUNC, derived by the logic of Figure 4.2.5.1.12.4-2 covers this case.

A biased exponent with value V is represented by an exponent field value of 64+V. The sum of two exponents is:

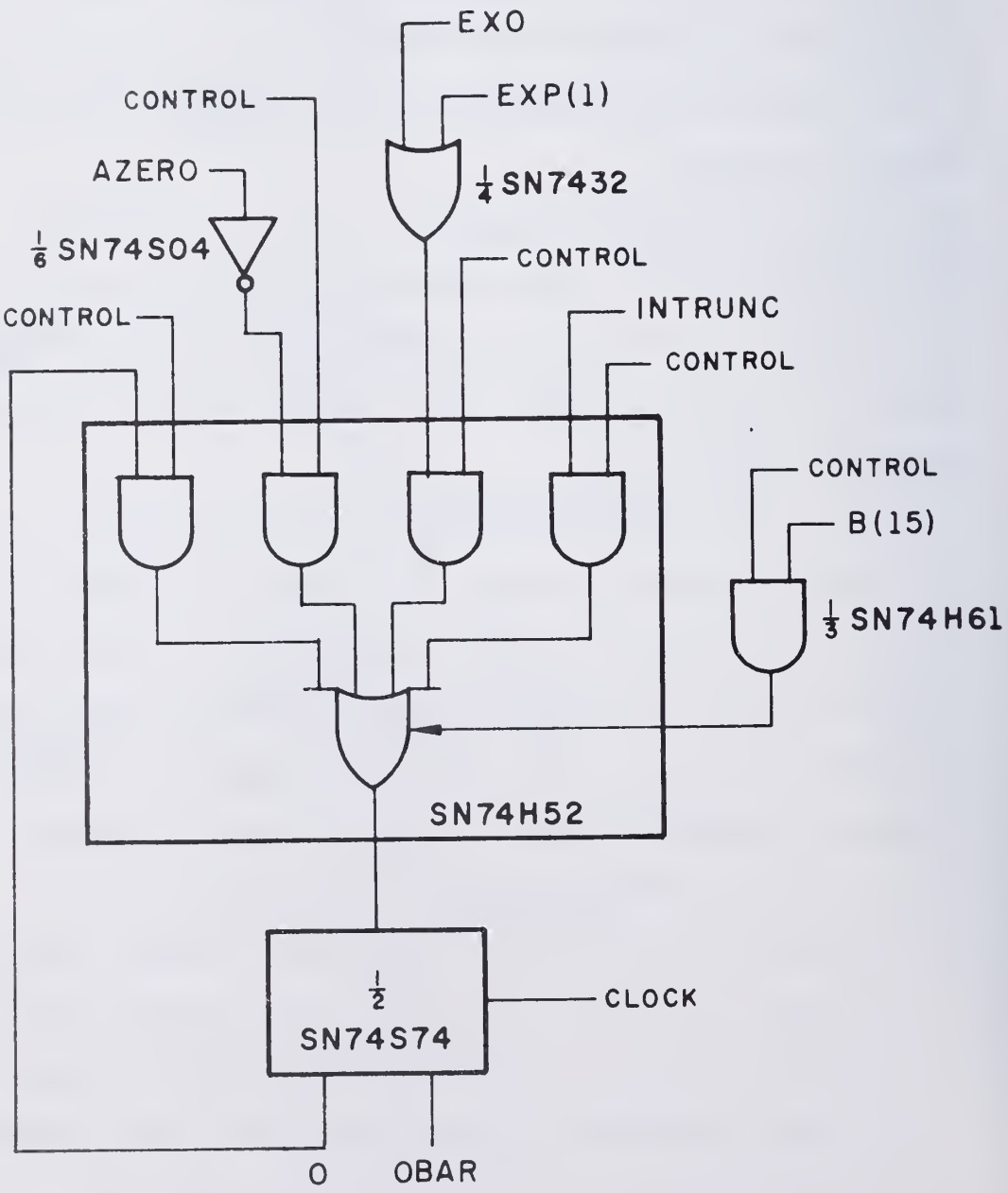


Figure 4.2.5.1.12.4-1 The Overflow Flip-flop Logic

$$\begin{array}{r} 64 + V1 \\ \underline{64 + V2} \\ 128 + V1 + V2 = 128 + V = S \end{array}$$

An overflow occurs when $64 \leq V \leq 63 + 63 = 126$, or when

$$192 \leq S \leq 254 \qquad (1)$$

A correct exponent results when $-64 \leq V \leq 63$, or when

$$64 \leq S \leq 191. \qquad (2)$$

Expressed in binary form, the above conditions are:

(1) 11xxxxxx

(2) 01xxxxxx(-64) or 10xxxxxx(63).

The difference of two exponents is:

$$\begin{array}{r} 64 + V1 \\ \underline{-(64 + V2)} \\ V1 - V2 = V \end{array}$$

An overflow occurs when $64 \leq V \leq 63 - (-64) = 127$. (3)

A correct exponent results when $-64 \leq V \leq 63$. (4)

Expressed in binary form, the above conditions are:

(3) 01xxxxxx

(4) 11xxxxxx(-64) or 00xxxxxx(63).

Conditions (1) through (4) can be implemented using an SN74S151 eight-to-one selector with the two high order bits of the result exponent and the exclusive OR of ABFUNC(2) and ABFUNC(3) bit selection code. Table .2.5.1.12.4-1 gives the truth table for this function.

ABFUNC(2) COR ABFUNC(3) 0 implies subtraction	EXC1(1)	EXC1(2)	EXO
0	0	0	0
0	0	1	1
0	1	0	x
0	1	1	0
1	0	0	x
1	0	1	0
1	1	0	0
1	1	1	1

Table 4.2.5.1.12.4-1 The Truth Table for Exponent Overflow Signal EXO

For both exponent addition and substraction, the straightforward arithmetic steps uniformly result in a bias bit which is incorrect. A correct biased result is produced when the bit in the bias position of the result is complemented after the arithmetic result has been computed.

During exponent correction, either one or zero is added to the component. The only way overflow can occur is that one is added to the biased exponent representation for an exponent of 63:

$$(64 + 63) + 1 = 128.$$

This has the binary form 10000000; in no other case does the result exponent have q high order one. Hence, the correct signal for overflow detection during exponent correction is EXP(1), the high order bit of the eight bit sum.

4.2.5.1.12.5 The Underflow Flip-flop

Figure 4.2.5.1.12.5-1 shows the underflow flip-flop and its associated control logic. Its state can be stored in a processor register (see section 4.2.5.1.7), and can be restored from a processor register by selecting the

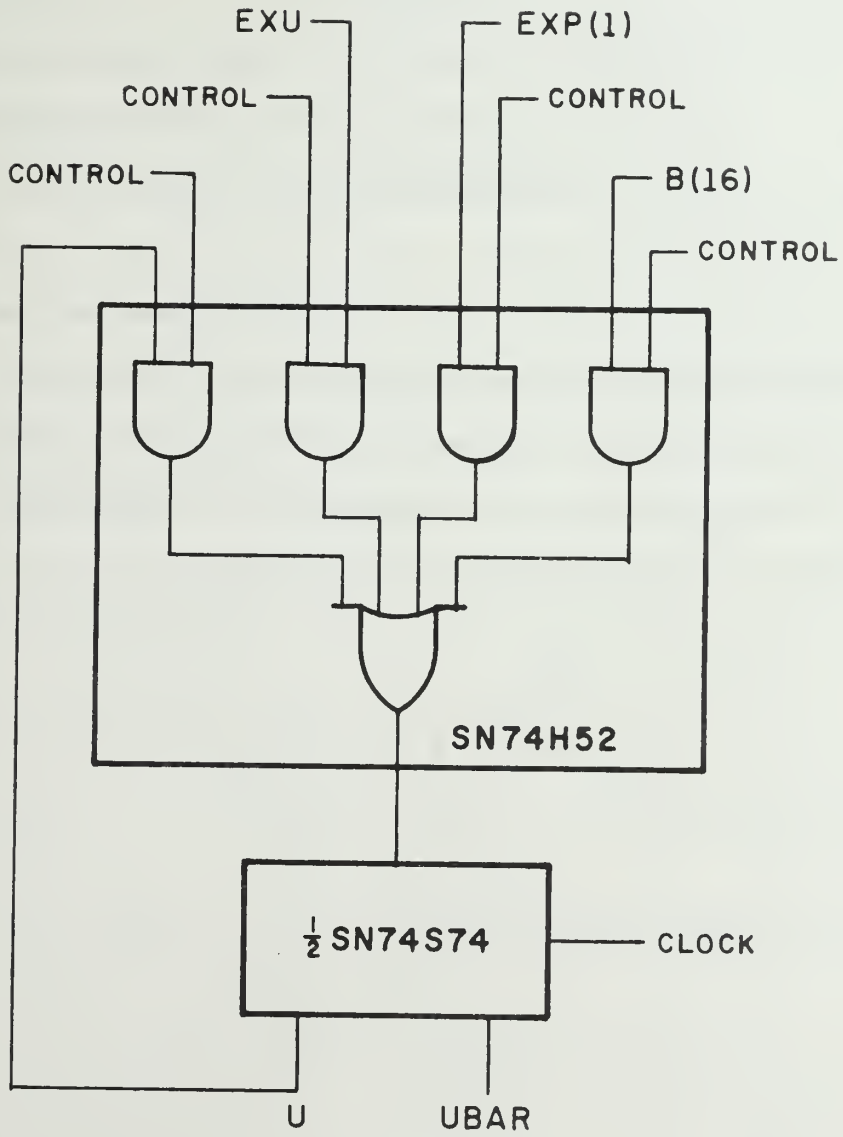


Figure 4.2.5.1.12.5-1 The Underflow Flip-flop Logic

path which includes B(16).

In this design, operand underflow occurs only when a result exponent which is less than -64 is computed. This can occur:

1. in the exponent adder during the computation of the result exponent for a multiplication or division; the signal EXU, described by the truth table in Table 4.2.5.1.12.5-1, is a logic one for this case.
2. when the value one is subtracted from an exponent value of -64 in the exponent correction adder. This occurs only during some division steps (see section 4.2.5.2.5). For this case, the initial biased exponent value is 00000000, and the result, 11111111, is the only case for which the high order result exponent bit, EXP(1), is a logic one.

A biased exponent with the value V is represented by an exponent field value of 64+V. The sum of two such exponents is:

$$\begin{array}{r} 64 + V1 \\ \underline{64 + V2} \\ 128 + V1 + V2 = 128 + V = S \end{array}$$

A underflow occurs when $-128 \leq V \leq -65$, or when

$$0 \leq S \leq 63. \qquad (1)$$

A correct exponent results when $-64 \leq V \leq 63$, or when

$$64 \leq S \leq 127. \qquad (2)$$

Expressed in binary form, the above conditions are:

(1) 00xxxxxxx

(2) 01xxxxxxx or 10xxxxxxx.

The difference of two exponents is:

$$\begin{array}{r} 64 + V1 \\ \underline{-(64 + V2)} \\ V1 - V2 = V \end{array}$$

An underflow occurs when $V \_ -65$. (3)

A correct exponent result when $-64 \_ V \_ 63$.

Expressed in binary form, the above conditions are:

(3) 10xxxxxx

(4) 11xxxxxx(-64) or 00xxxxxx(63).

Conditions (1) through (4) can be implemented using an SN74S151 eight-to-one selector with the two high order bits of the result exponent and the exclusive OR of ABFUNC(2) and ABFUNC(3) (see section 4.2.5.1.3) as the three bit selection code. Table 4.2.5.1.12.5-1 gives the truth table for this function.

ABFUNC(2) XOR ABFUNC(3) 0 implies subtraction	EXC1	EXC2	EXU
0	0	0	0
0	0	1	x
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	x

Table 4.2.5.1.12.5-1 The Truth Table for the Exponent Underflow Bit

For both exponent addition and subtraction, the straightforward arithmetic steps uniformly result in a bias bit which is incorrect. A correct biased result is produced when the bit in the bias position is complemented after the arithmetic result is computed.

During exponent correction, either one or zero is added to the exponent. The only way overflow can occur is for one to be added to the biased exponent representation for an exponent of 63:

$$(64 + 63) + 1 = 128.$$

This has the binary form 10000000; in no other case does the result exponent have a high order one. Hence, the correct signal for overflow detection during exponent correction is EXP(1), the high order bit of the eight bit sum.

4.2.5.2 Processor Function

The previous group of sections described several logic blocks in their own right without too much regard for their functions in support of processor operations. This set of sections describes how the logic blocks are integrated together to perform the high level operations. The details of the control signals and gating is given in these sections.

4.2.5.2.1 Normalization

A normalized floating point number in this design has a non-zero hexidecimal (four bit) digit as the leftmost digit of its fraction, unless the entire fraction is zero. The normalization process accepts an arbitrary floating point number and produces a normalized number with the same arithmetic value. A floating point zero is unchanged; a number whose fraction has a non-zero leftmost hexidecimal digit is unchanged. The fractions of all other floating point numbers are normalized by a left shift which makes the left-most fraction digit non-zero and introduces zero digits on the right for the zero digits shifted off the left. The exponent of the numbers so adjusted is reduced by one for each zero digit shifted off.

Figure 4.2.5.2.1-1 shows the control logic which computes the shift

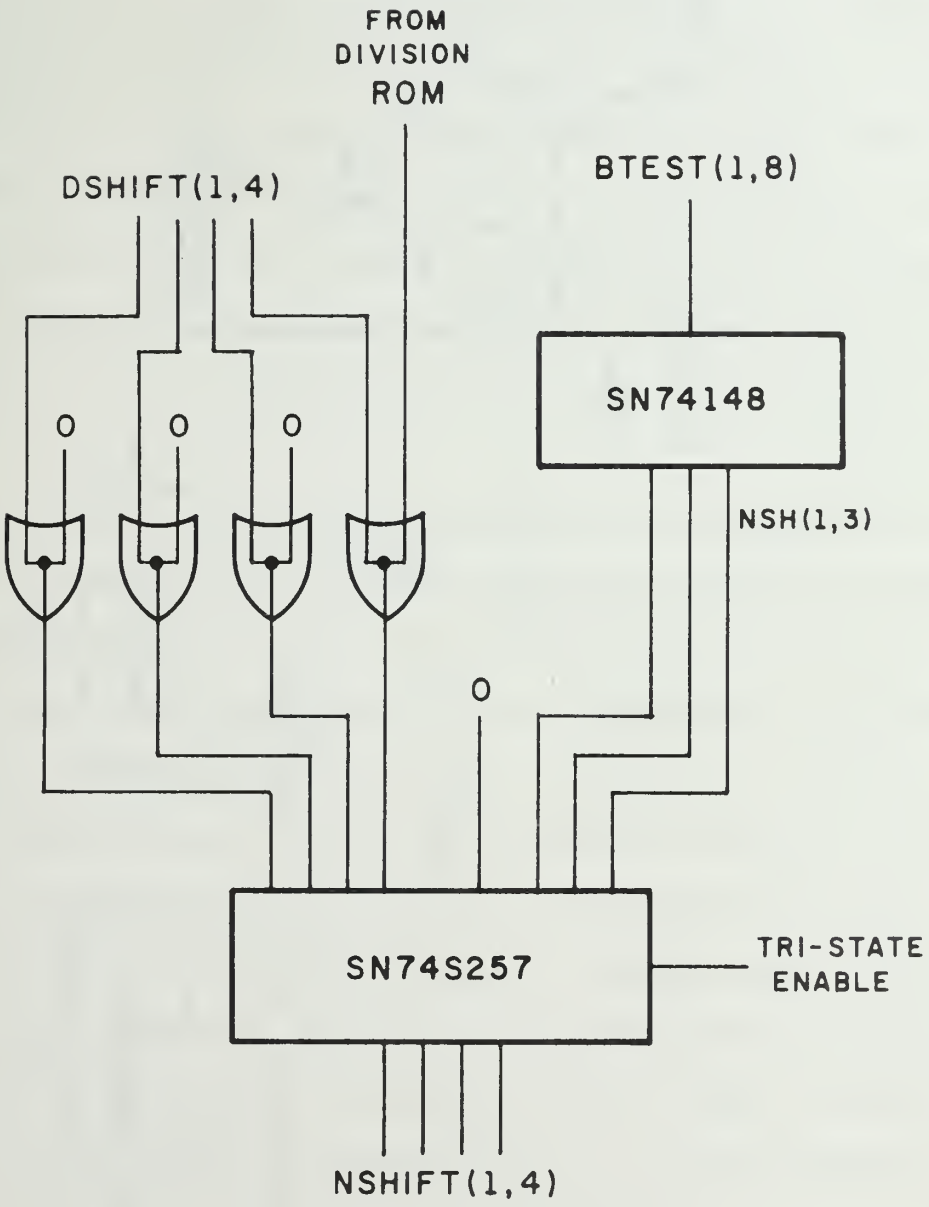


Figure 4.2.5.2.1-1 The Leading Zero Detection Logic

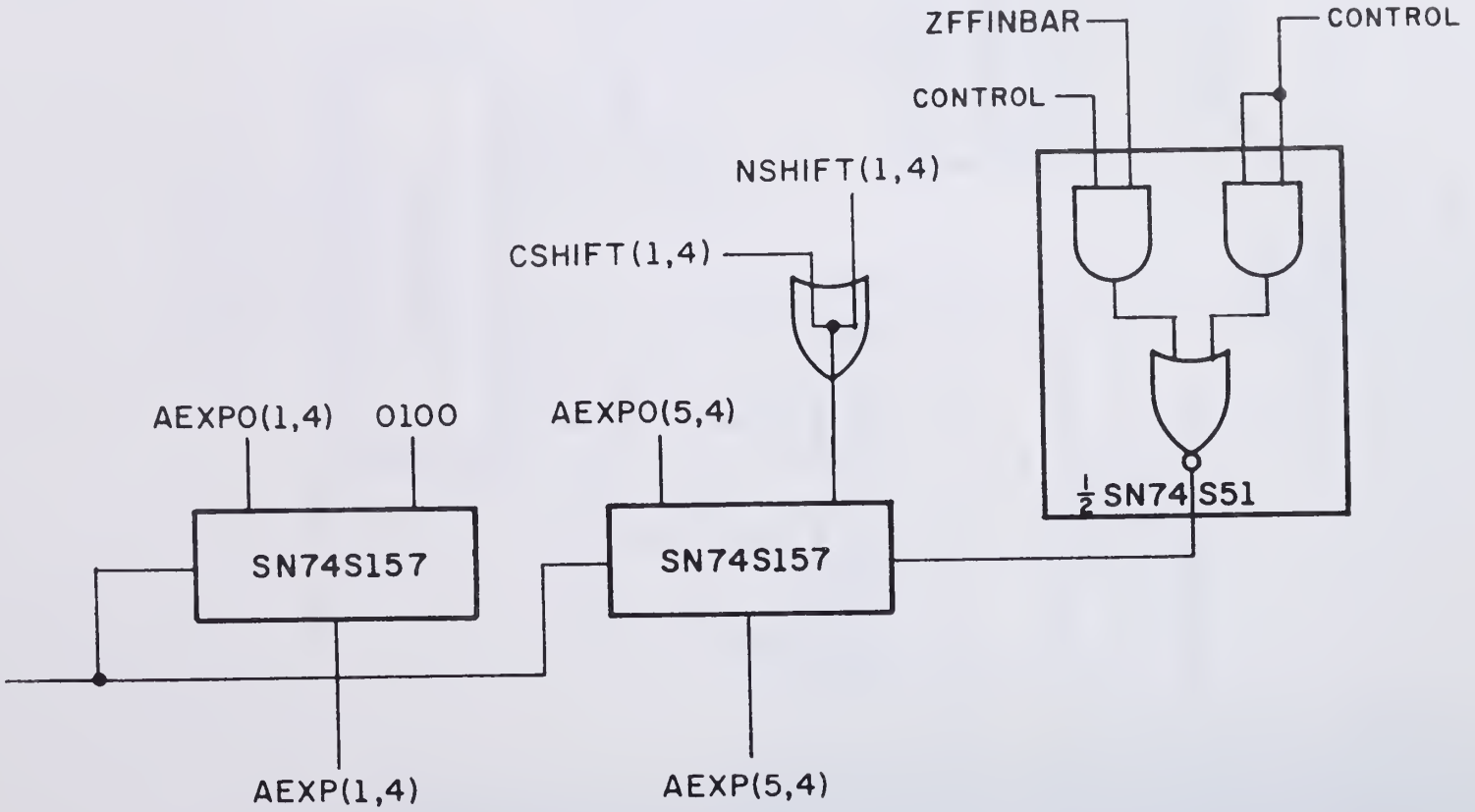


Figure 4.2.5.2.1-2 The Selection Logic for the "A" Exponent

amount for the normalize shift logic. The signal BTEST(1,8) comes from the SN74S260 gates of the zero detect logic for the right operand (see Figure 4.2-1 and Figure 4.2.5.1.1-1). BTEST(i) is a logic one if digit "i" of the left operand fraction is zero, numbering the digits from left to right. The SN74148 eight-line-to-three-line priority encoder accepts an eight bit input signal and produces a three bit output signal which is a count of the number of high order ones which occur in the input signal. The value seven is returned for input signals of all ones, which is the case for numbers with zero fractions.

During ordinary normalization, the output of the SN74148 is the left shift amount and also the number that must be subtracted from the exponent. It is selected by appropriate control by the SN74S157 two-to-one selector. NSHIFT(2,3) is sent to the normalize shift logic, and NSHIFT(1,4) goes to the selection logic for the exponent adder shown in Figure 4.2.5.2.1-2. This logic selects the "A" exponent for the exponent adder. Normally, it selects the exponent of "A" from the operand registers. For normalization, the operand (0100, NSHIFT(1,4)) is selected. Control signals enable the path for ZFFINBAR, the output of the zero detect logic for the result fraction, to the strobe input of the SN74S157 of Figure 4.2.5.2.1-2. When the fraction in question is zero, the output of the SN74S64 is one, so that the SN4S157 selector is disabled and supplies zeros rather than NSHIFT(1,4).

Although a shift of seven places is the largest that occurs during normalization, there are cases during double precision addition/subtraction when a value of up to twelve must be subtracted from the exponent. For these cases, a four bit NSHIFT value is provided. See section 4.2.5.2.7 for details.

The CSHIFT(1,4) signal is supplied by the control unit during multiplication and division by a power of two operations. See section 4.2.5.2.10 for the details of this operation.

4.2.5.2.2 Rounding

The fraction size of memory words and multiplier operands is twenty-four bits, and that of processor words is thirty-two bits. A rounding operation is included in the design to permit rounding a thirty-two bit processor fraction to a twenty-four bit memory and multiplier length fraction. The rounding is accomplished by adding one in bit position twenty-four of the fraction when position twenty-five is a one. The fraction passes through the logic as the right operand. Bit twenty-five of that fraction is selected by the left operand selector as bit twenty-four of a fraction that is zero in every other bit position (see section 4.2.5.1.5). The other bit positions are forced to zeros by disabling the left alignment shift network. The exponent of the result is that of the right operand, selected by control signals to the exponent selection part of the exponent correction adder (see section 4.2.5.1.8). The two fractions are added by the adder under control unit control, using CUAFUNC(1,3) for function specification (see section 4.2.5.1.6) Fraction overflow and the corresponding exponent adjustment by the exponent correction adder can occur. The sign of the result is the sign of the right operand.

4.2.5.2.3 Floating Point Addition

A floating point value in this design is represented by a sign bit, a non-negative proper fraction and an integer power of sixteen. The fraction parts cannot be correctly added until they are adjusted for the difference in

their exponents. In this design, this adjustment is made by shifting the fraction whose exponent is smaller right by the number of digit positions by which the exponents differ. The process, described in terms of Figure 4.2-1, proceeds as follows:

The exponent difference is computed by the exponent adder. The difference, together with a pair of one bit signals which each indicate whether one of the operand fractions is zero, is used by the pre-align control logic to specify which of the operands is to be shifted right. At least one of the alignment shift logic blocks performs a shift of zero places during each floating point addition. The other alignment shift logic is disabled when the shift amount exceeds seven. The pre-align control logic also selects the exponent of the result.

The correctly aligned fractions proceed through the operand selectors, adder, and fraction selector to the operand registers. The result of this processing cycle is an un-normalized floating point sum or difference with a correct exponent. If a normalized result is sought, another cycle is used. The fraction passes through the leading zero detection logic of Figure 4.2.5.2.1-1, which determines the left shift amount required for normalization. This shift amount is used by normalization shift logic to perform the fraction shift, and by the exponent adder to compute the correct exponent for the normalized result.

The addition process is complicated by the fact that sign-magnitude representation is used for floating point values in this design. The actual operation which the adder must perform depends not only on the instruction being executed, but also on the signs and the relative magnitudes of the

operands being processed. If one of the operands is zero, the result is the other operand. If two operands with equal exponents are to be added, the actual operation performed by the adder depends on their signs. When the signs are the same, the adder must add the two magnitudes; the result sign is that shared by the two operands. However, when the signs differ, the smaller magnitude must be subtracted from the larger, and the sign of the result is that of the larger operand. The SN74S381 arithmetic-logic unit is ideally suited to these circumstances, because it can perform the A+B, A-B, and B-A operations (see Table 4.2.5.1.3-1).

When the argument exponents differ, the operand with the larger exponent is the larger in absolute value without regard to the fraction values involved. Hence, an exponent comparison is also required to determine what SN74S381 operation to perform. Table 4.2.5.2.3-1 summerizes the ten input signals which are required to determine the operation which is performed by the SN74S381 arithmetic-logic units of the adder. Figure 4.2.5.2.3-1 shows the logic which implements Table 4.2.5.2.3-1. During floating point addition and subtraction, the wire OR network of Figures 4.2.5.1.6-2 and 4.2.5.1.6-3 makes AFUNC the same as AFUNC1 by appropriate enabling of the tri-state signals The ABEXEQ signal is derived by the logic of Figure 4.2.5.2.3-2. When the absolute value of the exponent difference is zero, the exponents are equal, and ABEXEQ is a logic zero.

A fraction overflow can occur only when the function performed by the SN74S381 arithmetic-logic units of the adder is A+B. The signal OVFLSEL is implemented by an SN74S151 eight-to-one selector which uses AFUNC(1,3), the SN74S381 function specification, as its selection signal. The input to

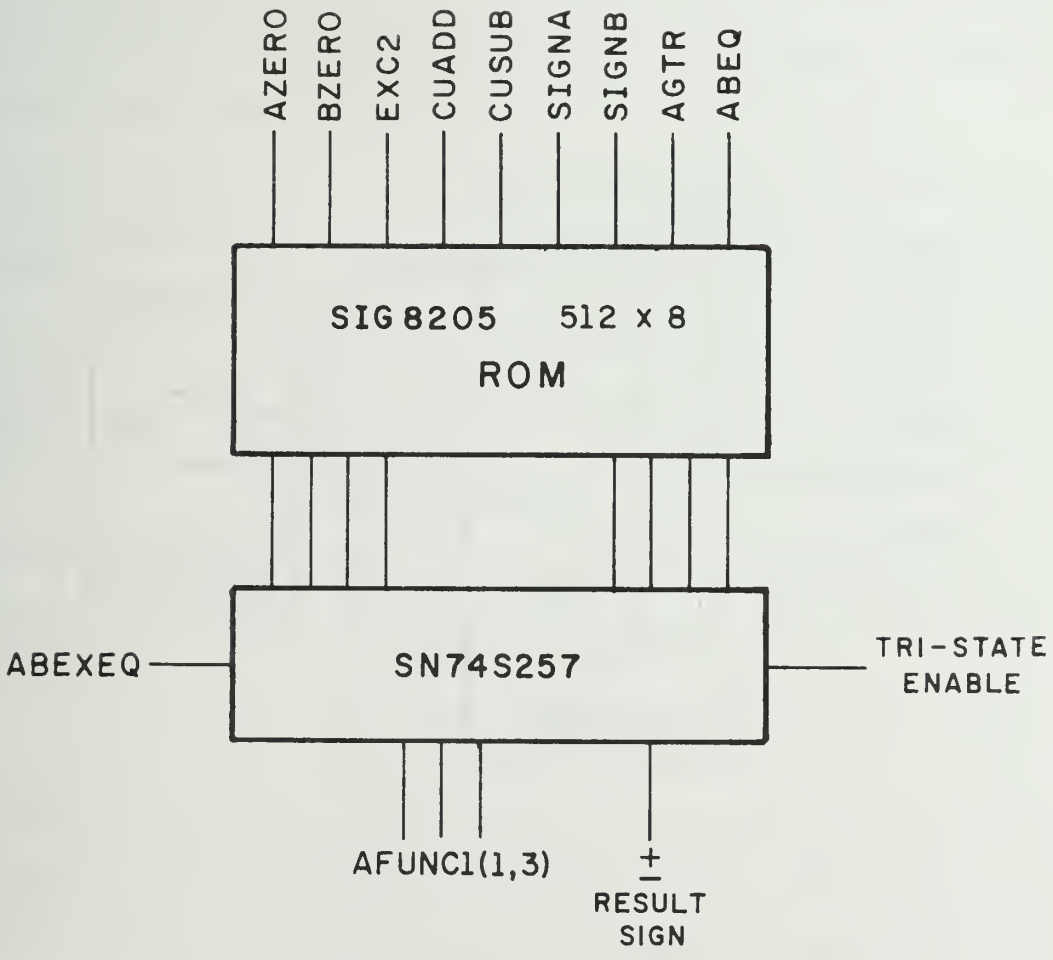


Figure 4.2.5.2.3-1 The Logic which Selects the Adder Function During Addition and Subtraction

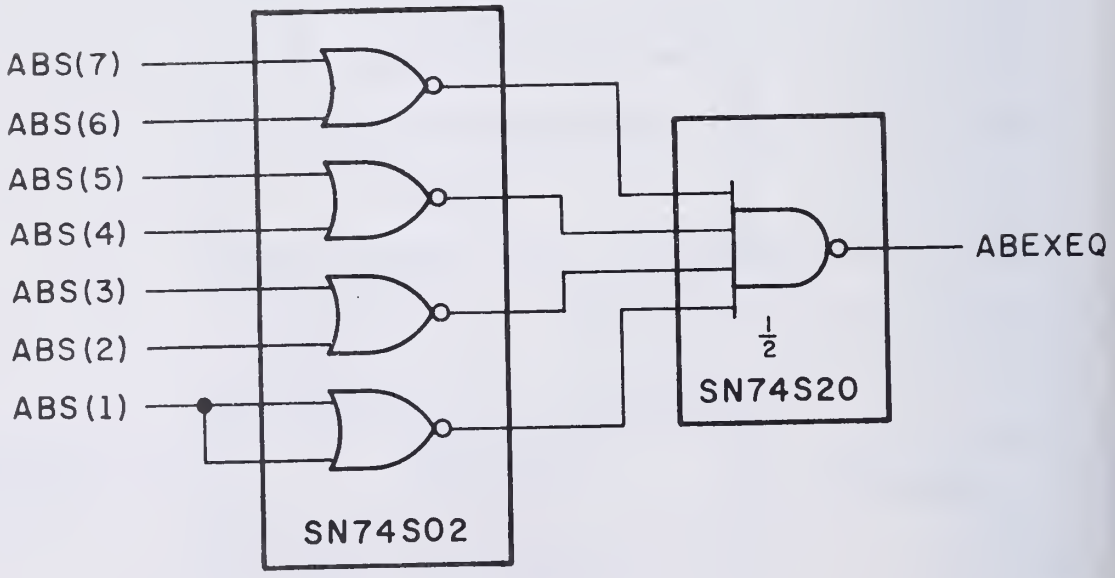


Figure 4.2.5.2.3-2 The Logic for the ABEXEQ Signal

the SN74S151 is a logic one in every position except that which corresponds to AFUNC(1,3)= 011; for the latter case, the selector input is ACOUT, the high order carry out of the adder. A logic zero value for OVFLSEL thus indicates a fraction overflow. The OVFLSEL signal is used by both the fraction selection and the exponent correction logic.

Signal	Value	Meaning
ABEQEQ	0	The two operand exponents have the same value.
AZERO	0	The left operand (A) fraction is zero.
BZERO	0	The right operand (B) fraction is zero.
EXC2	0	The exponent of the right operand exceed that of the left operand.
CUADD	1	The operation specified is addition.
CUSUB	0	When CUADD is zero, subtract the right operand from the left; that is B-A.
	1	When CUADD is zero, subtract the left operand from the right; that is A-B.
SIGNA	0	The left operand is greater than or equal to zero.
SIGNB	0	The right operand is greater than or equal to zero.
AGTR	1	The unshifted left fraction exceeds the unshifted right fraction.
ABEQ	1	The unshifted fractions are equal.

Table 4.2.5.2.3-1 The Input Signals for the Adder Function Logic

Table 4.2.5.1.3-1 which lists the functions and function codes for the SN74S381 arithmetic-logic unit of the adder indicates that the carry into the adder depends on the function code. The logic of Figure 4.2.5.2.3-3 shows how the carry into the adder is determined. Since the adder operates with

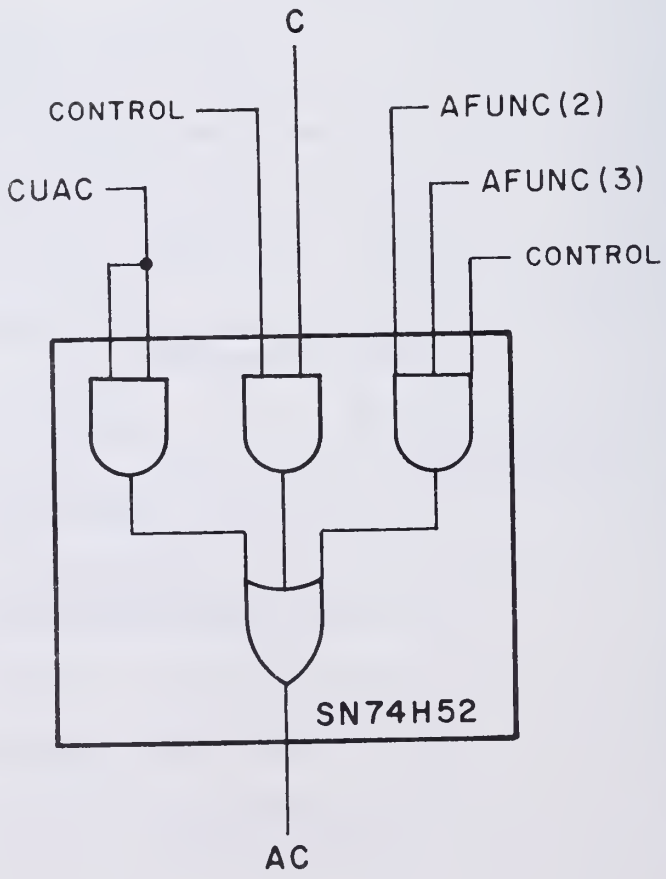


Figure 4.2.5.2.3-3 The Logic for the Carry into the Adder

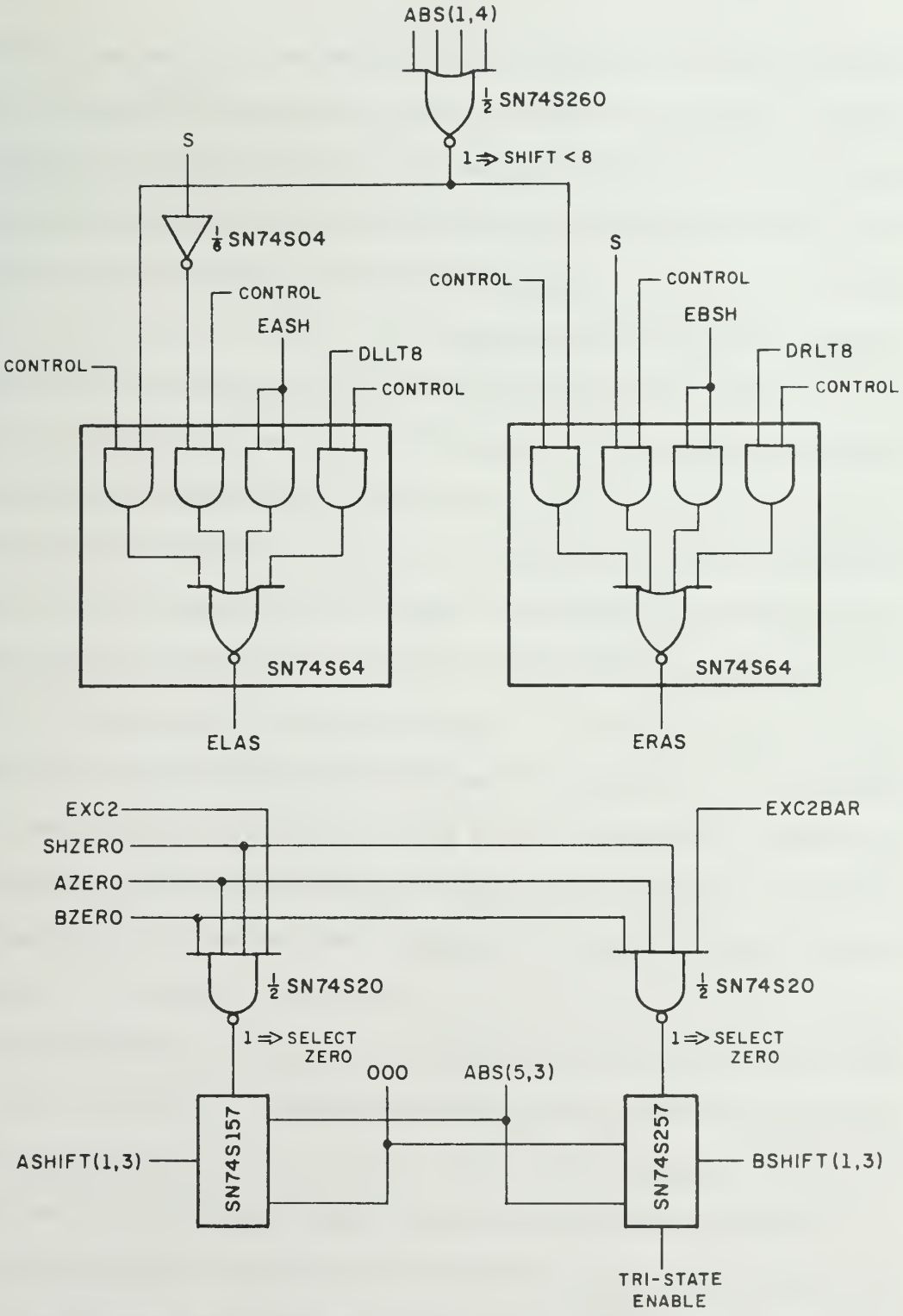


Figure 4.2.5.2.3-4 The Alignment Shift Conrtol Logic

active low data, a one carry in is required for addition and a zero for subtraction. The control unit can specify the carry by using control signal CUAC. When the adder function is determined in the processor by the logic of Figure 4.2.5.2.3-1, the path which uses added function bits produces the correct carry in. The carry flip-flop output, C, is used as the carry in to the adder during double precision operations.

The logic which controls alignment shifting during floating point addition and subtraction is shown in Figure 4.2.5.2.3-4. The signal ELAS is the enabling signal for the left alignment shift logic, and ERAS is that for the right alignment shift logic. The signals EASH and EBSH permit control unit specification of the shift enables without regard to local conditions. The two signals DLLT8 and DRLT8 come from the double precision control ROM, and the signal S is derived from the logic of Figure 4.2.5.2.7-3. Bits one through four of the absolute exponent difference, ABS(1,4), are combined by an SN74S260 NOR gate to yield a signal which is a logic one when the alignment shift amount is less than eight. The actual shift amount is either ABS(5,3) or zero under the control of a pair of shift selection signals which uses AZERO, BZERO and EXC2 of Table 4.2.5.2.3-1 along with a control unit signal SHZERO. When any of the preceeding signals is a logic zero, the shift selections signal one, and a zero shift amount is selected.

4.2.5.2.4 Multiplication

Measurements of the current model's execution on the IBM/360 revealed that approximately one-half of the floating point instruction executed are multiplications. Therefore, we have designed a high speed fully parallel multiplier. The details of this work are given in a Masters thesis by Mr. William

Stenzel (1975). Because the amount of hardware necessary for this multiplier varies as the square of the operand lengths, we chose to implement a twenty-four by twenty-four bit multiplier. The rounding operation described in section 4.2.5.2.1 rounds floating point values to this fraction precision. The integrated circuits used in the multiplier are:

1. the SN74S274 read only memory which accepts an eight bit address and returns an eight bit result. It is pre-programmed to accept two four bit digits and return their eight bit product (Texas Instrument Corporation, 1974; pp. 262-270),
2. the Signetics N8228 read only memory which accepts a ten bit address and returns a four bit operand. This device, available as Signetics part number N8228-CB1105, is programmed to add five two bit numbers and produce a four bit sum,
3. the SN74283 four bit binary full adder, which accepts two four bit inputs and a carry input, and produces a four bit sum and a carry output, and
4. the SN74S381 arithmetic-logic unit which is used together with SN74S182 look-ahead carry generators to a final addition step in the mulitplication process.

Figure 4.2.5.2.4-1 illustrates how to compute the product of two eight bit values using four SN74S274 read only memories. Each subscripted symbol in the figure represents a four bit digit. The four eight bit products are displayed in the familiar trapaziodal form and have also been rearranged in a triangular form. Four bit adders can be used to sum the partial products to yield the required product. Figure 4.2.5.2.4-2 shows the triangular rearrangement for all of the bits in the product of two twenty-four bit operands. A

$$
\begin{array}{cccc}
 & a_1 & a_0 & \\
 & b_1 & b_0 & \\
\hline
 & a_1b_0 & a_0b_0 & \\
a_1b_1 & a_0b_1 & \Rightarrow & a_1b_1 \quad a_1b_0 \quad a_0b_0 \\
 & & & \quad\;\; a_0b_1
\end{array}
$$

Figure 4.2.5.2.4-1 The Product of Two Eight Bit Values

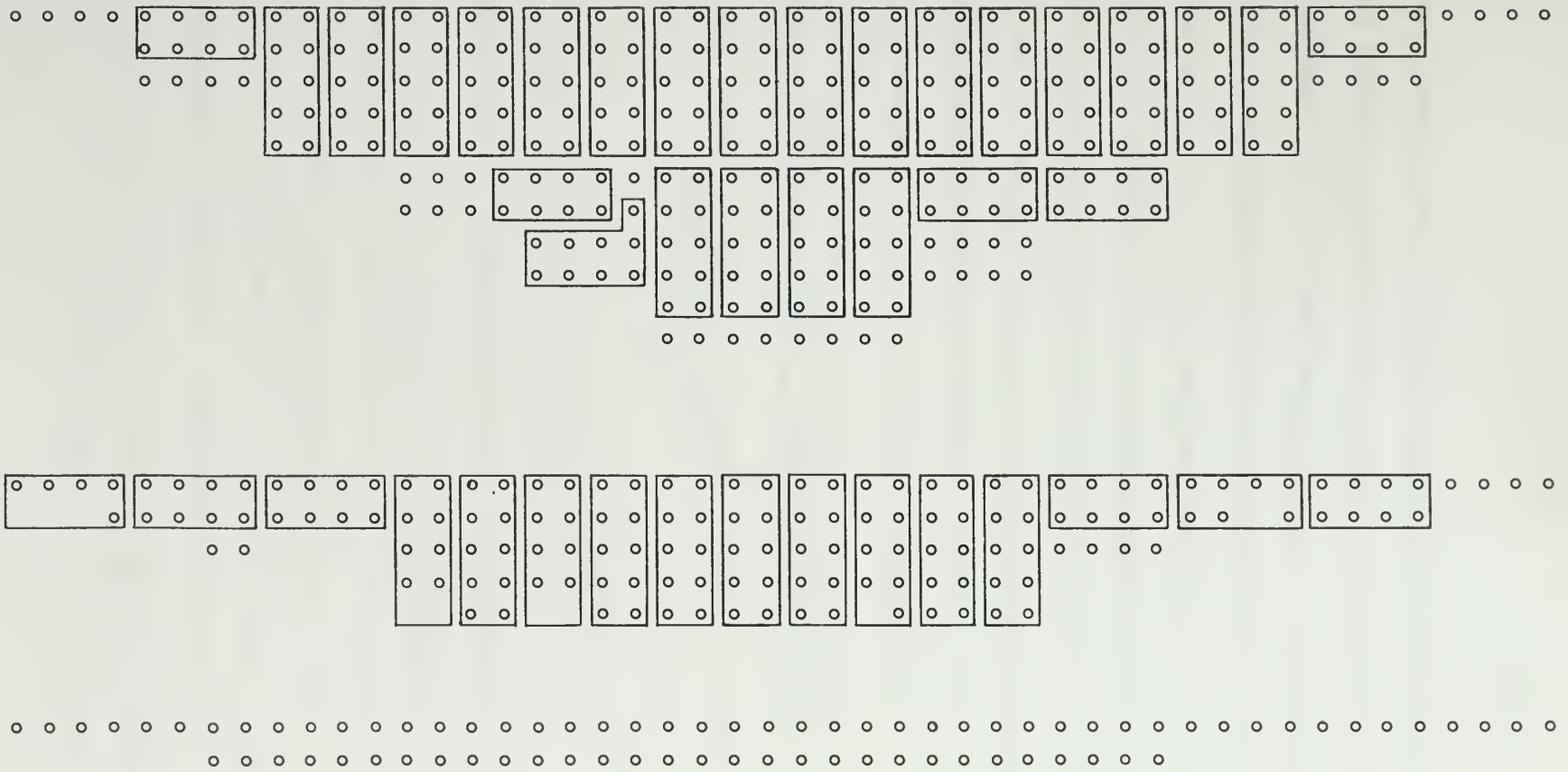


Figure 4.2.5.2.4-2 The Twenty-four by Twenty-four Bit Multiplier

three stage reduction processes results in the required product.

The vertical rectangles in the figure represent Signetics 8228-CB1105 read only memories. The five high order bits of the address, pins three through seven, accept the left column of bits - the high order bits of the five two bit input operands. The five low order bits of the address, pins one, two and thirteen through fifteen, accept the right column of bits - the low order bits of the five two bit input operands. The low order bit of the four bit sum appears on the output pin twelve, the low order bit of the output word.

The horizontal rectangles represent SN74283 four bit adders.

In the first reduction state, the eleven rows of partial product bits are reduced to five rows by using twenty Signetics 8228's and six SN74283's. In the second stage, ten 8228's and six SN74283's reduce the five rows to two. Nine SN74S381's and three SN74S182's produce the forty-eight bit product in the last stage.

4.2.5.2.5 Division

Three different division algorithms were examined as candidates for use in this design. They are all similar in two respects:

1. Each algorithm uses the multiplier.
2. Each algorithm uses read only memories to store values which it needs.

The first scheme used a quadratic Chebyshev fit to the reciprocal, stored the coefficients in read only memories, and used the multiplier to evaluate the quadratic polynomial. The scheme is not workable because the polynomial coefficients are relatively large and oscillate in sign, so that a reciprocal accurate to twenty-four bits could not be computed with the twenty-four bit multiplier.

The second scheme multiplies both numerator and denominator by cleverly chosen constants (Garcia, 1974). Two multiplications of both numerator and denominator reduce the denominator to one and the numerator to the required quotient. The denominator must be normalized so that there is a one in the high order bit. Call the high order eleven bits of this normalized denominator "A", and the low order thirteen bits "B". We can compute a twenty-four bit reciprocal of "A" with six Signetics N8228 read only memories which accept a ten bit address and report a four bit result. We can use only ten bits of "A" since the high order bit is known to be a one. The following sequence of equations illustrates the technique:

$$\frac{N}{D} = \frac{N}{A+B} = \frac{N(1/A)}{(A+B)(1/A)} = \frac{N(1/A)}{1+B/A} = \frac{N(1/A)(1-B/A+(B/A)^2)}{(1+B/A)(1-B/A+(B/A)^2)} =$$

$$\frac{N(1/A)(1-B/A+(B/A)^2)}{1+(B/A)^3}$$

By construction, B is less than 2^{-11}, and A is greater than or equal to one-half. Therefore, B/A is less than 2^{-10}, so that $(B/A)^3$ is less than 2^{-30}, and is therefore negligible in computing a twenty-four bit quotient. Four multiplications are necessary to compute the quotient using this scheme:

1. $N(1/A)$
2. B/A from B and $1/A$
3. (B/A)
4. $N(1/A)(B/A+(B/A)^2)$

The third scheme uses Newton's iterative methods. The function

$$f(x) = Dx-1$$

will converge to the reciprocal of "D". The derivative f'(x) = D, so that the equation for the iteration are

$$x_{n+1} = x_n - \frac{Dx_n - 1}{D} = x_n + \frac{1}{D}(1-Dx_n)$$

which is identically equal to 1/D. The term 1/D is the sought and unknown reciprocal. However, x_n is approximately equal to the reciprocal, so that the iteration becomes

$$x_{n+1} = x_n + x_n (1-Dx_n).$$

The analytically equivalent form

$$x_{n+1} = 2x_n - Dx_n^2$$

can not be computed with as much accuracy as can the preceeding form with the given processor.

The denominator "D" whose reciprocal is sought must be normalized in the usual binary sense; that is, its high order bit must be a one. An initial twelve bit approximation, x_0, is obtained from three Signetics N8228 read only memories by using A(2,10) (see Figure 4.2.5.2.5-1) as address bits; A(1) is known to be a one. In this scheme, however, the high order part of D should be rounded by adding 2^{-12} after the left shift which guarantees that the high order bit of D is a one.

Programs were written to simulate all three schemes. In the iterative case, two iterations were always performed; no convergence test was done. Therefore, the scheme requires a total of five multiplications to compute a quotient; two multiplications are needed for each iteration, and a final multiplication is required to compute the quotient from the reciprocal.

The simulation programs for the second and third schemes accepted four parameters:

1. the desired numerator,
2. the initial denominator,
3. the increment between successive denominators, and
4. the final denominator.

The programs computed all quotients for the indicated range of denominators. Two pairs of simulation programs were written. One pair computed quotients correct to twenty-eight bits and compared the approximate values to them. The second pair of programs computed a quotient rounded to twenty-four bits for each denominator, and compared similarly rounded approximate quotients to them. The results of tests using these programs are given in Table 4.2.5.2.5-1. These results led to the choice to implement the third scheme.

The implementation of the third division scheme uses four processor registers; registers zero to three are used. The first step in the process is to move the original denominator to register zero. This is necessary because one of two tri-state sources supplies the operand to the normalization shifters. The normal source is the two-to-one selectors in the upper right corner of Figure 4.2-1. The operand from memory enters the processor through these selectors. The other source is the zero-to-three bit shift logic discussed below. A denominator from memory would enter the normalization shift logic from two sources when a zero to three bit shift is performed if B(1,32) of Figure 4.2.5.2.5-1 were to come from the memory operand selectors of Figure 4.2-1. Hence, the B(1,32) operand unit must come from the registers. Another implication of this is that the two-to-one selectors

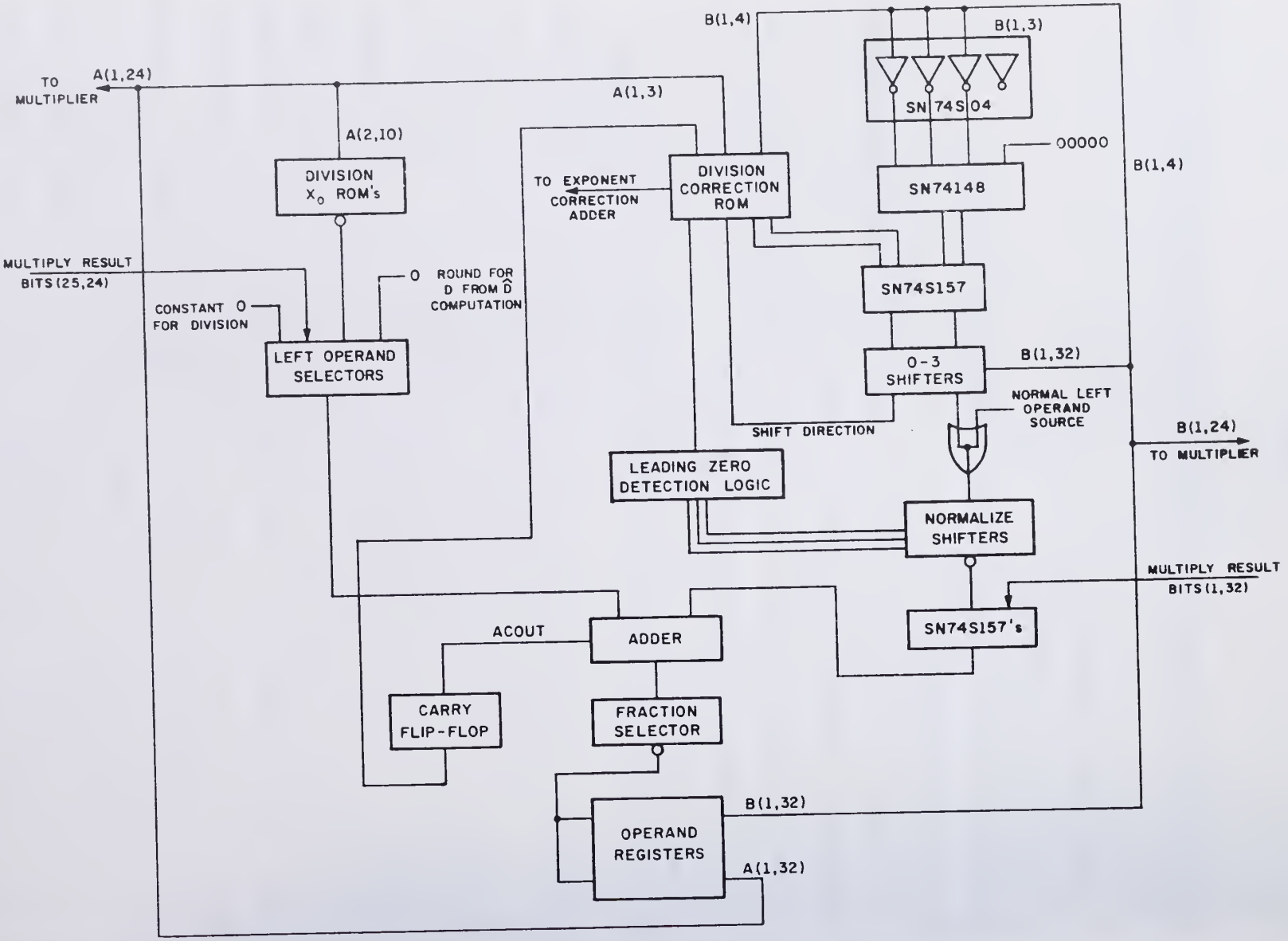


Figure 4.2.5.2.5-1 The Subset of the Processor Logic which Performs Division

Numerator	100000_{16}	(i.c. 1/16)
Initial denominator	100000_{16}	
Final denominator	200000_{16}	(i.c. 2/16)
Increment	1	(i.c. 2^{-24})

Item	28-bit Quotient		28-bit Rounded Quotients	
	Multiplicative Method	Newton's Method	Multiplicative Method	Newton's Method
Sum of Absolute Values of Errors	$7CDA1.E_{16}$	$850B2.3_{16}$	$7CE84.0_{16}$	$800C9.0_{16}$
Average Absolute Error (rounded)	$0.7D_{16}$	0.85_{16}	$0.7D_{16}$	0.80_{16}
Maximum Absolute Error	1.2_{16}	1.8_{16}	2.0	1.0
Sum of Signed Errors	$EDD8.6_{16}$	$546D.1_{16}$	$-ED2A.0_{16}$	$-2AF1.0_{16}$
Average Signed Error (rounded)	$0.0EE_{16}$	0.054_{16}	$-0.0ED_{16}$	$-0.02B_{16}$

Table 4.2.5.2.5-1 Results of Tests of the Two Division Algorithms

which select between the register and the memory operand in Figure 4.2-1 must be the tri-state SN74S257 for the fraction part of the operand.

The second step of the algorithm uses the zero to three bit shift logic of Figure 4.2.5.2.5-2 to shift the original denominator left by zero to three bit positions so that the high order bit is a one. Since the logic assumes that a three bit or smaller shift will suffice for this operation,

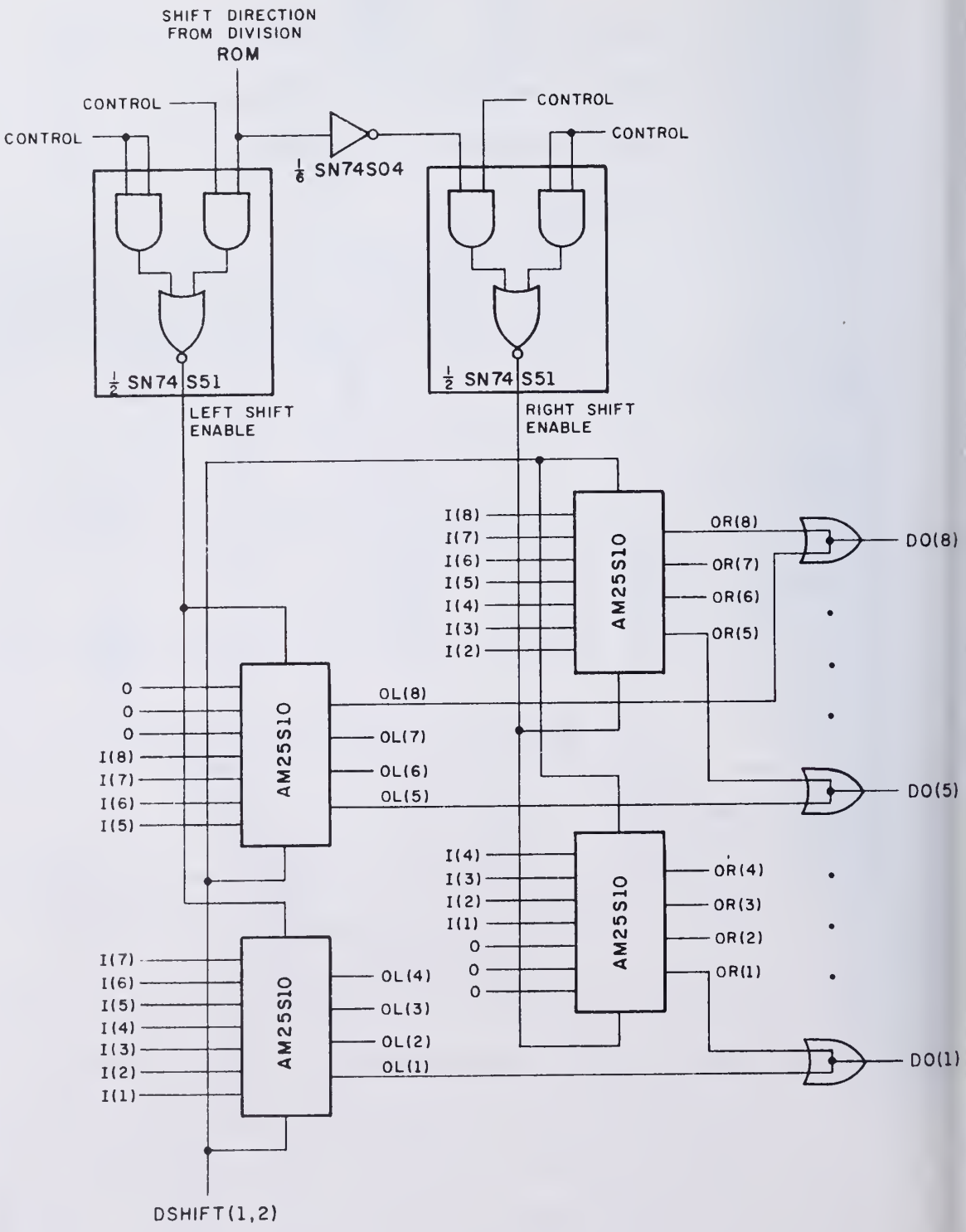


Figure 4.2.5.2.5-2 The Zero to Three Position Shift Logic

the original denominator must be a normalized value. The logic of Figure 4.2.5.2.5-2 relies on the AM25S10 tri-state four bit shifter. The figure illustrates both a left and a right shifting capability. Each AM25S10 accepts seven input bits, a two bit shift amount, and a tri-state enable signal. The two bit shift amount determines which of four sets of four contiguous input bits are output by the device. By using correct overlapping bit assignments to multiple AM25S10's, operands with more than four bits can be shifted. Figure 4.2.5.2.5-2 illustrates shift logic for eight bit input operands; shift logic for thirty-two bit values requires sixteen rather than four AM25S10's. Whether the ensamble of Figure 4.2.5.2.5-2 shifts left, as required by the second division step, or right, as required by a later step, is determined by the logic at the top of the figure. For this step, control signals from the control unit force a left shift, and cause the division ROM output to be ignored. The SN74148 of Figure 4.2.5.2.5-1 computes the shift amount for the zero to three bit shift logic by examining the three high order bits of the original denominator as stored in processor register zero. The shifted denominator is stored in processor register one.

The third step of the algorithm rounds the shifted denominator value by adding 2^{-12} to it. The constant for this rounding operation comes from the left operand selector described in section 4.2.5.1.5. Let us call the original denominator $\overline{D}$ and the shifted and rounded denominator D in the following discussion. The rounding step which produces D can result in an overflow; the carry out of the adder, ACOUT, is recorded in the C flip-flop of Figure 4.2.5.1.12.1-1 for the later use in the division process. If overflow occurs during denominator rounding, the special shift of one bit position in the fraction selector (section 4.2.5.1.7) is used to force the rounded result

800000_{16}, or exactly one-half.

The fourth step of the algorithm uses the division x_0 ROM's of Figure 4.2.5.2.5-1 and D to compute x_0, the first approximation to the desired reciprocal. This value varies from $FFF000_{16}$ for a D value of one-half, to 800000_{16} for the D value $FFF000_{16}$. The value actually stored by the ROM's must be a logic complement of the correct, rounded binary value, since the adder operates on active low data values and the fraction selector complements to account for this. The value from the ROM's is thus between one-half and $1-2^{-13}$ inclusive; since it represents the reciprocal of D, which is between one-half and $1-2^{-13}$ inclusive, it can be represented for the analysis below as $\frac{1}{2}x_0$. The resulting value is stored in register two.

In step five, we compute $\frac{1}{2} - \frac{1}{2} x_0D$ in one step by using the multiplier to supply the product term and using the left operand selector to supply the constant $\frac{1}{2}$. The result of this step is $\frac{1}{2}(1 - x_0D)$, which is a small value even for the first of the two iterations. Thus, step six adds the result of step five to itself to scale it up to the value $1 - x_0D$. Register three is used to store both of these results.

Step six computes $\frac{1}{2}x_0 (1 - x_0D)$ by using the multiplier with $\frac{1}{2}x_0$ from register two and $(1 - x_0D)$ from register three.

Step seven adds $\frac{1}{2}x_0$ from register two to the result of step six (from register three), and produces $\frac{1}{2}(x_0 + x_0(1 - x_0D))$ or $\frac{1}{2}x_1$.

Steps nine through twelve repeat steps five through eight, except that they use $\frac{1}{2}x_1$ instead of $\frac{1}{2}x_0$ throughout. The result is $\frac{1}{2}x_2$, or, in other words, $\frac{1}{2}$ of the reciprocal of D.

Step thirteen uses the multiplier to compute the exponent adder to

compute the result exponent and $\overline{Q}$ = (N/D). But we seek Q = N/$\overline{D}$. The form of $\overline{Q}$ is x.xxx..., where each "x" represents a bit. Since $\overline{D}$ was produced by shifting D left, that is by multiplying the original denominator, the correct Q is the result of a similar shit of $\overline{Q}$. This shift, conceptualized by a right shift of the binary point, results in a $\overline{Q}$ with one of the four following forms:

x.xxxxx...x (1)

xx.xxxx...x (2)

xxx.xxx...x (3)

xxxx.xx...x (4)

Since N, the original numerator, is also a floating point fraction, it has from zero to three leading zero bits. Hence, each of the four forms above can have from zero to four leading zero bits. Moreover, an overflow in step three of the division algorithm means that the original denominator, $\overline{D}$, was actually shifted left one less position than an examination of $\overline{D}$ would imply; this fact is recorded in the D flip-flop. Table 4.2.5.2.5-2 summarizes these conditions. The upper left part of each table entry indicates the amount and direction of a zero to three bit shift which is required to bring the binary point to one of the following positions.

.xxx....x, or (5)

xxxx.xx...x (6)

A left shift can occur when the number of high order zero bits in $\overline{Q}$ is greater than or equal to the number of bits to the left of the binary point in the form which $\overline{Q}$ takes among the forms (1) through (4) above. The lower right part of each table indicates the exponent alteration which is necessary

to convert $\overline{Q}$ to the proper quotient Q. The exponent correction is effected by the exponent correction adder. When form (5) results from the zero to three bit shift, no exponent correction is required. When form (6) results, the exponent must be reduced by one. When Table 4.2.5.2.5-2 indicates that a shift of four places is required, this is achieved by a shift of zero places in the zero to three position shift logic and a shift of one place in the normalization shift logic. In all other cases, the normalization shift logic shifts by zero places.

Leading Zeros in Q	Leading Zeros in D				
	1	0	1	2	3
0	0 / 0	R3 / -1	R2 / -1	R1 / -1	0 / +1
1	0 / 0	L1 / 0	R2 / -1	R1 / -1	0 / +1
2	0 / 0	L1 / 0	L2 / 0	R1 / -1	0 / -1
3	0 / 0	L1 / 0	L2 / 0	L3 / 0	0 / -1
4	L4 / -1	L1 / 0	L2 / 0	L3 / 0	L4 / 0

Table 4.2.5.2.5-2 0 to 4 Leading Zeros

Although the original denominator must be normalized, the numerator N need not be. A product with four (or more) leading zeros will result when the numerator is not normalized. The quotient is not normalized when the original numerator is not normalized. The quantity $\overline{Q}$ will also have four leading zero bits when the reciprocal is nearly ½ and N has its high order

to be truncated to an integer goes into the processor logic as the right operand. Its exponent is used as the address for a Signetics 8204 read only memory whose output, IFUNC(1,7) of Figure 4.2.5.1.6-2 controls the high order six SN74S381 arithmetic-logic units of the adder separately, and always forces ones in the seventh and eighth units. The logic assumes that the operand is normalized, and forces the correct number of fraction digits to ones (complimented to zeros by the fraction selection logic). The SN74S381's in the adder either add the operand fraction to a forced zero operand from the left operand selector, or they force ones as output. The function for addition is 011 and that for forcing ones is 111 (see Table 4.2.5.1.3-1). The high order bit is supplied by the SIG8205, and the two low order bits are supplied as CUAFUNC(2,3). The eighth ouput bit of the SIG8025 goes to the overflow flip-flop logic as INTRUNC, and is a logic one when the operand value cannot be represented in the six hexidecimal integer digits permitted one bit followed by several zero bits. The product of N with the reciprocal will then produce a non-normalized result, or one with four zeros.

A shift amount value of Rx in Table 4.2.5.2.4-1 means that a shift right of x bit positions is required. A shift amount of Lx means that a left shift of x bit positions is required.

4.2.5.2.6 Integers

The integers are represented and manipulated as floating point numbers in this design. The fractional part of an integer is zero. Logic is included to truncate the fraction part of an arbitrary floating point number. The largest integer that can be represented is $2^{24}-1$. A larger integer value can be represented by the thirty-two bit fraction of the processors, but memory can retain only twenty-four bit fractions. The value

in the design.

An exponent value of zero or less will produce an integer value of zero. An exponent value between one and six inclusive produces an integer with the corresponding number of potential non-zero hexidecimal digits. An exponent value of seven or more results in an integer truncation overflow condition.

4.2.5.2.7 Double Precision Addition and Subtraction

Measurements of the current model's execution on the IBM/360 reveals little required double precision operation. Therefore, we have designed a single precision processor which is augmented with the minimum additional hardware needed to permit double precision calculations. Twenty processor cycles are required to perform a normalized precision addition or subtraction. A double precision value consists of two single precision values, each with its own correct exponent and fraction. The high order part must always be normalized; the low order part contains the least significant six of the twelve fraction digits, whatever they may be, and therefore, has a normalized form only by coincidence. However, if the high order fraction is zero, the low order fraction must also be zero. The signs of both parts must agree.

Implementation of double precision addition and subtraction uses six processor registers. The normalized result is left with the high order part in register zero and the low order part in register one. Intermediate double precision operands in the processor have fourteen fraction digits, six in the high order part and eight in the low order part. The two low order digits of the high order parts are always zero at the completion of

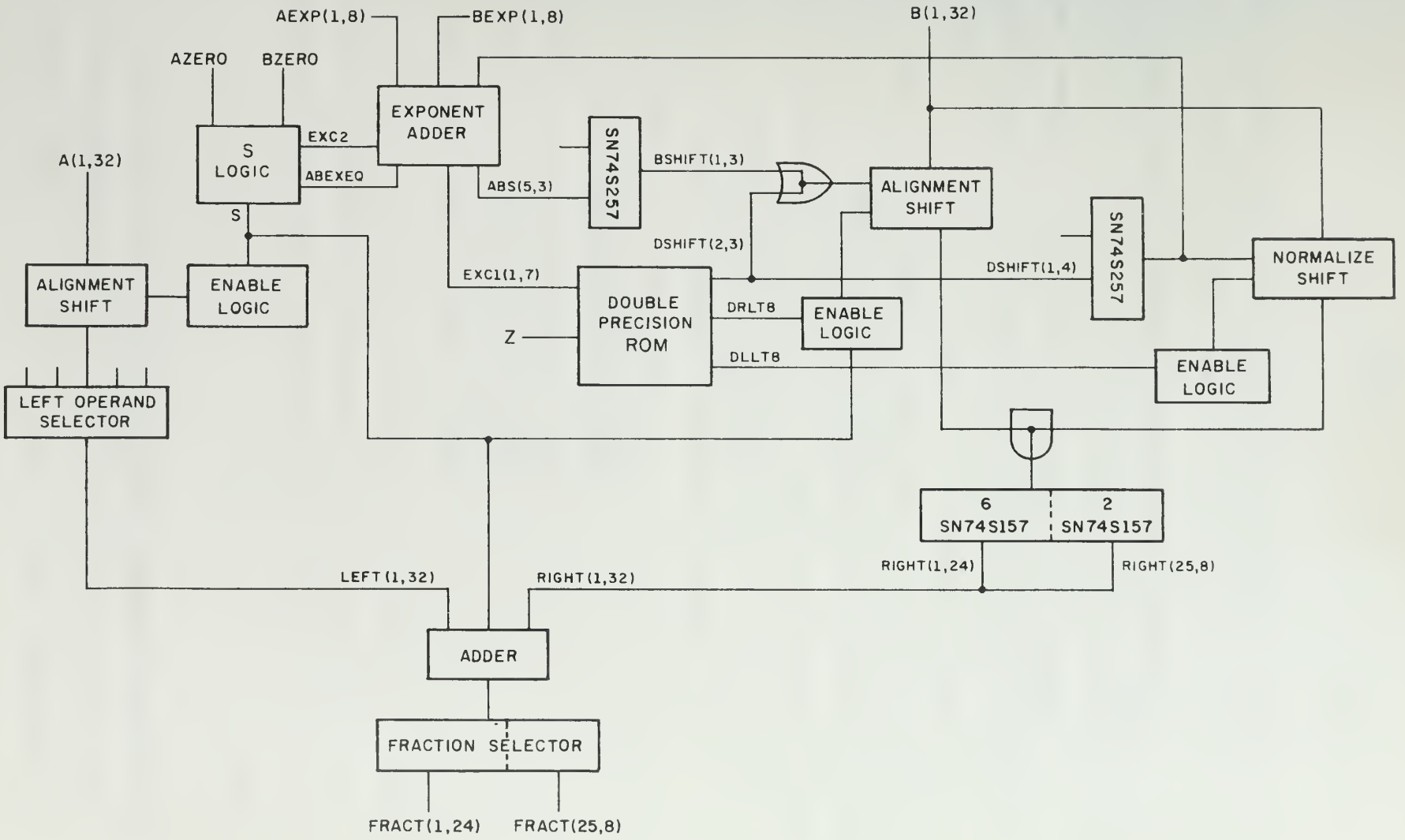


Figure 4.2.5.2.7-1 The Subset of the Processor for Double Precision Addition and Subtraction

an operation. The subset of the processor logic which performs double precision addition and subtraction is shown in Figure 4.2.5.2.7-1. The logic relies on the double precision read only memory of this figure for much of the specialized control which is required.

Several of the steps in the double precision addition and subtraction process are really fixed point addition of two fractions without regard to their signs or exponents. The exponent correction adder permits control from the control unit of which exponent is assigned to a result. The selection of the sign is also subject to complete control by the control unit. Hence, a fixed point addition of two fractions can be assigned to the exponent of either fraction and the sign of either fraction.

The complete double precision addition and subtraction process is illustrated by Figure 4.2.5.2.7-2, Figure 4.2.5.2.7-5, Figure 4.2.5.2.7.-9, and Figure 4.2.5.2.7-10. In these figures, the exponents and individual digits of all operands are shown. The digits of the two original operands, X and Y, are denoted by X1 through X14 and Y1 through Y14 respectively. The process determines which of the two operands is the larger and which is the smaller. The digits of the larger are denoted by L1 through L14; the digits of the smaller are denoted by S1 through S14. Finally, the digits of the sum or difference are denoted by T1 through T14. The operation portrayed by the figures is:

$$T = X \pm Y.$$

The original operands are shown in Figure 4.2.5.2.7-2(a). In the first step of the process, the high order part of X is written into registers zero and one; the operand registers permit writing a value to two different registers

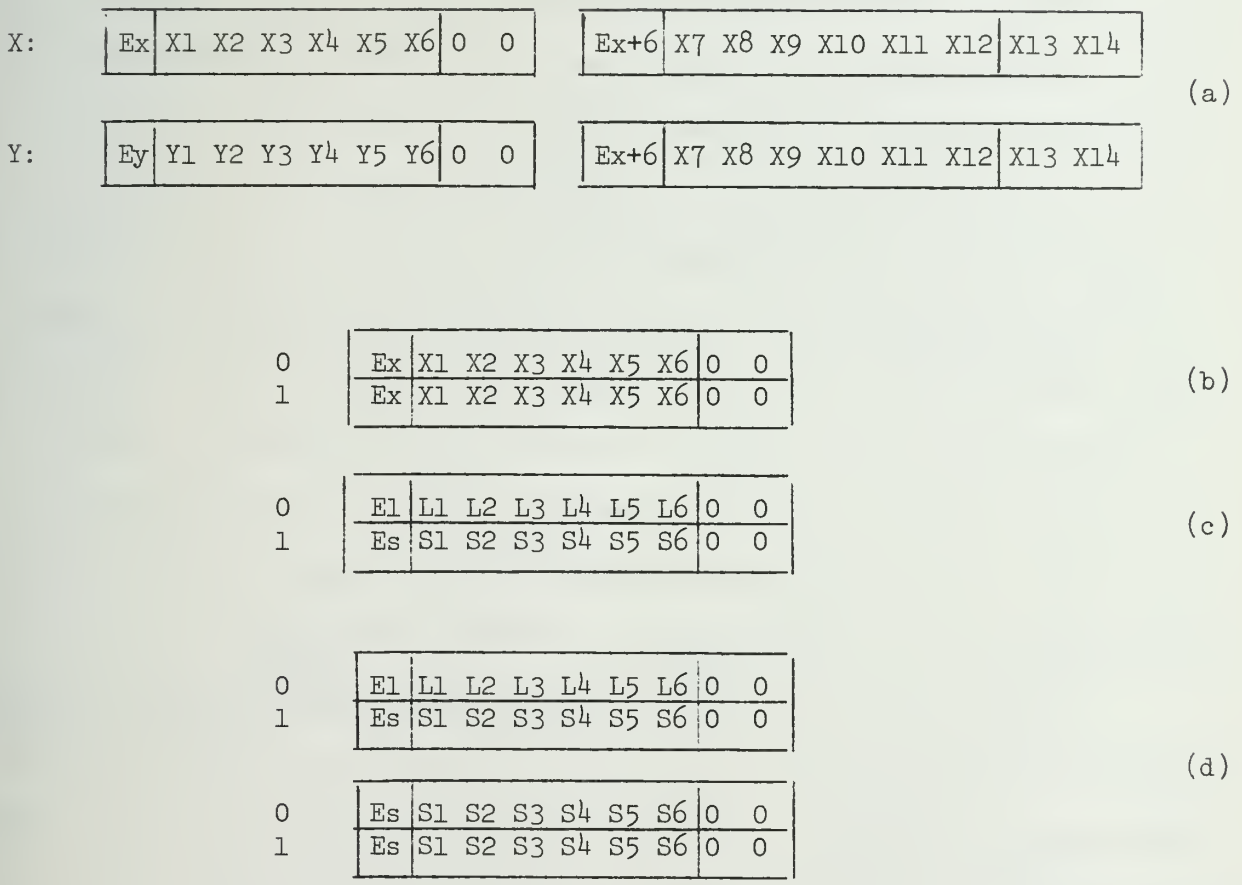


Figure 4.2.5.2.7-2 Preparatory Double Precision Addition and Subtraction Steps

in one operation (see section 4.2.5.1.10).

In the second step, the two high order parts of the operands are passed through the processor logic. The X operand is the left operand and the Y operand is the right operand since it may come from memory. The Y operand is always passed through the adder and fraction selector. The S logic, shown in Figure 4.2.5.2.7-3 determines whether the Y operand is larger or smaller than the X operand. A zero operand is always regarded as the smaller regardless of its exponent value. If the Y operand is larger, the S signal is zero; if the Y is smaller, the S signal is one. The result of the comparison, the S signal, is stored in the S flip-flop of Figure 4.2.5.2.7-3 for use in routing the low order halves of the operands in a later step. Table 4.2.5.2.7-1 explains the input signals for the S logic, and Table 4.2.5.2.7-2 gives the truth table for the S logic.

The logic which varies the operand register address bits to accomplish the local control needed by this and other steps in the double precision addition and subtraction process is shown in Figure 4.2.5.2.7-4. The signal is used, together with three zero address bits from the control unit, to select either register zero or register one during this step. The net result of step two is shown in part (c) of Figure 4.2.5.2.7-2; the larger operand is stored in register zero and the smaller in register one.

Step three duplicates the smaller operand in registers four and five. The Z flip-flop is set to indicate whether the smaller operand is zero. The rest of this step is shown in part (d) of Figure 4.2.5.2.7-2.

The next five steps align the fraction of the operands in preparation for the addition or subtraction steps. These five steps are shown in

Signal	Value	Significance
AZERO	0	The left operand fraction is zero.
BZERO	0	The right operand fraction is zero.
ABEXEQ	0	The operand exponents are equal.
EXC2	0	The left exponent is greater than or equal to the right exponent.
AGTR	1	The left fraction is greater than the right fraction.

Table 4.2.5.2.6-1 The Significance of the S Logic Input Signals

Signals				SN74S150 Input	Comments
AZERO or BZERO	ABEXEQ	EXC2	AGTR		
0	0	x	0	1	Y greater than or equal to X
0	0	x	1	x	X equals Y
0	1	0	x	0	X greater than or equal to Y
0	1	1	x	1	Y greater than X
1	x	x	x	BZERO	Exactly one operand is zero.

Table 4.2.5.2.7-2 The Truth Table for the SN74S150 of the S Logic

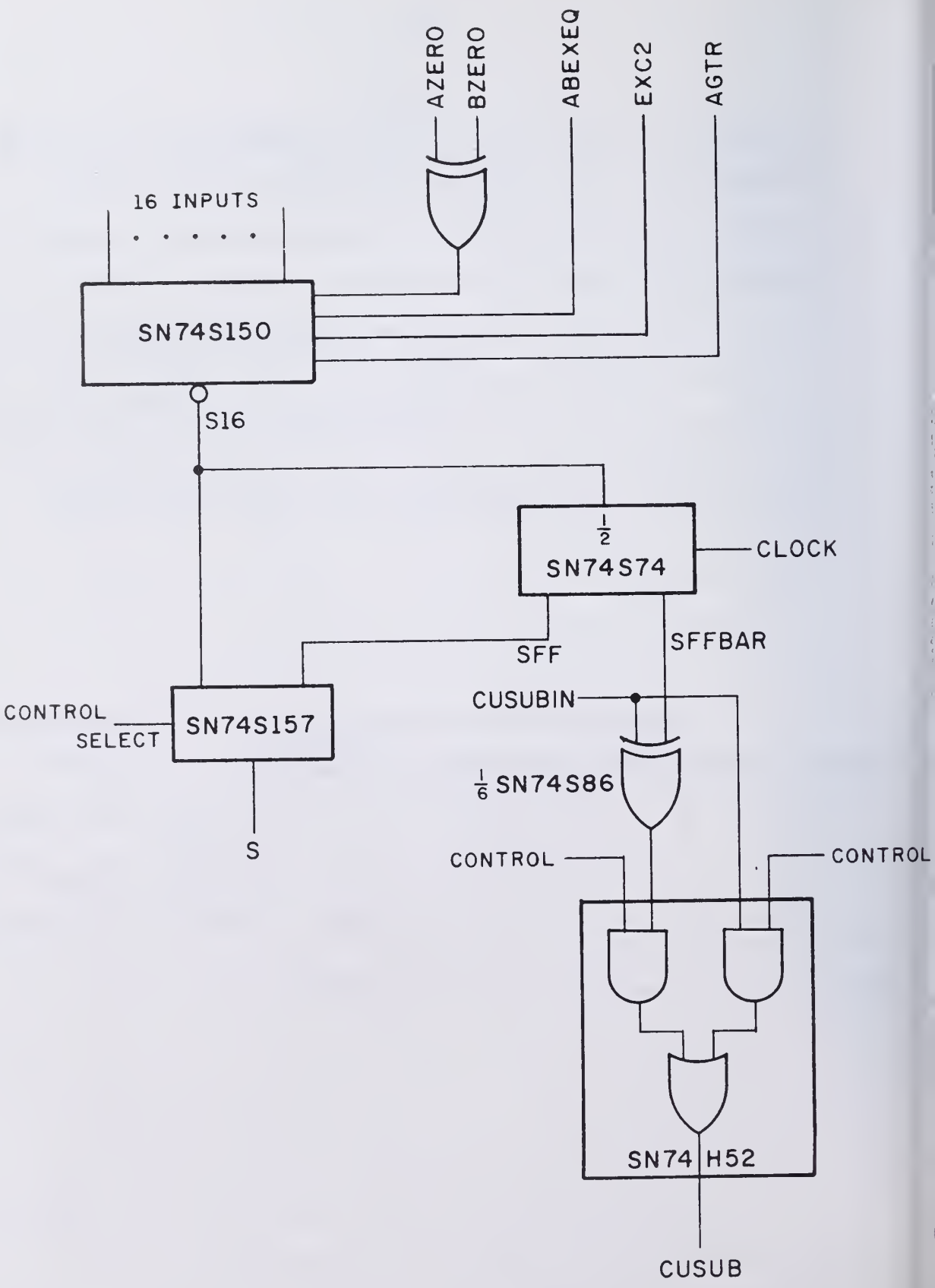


Figure 4.2.5.2.7-3 The Logic for the S Signal

Figure 4.2.5.2.7-5. The figure covers two cases. In the left column are successive register states for the case when the exponent difference is less than six; the right column covers the case where the exponent difference is greater than or equal to six. The exponent difference illustrated by the left column is three; that for the right is seven. The double precision ROM, which is crucial to many of the following steps, is shown in detail in Figure 4.2.5.2.7-6. It can be implemented with a Signetics 8204 read only memory. This ROM stores 256 eight bit words. The eight bit address is used as shown in the figure. One control signal from the control unit determines whether an alignment or normalization shift control result is desired; another control signal specifies whether a left shift or right shift is required. The other bits contribute to determining the shift amount. The operand which is to be shifted is always known beforehand, and is sent through the logic as the right operand. Table 4.2.5.2.7-3 summarizes the functions performed by the double precision control ROM during the operand alignment phase. The symbol "d" in the table represents the exponent difference.

Step four performs a left shift of the smaller operand by the amount given in Table 4.2.5.2.7-3. The control ROM uses signals DCADDR(1) and DCADDR(3) as shown in Figure 4.2.5.2.7-4 to store the result in register four when the exponent difference is less than six and in register one when that difference is greater than or equal to six. The results of step four are shown in Figure 4.2.5.2.7-5(a).

Step five performs a right shift of the smaller operand, taken from register five, by the amount given in Table 4.2.5.2.7-3. The control ROM again uses DCADDR(1) and DCADDR(3) as shown in Figure 4.2.5.2.7-4 to

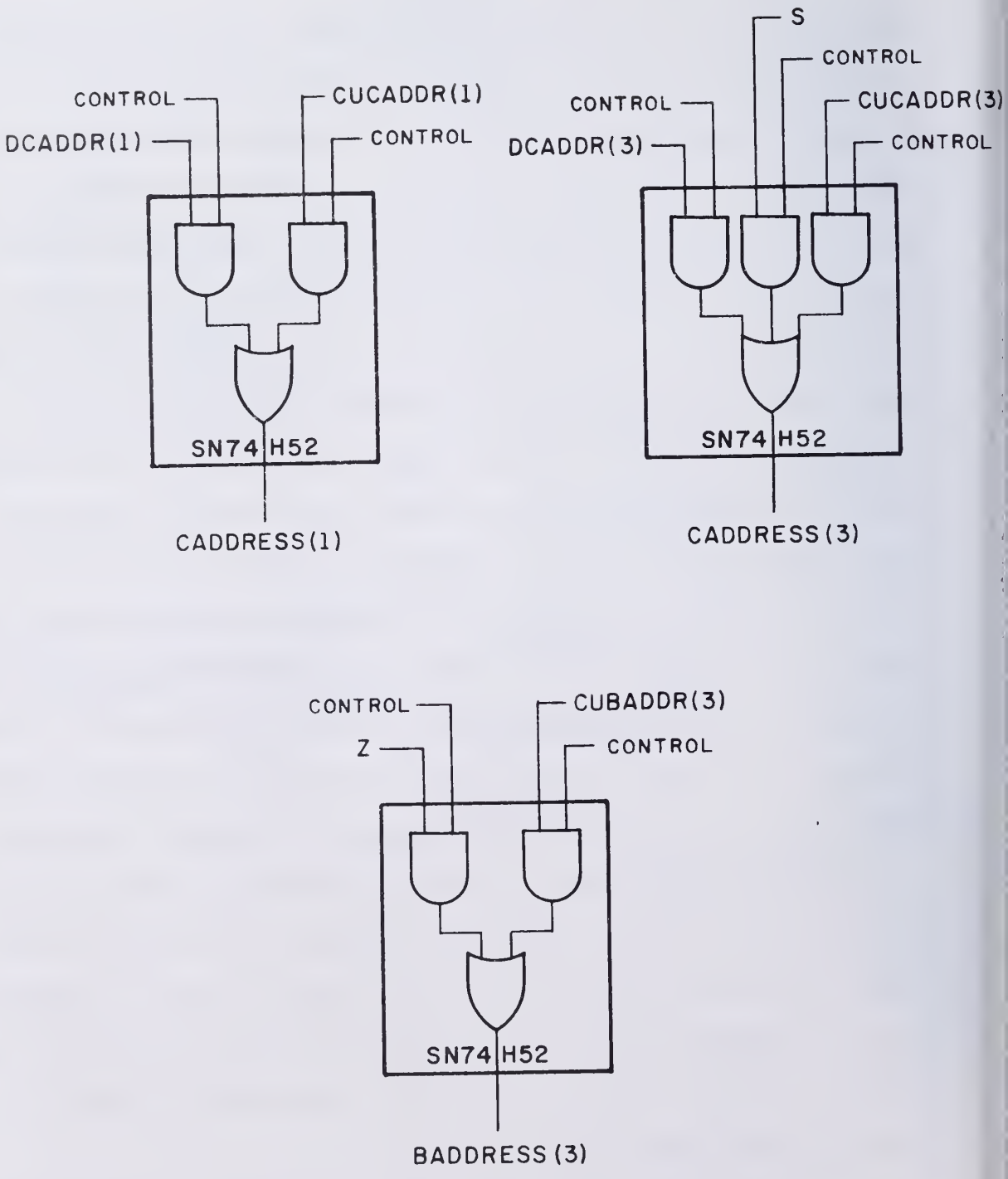


Figure 4.2.5.2.7-4 Logic for Local Control of Operand Register Addresses

Exponent Difference 6

0	E1	L1	L2	L3	L4	L5	L6	0	0
1	Es	S1	S2	S3	S4	S5	S6	0	0
4	E1	S4	S5	S6	0	0	0	0	0
5	Es	S1	S2	S3	S4	S5	S6	0	0

Exponent Difference 6

0	E1	L1	L2	L3	L4	L5	L6	0	0
1	Es	0	0	0	0	0	0	0	0
4	Es	S1	S2	S3	S4	S5	S6	0	0
5	Es	S1	S2	S3	S4	S5	S6	0	0

(a)

0	E1	L1	L2	L3	L4	L5	L6	0	0
1	Es	0	0	0	S1	S2	S3	0	0
4	E1	S4	S5	S6	0	0	0	0	0

0	E1	L1	L2	L3	L4	L5	L6	0	0
1	Es	0	0	0	0	0	0	0	0
4	E1	0	S1	S2	S3	S4	S5	S6	0

(b)

0	E1	L1	L2	L3	L4	L5	L6	0	0
1	Es	0	0	0	S1	S2	S3	0	0
2	Ex-6	X1	X2	X3	X4	X5	X6	0	0
3	Ex-6	X1	X2	X3	X4	X5	X6	0	0
4	E1	S4	S5	S6	0	0	0	0	0

0	E1	L1	L2	L3	L4	L5	L6	0	0
1	Es	0	0	0	0	0	0	0	0
2	Ex-6	X1	X2	X3	X4	X5	X6	0	0
3	Ex-6	X1	X2	X3	X4	X5	X6	0	0
4	E1	0	S1	S2	S3	S4	S5	S6	0

(c)

0	E1	L1	L2	L3	L4	L5	L6	0	0
1	Es	0	0	0	S1	S2	S3	0	0
2	E1-6	L7	L8	L9	L10	L11	L12	L13	L14
3	Es-6	S7	S8	S9	S10	S11	S12	S13	S14
4	E1	S4	S5	S6	0	0	0	0	0

0	E1	L1	L2	L3	L4	L5	L6	0	0
1	Es	0	0	0	0	0	0	0	0
2	E1-6	L7	L8	L9	L10	L11	L12	L13	L14
3	Es-6	S7	S8	S9	S10	S11	S12	S13	S14
4	E1	0	S1	S2	S3	S4	S5	S6	0

(d)

0	E1	L1	L2	L3	L4	L5	L6	0	0
1	Es	0	0	0	S1	S2	S3	0	0
2	E1-6	L7	L8	L9	L10	L11	L12	L13	L14
3	Es-6	0	0	0	S7	S8	S9	S10	S11
4	E1	S4	S5	S6	0	0	0	0	0

0	E1	L1	L2	L3	L4	L5	L6	0	0
1	Es	0	0	0	0	0	0	0	0
2	E1-6	L7	L8	L9	L10	L11	L12	L13	L14
3	Es-6	0	0	0	0	0	0	0	0
4	E1	0	S1	S2	S3	S4	S5	S6	0

(e)

0	E1	L1	L2	L3	L4	L5	L6	0	0
1	Es	0	0	0	S1	S2	S3	0	0
2	E1-6	L7	L8	L9	L10	L11	L12	L13	L14
3	Es-6	S4	S5	S6	S7	S8	S9	S10	S11

0	E1	L1	L2	L3	L4	L5	L6	0	0
1	Es	0	0	0	0	0	0	0	0
2	E1-6	L7	L8	L9	L10	L11	L12	L13	L14
3	E1-6	0	S1	S2	S3	S4	S5	S6	S7

(f)

Figure 4.2.5.2.7-5 Alignment Steps in Double Precision Addition and Subtraction

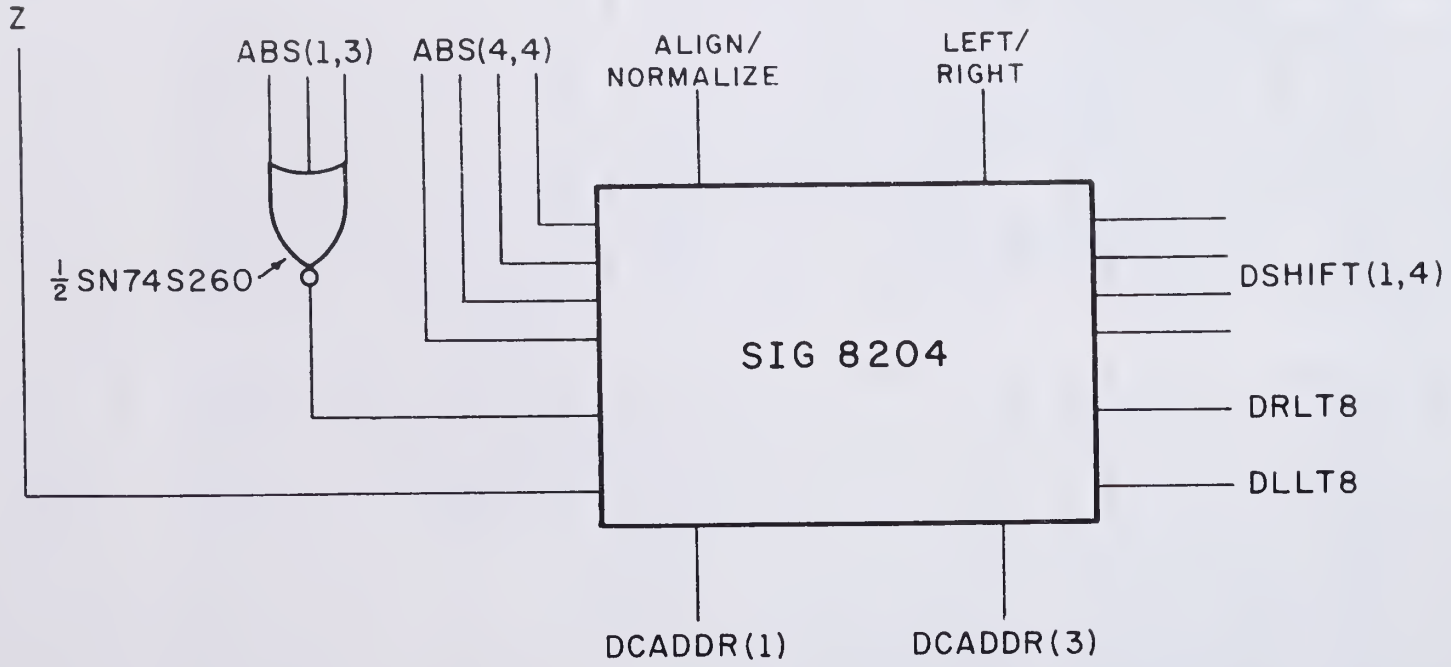


Figure 4.2.5.2.7-6 The Details of the Double Precision Control Read-Only Memory

Shift Direction	Shift Amount	
	d < 6	d > 6
Left	6-d	d
Right	d	d-6

Table 4.2.5.2.6-3 Signetics 8205 Control ROM Shift Amount During the Operand Alignment Phase

store the result in register one when the exponent difference is less than six and in register four when that difference is greater than or equal to six. This shifted result must have its two low order digits both zero. This is necessary for step eleven to compute a correct high order part. The two low order digits, FRACT(25,8), are forced to zero by causing the two SIG8263 selectors of the fraction selection logic (Figure 4.2.5.1.7-1) which produce these bits to emit zeros during this step. This is accomplished by setting both bits of their selection signal to zero and their complement signal also to zero (see Table 4.2.4.2-1). The results of this step are shown in Figure 4.2.5.2.7-5(b).

Step six loads registers two and three with the low order part of X. Step seven is similar to step two. The contents of the S flip-flop, as shown in Figure 4.2.5.2.7-4, are used to direct the low part of Y to register two when Y was the larger operand in step two, and to register three when Y was the smaller operand in step two. The state of the registers after step seven is shown in Figure 4.2.5.2.7-5(d).

In step eight, a normal floating point alignment operation results in a shift right of the smaller lower order part, taken from and returned to register three, by the amount of the exponent difference. The result of this

step is shown in Figure 4.2.5.2.7-5(e). Of course, when the exponent difference exceeds seven, the contents of register three after this step is zero. Step eight combines the contents of register three and four by addition with forced alignment shifts of zero places to produce the correct low operand for the addition or subtraction step. The result of this step is shown in Figure 4.2.5.2.7-5(f). At this point, the two high order operands are in registers zero and one, and the two low order operands are in registers two and three.

The actual addition or subtraction process is complicated by the fact that sign-magnitude representation is used for floating point values in this design. The actual operation which must be performed depends not only on the instruction being executed but also on the signs and relative magnitudes of the operands being processed. If one of the operands is zero, the result is the other operand, possibly with its sign reversed. If two operands with equal exponents are to be added, the actual operation performed depends on their signs. When the signs are the same, the magnitudes are simply added, and the sign of the result is that shared by the two operands. However, when the signs differ, the smaller magnitude must be subtracted from the larger, and the sign of the result is that of the larger operand. During double precision addition and subtraction, the function which the adder must perform is usually determined by the high order parts of the operands. But, for example, when the signs are unlike during an addition, the relative magnitudes of the low order parts of the operands will determine the operation when the high order parts are equal. In step nine, the D flip-flop of Figure 4.2.5.2.7-7 is set according to the truth table in

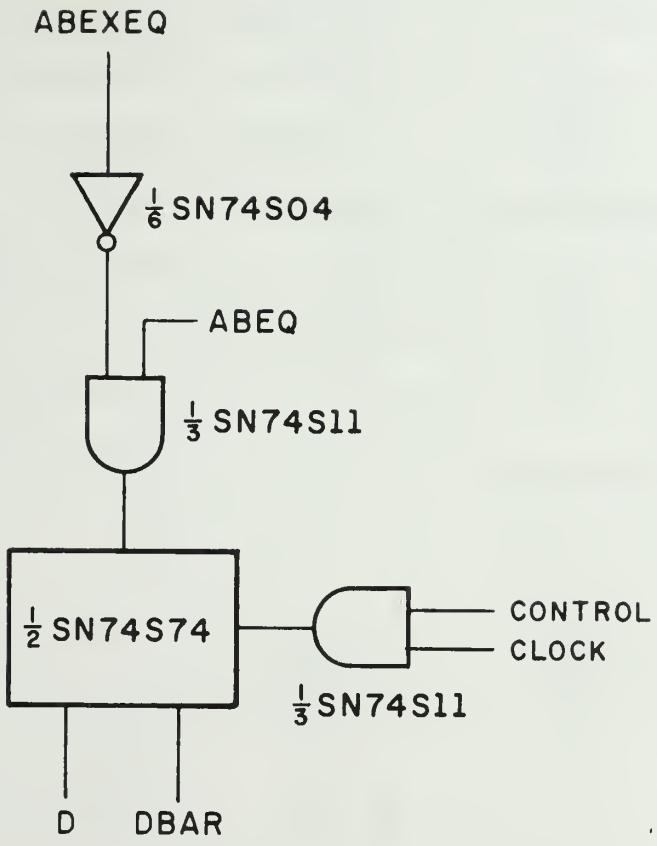


Figure 4.2.5.2.7-7 The D Flip-flop Logic

Table 4.2.5.2.7-3. For this step, the two high order parts are passed through the logic, and the adder function which they require is determined by the Signetics 8205 read only memory of Figure 4.2.5.2.7-8. The adder function is stored in the NAT8551 tri-state register, but the result of the operation is not stored in the operand registers. The D flip-flop is set to a logic zero when the high order parts of the operands determines the function; the D flip-flop is set to one only when both the high order exponents and fractions are equal, so that the low order parts must determine the function. The operand registers at the end of step nine are the same as they were previous to this step. However, the D flip-flop and the NAT8551 are set by the step for use in step ten.

Input Signals		D Flip-flop Setting	Comments
ABEXEQ	ABEQ		
0	0	0	Operands not equal
0	1	1	The operands are equal
1	0	0	Operands not equal
1	1	0	Operands not equal

Table 4.2.5.2.7-3 Truth Table for the D Flip-flop

In step ten, the low order parts of the operands from registers three and four are added or subtracted using the contents of the NAT8551 when the D flip-flop setting from step nine is zero and using the output of the SIG8205 control ROM when the D flip-flop setting from step nine is one. When the relation of the low order operands should determine the adder function (that is, when the D flip-flop is one), the SIG8205 function output is clocked into the NAT8551 during step ten processing. The high order carry out of the adder during step ten is saved in the carry flip-flop, C. This

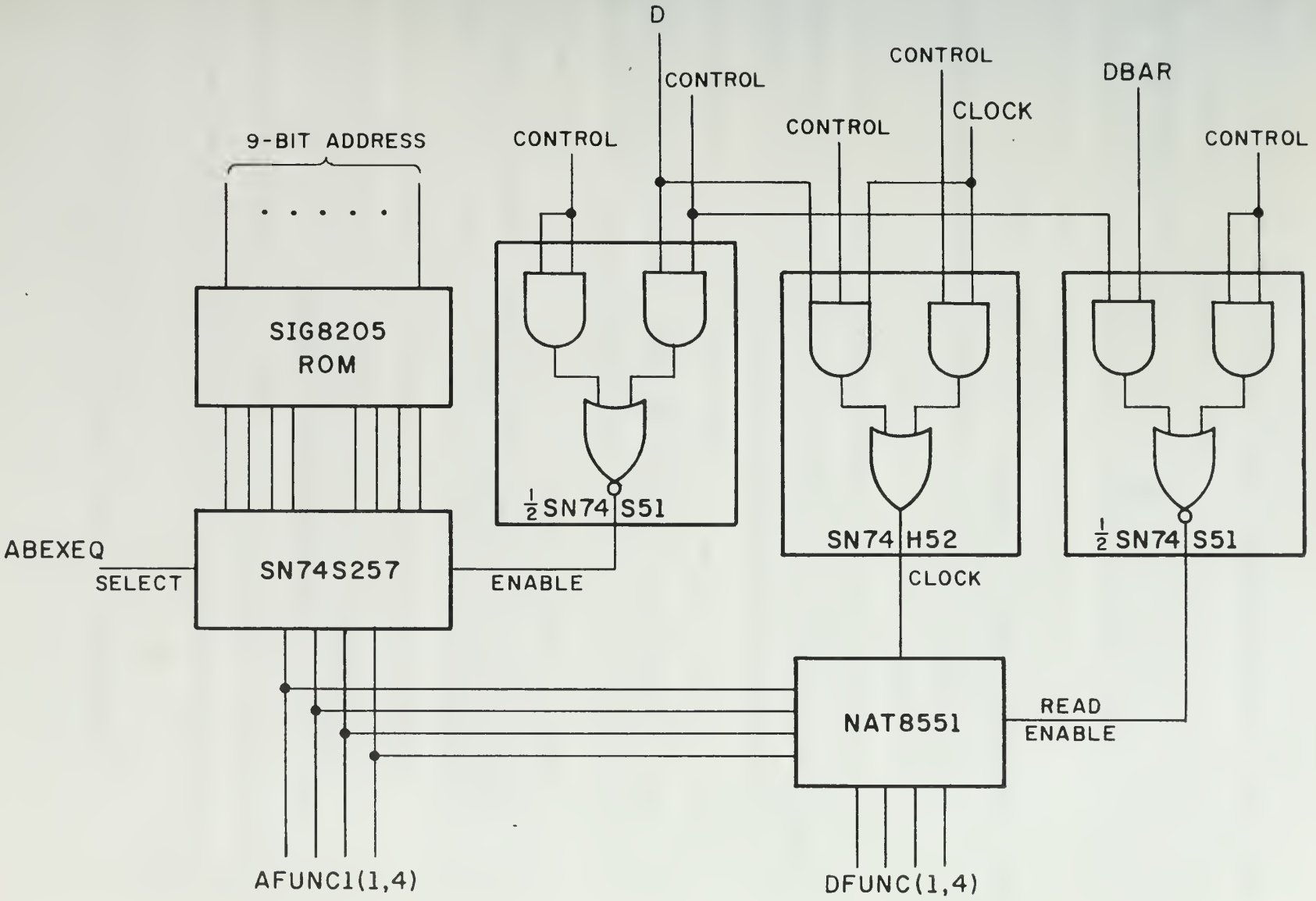


Figure 4.2.5.2.7-8 The Logic for the AFUNC(1,4) and DFUNC(1,4) Control Signals

carry must be propagated to the high order operation, which occurs in step eleven. The results of step ten are shown in Figure 4.2.5.2.7-9(a). The low order result is stored in register three. The normal operation of the fraction selection logic is aborted for this step; no right shift is performed if a fraction overflow occurs. Instead, the carry flip-flop contents propagate the overflow condition to the high order operation.

Step eleven uses the function stored in the SN74S670 and the carry stored in the carry flip-flop, C, to compute the high order part of the results. So that the carry can propagate across the eight low order bits which are ones in both operands (active low zeros), the two low order SN74S157 quadruple two-to-one selectors which select the output of the wire AND shown in Figure 4.2.5.2.7-1 are made to supply zeros (active low ones) by setting their strobe inputs to one for this step only. The result of this operation is shown in Figure 4.2.5.2.7-9(b). The left part of the figure shows the case for which no fraction overflow occurs; the right part shows the result when fraction overflow does occur. The high order part of the result is left in register zero and the low order part in register two by this step.

The one bits introduced to propagate the carry must be removed by the fraction selection logic. The two SIG8243 three-to-one selectors which forced the two low order digits to zero in step five are used. They operate under processor control to force two digits to zero when no fraction overflow occurs, and they force one digit to zero when a fraction overflow does occur.

Step twelve shifts the high order part of the result left six

Set Function from (0) and (1)

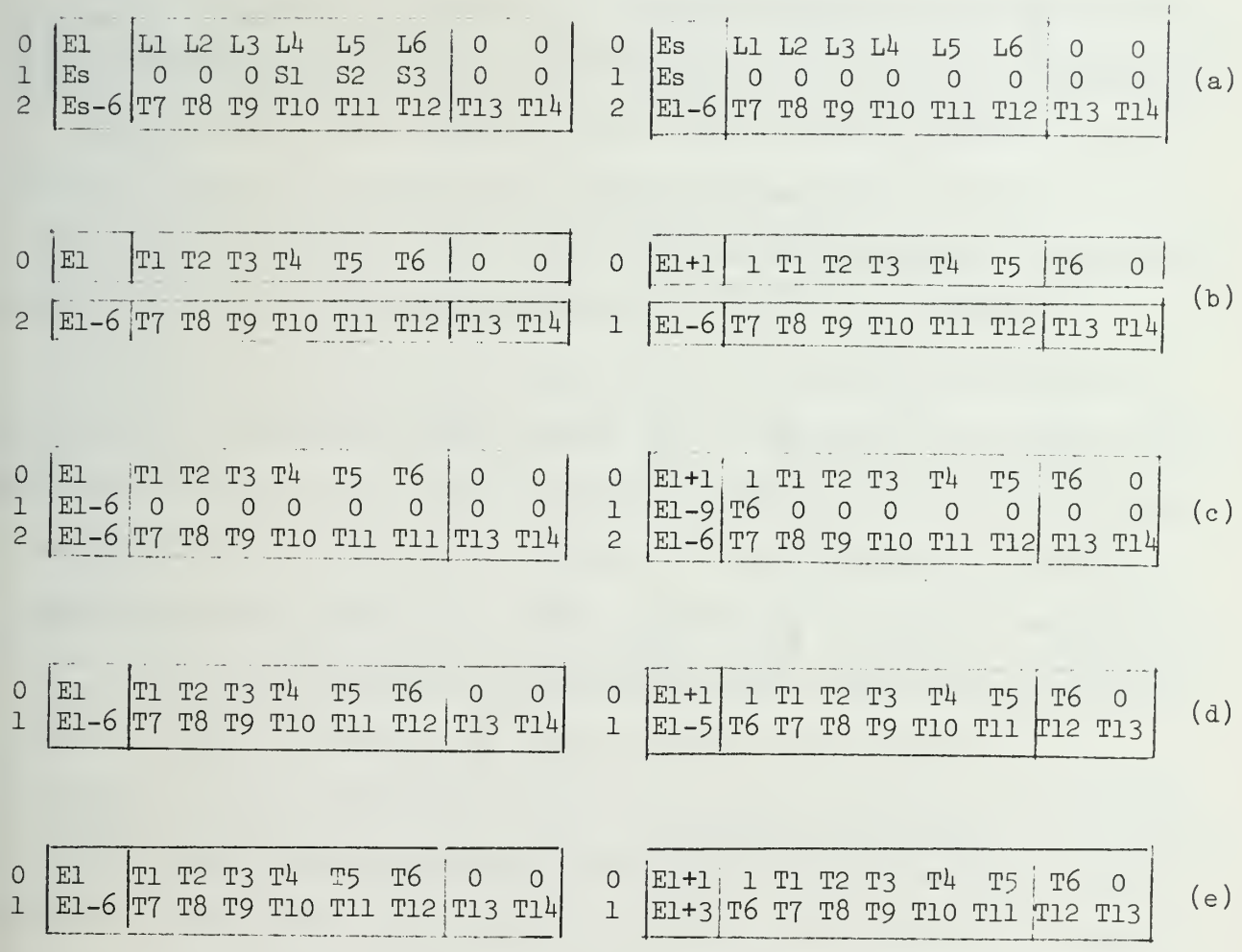


Figure 4.2.5.2.7-9 The Addition Steps in Double Precision Addition and Subtraction

places and stores the shifted value in register one. The control ROM will output the value six required to control the shift if the register zero operand is sent through the logic as both the left and right operands. One of the operands is forced to zero by its alignment shift logic, and the other shifted six left passes through to register one. The results of step twelve are shown in Figure 4.2.5.2.7-9(c).

Step thirteen is an ordinary unnormalized addition of the contents of registers one and two. The result is stored in register one, and it is the correct low order part for the double precision operation. Steps twelve and thirteen served to transfer a possible T7 digit from the high to the low order part of the double precision fraction. The results of step thirteen are shown in Figure 4.2.5.2.7-9(d). The zero flip-flop is set to indicate whether the high order fraction result of this step is zero.

In step fourteen, the high order part is passed through the logic and two low order zero digits are forced by the fraction selection logic to clear a possible T7 digit from the high order part of the result. The results of step fourteen, a correct but unnormalized double precision floating point addition or subtraction result, are shown in Figure 4.2.5.2.7-9(e).

The result must be normalized. If the high order fraction is zero but the low order one is not, the logic which controls the adder function selection for double precision operations will not work correctly. The five steps which are required to normalize the result are shown in Figure 4.2.5.2.7-10. The left column of the figure details with the case in which the high order fraction is zero; the right column treats the case in which the high order fraction is not zero.

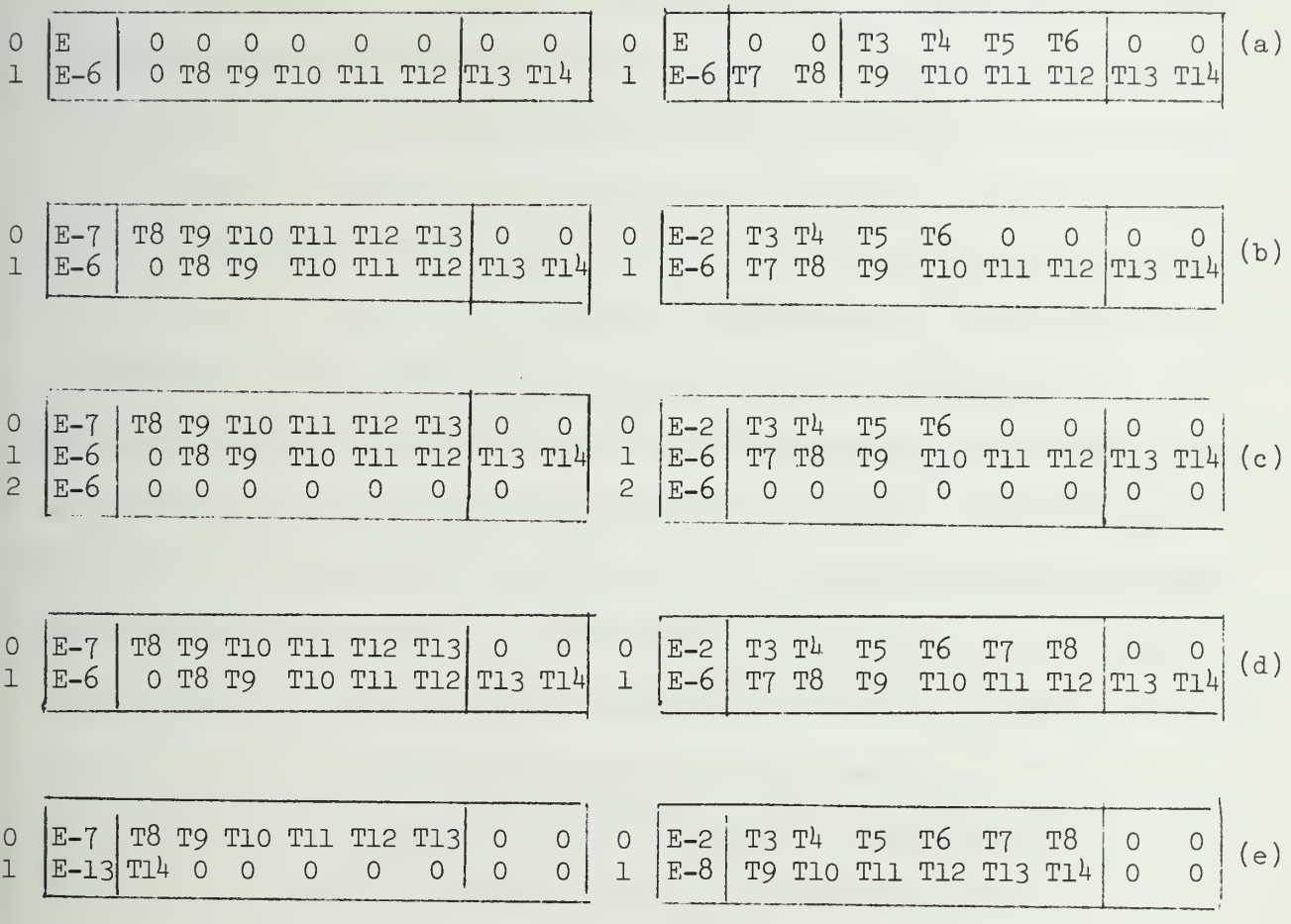


Figure 4.2.5.2.7-10 The Normalization Steps in Double Precision Addition and Subtraction

The first step of the normalization process uses the Z flip-flop state and the logic of Figure 4.2.5.2.7-4 to select the register zero operand when the high order fraction is non-zero and the register one operand when the high order fraction is zero. The initial operands for normalization, assumed results of the addition or subtraction, are shown in Figure 4.2.5.2.7-10(a). The results of this step, an ordinary normalization step, are shown in Figure 4.2.5.2.7-10(b).

The second normalization step uses the values from register zero and register one. The exponent difference is used by the control ROM in the normalization mode to compute a right shift amount. Table 4.2.5.2.7-4 summarizes the function of the SIG8205 control ROM for the normalization phase of double precision operations. The symbol "d" in the table represents the exponent difference between the register zero and register one operands.

Shift Direction	High Order Fraction	
	Zero	Not Zero
Left	6+d	6-d
Right	6-d	d

Table 4.2.5.2.7-4 Signetics 8205 Control ROM Shift Amount During the Normalization Phase

The second normalization step shifts the low order fraction right by the amount specified by the SIG8205 control ROM. The two low order digits of the shifted result are forced to zero by the FRACT(25,8) selectors of the fraction selection logic. The results of this step are shown in Figure 4.2.5.2.7-10(c). The shifted result is stored in register three.

The third normalization step adds the contents of registers three and zero and stores the result in register zero. The result of this step is shown in Figure 4.2.5.2.7-10(d). The net effect of steps two and three is the transfer of fraction digits from the low to the high order part of the double precision fraction.

The fourth normalization step shifts the low order fraction left by the amount specified by the SIG8205 control ROM. The shift amount computed by the ROM is subtracted from the exponent of the low order operand so that the final exponent result is correct. The amount subtracted from the exponent is thirteen for the case when only one non-zero fraction digit is produced as digit T14 of the addition or subtraction result. Thus, although the normalization shifter is disabled so that it outputs a zero when the shift amount exceeds seven, an amount of up to thirteen must be able to go from the SIG8205 to the exponent adder. The result of this step is a correct normalized double precision addition or subtraction result. The zero flip-flop is set on this step to indicate whether the low order fraction is zero.

The last normalization step tests the high order fraction for zero, and ANDs the result of the test into the zero flip-flop (see Figure 4.2.5.2.12-1). Hence, the flip-flop will be zero after a floating point double precision addition or subtraction only if both fraction parts are zero.

4.2.5.2.8 Double Precision Multiplication

Figure 4.2.5.2.8-1 shows the partial products which contribute to a double precision multiplication result. In this design, two double precision operands are multiplied to yield a double precision result. The low order part of that result is not produced. The figure displays the product

A = (A1, A0)

B = (B1, B0)

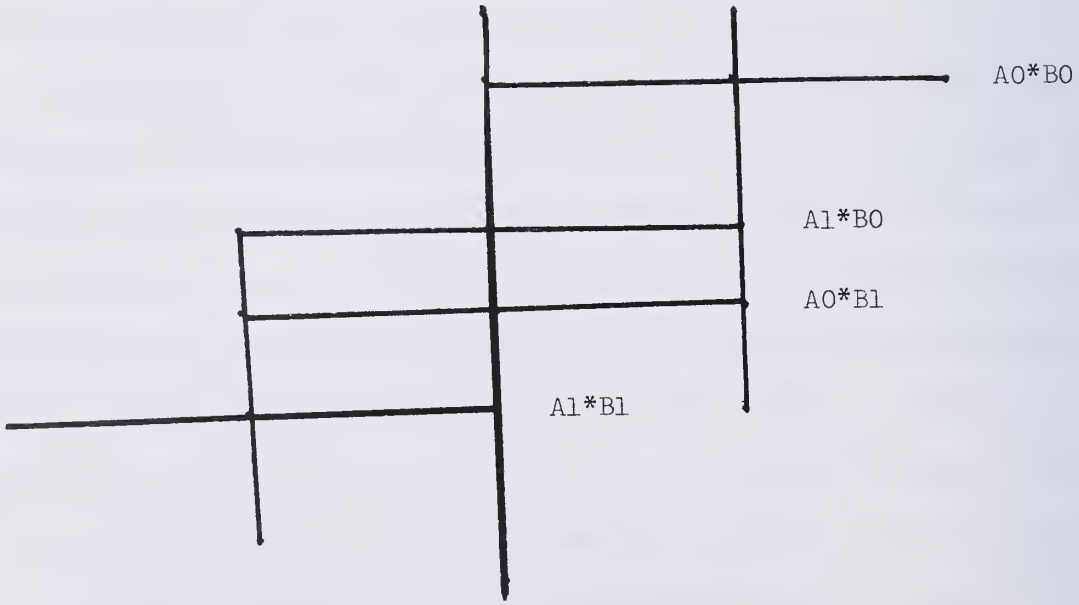


Figure 4.2.5.2.8-1 The Partial Products in Double Precision Multiplication

of A=(A1, A0) by B=(B1, B0); A1 and A0 are the most and least significant part of the double precision number A, respectively. The products A1*B1 and A0*B1 are computed first; four registers store the product results. They are combined into two values by addition of the low order parts and propagation of the carry to the addition of the high order parts. The carry from the high order addition is saved for later addition to the high order part of the product A1*B1. The product A1*B1 is computed and the saved carry is added to the high order part. The high order part of the sum of the middle partial products is then added to the product A1*B1. The carry is propagated across. Finally, the product A0*B0 is computed. It is added to the low order part of the sum of the middle partial products, and the cary - if any - is propagated by two additions.

Twentysteps are needed to complete the process. They are:

1. Multiply: Compute A1*B0 and store the high order part in register one. The low order exponent of the final product is computed in this step.

2. Store: Store the low order part of the product in register two.

3. Multiply: Compute A0*B1 and store the high order part in register zero.

4. Store: Store the low order part in register three. The addition with the low order part of A1*B0 which follows cannot be done on the fly because the operands for the multiplication must continue to be supplied by the operand registers.

5. Add: Add the low order parts of the above products and save the carry. Store the result in register two.

6. Add with carry: Add the high order parts of the above products together with the saved carry from the low order parts. Store the result in register one. Save the carry from this addition.

7. Multiply: Compute A1*B1 and store the high order part in register zero. The high order exponent of the final product is computed in this step.

8. Store: Store the low order part of the A1*B1 product in register three.

9. Add carry: Add the carry saved from the previous addition to the high order part to the A1*B1 product.

10. Add: Add the contents of register one to the low order part of the A1*B1 product from register three. Save the carry out of this addition.

11. Add carry: Add the saved carry from step (10) to the high order part of the product in register zero.

12. Multiply: Compute A0*B0 and store the high order part in register three.

13. Add: Add the high order part of A0*B0 to the low order part of the sum of the middle partial products. Save the carry from this addition.

14. Add carry: Add the saved carry to the low order part of the final result in register one. Save the carry from this addition.

15. Add carry: Add the saved carry to the high order part of the final result in register zero.

The result of the above fifteen steps is the unnormalized double precision product of the initial double precision operands. Five normalization steps exactly like those which were used to normalize the double precision addition or subtraction result complete the operation.

4.2.5.2.9 Double Precision Division

Double precision division can be implemented by a process which parallels that for single precision division described in section 4.2.5.2.5. The initial approximation to the reciprocal is computed by a single precision division. An iterative procedure based on the equation

$$X_{n+1} = X_n + X_n(1 - Dx_n)$$

is carried out. We did not determine the number of iterations which would be required, but it would be two - perhaps three. The term "D" above is the

original double precision denominator, and the successive x terms are approximations to the reciprocal. Double precision multiplications are used to perform the iterations, and fixed point double length additions combine the terms as they did in the single precision division case. A final floating point multiplication by the original numerator computes the computation of the required quotient.

4.2.5.2.10 Multiplication and Division by a Power of Two

In many of the multiplications and divisions which the model executes, one of the operands is a power of two. The logic described in this section performs a multiplication or division by a power of two in one processor cycle. The power of two in the operation is specified by a six bit value, CSHIFT(1,6) of Figure 4.2.5.2.10-1. In a machine with an exponent radix of two, all of these bits would be added to the exponent for multiplication by a power of two and subtracted from it for division by a power of two. In this design, however, the exponent radix is sixteen. Thus, the two low order bits of the power of two determine a shift of the fraction, and the four high order bits of the power of two are added to or subtracted from the exponent. The control aspects of the logic are shown in Figure 4.2.5.2.10-1. The heart of the process is the Signetics 8204 read only memory. It accepts CSHIFT (5,2), the two low order bits of the power of two, the three high order bits of the fraction, and a signal which specifies whether multiplication or division by a power of two is desired. The output from the read only memory controls the zero-to-three position shifter with a two bit amount and a one bit shift direction signal, and it controls the exponent correction adder with

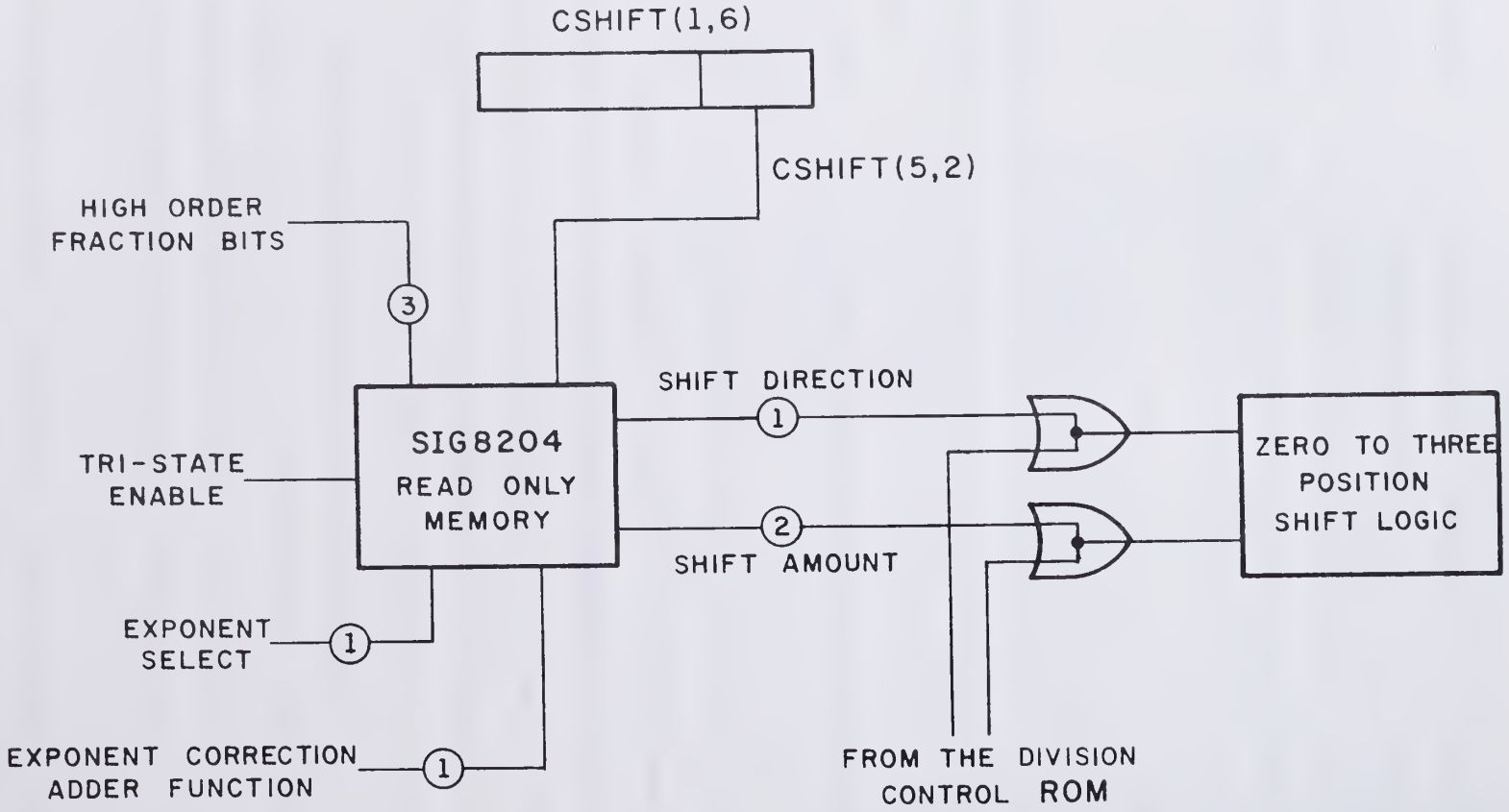


Figure 4.2.5.2.10-1 The Control Logic for Multiplication and Division by a Power of Two

High Order Fraction Zero Bits	Shift			
	0	1	2	3
0	0 / 0	R3 / +1	R2 / +1	R1 / +1
1	0 / 0	L1 / 0	R3 / +1	R2 / +1
2	0 / 0	L1 / 0	L2 / 0	R1 / +1
3	0 / 0	L1 / 0	L2 / 0	L3 / 0

Table 4.2.5.2.10-1 Control Details for Multiplication by a Power of Two

High Order Fraction Zero Bits	Shift			
	0	1	2	3
0	0 / 0	R1 / 0	R2 / 0	R3 / 0
1	0 / 0	R1 / 0	R2 / 0	L1 / -1
2	0 / 0	R1 / 0	L2 / -1	L1 / -1
3	0 / 0	L3 / -1	L2 / -1	L1 / -1

Table 4.2.5.2.10-2 Control Details for Dibision by a Power of two

a one bit function signal and a one bit selection signal.

Table 4.2.5.2.10-1 gives the details of the control signals for multiplication by a power of two, and Table 4.2.5.2.10-2 gives the details for division by a power of two. The upper left part of each table entry gives the shift amount and direction; the lower left part gives the exponent adjustment.

4.2.6 The Instruction Set for the Processors

The instruction set for the processors is given in Table 4.2.6-1. Separate classes of instructions with three, two, one and zero addresses are included. An address usually designates a processor register or memory location, but no more than one memory address is permitted in an instruction. In some special cases noted in Table 4.2.6-1, an address designates and operand other than a processor register or a memory location.

The first four operations in the table - addition and subtraction, multiplication and division - were covered in detail in sections 4.2.5.2.3, 4.2.5.2.4, and 4.2.5.2.5 respectively. The AND, OR and XOR (exclusive or) logical operation are implemented by using the corresponding logical operation of the SN74S381 arithmetic-logic unit of the adder (see Table 4.2.5.1.3-1) Logical NOT is implemented by using an exclusive OR with a forced one operand from a disabled alignment shift network. The MOVE operations are simple transmissions of operands from one place to another. Normalization is discussed in section 4.2.5.2.1; the integerize operation is discussed in section 4.2.5.2.6. Comparison operation are simply subtractions which set the condition flip-flops, but not the operand registers. The mode setting instructions use the mode logic of section 4.2.5.1.9. Combinations of sequences of

Address	Operation	Options	Comments
3	Add	Round, Normalize, Sign	Single & double precision
3	Subtract	Round, Normalize, Sign	Single & double precision
3	Multiply	Round, Normalize, Sign	Single & double precision
3	Divide	Sign	Single & double precision
3	Shift	Normalize	Multiply by a power of two
3	Logical AND	Exponent source	
3	Logical OR	Exponent source	
3	Logical XOR	Exponent source	
2	Move	Register ← Memory	Single & double precision, Sign
		Memory ← Register	Single & double precision
		Routing pattern ← Register	
		Register ← Register	Single & double precision, Sign
2	Compare		Set the condition register Single & double precision
2	Normalize	Sign	Single & double precision
2	Integerize	Normalize, sign	
2	Logical NOT		
2	Round	Sign	
2	Set	Status(i) ← Mode @ Status(j)	The "@" sign represents any one of the sixteen possible Boolean operations on two variables. The two addresses designate the bit numbers "i" and "j" which select amoung the eight status register bits.
		Mode, Status(i) ← Mode @ Status(j)	
1	Move	Register ← Routing data	
		Routing data ← Register	
		Routing data ← Memory	
		Register ← Status	
		Status ← Register	
		Register ← 0	Single & double precision
1	Set Mode	Mode ← Mode @ Status(i)	
1	Route		Addresses pattern
0	Set Mode	Mode ← 1	
0	CU ← Modes	Mode ← 0	
0	Table-look-up		

Table 4.2.6-1 The Instruction Set for the Processors in the Array

condition states can be stored in the status register of the mode logic, and provide a simple way to implement complex testing procedures. Several instructions include the option to require a particular sign for the result. With a sign-magnitude representation, absolute value and complementation operations reduce to simple sign manipulations. The sign logic of section 4.2.5.2.12.3 permits the normal result sign, its complement, a positive sign, a negative sign, or the exclusive OR of the operand signs to be assigned as the sign of the result.

The route instruction supplies a routing pattern address to the routing network. The network stores sixteen pre-loaded routing patterns. A routing instruction calls for the use of one of these pre-loaded patterns. A built-in operand broadcast is also included. It causes an operand in one of the 256 routing dis-assembly registers to be sent to every routing re-assembly register. The control unit can load values into the original dis-assembly register and retrieve value from the corresponding re-assembly register. See section 4.3 for the details of the routing network.

The shift operation permits multiplication or division by a power of two as discussed in section 4.2.5.2.10. The power of two is a control unit operand of six bits in length.

The exponent selection feature of the logical operations permits a mask to be used for both selecting bits from a fraction and assigning an exponent value from the mask word to the result. The final binary point alignment can be achieved by a shift operation.

4.3 Processor Intercommunication - The Routing Network

In virtually every problem for which an array processor is suited, the processors in the array need to exchange data values from time to time. Indeed, the scope of the problems for which a particular array processor is suited can depend on the flexibility of its data interchange network. The data interchange network of this design - hereafter called the routing network - is a three stage Clos network (Clos, 1953; Benes, 1965). Although Clos proved that such a network can perform any permutation of the input signals to the output ports, his proof did not provide a guide to a general algorithm for controlling the network. This author is among a growing group of people who would like to have such an algorithm.

The general form for a Clos network is shown in Figure 4.3-1, and the specific form used in this design is shown in Figure 4.3-2. The author is indebted to William Stenzel for many of the ideas which lead to the formulation of the routing network in this form.

The last two stages of a Clos network form what Lawrie (1973) has called an omega network. In his thesis, Lawrie shows that an omega network, among other operations, can perform uniform circular shifts of arbitrary distance and direction. In later work, Lawrie and Wen (1975) have discovered simple control algorithms for the omega network which permit its use in partitioned form to perform several simultaneous circular shifts of independent amount and direction within the separate partitions. For an omega network such as we have in this design, the size of all partitions must be an integer power of two, although the partitions may have various sizes. What must hold for each partition, however, is that with the input ports numbered

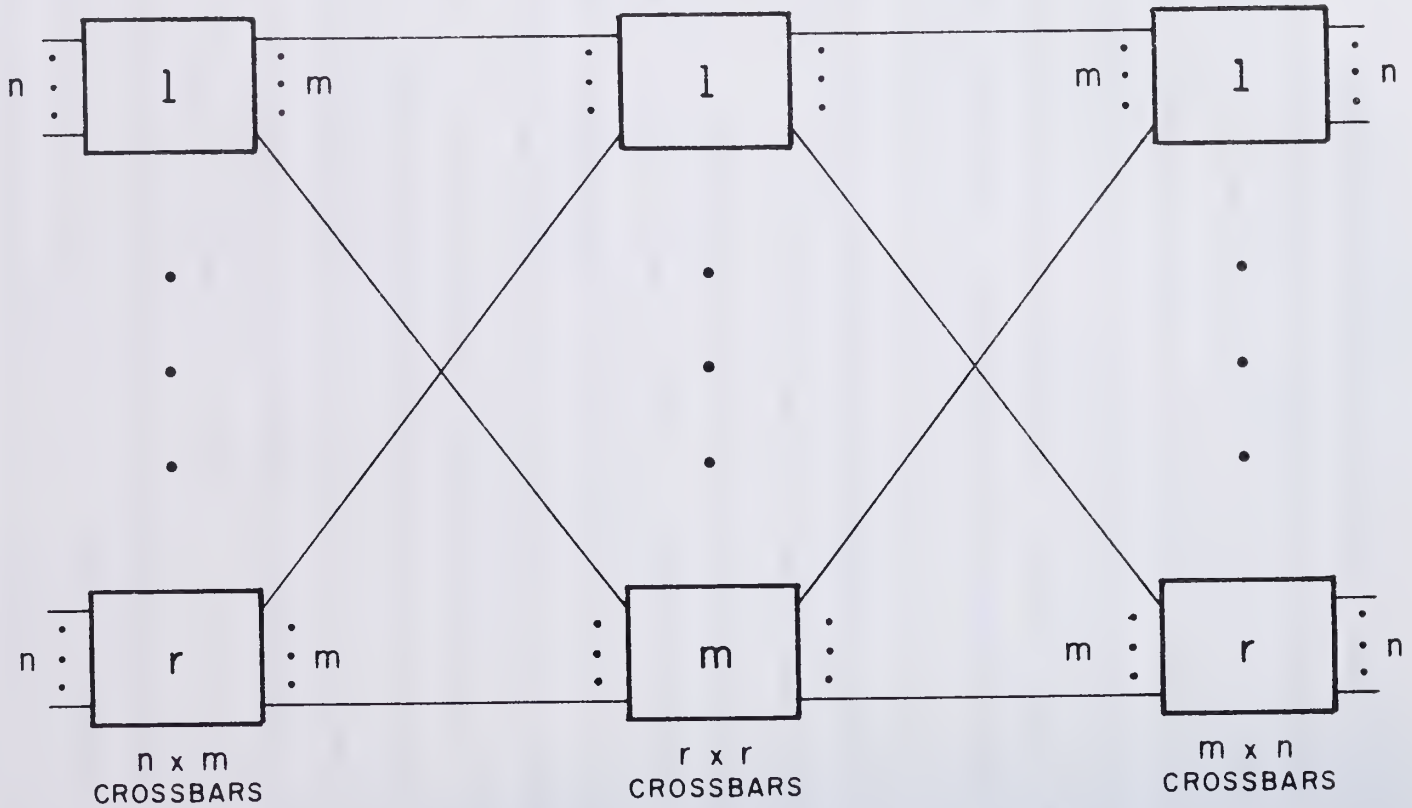


Figure 4.3-1 The General Clos Three-Stage Network

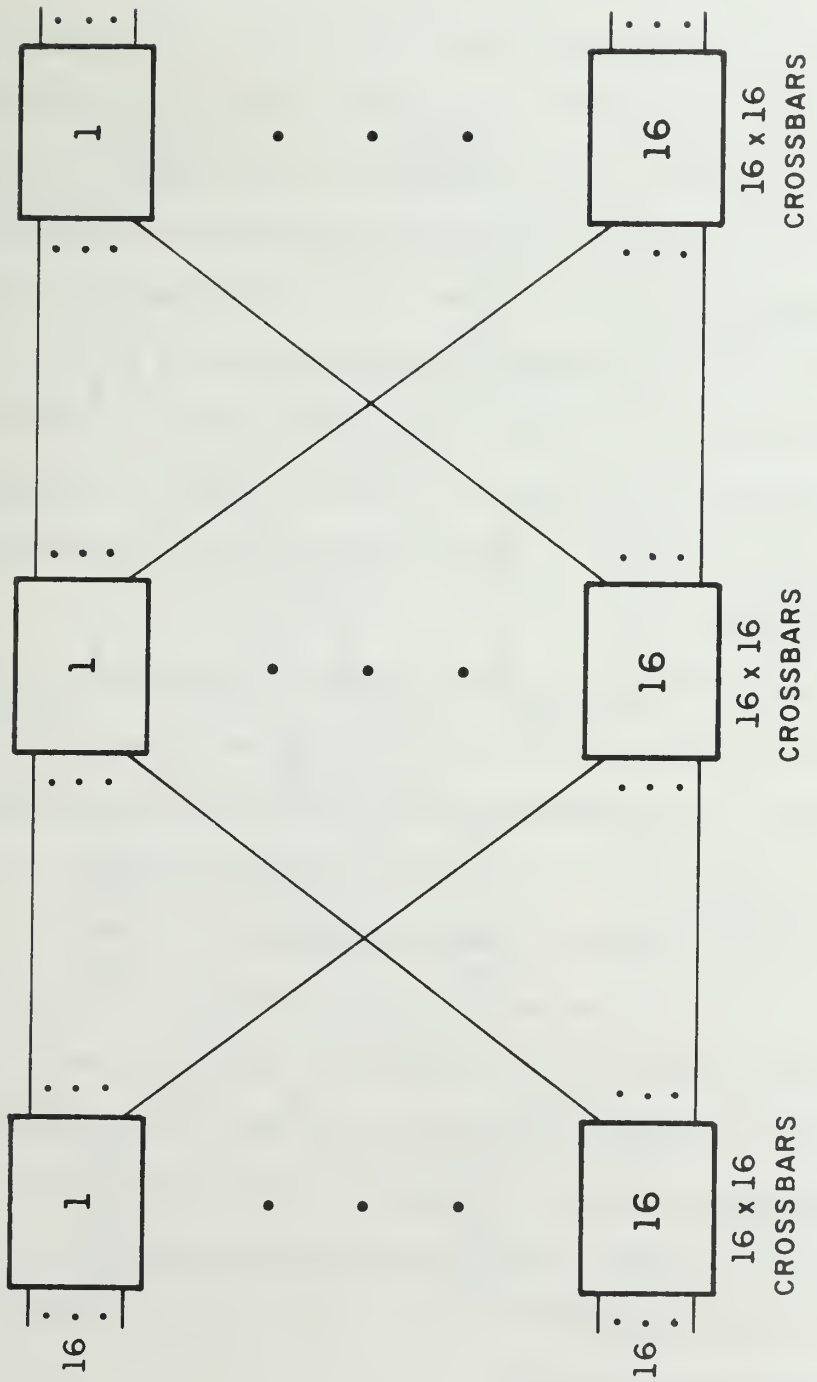


Figure 4.3-2 A Clos Three-Stage Network for the Design

from zero to N-1, the index number of the lowest numbered input port of a partition must be congruent to zero modulo the size of the partition. The Clos network, of course, permits arbitrary partitions, but we have only been able to find an algorithm for uniform shifts of one in either direction within arbitrary partitions. Where other shift amounts are necessary, one must either conform to the partition restrictions of the omega network and use the Clos network as an omega network by sending the input operands straight through the first stage of crossbars without interchange, or make multiple passes through the general Clos network if non-omega suited partitions must be used.

The details of the interconnections between the crossbars in the Clos network are given in Figure 4.3-3 for a two stage network of four by four crossbars. The figure shows the sixteen input ports of the network divided into four groups of four. The destination number, d, of a lead from an ouput port source of the first stage, s, is given by

$$d = (s*N + g) \text{ modulo } N^k,$$

where all port numbers begin at zero, g is a crossbar number (beginning with zero), N is the number of input and output ports for an individual crossbar, and N^k is the total number of input and output ports of the network as a whole. Every transmitting switch sends exactly one value to every receiving switch in the next stage.

4.3.1 Routing Network Control

The following two sections describe the techniques needed to control the two stage omega network and the three stage Clos network. No hardware is in the design to support run time execution of these algorithms.

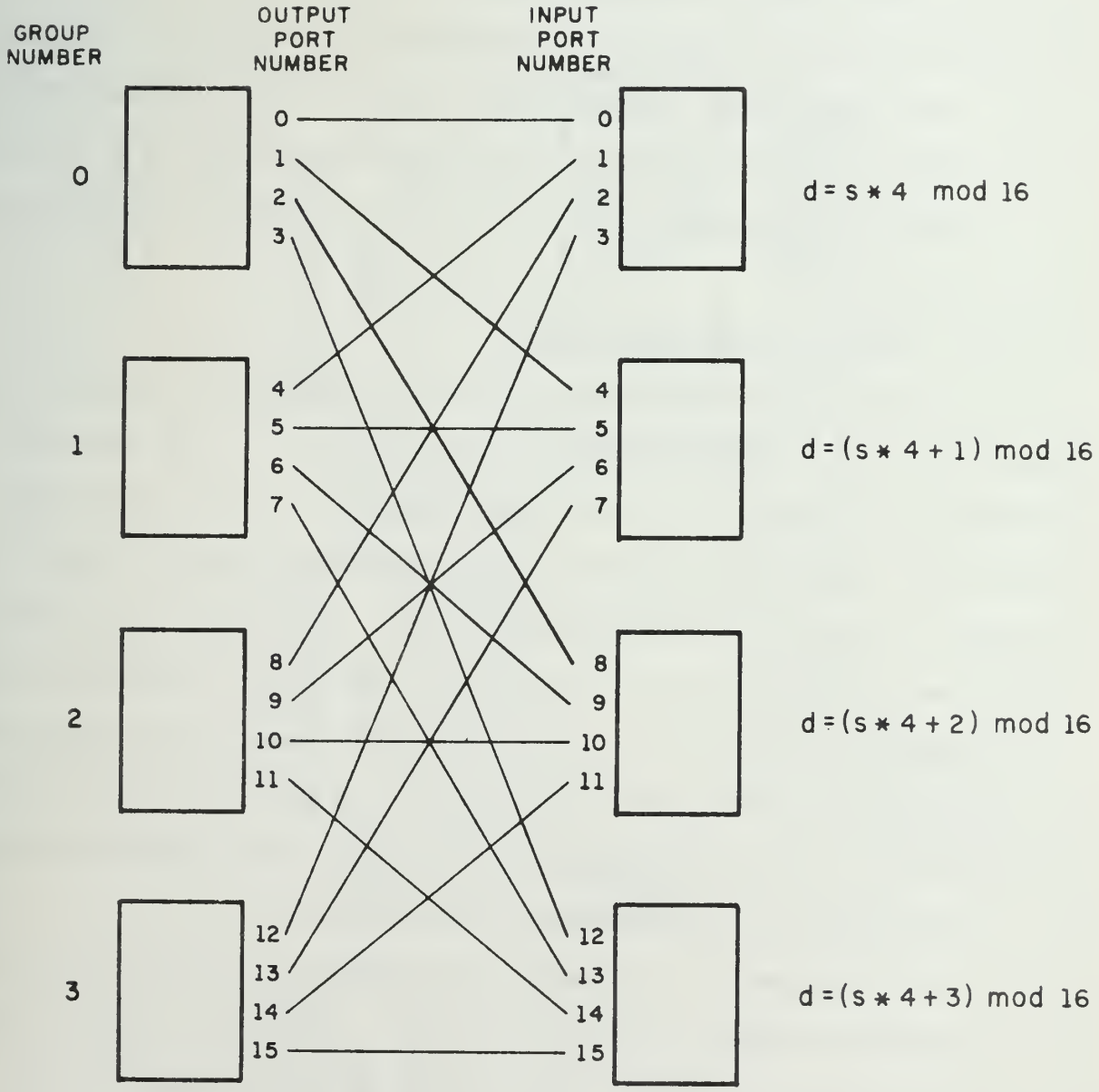


Figure 4.3-3 The Details of Inter-Stage Connections within the Routing Network

The crossbar implementation includes a memory to store sixteen four bit routing control words for each data path (the 10145 of Figure 4.3.3.1-1). A path from the data register to the memory input permits the control memories to be loaded with values computed by the compiler or other software external to the machine. As we will see in section 6.2, this capability is sufficient to support the general circulation model and several other algorithms of practical interest.

4.3.1.1 Control of the Omega Network

The omega network in this design is composed of two stages of sixteen by sixteen crossbars. Sixteen is the square root of 256, the total number of input ports. The destination address for any data value which enters the omega network from the first Clos network stage is an eight bit number; the four high order bits are the number of the third Clos stage to which the value must be sent. The low order four bits of that address give the number of the output port of that crossbar to which the data value should be sent. Lawrie (1973) and Wen (1975) have shown that the omega network can perform all of the following useful data routings within suitable partitions:

1. Circular shifts in either direction of any amount.
2. Uniform separation of a group of contiguous values (unless p, the ultimate separation distance, is relatively prime to the partition size, P, only P divided by the greatest common divisor of p and P elements can be "expanded"),
3. Elements originally separated by uniform separations p can be brought together. Again, unless p and the partition size P are relatively prime, elements separated by p units distance fail to wrap around, and only P

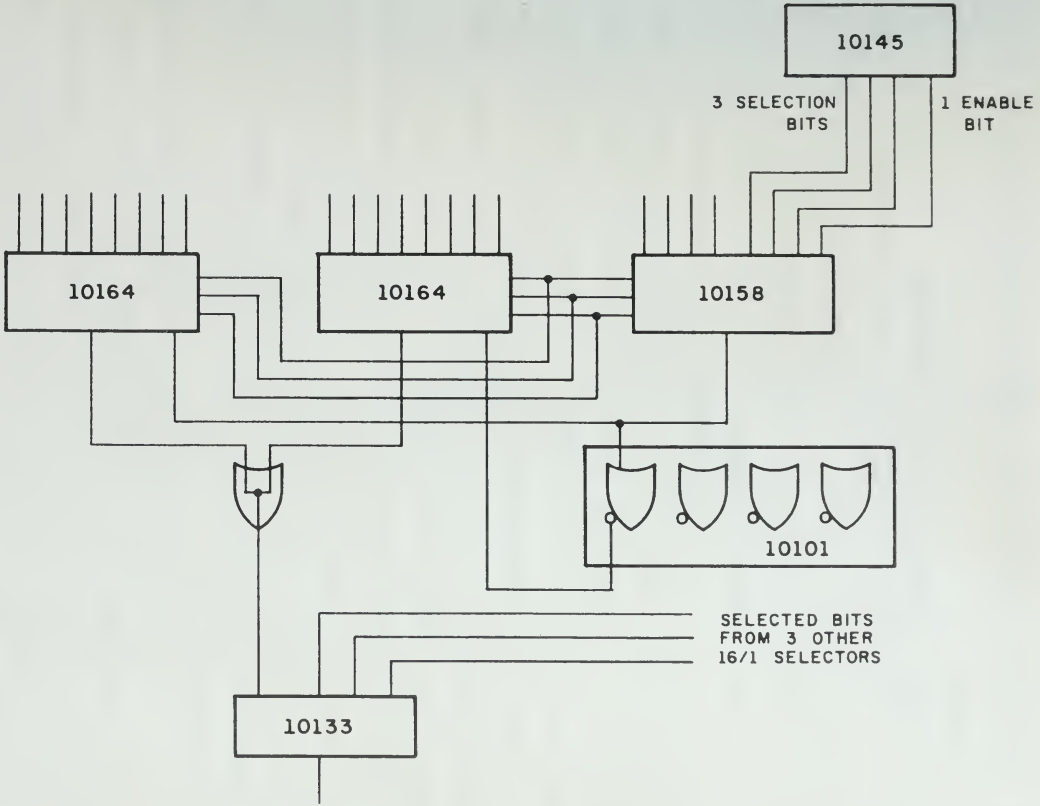


Figure 4.3.3.1-1 The Logic for a One Bit Path Through a Sixteen by Sixteen Crossbar Switch

divided by the greatest common divisor of p and P elements can be processed.

4.3.1.2 Shifts of One Position in a Clos Network

The argument of this section presents a description of the cases illustrated in Figure 4.3.1.2-1. Three types of interactions of partitions with the crossbar switches of the routing network are shown.

As the diagram shows, no more than one value needs to move up from one switch in the first stage to another in the third stage, and no more than one value needs to move down from one first stage switch to another third stage switch. If we send all values which must move up to the top switch in the second stage and all values which must move down to the last switch of that stage, we are guaranteed that there will be no more than sixteen such values, and moreover, that no two such values need to go to the same third stage switch. Values in partitions like "A", "D" or "E" can be routed straight through to the third stage, which can interchange them as required. Only if there are partitions such as "D" or "E" will there be less than sixteen values which must move up and down. One value from such partitions can arbitrarily be sent to the top and bottom second stage switches to fill otherwise unused positions.

This argument is difficult to extend to the case where shifts of more than one position are involved, for then it is difficult to account rigorously for all switch positions, and to insure that no second stage switch recieves two or more values destined for the same third stage switch.

4.3.2 ECL Logic

The choice of ECL current mode non-saturating logic for the imple-

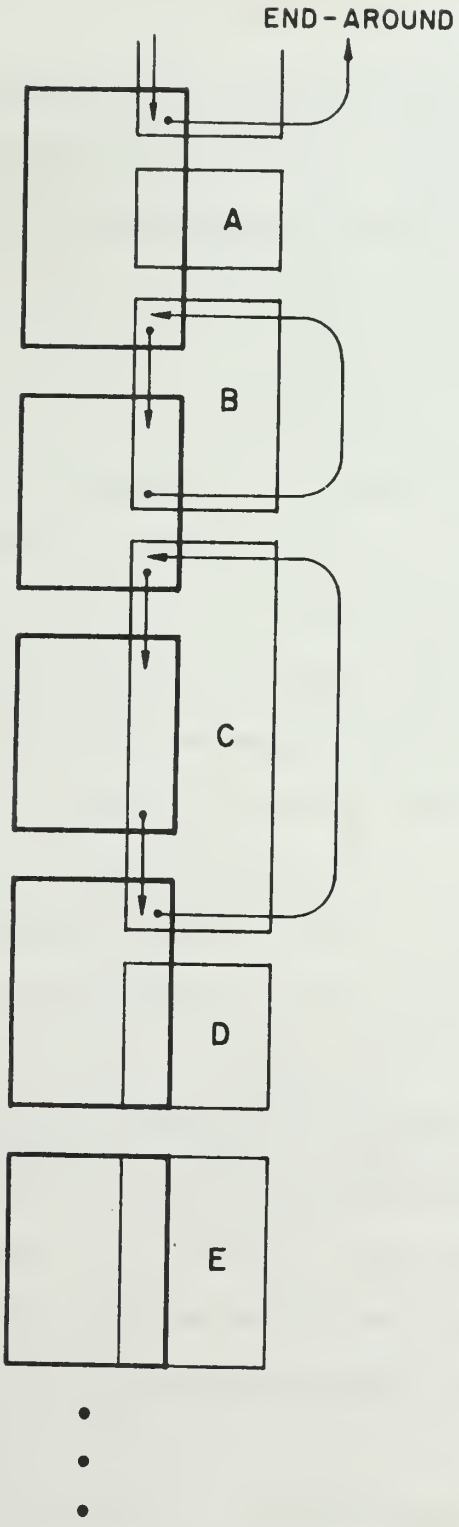


Figure 4.3.1.2-1 The Possible Interactions of Partitions with Crossbar Switches

mentation of the routing network was dictated by two factors: first, we want to be able to route a set of operands through the network in a time comparable to that of a processor operation, and second, we want to minimize problems with noise and signal cross-talk in the many cables of the routing network. The differential pairs of the ECL family, while necessitating rigorous balancing of line impedances, give - in return - effective isolation of the ground and signal levels of the driving and receiving logic. These two advantages of ECL logic over TTL prompted the decision to design the routing network with ECL logic.

The ECL logic packages used in this design are those in the series developed by the Motorola Corporation and usually referred to as MECL 10000. Many other manufacturers provide a second source for these circuits, and the reference used for the data on 10000 series circuits used in this paper is Signetics Corporation (1974A). In logic diagrams, ECL packages are labelled with their part number, which is uniformly five digits beginning with one and zero.

4.3.3 Routing Network Time and Component Count Estimates

The routing network can be built either as a pure switching system through which values flow in one step, or it may be built with registers in each stage so that successive values may flow through it in pipeline fashion. A third option, not considered further here, is to build one stage of crossbars and cycle values through it twice for omega network operations and three times for Clos network operations. In any case, crossbar switches for less than the full forty bit width can be built and used in byte serial fashion. Table 4.3.3-1 gives the details of a component count analysis for the pipe-

Components	Component Counts			
	Pipelined Unit		Non-Pipelined Unit	
	Per Bit	Per Crossbar	Per Bit	Per Crossbar
10101	-	4	-	4
10133	¼	-	-	-
10145	-	16	-	16
10158	-	16	-	16
10164	2	-	2	-
Totals	16 * 2¼ * B + 36		16 * 2 * B + 36	

Table 4.3.3-1 Crossbar Component Counts

Clos Network			Omega Network		
Pipelined		Non-Pipelined	Pipelined		Non-Pipelined
Total Time	Last Stage		Total Time	Last Stage	
286	72	244	227	72	189

Table 4.3.3-2 Routing Network Propagation Times

lined and non-pipelined designs for a sixteen by sixteen crossbar in terms of the parameter B, the width in bits of the data path through the crossbar. Table 4.3.3-2 presents the propagation time in nanoseconds through various networks. Its values are derived by consideration of Figure 4.3.3-1 which illustrates the hardware components through which a signal must flow in a Clos network. (Also see section 4.3.3.1.) The total network switching time and the component count for one crossbar given in Table 4.3.3-3 for crossbars of all reasonable byte sizes. The expected cycle time of memory for the system is nominally 500 nanoseconds. Table 4.3.3-3 shows that to keep the time for one routing step commensurate with this time, either a twenty bit non-pipelined network, a pipelined Clos network for ten bit bytes, or a pipelined omega network for eight bit bytes should be built. The component count aspect of the issue makes it clear that the pipelined design is to be preferred.

The essential steps in the piplined implementation are:

1. Transformation of the data from the parallel form of the processors to the byte serial form for the routing network,
2. Transmission of the byte serial data through the routing network, and
3. Transformation of the byte serial data back to fully parallel form.

The following two sections discuss the tranformation and transmission aspects of the routing network hardware.

4.3.3.1 Data Transmission and Broadcasting

The data transmission logic is two or three stages of byte serial sixteen input by sixteen output crossbar switches. The essential elements of this network, the crossbar switches, are implemented by the logic of Figure 4.3.3.1-1, which shows the logic necessary to implement a one bit path.

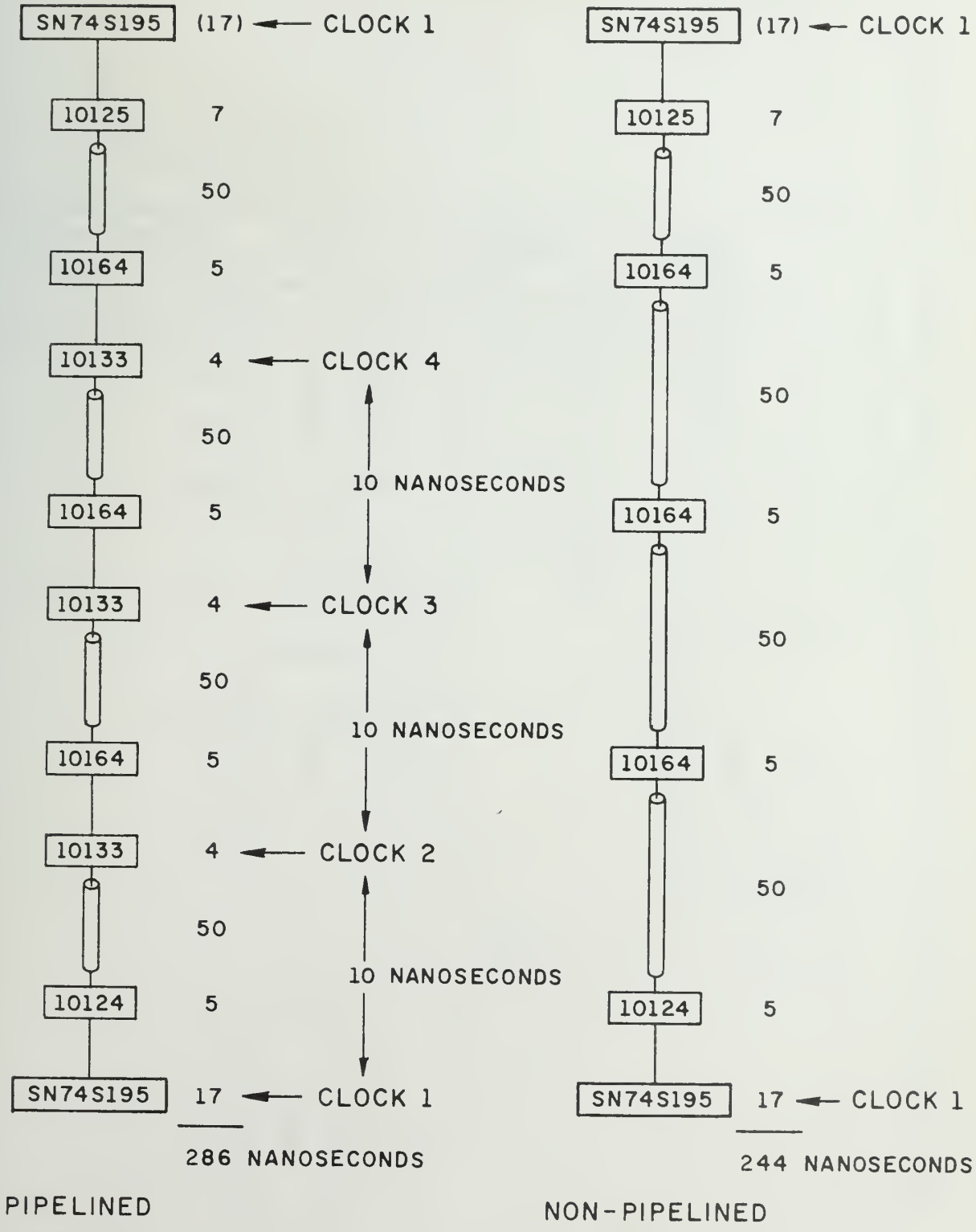


Figure 4.3.3-1 Clos Routing Network Timing Estimates

Byte Size	Pipelined			Non-Pipelined		
	Crossbar Components	Namoseconds		Crossbar Components	Namoseconds	
		Clos	Omega		Clos	Omega
40	1476	286	227	1316	244	189
20	756	358	299	676	488	378
10	396	502	443	356	976	756
8	324	574	515	292	1220	940
5	216	790	731	196	196	1512

Table 4.3.3-3 Component Counts and Network

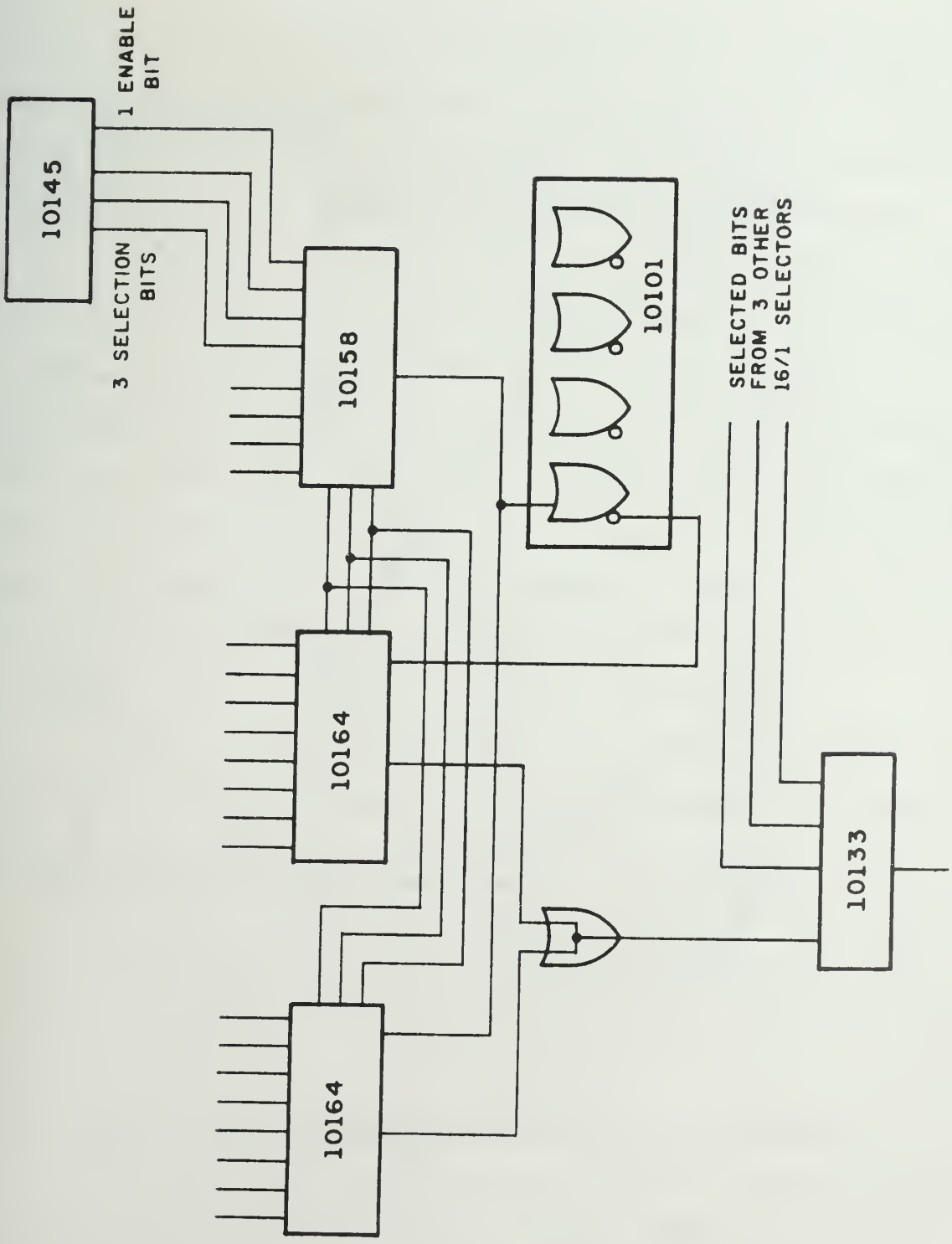


Figure 4.3.3.1-1 The Logic for a One Bit Path Through a Sixteen by Sixteen Crossbar Switch

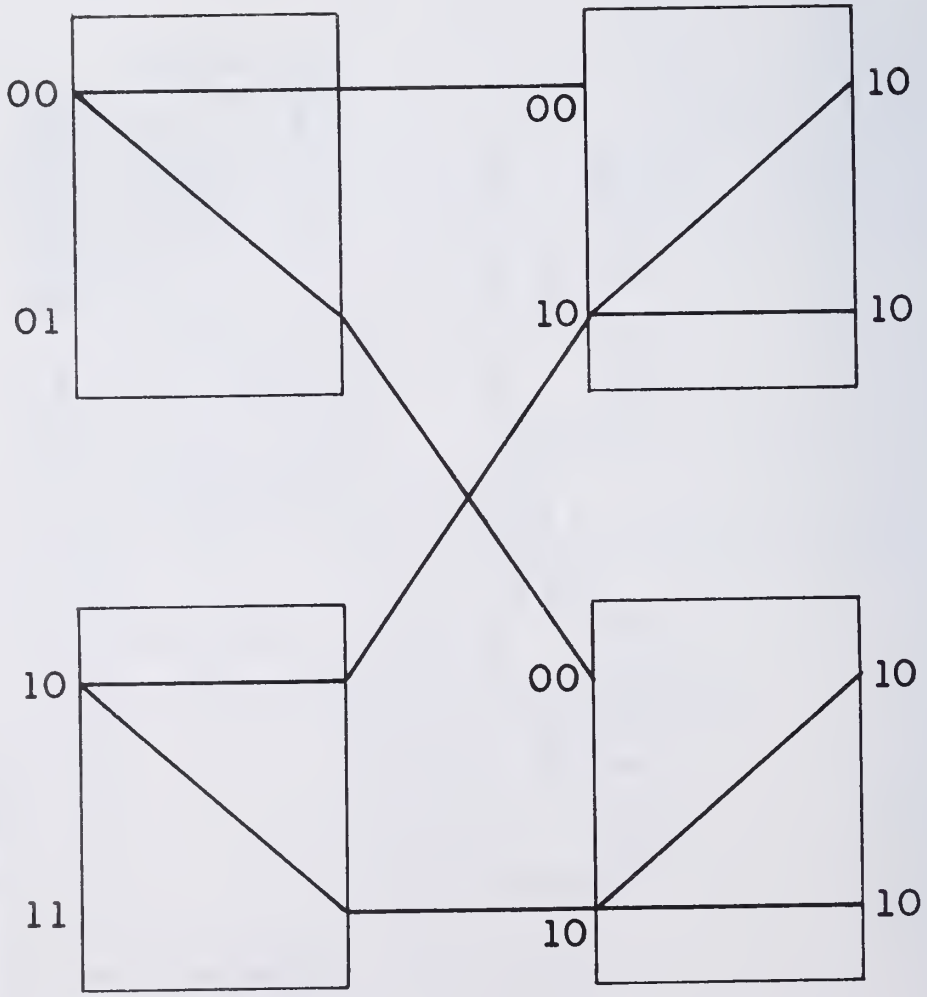


Figure 4.3.3.1-2 Broadcasting with a Routing Network

The 10145 storage register shown in the figure stores the control bits for all eight paths for one of the sixteen bytes through the crossbar. Three of the four bits in a control signal select one of eight inputs as the output of two 10164 eight-to-one selectors whose outputs are wire ORed together. The fourth control bit, complemented by the 10101 inverter, serves to decide which of the two selectors is enabled and which is disabled. The 10158 quadruple two-to-one selector permits either local or global control of the switching path to be selected. The 10133 four bit latch holds the selected result for the stage; these latches are the registers which permit pipelining of the byte signals through the three stage network. Thus, each bit switched through the crossbar requires two 10164 selectors, one quarter of a 10133 latch and a 1010 quadruple inverter, and one eighth of a 10158 selector and a 10145 register file.

A value from any of the 256 input ports of the routing network can be broadcast to all 256 output ports using only two stages of crossbars. The process is illustrated in Figure 4.3.3.1-2 for a two stage network of two by two crossbars. The low order part of the address of the desired broadcast input determines the setting for all first stage crossbars, and the high order part of that address determines the setting of all second stage crossbars.

4.3.3.2 Data Parallel-to-Serial and Serial-to-Parallel Conversion

The hardware which performs parallel-to-serial and serial-to-parallel conversions resides in the processors as the dis-assembly and re-assembly logic of Figure 4.5.2.17-2. This hardware is shown in successively more detail in Figure 4.3.3.2-1 and Figure 4.3.3.2-2. Figure 4.3.3.2-1 shows a complete

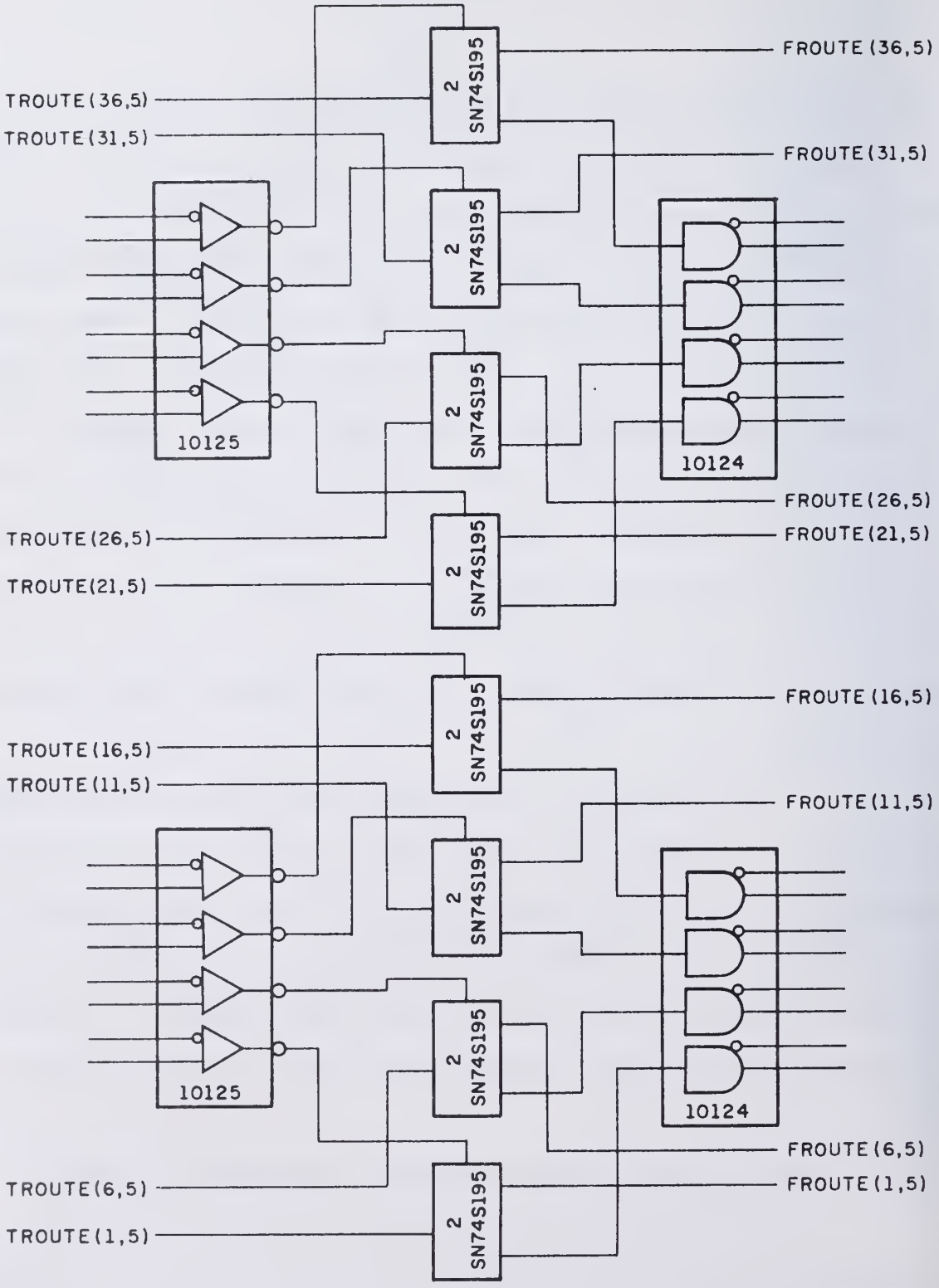


Figure 4.3.3.2-1 The Parallel-to-Serial and Serial-to-Parallel Conversion Logic

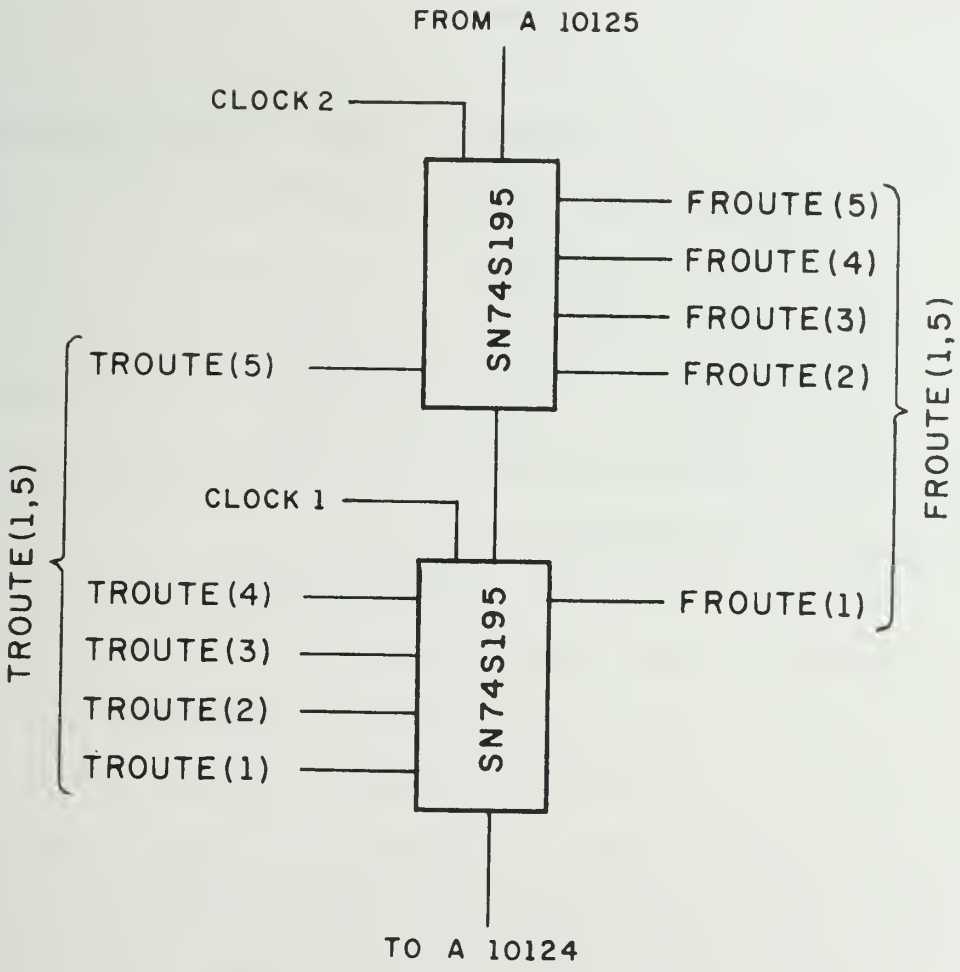


Figure 4.3.3.2-2 The Details of a Block of the Parallel-to-Serial and Serial-to-Parallel Conversion Logic

forty bit dis-assembly and re-assembly register together with its associated drivers and receivers. The SN74S195 four bit parallel in and parrallel out shift registers are TTL circuits which receive values from the operand registers of the processor and transmit values to the fraction selector of the processor. The 10124 differential drivers receive TTL signals from the SN74S195 shift registers, convert them to standard ECL levels, and transmit them in differential pair form to the ECL logic of the routing network. The 10125 differential receivers accept ECL differential signal pairs from the routing logic and convert them to TTL levels.

The assembly-disassembly register hardware can be implemented with fewer components for eight bit byte operation than for ten bit byte operation. The discussion of the next paragraph discusses an eight bit byte design. The eight bit design requires sixteen SN74S195 register whereas the ten bit design requires twenty. Furthermore, the eight bit design uses only four ECL 10000 series components; the ten bit design uses six.

Figure 4.3.3.2-2 shows the details of one of the SN74S195 blocks of Figure 4.3.3.2-1. Table 4.3.3.2-1 lists the eight steps which are used to transmit a forty bit value through a Clos routing network in five eight bit bytes. In step one, five consecutive bits from the operand registers of the processor are loaded in parallel into the SN74S195's shown using CLOCK1 and CLOCK2 in synchrony. The results of step one, taken from the serial output pins of the eight SN74S195's (pin twelve), are available to the routing network as byte one of the input.

Step two uses CLOCK1 and CLOCK2 in synchrony again to perform a serial shift which makes the eight bits of byte two available to the routing

Cycle	CLOCK1	CLOCK2	Input	Output	Comments
1	1	1	forty bits	byte one	Parallel load from operand registers
2	1	1	none	byte two	Serial shift
3	1	0	none	byte three	Serial shift
4	1	1	byte one	byte four	Serial shift
5	1	1	byte two	byte five	Serial shift
6	0	1	byte three	none	Serial shift
7	0	1	byte four	none	Serial shift
8	1	1	byte five	none	Serial shift

Table 4.3.3.2-1 The Steps in Data Transmission Through a Clos Routing Network

Cycle	CLOCK1	CLOCK2	Input	Output	Comments
1	1	1	forty bits	byte one	Parallel load from operand registers
2	1	1	none	byte two	Serial shift
3	1	1	byte one	byte three	Serial shift
4	1	1	byte two	byte four	Serial shift
5	1	1	byte three	byte five	Serial shift
6	0	1	byte four	none	Serial shift
7	1	1	byte five	none	Serial shift

Table 4.3.3.2-2 The Steps in Data Transmission Through an Omega Network

network; at the end of this step, no data remains in the upper SN74S195 of each pair. Step three uses CLOCK1 alone to shift the third byte into output position. At the end of step three, the first three data bytes are in the register of the routing network pipeline. On step four, CLOCK1 is used to supply byte four to the network and CLOCK2 is used to receive the first byte of the routed result from the network. Steps five through eight complete the routing process. On step eight, CLOCK1 and CLOCK2 are used in synchrony to accept fifth and last byte of the routed result. Although the design presented is used with forty bit parallel inputs, it is clear that the technique described by Table 4.3.3.2-1, with the addition of one more step which uses both clocks in synchrony, could be used to transmit data words of up to forty-eight bits in six bytes of eight bits each. Because latches and not master-slave flip-flop are suggested for use in the crossbar switches, clock signals controlling the flow of data through the network and logic of this section would probably have to be applied in time starting with CLOCK2 (and for step eight, CLOCK1 and CLOCK2) of Figure 4.3.3.2-2 and proceeding in succession from right to left through the three stages of the routing network of Figure 4.3-2. In particular, CLOCK2 could never be used to both shift a bit out for output use and in for input use at the same time.

The seven steps in the data transmission process for a two stage omega network are given in Table 4.3.3.2-2. Because the two stages only hold two data bytes in the pipeline, there is no spare step, similar to that in the Clos process, so that the capacity of the network is limited to forty bits in five eight bit bytes if the logic of Figure 4.3.3.2-1 is used for the parallel-to-serial and serial-to-parallel conversion process.

4.3.4 Table Look Up

A table look up facility is provided within the routing hardware to support the table look up needs of the model, primarily those of the long wave radiation calculations. The table look up unit is shown in Figure 4.3.4-1. One table look up unit is included for each of the sixteen routing units. The hardware includes one processor memory module, an assembly dis-assembly register like that of Figure 4.3.3.2-1, four SN74LS193 low power Schottky four bit counters which form an address register, and four SN74157 quadruple two-to-one selectors to determine the source of the memory address. The assembly register receives data from port one of its corresponding first stage crossbar. The dis-assembly register delivers data to input port one of its corresponding last stage crossbar.

The unit operates in two different modes. In the first mode, each processor computes the address of the table value which it wants, using integer arithmetic and the index adder discussed in section 4.2.5.1.11. The address for the table entry for processor zero of each first stage routing crossbar is clocked into the assembly register in two cycles. The data is read from memory, dis-assembled and sent via the last stage crossbar back to processor zero. The two address bytes from processor one could be clocked into the assembly register as the last two bytes of data are clocked out to register zero. This process continues until all sixteen words requested by the processors have been delivered.

The second mode of table look up operation is table loading in this mode, as initial table address is sent from an appropriate source. In some cases, the address may be broadcast from the control unit; in other cases,

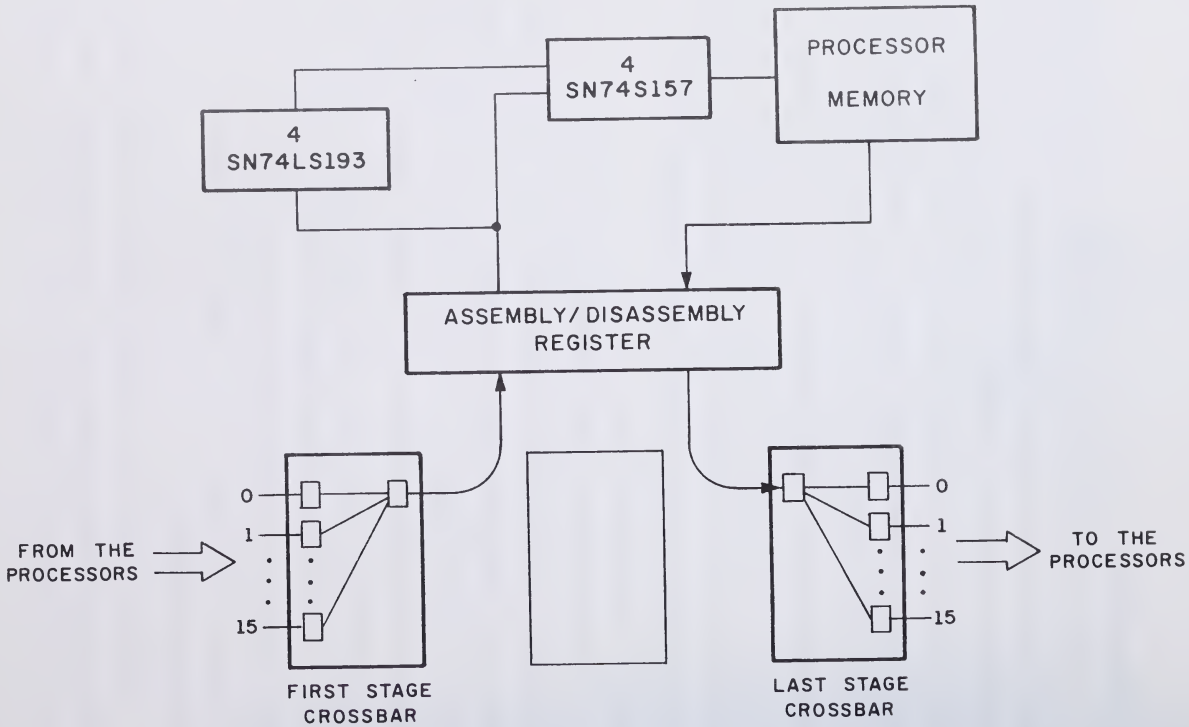


Figure 4.3.4-1 One of the Sixteen Table Look Up Sites

an address unique to each table look up memory may be used: it is not necessary that all look up tables have the same contents. The set of processors can be partitioned by using the routing network to execute several programs with different table contents simultaneously. The initial block address is clocked into the register composed of the four SN74LS193 up-down counters. A succession of table words from an appropriate source are sent; between words the storage address is incremented or decremented by one as appropriate.

At this point, a further remark about the logic of Figure 4.3.3.2-1 is in order. If the bit assignments shown in the figure were strictly adhered to, the eight bit bytes transmitted by the routing network would not correspond to contiguous eight bit segments of processor operands. In particular, if the processor is to be able to compute a table address and transmit it in two byte transmissions to the table look up unit, an input bit order from that shown in Figure 4.3.3.2-1 is required. Of course, the arrangement of the output bit assignments can be reordered so that values are transmitted correctly by the routing network. Suffuce it to say that the input arrangement is arbitrary, and that an arrangement which supports the needs of efficient use of the table look up unit can be used without harming the other operational needs of the routing system.

4.3.5 Communication with the Control Unit and the Input-Output Channel

The routing unit forms the basis for intercommunication among the elements of the machine as well as with the input-output channel and any possible future secondary storage. The main function of the routing unit, that of providing communication paths between the processors, has been discussed in previous sections. The following two sections discuss the use of the

routing unit in support of data flow between the control unit and the processors, and also in support of data flow between the machine and the perpheral world envisioned for this design.

4.3.5.1 Communication Between the Array and the Control Unit

As we saw in section 4.3.3.1, two stages of the routing network permit a value to be broadcast from any one input port to all output ports. The control unit can, therefore, send a value to all processors if it can transmit that value to any one of the input ports of the first routing unit stage. It can receive a value from any of the processors by accepting a value from any of the second stage output ports if that value has been broadcast to all of those ports by the first two stages of the routing network.

4.3.5.2 The Routing Unit in Support of Input and Output

Data transmission to and from a sequential external device on the input-output channel can be supported by using the 256 eight bit registers of stage one of the routing network as a large circular shift register. Information to the control unit would enter any stage one input port and be broadcast to the output port for the control unit in stage two. Information from the control unit to the channel would flow through the control unit's input port and be broadcast to an output port which is connected to the channel.

For volume data input from a sequential device, successive bytes can be sent in through any stage two input port, broadcast to the third stage, and clocked into the appropriate processor assembly register for subsequent storage in array memory. Volume data output to a sequential device can be broadcast from the first stage input ports in any desired order to all second stage output ports. Any one of these can be connected to the channel.

Paths from a parallel access secondary storage device - not proposed for the general circulation model - could be attached to consecutive input ports of one stage shifted uniformly to the desired position in the next stage. Although 256 parallel paths are conceptually simpler to deal with, any number less than that can be accomodated by the joint use of mode and routing control. Paths to a parallel access secondary storage device could be attached to the second or third stage output ports, and blocks of data could be shifted to those ports from either processor or control unit memory.

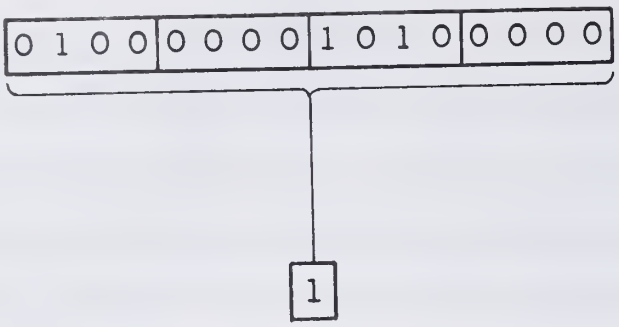
4.4 The Control Unit

The control unit must provide control signals to operate the three other main components of the design: the processors in the array, the routing unit, and the input-output channel interface. As we have seen in section 4.3.4, the bulk of the load for input-output control is the task of the routing unit control logic.

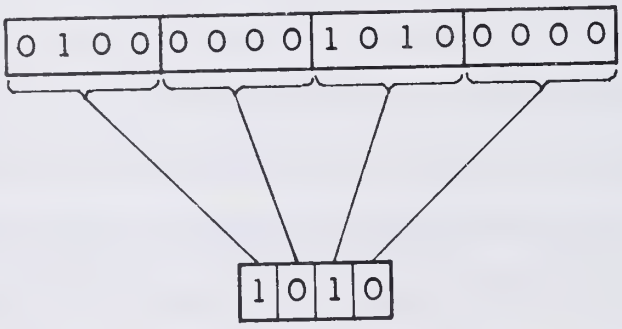
4.4.1 Control of the Processor Array

By design, the processors are simple to control. For each step, a set of control signals and one clock pulse are all that is required. The obvious control mechanism is a read only memory in which the proper control signal sequence are stored together with simple hardware to interpret the instruction stream and send the appropriate sequence of control signals to the array.

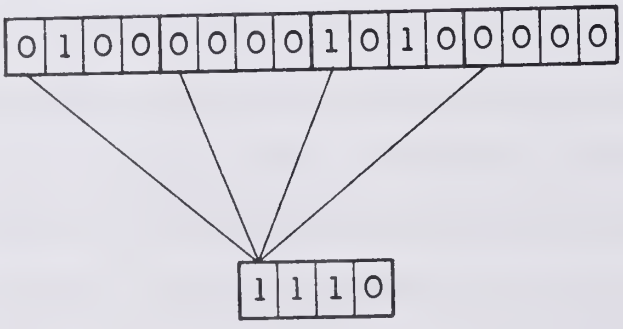
The control unit can sample the status of any processor by examining its mode, condition and status register contents by way of the routing network. Figure 4.4.1-1 illustrates the three ways in which the control unit can access the 256 MODEOUT signals from the mode logic of the 256 processors in the array. An array of sixteen processors is shown in the figure, arranged in four groups



(a)



(b)



(c)

Figure 4.4.1-1 Reception by the Control Unit of the MODEOUT Signals

of four. In the design, the 256 processors would be arranged in sixteen groups of sixteen; each four bit group of Figure 4.4.1-1 thus corresponds to a sixteen bit group in the system. The control unit can access the logical OR of all 256 MODEOUT bits as shown in Figure 4.4.1-1(a). It can access a sixteen bit value whose bits represent the logical OR of the MODEOUT bits of the processors in a sixteen bit group either of ways. In part (b), sixteen contiguous MODEOUT logic bits are ORed to form one bit. In part (c), the sixteen bits from corresponding positions in each of the sixteen groups of contiguous processors are ORed.

Figure 4.4.1-2 illustrates the three ways the control unit can supply the MODEIN bit to the mode logic of the 256 processors. All 256 MODEIN signals can be the same, as shown in Figure 4.4.1(a). Sets of sixteen processors can be supplied with a common MODEIN bit value in the two way illustrated by parts (b) and (c) of Figure 4.4.1-2. In all cases, of course, the MODEIN value can be combined with local control information stored in the mode register and status register of each processor.

4.4.2 Control of the Routing Network

Control of the routing network - as section 4.3 makes clear - required sequences of synchronized and phased clock pulse interspersed with shift control and selection signals. Although the precise nature of the control signals differs in kind from those for the array of processors, the same technique can be used for the routing network as was used for the processor array. The question as to whether two asynchronous control devices, one for the processors, the other for the routing network, would prove cost effective was not answered before work on the design ceased.

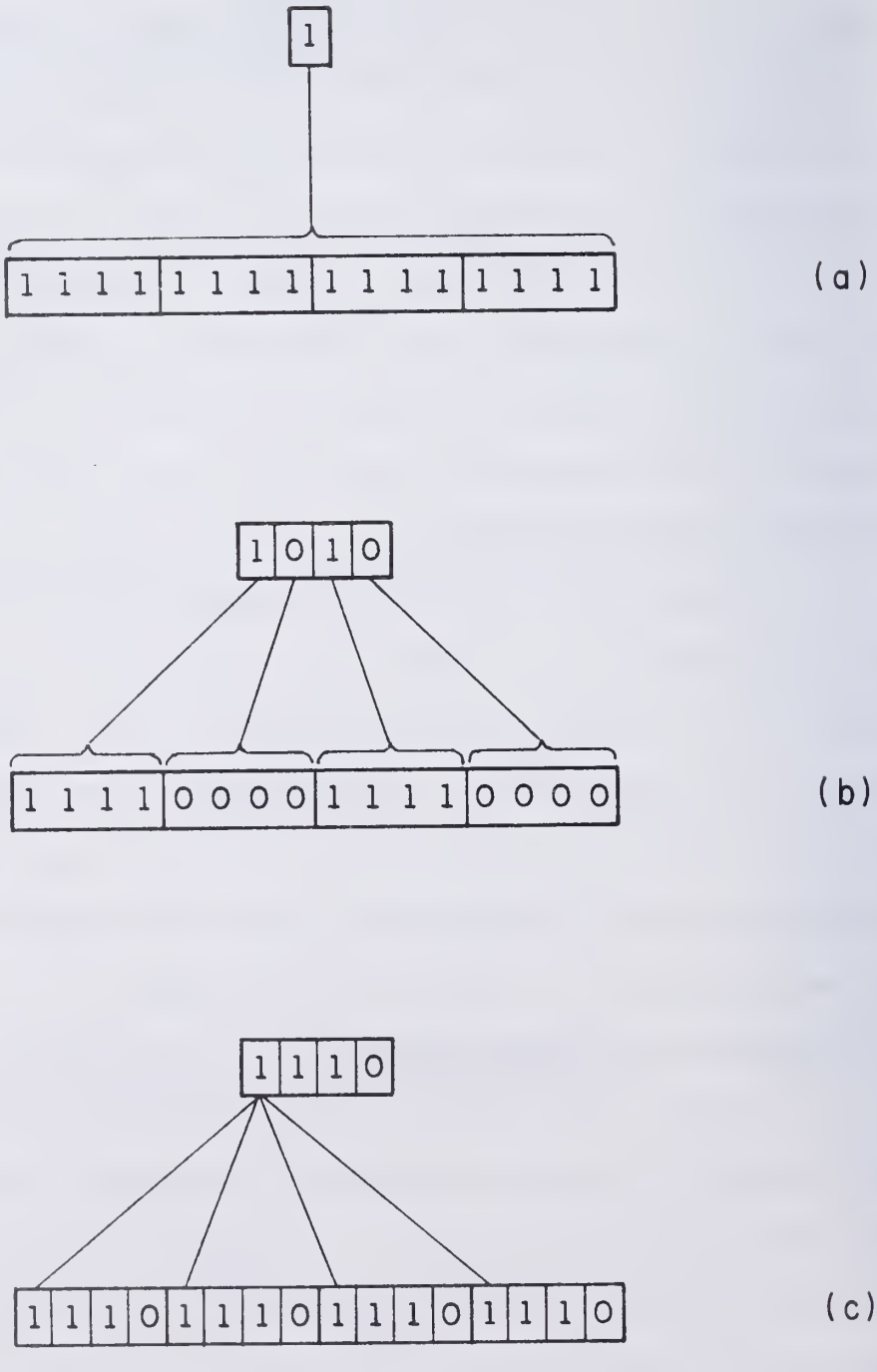


Figure 4.4.1-2 Transmission to the Processor Array of the MODEIN Signal

5. Design Testing

The multiplier design was tested by constructing a hardware prototype, and the floating point addition logic was tested by simulation. The following two sections discuss these two efforts.

5.1 The Logic Simulation System

Breuer has edited a book on simulation of computer systems, and one of its chapters (Breuer, 1972) discusses logic simulators. Two classes of simulation techniques are identified: the compiled code model and the table driven model. In these terms, the logic simulator described here is a compiled code simulator.

In the bibliography for the logic simulation chapter, there are references to many papers about logic simulation. The larger majority of both the references and the chapter deals with gate level simulation. The simulator of this paper is a package level simulation. The references uniformly discuss how their authors constructed simulators; no off-the-shelf simulation system suitable for package level simulation exists that does not require the user to write his own package simulation routines. This view was confirmed by conversation with Dietmeyer (1975). Since the bulk of the work in constructing the simulator presented here was exactly that of writing the package simulation routines, the author feels that no duplication of available material is represented by the simulator construction effort described here.

Figure 5.1-1 is a diagram of the logic simulation system. The primary input to the system is a description of logic to be simulated. A preprocessor accepts this description and produces two items:

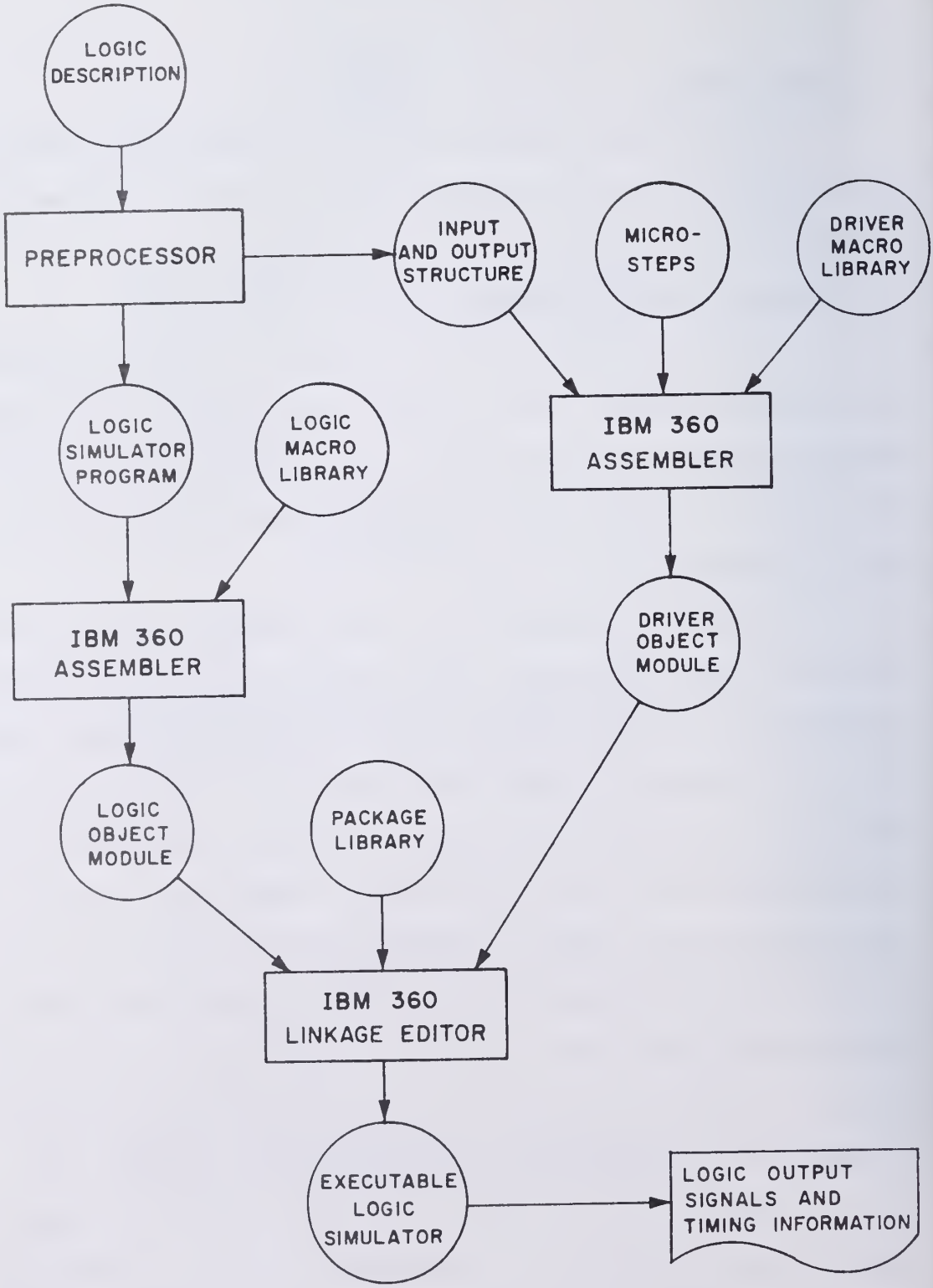


Figure 5.1-1 Diagram of the Logic Simulation System

1. An assembly language program, consisting entirely of macro calls, which simulates the input logic, and
2. A macro and a macro call which define the structure of a driving module for the input logic.

Except for a few lines, the macro calls in output (1) above correspond one-to-one with packages in the logic. Each logic function is represented by a macro which, when assembled, simulates the action of the package. Some of these macros expand into executable code directly, while others expand into subroutine calls on simulation modules which reside in a package library. The macros, not the preprocessor, determine whether a compiled code or table driven simulator results from the approach described here. Note also that the complexity of the packages simulated can vary from simple AND, OR level gates to single packages which perform a full fraction multiplication. Although the set of macros chosen for the particular simulator described here do not permit it, a package could well be simulation module produced by the system for a part of the subject logic, so that modular investigation and debugging of a design can be supported by the technique described here.

Output (2) above consists of a macro called STEP, written by the preprocessor, which is called by the user of the package. A STEP call results in one execution of the subject logic with the values for the input variables given in the call. The only other output included in (2) is a call on the macro BEGIN with all of the input and output signals for the subject logic as parameters. Execution of this call begins each execution cycle by setting the time portion for each input signal to the maximum of the times from the output signals of the previous cycle. Assembly of output (2) together with a

handwritten series of STEP calls produces a module which exercises the subject logic.

By saving the logic object module and the input and output structure description shown in Figure 5.1-1, the user of the simulation system can execute the subject logic as many times as desired, having assembled it only once.

5.1.1 The Logic Simulator Language and the Preprocessor

Tessler (1968) has defined a single assignment language as one with the following properties:

1. Every statement is an assignment statement.
2. No two statements assign a value to the same variable.
3. No loops occur which cause the value of a variable to depend on itself.

With the relaxations of the third restriction described in later sections, this language form is ideal for describing computer logic. The proper order for execution of the assignment statements depends on the partial order implicit in them: variables which never are assigned values are input signals to the logic; variables which are only assigned values and never referenced are output signals from the logic. All other variables are internal signals. The first executable statement uses only input signals on its right side, and defines an internal variable or output signal. The process of selecting executable statements continues until all statements have been selected or a loop occurs.

The preprocessor accepts a set of assignment statements which describe the logic. These statements can be in any order. The topological sorting algorithm given by Knuth (1968, pp. 258-263) is used to output the

lines in a correct order for execution. Loops and multiple definition of variables are detected.

A line in the input language is an assignment statement which describes the action of one element (or package) of the logic. An input line includes the signals which are outputs of the package, the function of the package, and the signals which are the inputs to the package. Each line begins with a list of the output signals from the package; this list is followed by a colon. The function name follows the colon and is followed in turn by a list of the input signals to the package. The line ends with a semicolon.

Signals names must be given to all signals which flow between packages; each bit of a given named signal maps one-to-one into a wire in the physical realization of the logic. A signal name is an identifier which begins with a capital letter and is followed by seven or less capital letters or digits. (The signal name convention of the logic language was also used in section 4 for the hardware description.) The eight character limit is imposed by the use of the IBM 360 assembler which puts an eight character limit on the symbol names which it accepts.) The identifier part of the signal can optionally be followed by a bit specification. A bit specification is one, two or three integers enclosed in parentheses and separated by commas, and is required when the named signal consists of more than one bit. The bits of an N bit signal are numbered from one for the most significant to N for the least significant bit. A bit specification with a single integer specifies that bit of the signal which has that integer as its bit number. In a bit specification with two integers, the first specifies the bit number of the most significant bit of the signal and the second specifies the number

of contiguous bits in the signal. The third integer of a three integer bit specification gives the difference between successive bit numbers for the bits in the signal when that difference is not one. Table 5.1.1-1 summarizes the signal naming conventions.

Signal Name	Meaning
A	The one bit signal "A"
B(3)	Bit three of the multi-bit signal "B"
B(1,32)	Bits one through thirty-two of the multi-bit signal "B"
B(5,4)	Bits five through eight of the multi-bit signal "B"
C(1,2,4)	Bits one and five of the multi-bit signal "C"

Table 5.1.1-1 Summary of the Signal Name Conventions

The individual bits of the signals are the variables assigned by execution of the lines. The preprocessor guarantees that no bit is assigned a value more than once, and that every bit which is referenced has been assigned a value.

Many packages, such as the SN74S157 two-to-one selector, have one output signal. Others, such as the SN74S182 look ahead carry generator, have as many as five output signals. Every line which uses the same package type should have the same number of input and output signals. The preprocessor prints a function usage summary for each package type which lists any deviations in usage.

Frequently in the logic design described in section 4, there was a need for constant logic one or zero signals. The logic description language includes the variables ZERO, ZEROS, ONE and ONES as built in variables

with the constant logic values which their names suggest. It also happens that some of the output signals from a package with multiple outputs are not used. Since the preprocessor questions (but does permit) the use of a package with different numbers of output signals in different instances, the built in variable UNUSED is permitted; its use is encouraged for the sake of clarity.

The preprocessor also includes two built in functions. The OUTPUT function prints the values of the input signals written for it as the first time that all of those signal values are set in a logic simulation cycle; it appears in the place assigned to it by the partial ordering process. An OUTPUT statement names no output variables, so that it begins with a colon. The FORM statement is used to build multi-bit signals from shorter signals. One instance of its use is to build an eight bit signal composed of ZERO and ONE bits for input to the SN74S151 eight-to-one selector which supplies the EXO overflow indication signal described in section 4.2.5.1.12.4.

5.1.2 Timing by the Simulator

At run time, each named signal which occurs in the logic specification is represented by the structure shown in Figure 5.1.2-1. The signal name left justified in a blank filled eight byte field. The name is followed by a half-word integer which is used to store the time at which the signal received its value. The time for multi-bit signals which are set by the output from several different packages is the maximum of the times for all such package outputs. When knowledge of such time differences is important, multi-bit signals can be split into several different parts for more detailed timing information. The bits of a named signals are each represented by a byte; the

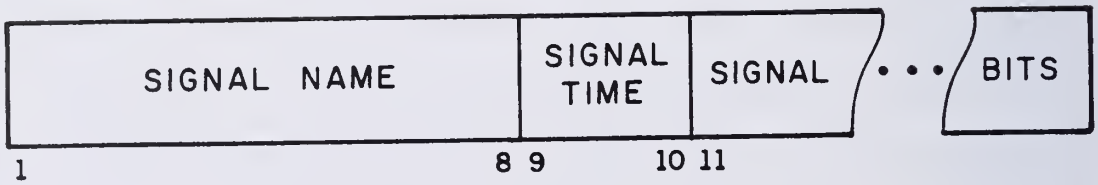


Figure 5.1.2-1 The Format of the Representation of a Signal During Simulation

string of bytes which represents the bits of the signal follows the time halfword. The execution of an OUTPUT function prints the signal name, the bit specification numbers, the signal time, and the values of the specified bits.

Each package that receives a clock pulse sets the time of that pulse. In this way, the first possible time at which the clock pulse could occur is determined.

The following discussion describes the calculation for the value assigned to the time for the output signal of an SN74S157 two-to-one selector. The discussion will clarify the nature of the output signal time calculations. As shown in Figure 5.1.2-2, the SN74S157 has four input signals and one output signal. If the strobe signal is a logic one, the output signal is always zero regardless of what the selection and A and B input signal values are. In this case, the time assigned to the output signal is that for the strobe signal plus the delay time through the package for this case given by Texas Instrument Corporation (1973). When the strobe signal is a logic zero, the value of the selection signal determines whether the package output is "A" or "B". In this case, the time assigned to the output signal is the maximum of the selection signal time plus its delay and the time of the selected input signal plus its delay. The time of the non-selected input signal is ignored.

5.1.3 Debugging Aides in the Simulation System

The simulation process for each package includes a test of each bit of the input operand. Because each bit is represented by a byte of 360 memory, it can assume more than the two states found in conventional digital logic. Input signals which are ignored by the package are not tested; thus, the simulation of an SN74S157 selector does not test the input and selection signals

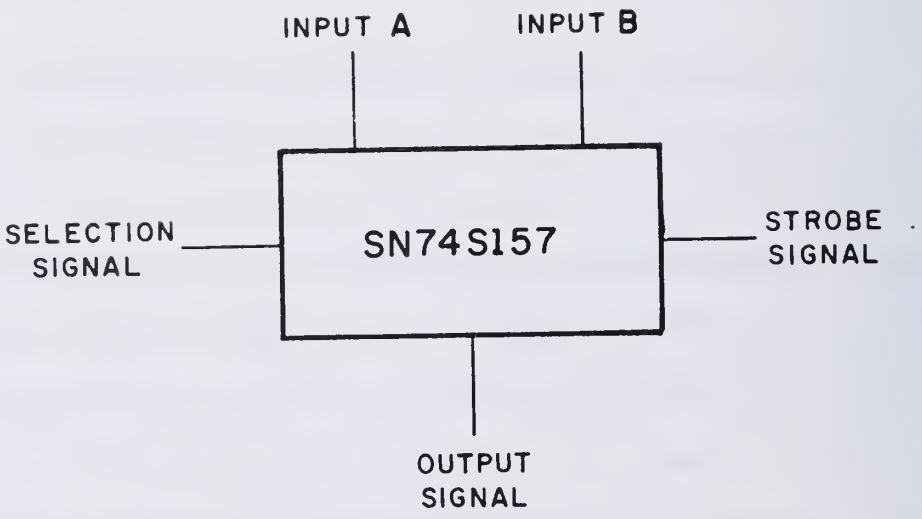


Figure 5.1.2-2 The SN74S157 Two-to-One Selector

if the strobe signal value is a logic one. It always tests the strobe bit value.

During the early debugging of the simulator, this testing process helped to identify the source of the error. The standard simulator response to an improper bit value in a tested signal is to print an error message together with the standard output for the errant signal (that is; its name, bit specification, time and bit values). Logic ones and zeros print as ones and zeros; improper bits print as dots. The simulator halts and dumps memory when an error occurs. Although the investigation was not carried to this point, the simulator could easily be altered, so that it would continue rather than halting when an improper bit value is detected. This action would help in designing fault detection programs for the logic, since it would permit easy determination of the propagation effects of an error. Moreover, it would permit identification and verification of those signals whose values, for a particular cycle, are of no consequence.

5.1.4 Simulated Packages with No Exact Hardware Analog

In the description of the left operand selection logic (section 4.2.5.1.5), the block in Figure 4.2.5.1.5-1 represented selection functions rather than hardware packages. In many cases, simulation results are not effected, but simulation time is reduced by permitting the simulation macros to perform package functions in this approximate way. Thus, the macro which simulates the SN74S157 two-to-one selector will accept input operand pairs of any bit length from one to 256, and will produce an output signal with the corresponding bit length. This deviation from exact simulation does no violence to the logic function or the logic execution time of the simulated

logic.

5.1.5 Loops

In section 5.1, we referred to relaxations of the restriction on a single assignment language which prohibits loops. In real hardware designs, loops do occur. Three different types of loops are present in the simulated floating point addition hardware, and they are discussed in the three sections which follow.

5.1.5.1 Loops and Storage Registers

The value of the zero flip-flop from a previous cycle must be used to determine the action of the normalization process (see section 4.2.5.2.1 and Figure 4.2.5.2.1-2). Another example (which was not simulated) occurs in the cases of the overflow flip-flop of Figure 4.2.5.1.12.4-1 and the underflow flip-flop of Figure 4.2.5.1.12.5-1. In both of these cases, the previous value of the flip-flop occurs as a possible input to determine its subsequent value. The loops which these cases give rise to should be broken by delaying the execution of the line which assigns a new value to the register or flip-flop until after all lines which reference the old value have been executed. Preceeding the output signal name with an asterick has precisely this effect: a line which contains an output symbol preceeded by an asterick is placed in the output program after all lines which refer to the named output signal.

5.1.5.2 Apparent but not Real Loops

The logic of the index adder, shown here again as Figure 5.1.5.2-1, appears to include a loop. The SN74S182 receives the carry generation and propagation signals; IXG(1,4) and IXP(1,4), from the four SN74S181 arithmetic-logic units, and returns the three carry signals, IXC4, IXC8, and IXC12, to

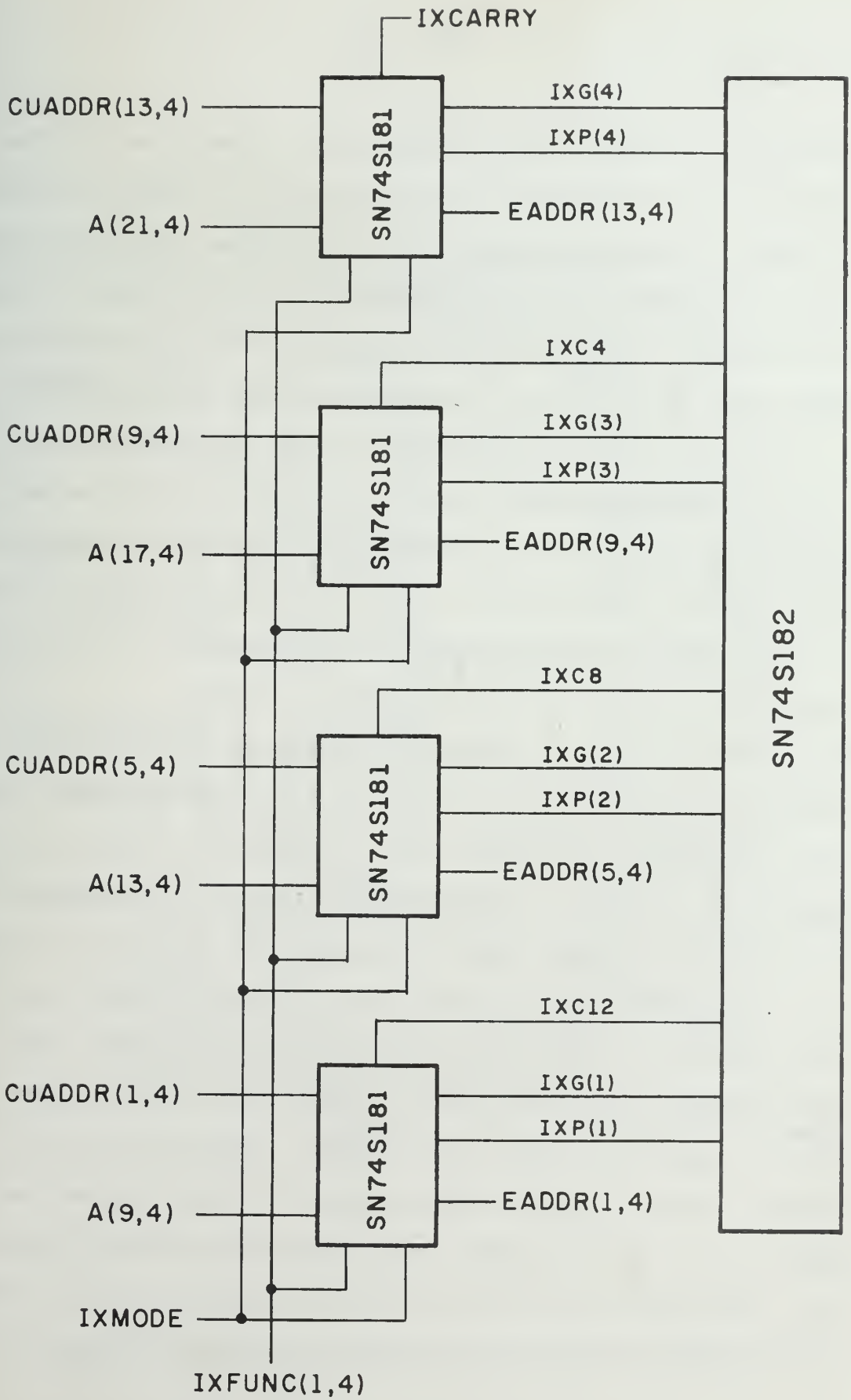


Figure 5.1.5.2-1 The Index Adder Logic

three of the SN74S181's. On closer examination, however, we find that the functions of the SN74S181 can be partitioned into two separate operations. The generate and propagate signals depend only on the values of the inputs A(9,16) and CUADDR(1,16) and are independent of the carry inputs IXCARRY, IXC8, and IXC12. The sum EADDR(1,16) depends on the input operands and the carries. The apparent loop is broken in the simulator by implementing the two separate functions of the SN74S181 (and also the SN74S381) as two separate pseudo-packages as shown in Figure 5.1.5.2-2. The S181GP package uses the input operands A(9,15) and CUADDR(1,16) to produce the generate and propagate signals for the S182. The carries from the S182 package are used by the S181 package, together with the input operand values, to produce the required sum.

Figures 5.1.5.2-3 through 5.1.5.2-8 are the computer output for the simulation of the index adder. Figure 5.1.5.2-3 shows the SYSPRINT file which lists the logic description which was input, and summarizes the functions used in logic and the signals which are inputs to the logic and outputs from the logic. The first seventy-two characters of each input line are processed by the logic simulator. Card input is assumed, and the last eight columns of each card can be used for card sequence information. The entire eighty columns of each input card are listed, and the function summary lists the card number of the function card printed. If a function is used with different numbers of input or output signals, all cards for that function are printed in the function summary. This situation may or may not represent an error, and the user can proceed to assemble and execute a simulator with this sort of input. The response is completely determined by his macros which

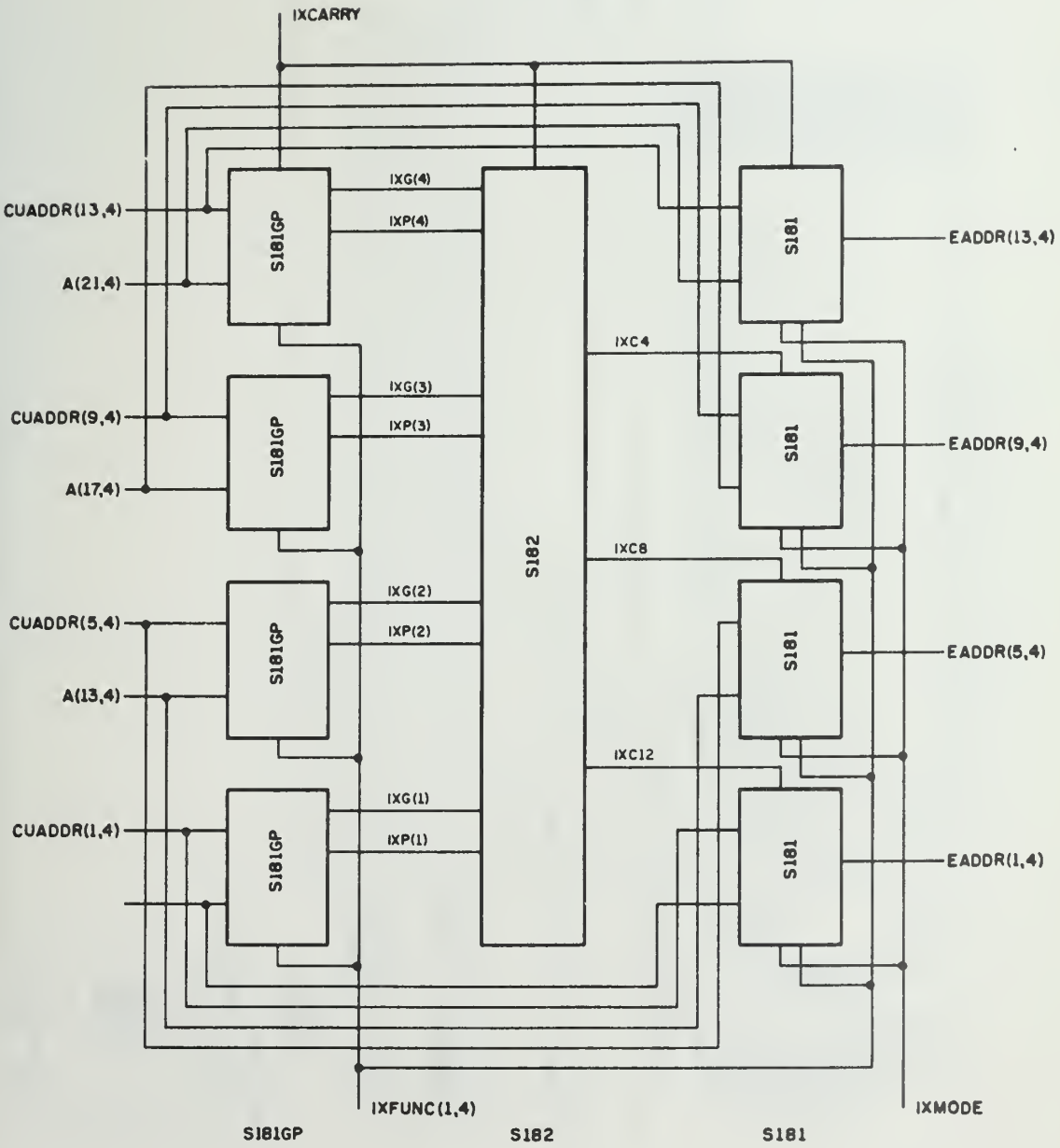


Figure 5.1.5.2-2 The Apparent Loops in the Index Adder Logic Removed

```
: OUTPUT CUADDR(1,16) A(17,16) IXFUNC(1,4) IXMODE IXCARRY ;                02/00100
: OUTPUT IXG(1,4) IXP(1,4) ;                                              02/00200
: OUTPUT IXC4 IXC8 IXC12 ;                                                02/00300
IXG(1) IXP(1) : S181GP CUADDR(1,4) A(17,4) IXFUNC(1,4) ;                  02/00400
IXG(2) IXP(2) : S181GP CUADDR(5,4) A(21,4) IXFUNC(1,4) ;                  02/00500
IXG(3) IXP(3) : S181GP CUADDR(9,4) A(25,4) IXFUNC(1,4) ;                  02/00600
IXG(4) IXP(4) : S181GP CUADDR(13,4) A(29,4) IXFUNC(1,4) ;                 02/00700
UNUSED UNUSED IXC4 IXC8 IXC12 : S182 IXCARRY IXG(1,4) IXP(1,4) ;          02/00800
EADDR(1,4) CARRY : S181 CUADDR(1,4) A(17,4) IXC12 IXFUNC(1,4) IXMODE ;    02/00900
EADDR(5,4) UNUSED : S181 CUADDR(5,4) A(21,4) IXC8 IXFUNC(1,4) IXMODE ;    02/01000
EADDR(9,4) UNUSED : S181 CUADDR(9,4) A(25,4) IXC4 IXFUNC(1,4) IXMODE ;    02/01100
EADDR(13,4) UNUSED : S181 CUADDR(13,4) A(29,4) IXCARRY IXFUNC(1,4)        02/01200
     IXMODE ;                                                             02/01300
: OUTPUT EADDR(1,16) CARRY ;                                              02/01400
```

```
UNDEFINED SIGNALS :


OUTPUT SIGNALS :

   CARRY
   EADDR


INPUT SIGNALS :

   IXCARRY
   A
   IXFUNC
   IXMODE
   CUADDR


FUNCTIONS :

    UNUSED UNUSED IXC4 IXC8 IXC12 : S182 IXCARRY IXG(1,4) IXP(1,4) ;

    EADDR(13,4) UNUSED : S181 CUADDR(13,4) A(29,4) IXCARRY IXFUNC(1,4)  IXMODE ;

    IXG(4) IXP(4) : S181GP CUADDR(13,4) A(29,4) IXFUNC(1,4) ;
```

Figure 5.1.5.2-3 The SYSPRINT Output of the Logic Simulator

```
 DATA
CUADDR INPUT 16
IXC4 SIGNAL 01
IXC8 SIGNAL 01
IXC12 SIGNAL 01
IXMODE INPUT 01
IXFUNC INPUT 04
EADDR OUT 16
IXG SIGNAL 04
CARRY OUT 01
IXP SIGNAL 04
A INPUT 32
IXCARRY INPUT 01
 PROGRAM
 S181 (EADDR,13,04),(UNUSED),(CUADDR,13,04),(A,29,04),(IXCARRY),(IXFUNC*
                ,01,04),(IXMODE)
 S181GP (IXG,04),(IXP,04),(CUADDR,13,04),(A,29,04),(IXFUNC,01,04)
 S181GP (IXG,03),(IXP,03),(CUADDR,09,04),(A,25,04),(IXFUNC,01,04)
 S181GP (IXG,02),(IXP,02),(CUADDR,05,04),(A,21,04),(IXFUNC,01,04)
 S181GP (IXG,01),(IXP,01),(CUADDR,01,04),(A,17,04),(IXFUNC,01,04)
 OUTPUT ,(CUADDR,01,16),(A,17,16),(IXFUNC,01,04),(IXMODE),(IXCARRY)
 S182 (UNUSED),(UNUSED),(IXC4),(IXC8),(IXC12),(IXCARRY),(IXG,01,04),(IX*
                P,01,04)
 OUTPUT ,(IXG,01,04),(IXP,01,04)
 S181 (EADDR,01,04),(CARRY),(CUADDR,01,04),(A,17,04),(IXC12),(IXFUNC,01*
                ,04),(IXMODE)
 S181 (EADDR,05,04),(UNUSED),(CUADDR,05,04),(A,21,04),(IXC8),(IXFUNC,01*
                ,04),(IXMODE)
 S181 (EADDR,09,04),(UNUSED),(CUADDR,09,04),(A,25,04),(IXC4),(IXFUNC,01*
                ,04),(IXMODE)
 OUTPUT ,(IXC4),(IXC8),(IXC12)
 OUTPUT ,(EADDR,01,16),(CARRY)
 FINIS
 END PROGRAM
```

Figure 5.1.5.2-4 The Assembler Program Output from the Logic Simulator which is Written in the File DECK

```
 MACRO
&G STEP &CUADDR=,&IXMODE=,&IXFUNC=,&A=,&IXCARRY=
&G FIELD (IXCARRY,1,01),&IXCARRY
&G FIELD (A,1,32),&A
&G FIELD (IXFUNC,1,04),&IXFUNC
&G FIELD (IXMODE,1,01),&IXMODE
&G FIELD (CUADDR,1,16),&CUADDR
 STEPEND
 MEND
 EJECT
MICRO BEGIN 02,CARRY,EADDR,IXCARRY,A,IXFUNC,IXMODE,CUADDR
```


Figure 5.1.5.2-5 The STEP Macro Output of the Logic Simulator which is Written in the File MICRO

```
STEP    IXCARRY=0,IXFUNC=1010,IXMODE=1,CUADDR=X0,A=X1234          05/00100
STEP    IXFUNC=1111,CUADDR=X22                                    05/00200
STEP    IXFUNC=0000,IXMODE=0                                      05/00300
STEP    IXCARRY=1                                                 05/00400
STEP    IXFUNC=1001,IXCARRY=1                                     05/00500
STEP    IXCARRY=0                                                 05/00600
STEP    IXFUNC=0110                                               05/00700
STEP    IXFUNC=1111,IXCARRY=1                                     05/00800
STOP                                                              05/00900
END                                                               05/01000
```


Figure 5.1.5.2-6 The STEPs Written to Drive the Index Adder Simulation

```
BEGIN MICRO-STEP EXECUTION
CUADDR    01  16  00034  0000000000100010
A         17  16  00034  0001001000110100
IXFUNC    01  04  00034  1111
IXMODE    01  01  00034  1
IXCARRY   01  01  00034  0
IXG       01  04  00049  1111
IXP       01  04  00049  0011
IXC4      01  01  00056  0
IXC8      01  01  00056  0
IXC12     01  01  00056  0
EADDR     01  16  00068  0000000000100010
CARRY     01  01  00067  0

BEGIN MICRO-STEP EXECUTION
CUADDR    01  16  00068  0000000000100010
A         17  16  00068  0001001000110100
IXFUNC    01  04  00068  0000
IXMODE    01  01  00068  0
IXCARRY   01  01  00068  0
IXG       01  04  00083  0000
IXP       01  04  00083  0000
IXC4      01  01  00090  1
IXC8      01  01  00090  1
IXC12     01  01  00090  1
EADDR     01  16  00102  0000000000100011
CARRY     01  01  00101  1

BEGIN MICRO-STEP EXECUTION
CUADDR    01  16  00102  0000000000100010
A         17  16  00102  0001001000110100
IXFUNC    01  04  00102  0000
IXMODE    01  01  00102  0
IXCARRY   01  01  00102  1
IXG       01  04  00117  0000
IXP       01  04  00117  0000
IXC4      01  01  00124  1
IXC8      01  01  00124  1
IXC12     01  01  00124  1
EADDR     01  16  00136  0000000000100010
CARRY     01  01  00135  1

BEGIN MICRO-STEP EXECUTION
CUADDR    01  16  00136  0000000000100010
A         17  16  00136  0001001000110100
IXFUNC    01  04  00136  1001
IXMODE    01  01  00136  0
IXCARRY   01  01  00136  1
IXG       01  04  00151  0000
IXP       01  04  00151  0010
IXC4      01  01  00158  1
IXC8      01  01  00158  1
IXC12     01  01  00158  1
EADDR     01  16  00170  0001001001010110
CARRY     01  01  00169  1
```

Figure 5.1.5.2-7 The First Half of the Output from the Simulation of the Index Adder

```
BEGIN MICRO-STEP EXECUTION
CUADDR    01   16   00170   0000000000100010
A         17   16   00170   0001001000110100
IXFUNC    01   04   00170   1001
IXMODE    01   01   00170   0
IXCARRY   01   01   00170   0
IXG       01   04   00185   0000
IXP       01   04   00185   0010
IXC4      01   01   00192   1
IXC8      01   01   00192   1
IXC12     01   01   00192   1
EADDR     01   16   00204   0001001001010111
CARRY     01   01   00203   1


BEGIN MICRO-STEP EXECUTION
CUADDR    01   16   00204   0000000000100010
A         17   16   00204   0001001000110100
IXFUNC    01   04   00204   0110
IXMODE    01   01   00204   0
IXCARRY   01   01   00204   0
IXG       01   04   00219   0000
IXP       01   04   00219   0001
IXC4      01   01   00226   1
IXC8      01   01   00226   1
IXC12     01   01   00226   1
EADDR     01   16   00238   1110110111101110
CARRY     01   01   00237   1


BEGIN MICRO-STEP EXECUTION
CUADDR    01   16   00238   0000000000100010
A         17   16   00238   0001001000110100
IXFUNC    01   04   00238   1111
IXMODE    01   01   00238   0
IXCARRY   01   01   00238   1
IXG       01   04   00253   1111
IXP       01   04   00253   0011
IXC4      01   01   00260   0
IXC8      01   01   00260   0
IXC12     01   01   00260   0
EADDR     01   16   00272   0000000000100001
CARRY     01   01   00271   0


BEGIN MICRO-STEP EXECUTION
END OF SIMULATION
```

Figure 5.1.5.2-8 The Last Half of the Output from the Simulation of the Index Adder

define the package operations. Macros which accept a variable number of inputs can be written and used where desired. Figure 5.1.5.2-4 shows the assembly program written to simulate the index adder in response to the input shown in Figure 5.1.5.2-3. Figure 5.1.5.2-5 shows the STEP macro written to facilitate control of the logic. Figure 5.1.5.2-6 shows a list of input STEPS which produced the simulator output shown in Figure 5.1.5.2-7 and 5.1.5.2-8.

In the appendix, the complete control card and input set up for the simulation of the floating point addition and subtraction for the processor is given. As shown in the listing, the OUTPUT built in function will accept an integer value in the output field. This value can be used together with an integer PARM to supress output. Only output with an output number less than the PARM number is printed during simulation.

5.1.5.3 Sequential Logic: Real Loops

An alternative design for the exponent adder, shown in Figure 5.1.5.3-1 includes the feedback characteristic of sequential logic. This design was not used as the eventual exponent adder described in section 4.2.5.1.3 because it is significantly slower than the adder described in that section, and the exponent adder stands directly in the center of a time-critical path in the logic. This slower form performs a one's complement subtraction; feedback of the high order carry is required to compute a correct result. The absolute value of the difference is produced by SN74S86 exclusive OR gates which complement the one's complement result when it is negative and pass it through in true form when the difference is positive or zero. The logic was correctly simulated; the technique used is shown in Figure 5.1.5.3-2. In this particular case, even when the so-called end around carry of the one's complement is

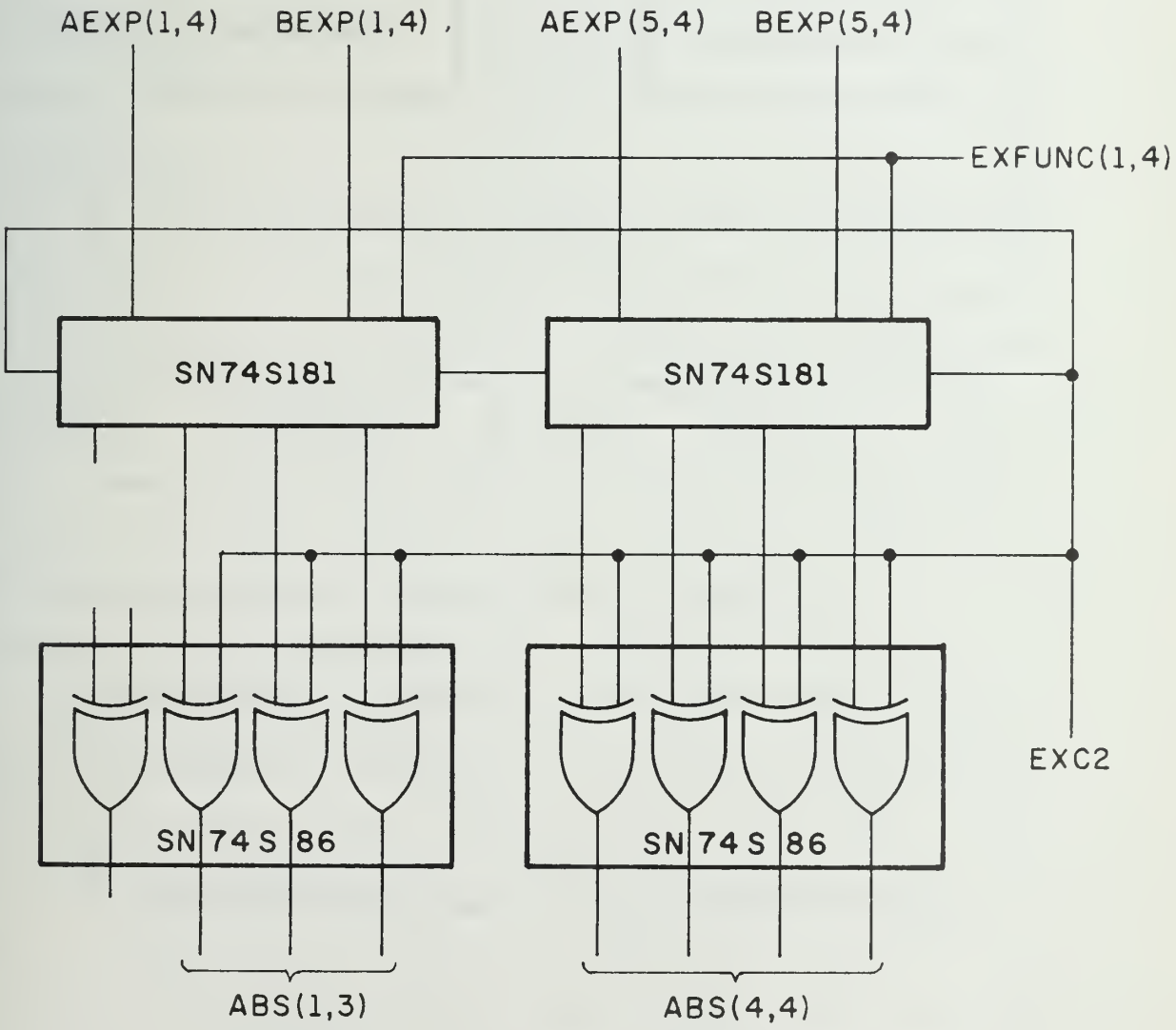


Figure 5.1.5.3-1 The Sequential Circuit for the Exponent Adder

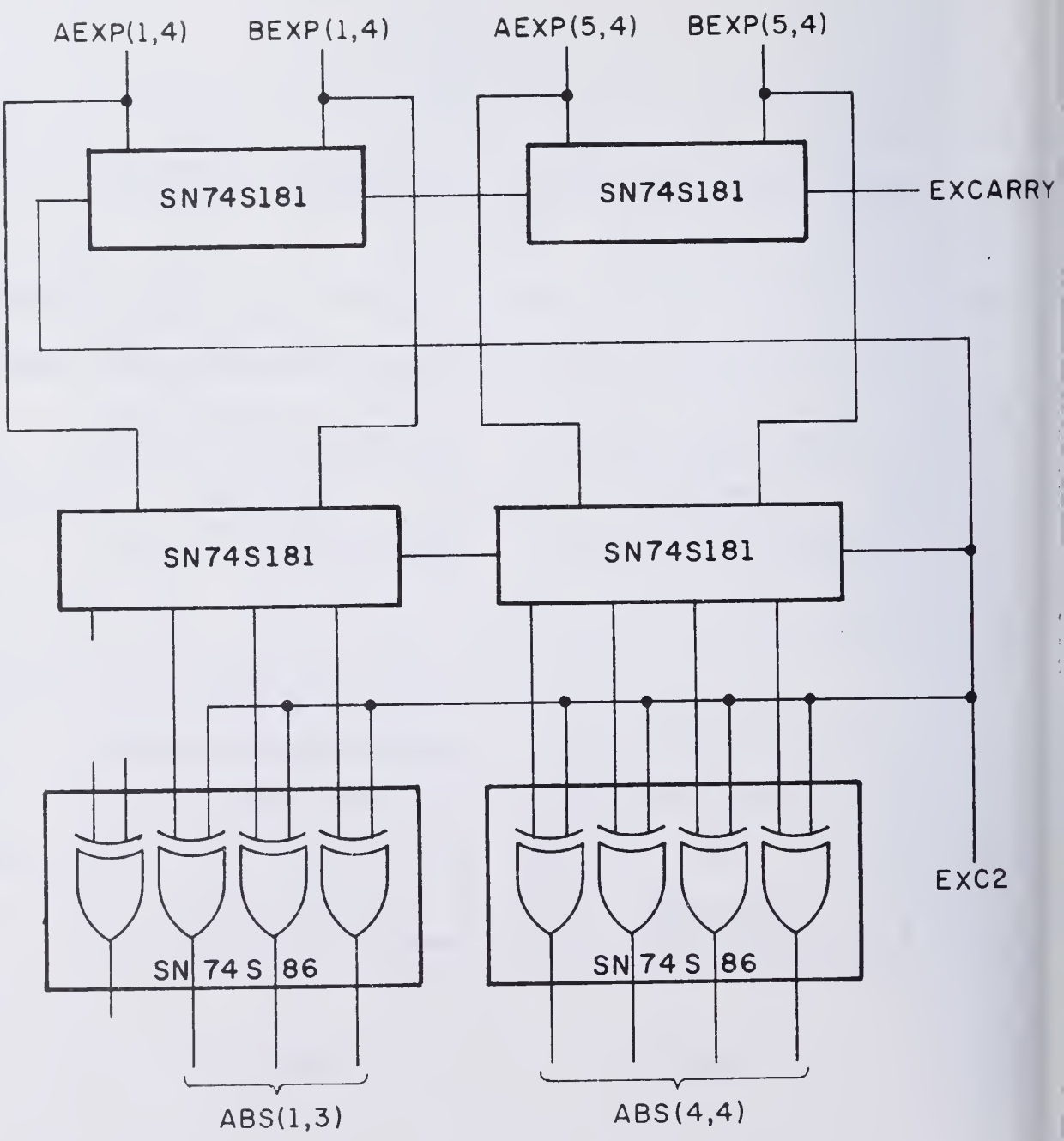


Figure 5.1.5.3-2 The Sequential Circuit for the Exponent Adder as it was Unwound for Simulation

a one, addition of that carry to the difference will not alter the carry. The ones complement negative zero is complemented by the SN74S86 exclusive OR gates. Hence, one pass around the loop always produces the correct result. The simulation unwinds the loop and expresses it as shown in the figure.

5.1.6 Wiring Lists

An original goal of the logic simulation system was the production of wiring lists from the logic description for the debugged logic. Work toward this goal was not performed, and the techniques used to avoid loops described in section 5.1.5.2 and 5.1.5.3 make the production of wire lists more difficult. The use of packages for arbitrary length operands, described in section 5.1.4, adds to the problem of wire list production. The technique of section 5.1.4 is a convenience used to reduce the length of the logic description and speed up the simulation execution. The loop avoidance techniques, on the other hand, are necessary deviations from an exact line to package one-to-one correspondence. Another obstacle in the way of wire list production is the use of implicit input signals, such as constant logic one inputs to AND gates which, in physical form, have more input than the particular use requires. In the simulation of the floating point addition and subtraction hardware which was performed, several packages which have strobe input signals like that of the SN74S157 were simulated without providing for this input. The assumption implicit in this practice is that the missing strobe signal is always to be connected, in the actual hardware, to a logic zero.

All of the cases which appear to cause trouble can be treated in a simple way except the sequential circuit case. Implicit input signals and non-standard signal lengths can be easily accounted for. The correct associ-

ation of the S181GP and S181 pseudo-package can easily be made on the basis of the common signals which both share. In the sequential circuit case, however, different signals names are required by the very nature of the feedback situation to break the loop brought on by that feedback situation. The author sought but was unable to find a technique like that of the asterick notation for register values for such signals.

5.2 The Multiplier Prototype

The great bulk of the multiplier design described here was done by William Stenzel and will be described in detail in his master's thesis (Stenzel, 1975).

The facilities of the Computer Science Department shop limited us to two-sided boards with maximum dimensions of fifteen inches by eighteen inches. In practice, these are not confining limits, since we had decided to use two-sided boards throughout the design, and a fifteen by eighteen inch board is about as large as one can practically use. The multiplier logic contains ninety integrated circuits which require a complicated data interconnection pattern. With the help of the etched power and ground buss structure suggested by Mr. Frank Serio, we were able to design and build a one board multiplier prototype. Power and ground distribution, often the third and fourth layers of a multi-layer board, were provided by etched distribution systems. A diagram of the scheme is shown in Figure 5.2-1, and Figures 5.2-2 and 5.2-3 show the artwork for the power and ground systems, respectively. The thin strips of the buss systems run between the rows of pins of the dual-in-line circuit packages of the logic. Pins at the appropriate points connect the integrated circuits to the power and ground distribution system. The

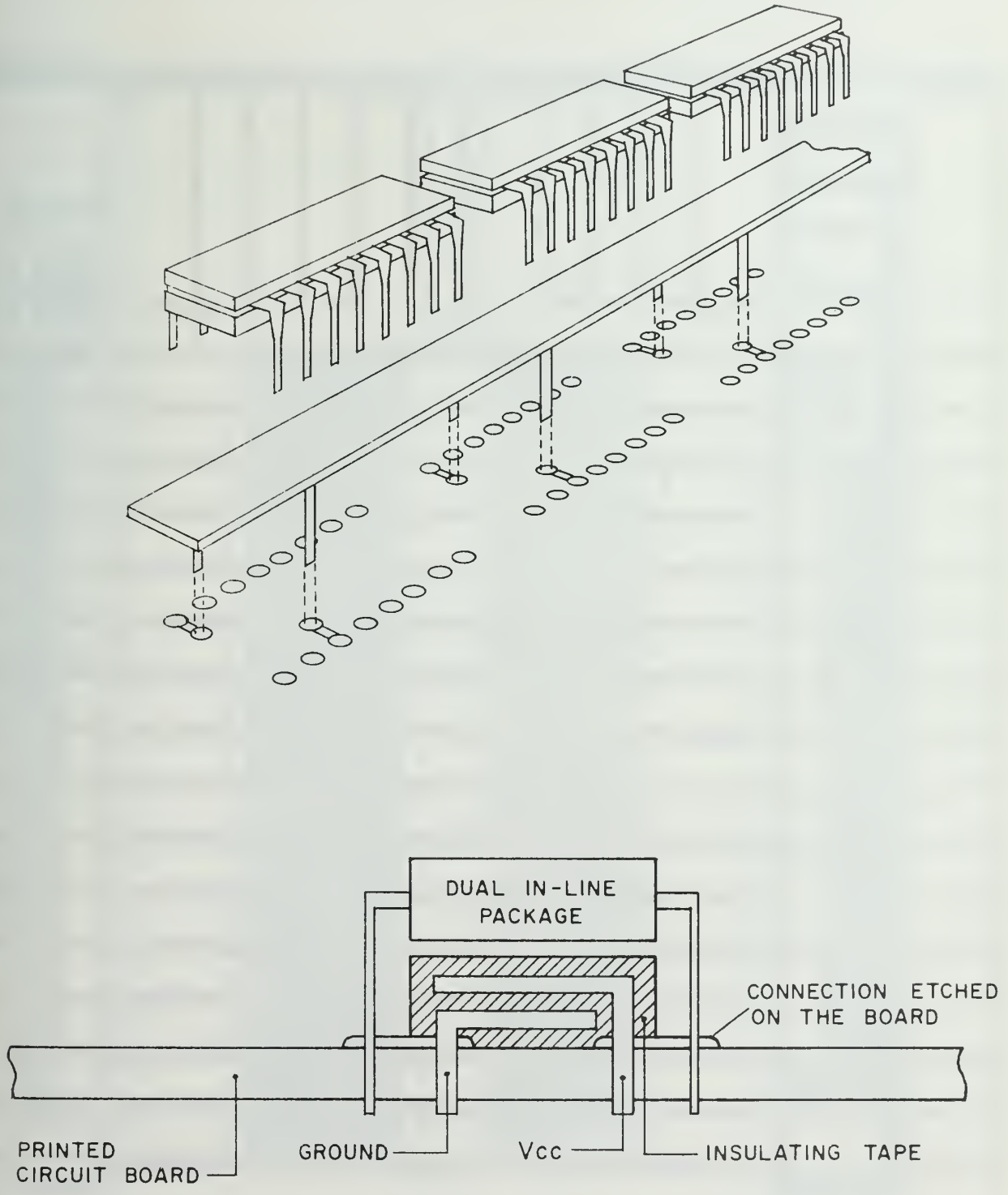


Figure 5.2-1 Details of the Power and Ground Bussing System

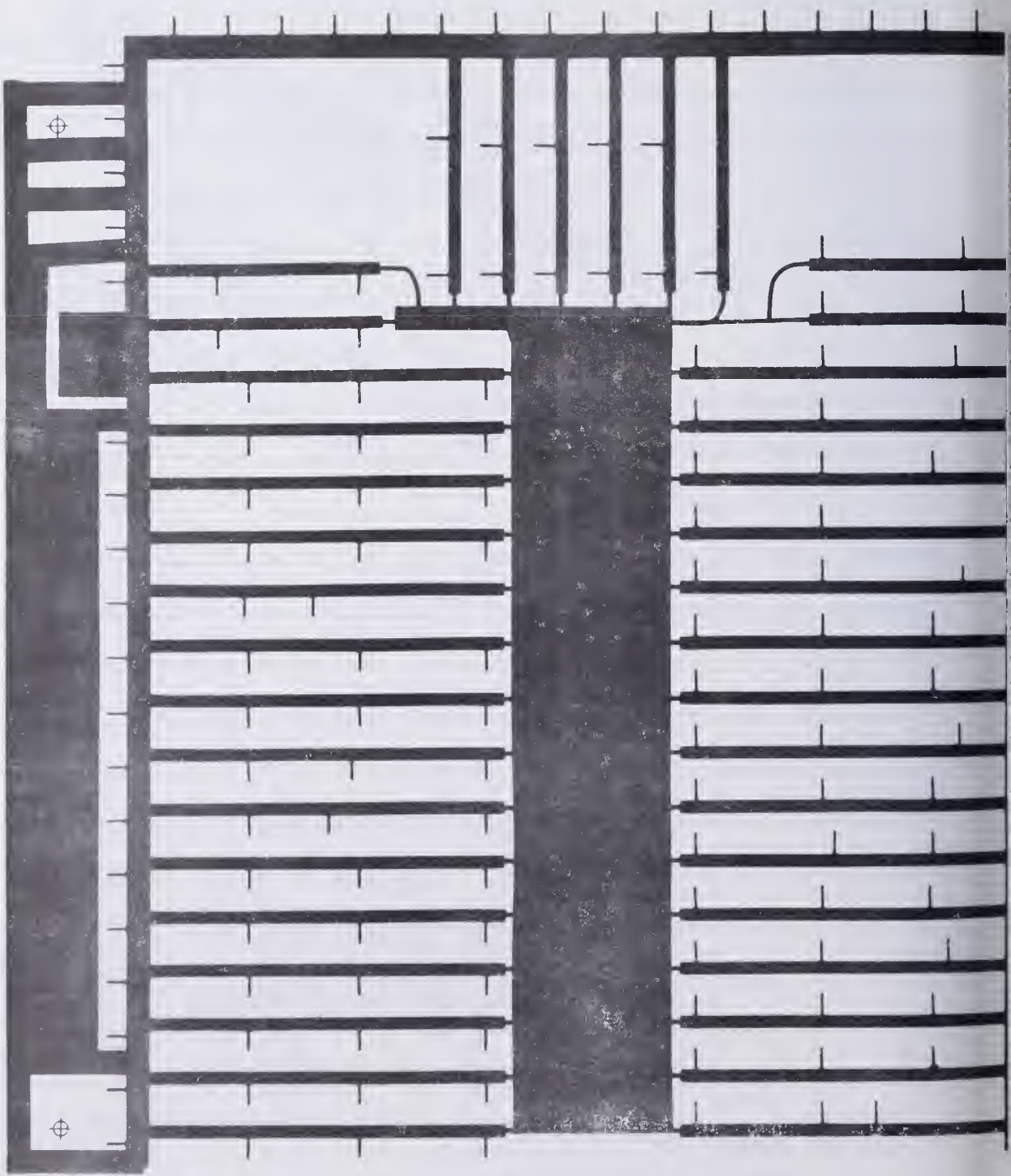


Figure 5.2-2 Power Distribution Artwork

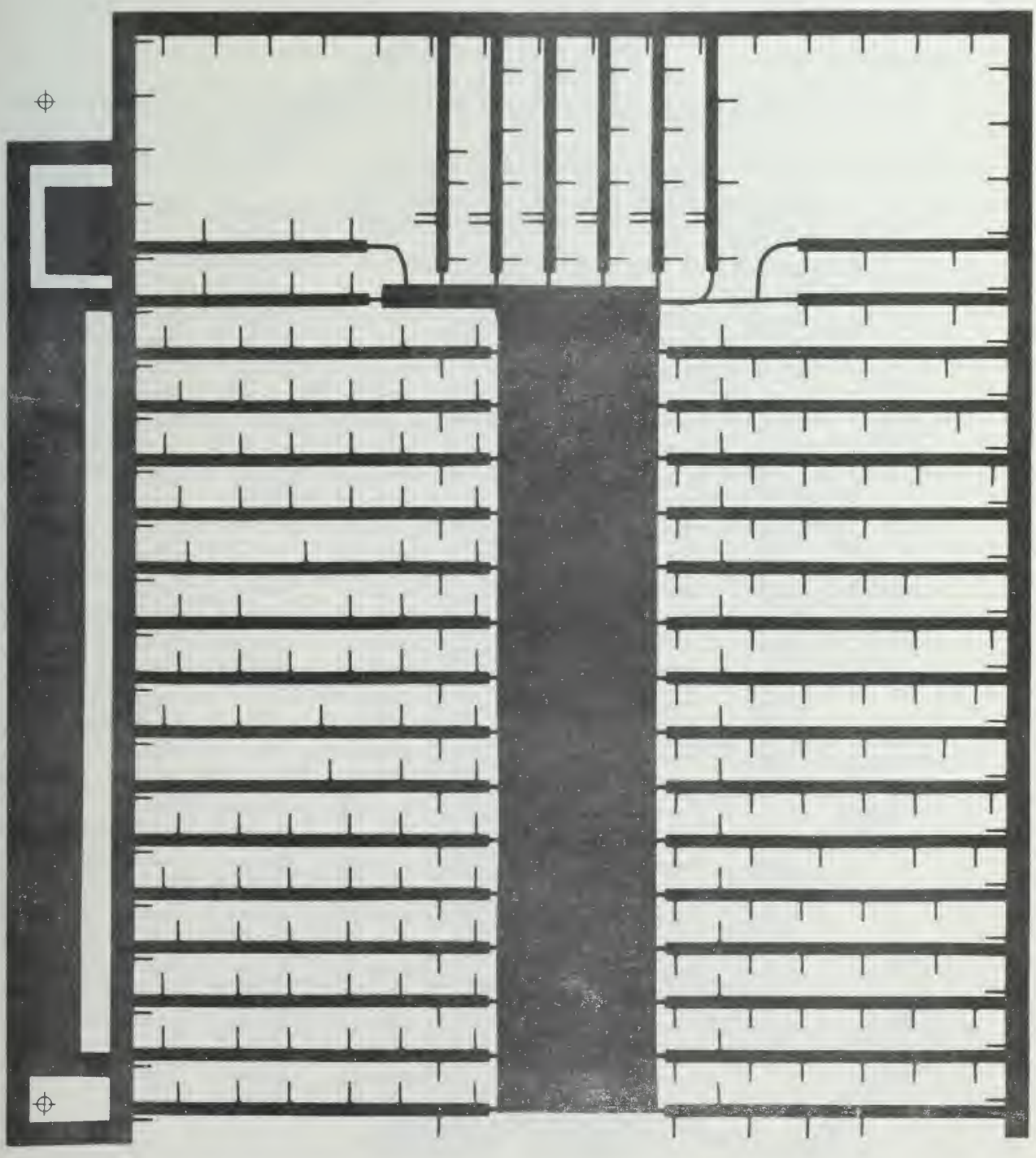


Figure 5.2-3 Ground Distribution System Artwork

etched circuits of the system are insulated and attached to the board by insulating tape.

After several interations, Ms. Stenzel decided on a board layout which places the integrated circuit components in a horseshoe arrangement at the periphery of the board with the input lines running up the center of the component side of the board and the output signals running down its outside edges. The component and solder sides of the resulting board are shown in Figure 5.2-4 through Figure 5.2-7.

The sum of the maximum operating times of the integrated circuits in the multiplier logic is 264 nanoseconds, and the sum of the typical operating times is 189 nanoseconds. Several stages of testing and refining the ground transmission by the cabling have shown that the multiplier will operate reliably at cycle times as low as 200 nanoseconds. The original cables which provided the input to the board and received its output were twenty-six conductor flexible ribbon-type cables. Twenty-four conductors of each of four cables were used to transmit the twenty-four bits of each of the two input operands and the forty-eight product bits. To obtain satisfactory time and noise performance from these cables, we found it necessary to shield each of them with copper tape ground planes. Therefore, we feel that the eventual system should use nothing less than cabling which will transmit interleaved ground and signal pairs between boards.

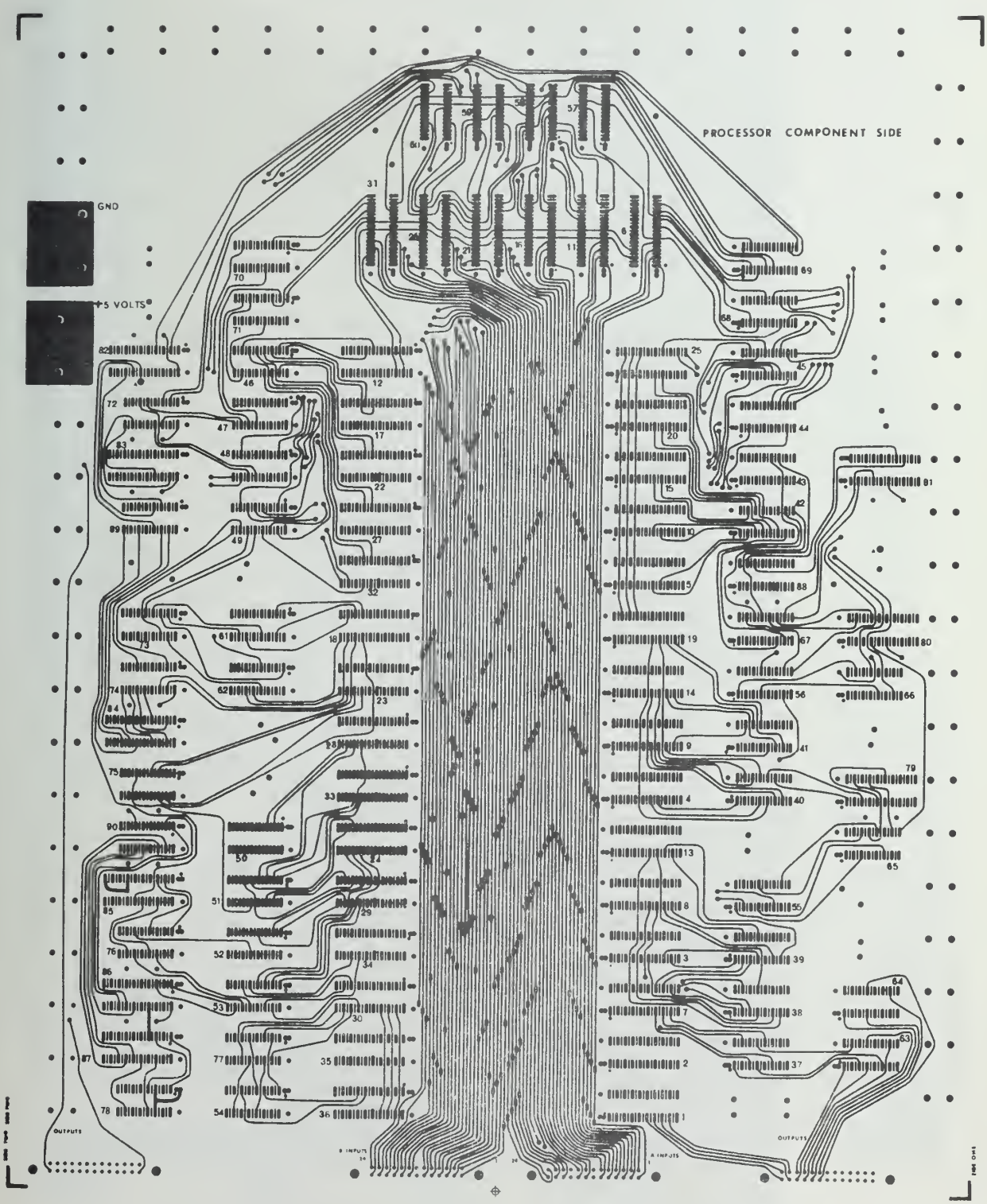


Figure 5.2-4 Multiplier Prototype Board; Component Side

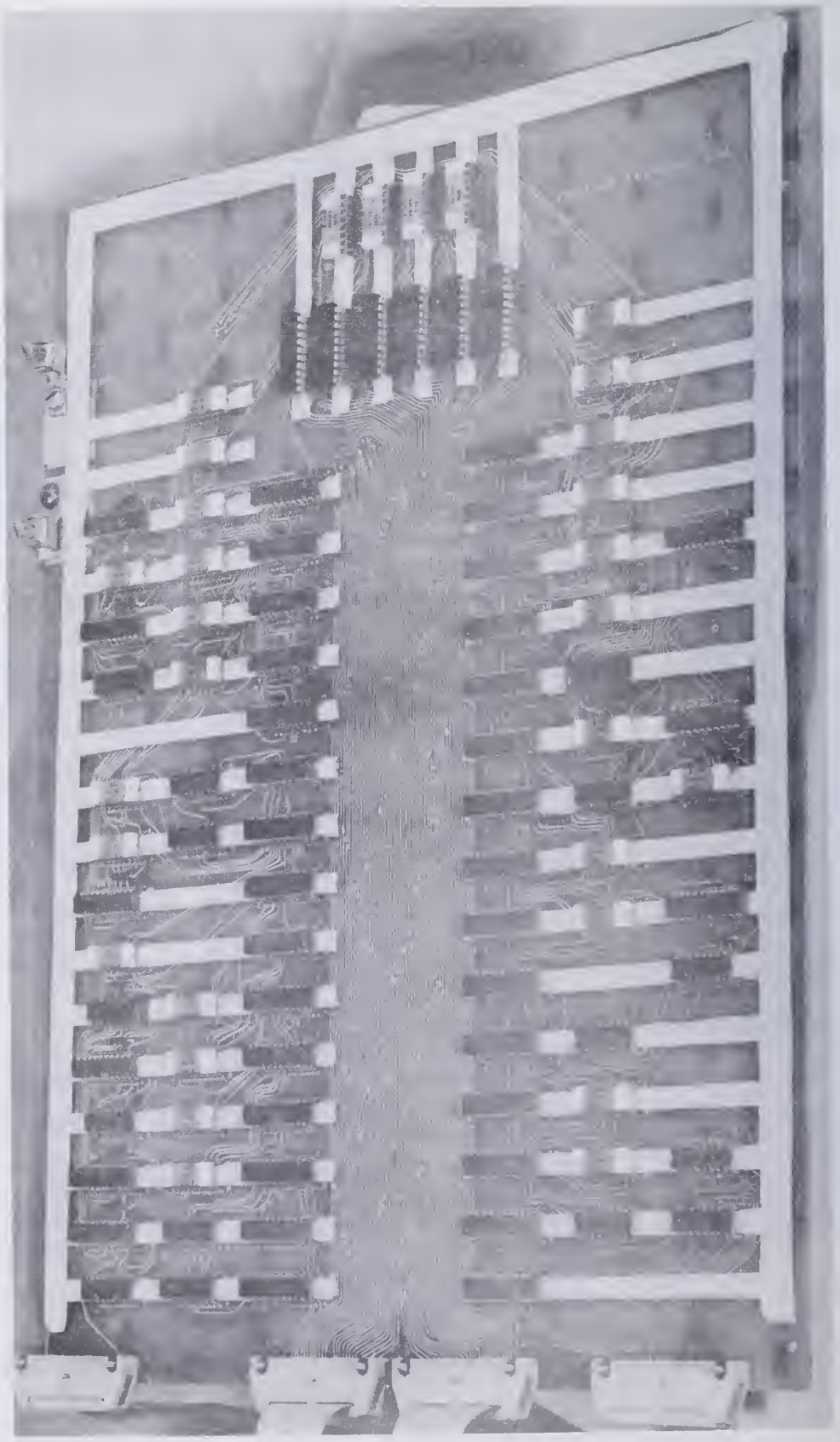


Figure 5.2-5 Photograph of the Component Side of the Multiplier Board

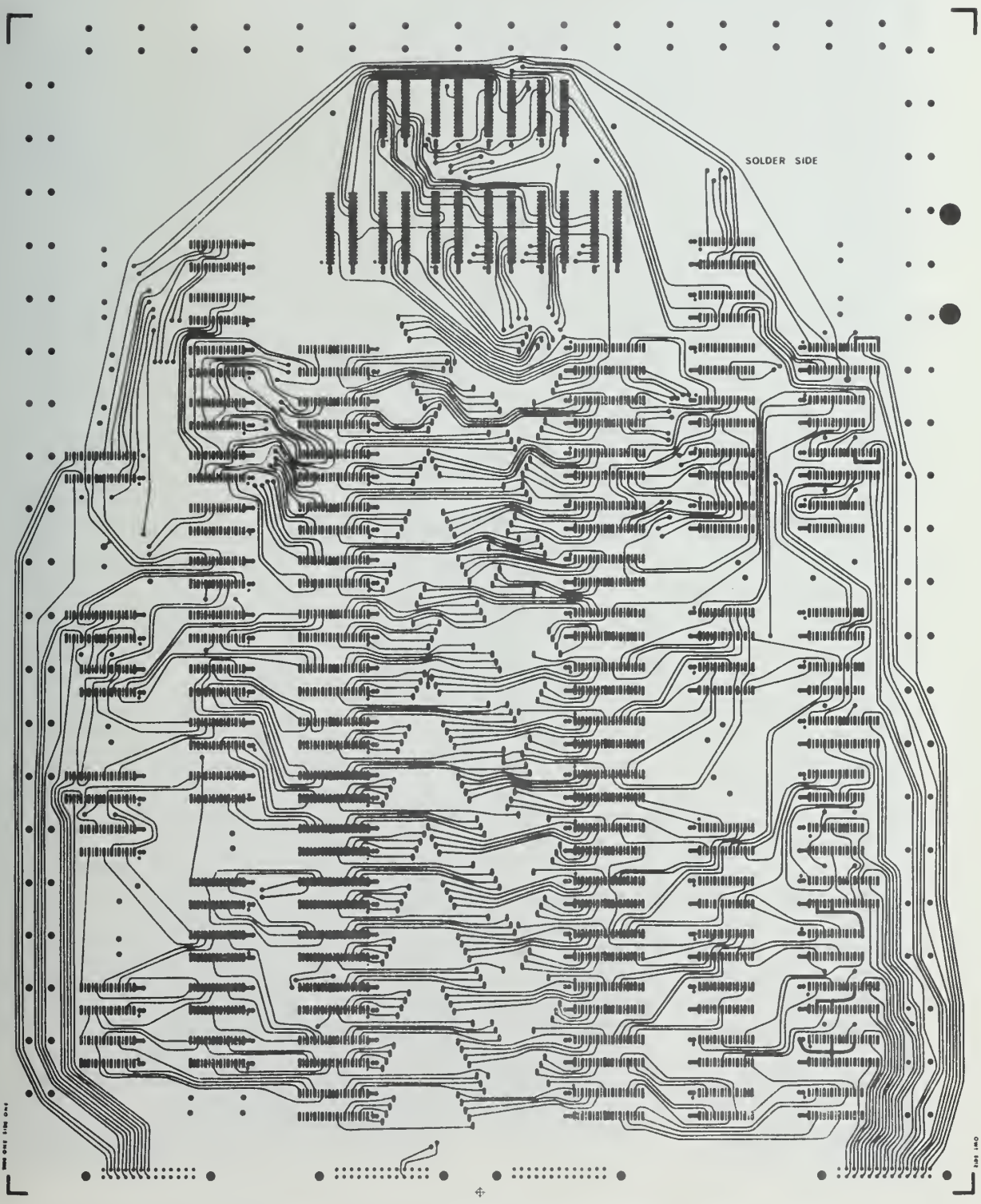


Figure 5.2-6 Multiplier Prototype Board, Solder Side

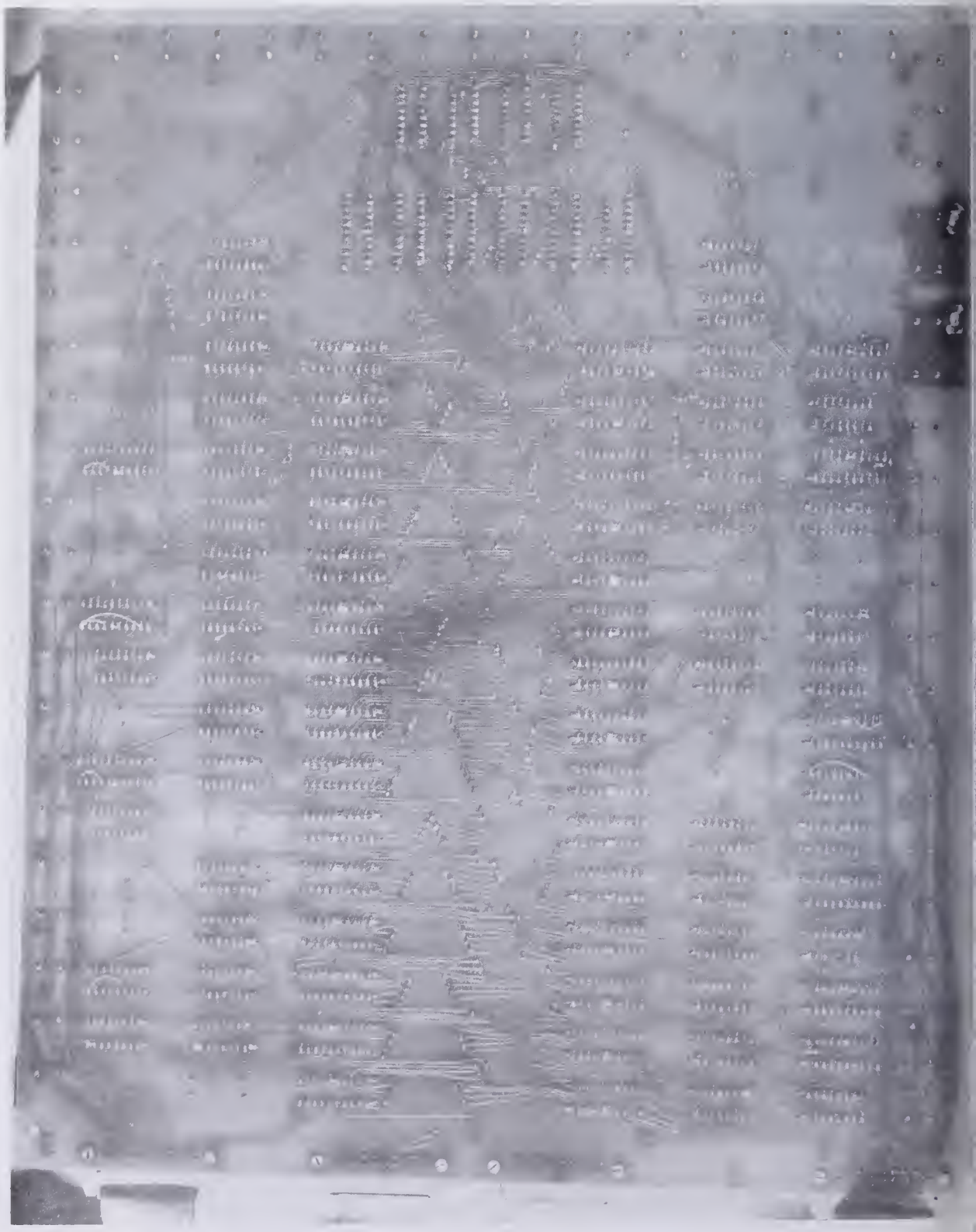


Figure 5.2-7 Photograph of the Solder Side of the Multiplier Board

6. System Performance

This group of sections will evaluate several aspects of the performance of the machine. In the first section, we will discuss the execution on time for operation cycles of the processors with information derived from the logic simulation work. The other sections will evaluate the effectiveness of the design for the weather model, matrix inversion, image data processing and information retrieval.

6.1 Processor and Routing Unit Cycle Times

The simulator indicated that the time for a floating point addition or subtraction was 256 nanoseconds. Two selector stages and the operand registers, all of which are in the operation cycle for the complete processor, were not included in the simulation. Inclusion of these elements would increase the time measured by the simulator to 336 nanoseconds. This figure represents the sum of the maximum propagation time through the logic elements. As the experience with the multiplier has shown, it is not unreasonable to expect this time to be achievable. On this basis, we estimate that a reasonable operation cycle time for the processor logic is 350 nanoseconds. The logic description of the processor given in section 4 did not include any extra logic to reduce the cycle time for frequently occuring special cases. Replacing the fraction selection logic of Figure 4.2.5.1.7-2 with that shown in Figure 6.1-1 removes the adder and the left and right operand selection gates from the path taken by normalization and multiplication results. The simpler but slower design assumes the use of one constant clock frequency to control the operation cycle of the processor. Adding extra paths implies the need for different operation cycle times, so that more complicated clocking

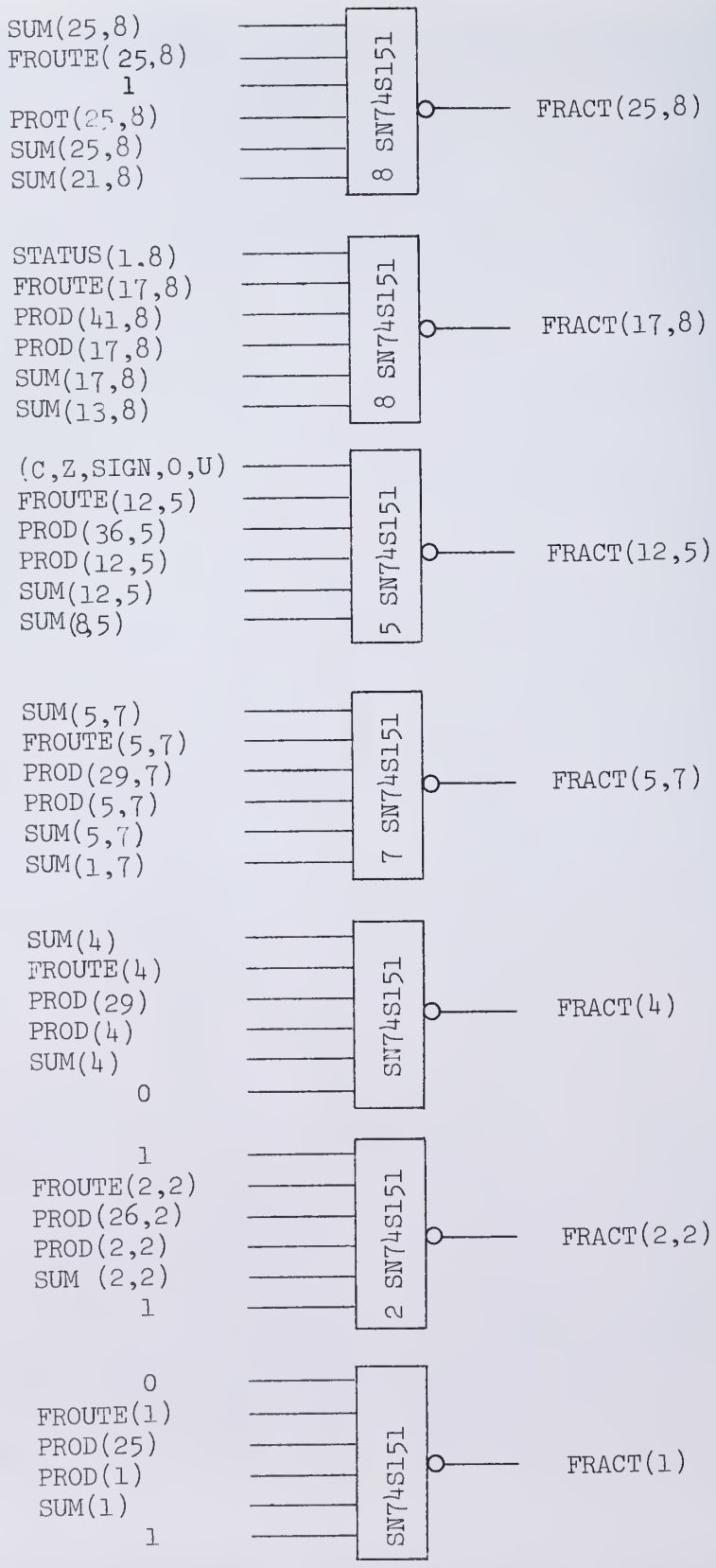


Figure 6.1-1 An Alternative to the Fraction Selection Logic

logic would be required. The increased complication occurs only in the control unit, however, not at the processor level. A complete analysis beyond that permitted by the information we now have about the system is required to decide how cost effective such enchancements would be.

Few of the arithmetic operations which the model will actually use can be performed in only one processing cycle. All normalized results require at least two cycles. A normalized multiplication will probably require three cycles unless a logic enchancement like that mentioned in the previous paragraph is used. On the other hand, the compare, normalize, integerize and all of the move operations will take only one cycle.

Work which was not completed was to have experimented with prototype routing hardware. The results of this work would have provided a basis for estimating the operation time of the routine network. The principle unknown factor in this part of the design is the time required to send the signals through the cables connecting the switches in the routing network. In section 4.3.3, we estimated the times for the routing unit by assuming cable transmission times of fifty nanoseconds. The estimate given there for the operation time of a pipelined unit with eight bit paths was 542 nanoseconds. This estimate will have to stand, since we have no information about the actual behavior of a prototype for this logic.

6.2 Performance of the System on the General Circulation Model

There is no subroutine of the general circulation model which is small enough to serve as a reasonable test case for timing estimates. The only parts of the model for which 360/95 times are available are the large COMP1-COMP2, COMP3, and the radiation subroutines. The subroutines COMP1 and

which form the core of the model, exist as two separate subroutines only because the logical unit which they form is too large for complication by the IBM FORTRAN H compliler (Karn, 1974). Evidence for the applicability of the array computer architecture is found, however, in the results of the effort by GISS to run their model on the ILLIAC IV, (Karn, 1975) which are presented in Table 6.2-1. The table shows the ratio of ILLIAC IV to 360/75 processing times for three parts of the model. During the time these figures were measured, the extensive facilities of the ILLIAC IV control unit, which are intended to speed instruction decoding and overlap the execution of parts of array instructions, were disabled; this accounts for the relatively low ratio. With all of the features of the control unit operational, these ratios should all increase by a factor of three. The poor performance of ILLIAC IV on the radiation routine is a direct result of the fact that the 3000 word table which is used by this routine had to be distributed across the memories of all sixty-four processing unit memories in the array. As a consequence, table access by a processor to a particular table value was very time consuming. This very result prompted the inclusion of the table look up facilities in the current design. The last line of the table gives the performance figures for a new radiation algorithm designed for use on parallel machines. It uses more computation and less table space, so that - on ILLIAC IV - the required table can be stored within the memory of every processor.

Rather than attempting a timing exercise for the model on the design, we will present an analysis of the efficacy of the routing network in supporting the data communication needs of the model. Figure 6.2-1 is a schematic representation of the grid of the general circulation model. Each

Code Segment	360/95 Time (seconds)	ILLIAC IV Time (seconds of CPU time only)	Time Ratio
COMP1	12.78	2.36	5.42 : 1
COMP3	6.54	1.54	4.25 : 1
Radiation (Large Table)	57.90	187.65	1 : 3.25
Radiation (Parallel algorithm)	*****	33.00	1.76 : 1

Table 6.2-1 Relative Timing of the ILLIAC IV and 360/95 Models

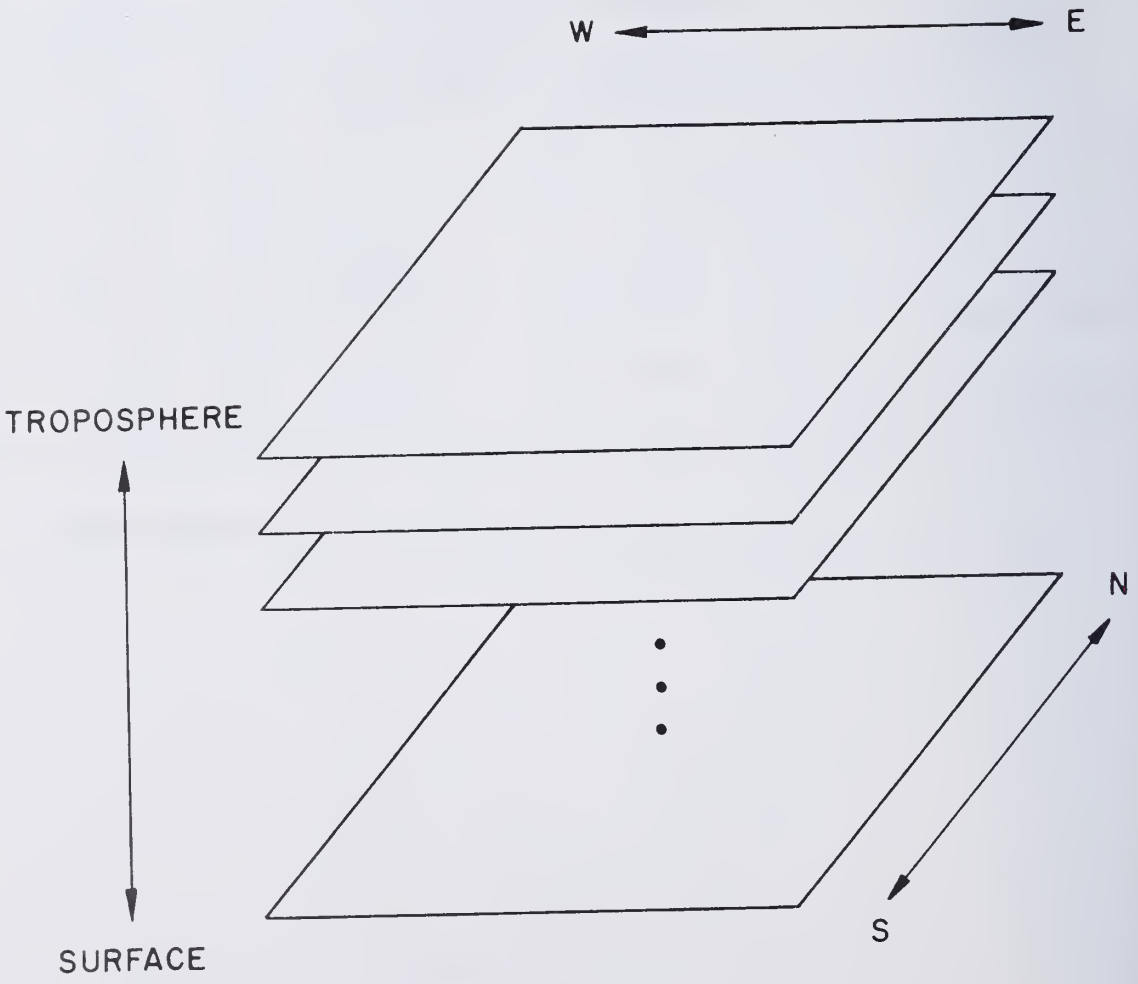


Figure 6.2-1 A Schematic Representation for the Grid of the General Circulation Model

spherical shell is shown as a rectangle. The north and south edges of each rectangle represent the north and south poles at the various vertical levels. Figure 6.2-2, based on Arakawa (1972), Tsan (1973) and Mintz (1974), shows the types of interactions between points of the grid which occur in the model. The interaction of the vertical levels is very simple. All of the horizontal interactions require simple access to one neighboring value (or a sequence of these operations) except the case which requires that the set of polar values be averaged to produce one common value.

The horizontal averaging shown in the figure is required to overcome the effect of the convergence of the meridians at the poles. If the Courrant stability condition - $c\Delta t < \Delta x$ - (Fox, 1961) which relates the maximum velocity to the inter-grid point spacing would require a very small time step over the entire grid for numerical stability. All models violate this condition, and use a larger time step than the small polar inter-grid distances permit. The resulting instabilities in the polar regions are removed by averaging several meriodnal values; the number of averaging iterations increase as the latitude approaches the polar regions. This zonal smoothing occurs even in the split grid model, although to a lesser degree. Because of this zonal smoothing, there is a clear inherent preference for parallel computation on circles of constant latitude. This approach is the best way to maximize the efficiency of the computation by maximizing the number of processors actively contributing to the results at any time.

For the next decade, GISS will be interested in models of two different horizontal resolutions (Halem, 1974). Both models have fifteen vertical levels. The two horizontal resolutions are:

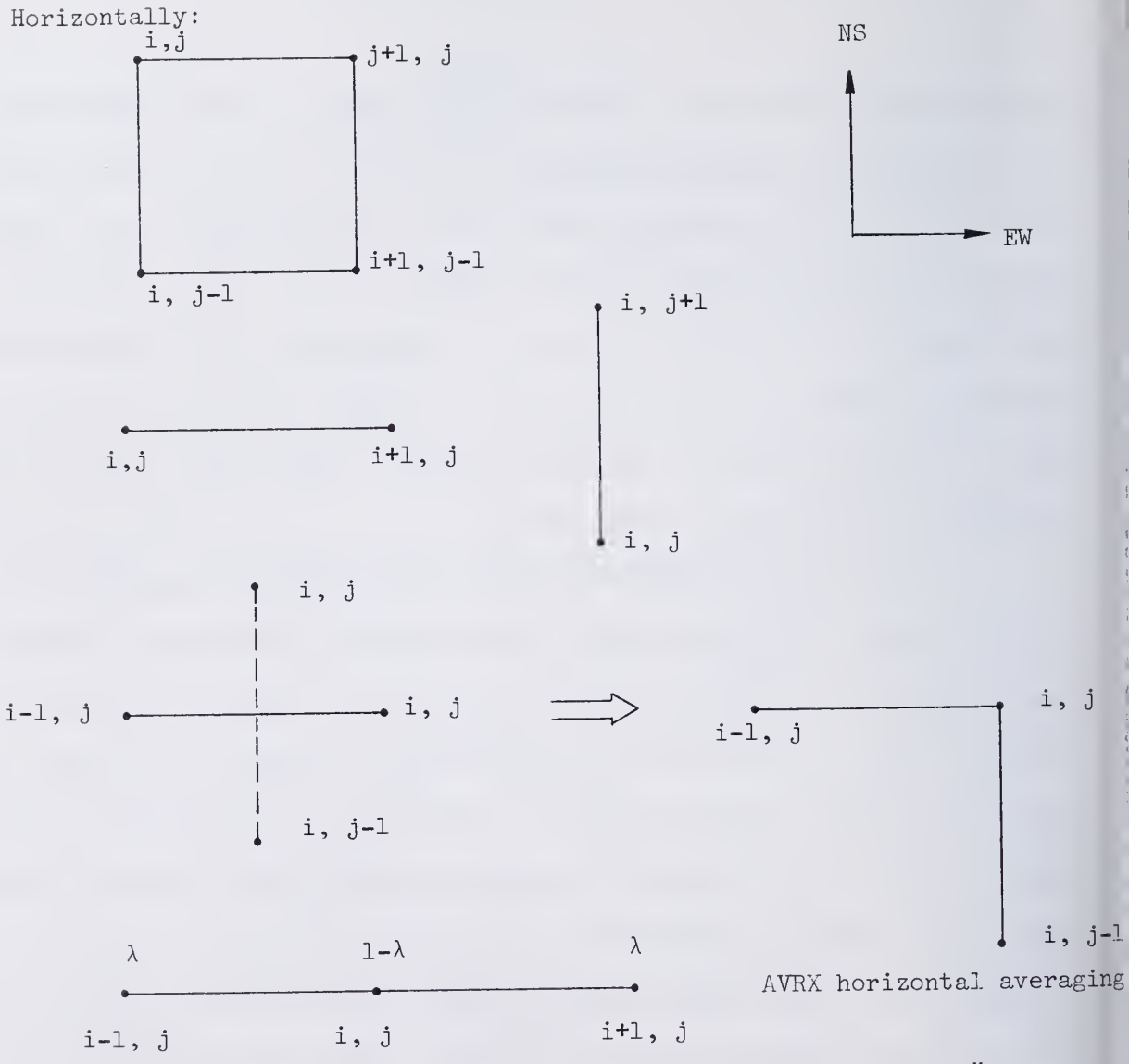


Figure 6.2-2 The Ranges of Interactions Between Points in the Finite Difference Grid

1. a model with 128 points around its equator and ninty-six circles of latitude, which we will call the 96x128 grid, and
2. a model with 256 points around its equator and 192 circles of latitude, which we will call the 192x256 grid.

In the next two sections, we will discuss the two primary variations of the model: the UCLA rectangular model and the Giss split grid model. A third section will discuss the common problem of computing the average of all polar values.

6.2.1 The Rectangular Model

In this model, all latitude circles have the same number of points. The 192x256 grid fits the machine very well; the entire array is treated as one circle of size 256. All of the processors are always fully employed. For the 96x128 model, the array can be treated as two circles of size 128. Fourteen of the fifteen vertical levels for a given latitude can be processed in parallel in seven cycles. One level from each of two different latitude lines can be processed in an eighth cycle, so that two complete latitude circles can be processed in fifteen computation cycles. In high latitude regions, half of the processors will be inactive during part of one of these cycles while the other half complete the extra zonal averaging steps required at the higher latitude. The machine will be very efficient for these models. Only shifts of one position left or right are required for east-west communication. An occasional shift of 128 positions is required for north-south communication in the 92x128 grid. All of the required shifts can be accomplished by the omega network in one routing network pass.

6.2.2 The Split Grid Model

We will discuss two different techniques for the split grid model. In the first of these, points deleted from the rectangular grid will be used, and missing points will imply unused processors. Figure 6.2.2-1 shows one rectangle of the resulting grid for the 96x128 model. To retain contiguity of values on the same meridian, points are stored with increasing separation between active processors as the latitude increases. Table 6.2.2-1 shows how the number of split grid regions - regions with the same number of points on a latitude circle - increases as the horizontal grid is refined. Table 6.2.2-2 shows a possible distribution of latitude circles of the various sizes which occur in the 96x128 and 192x256 grids.

Meridians at the Equator	Number of Split Grid Regions
72	5
128	7
256	11
512	15

Table 6.2.2-1 The Number of Split Grid Regions for Various Model Sizes

Just as in the rectangular model, the 192x256 grid uses the processor array as one circle of size 256, and the 96x128 grid uses two circles of size 128. In the rectangular model, a uniform shift of one position was always required for east-west communication. Hence, however, shifts of from one to as much as thirty-two positions (for the eight point high latitude circles in the 192x256 grid) are required. North-south communication in the 96x128 requires an occasional shift of 128 positions as before. All of the required shifts are supported by the omega network included in the routing network in one routing cycle.

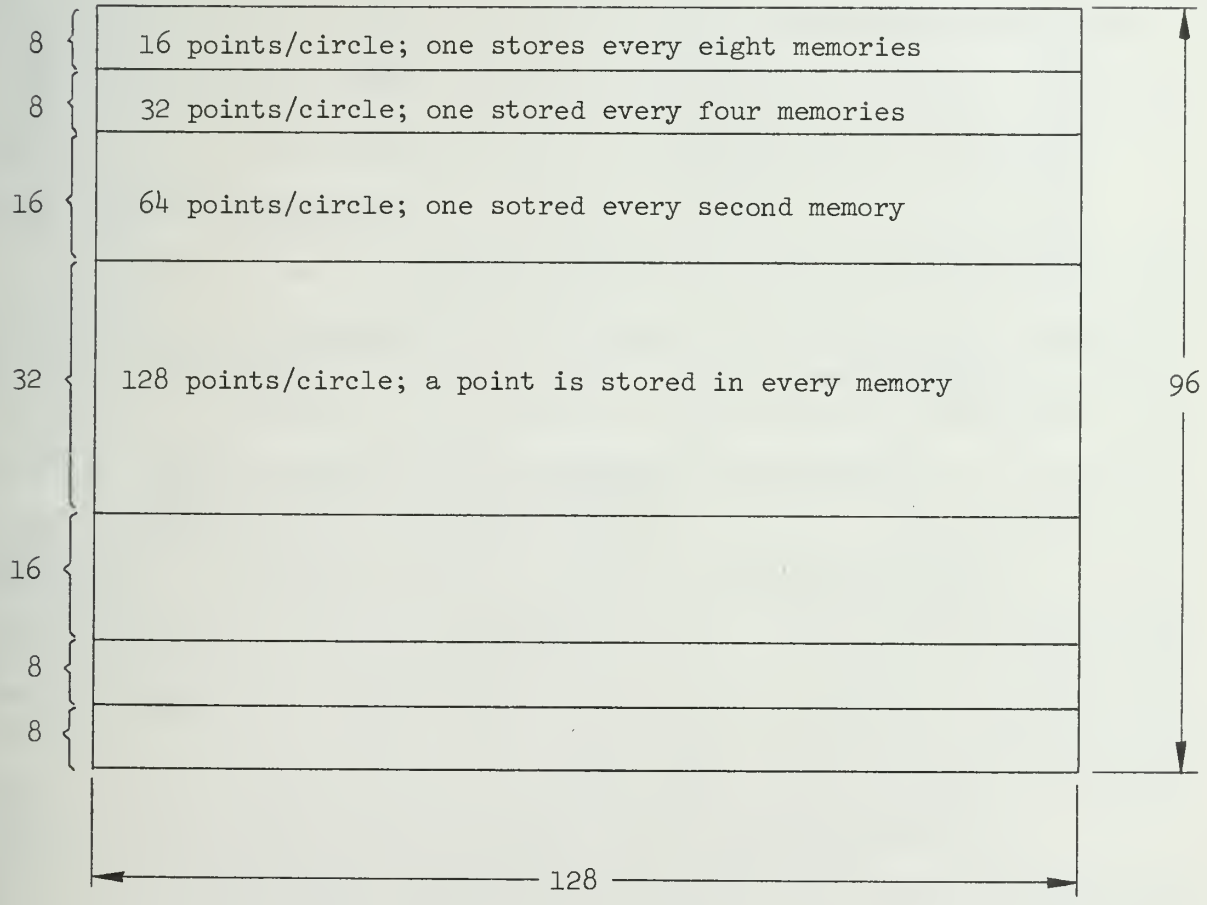


Figure 6.2.2-1 One of the Vertical Level of the Rectangular Mapping for the 96 x 128 Split Grid Model

192 x 256		96 x 128	
points per latitude circle	number of such circles	points per latitude circle	number of such circles
8	4	16	8
16	4	32	8
32	8	64	16
64	16	128	32
128	32	64	16
256	64	32	8
128	32	16	8
64	16		
32	8		
16	4		
8	4		
27328 points per variable per level		6912 points per variable per level	

Table 6.2.2-2 Distribution of the Various Sizes of Latitude Circles for one Level

In each of the split grid sizes, fifty-six percent of the processors are occupied by data. This seeming loss of efficiency is more than repaid by the fact that the time step for the split grid model is at least twice that for the corresponding rectangular model.

The second approach to the split grid model uses latitude circles of size sixteen through 128 for the 96x128 model and eight through 256 for the 192x256 model as indicated by Table 6.2.2-2. All shifts of data to support east-west communication in this approach are shifts of one position. For most cases, north-south communication requires a shift between different latitude circles by the size of the circles involved. For example, when the array of processors is treated as a collection of circles of size eight, an eight position shift which treats the array as one circle of size 256 will facilitate north-south communication. The exception noted above occurs when communication

between circles of different sizes must occur, as it must at split grid region boundaries. For these cases, an omega network expansion or contraction of interprocessor distance will suffice. How much of the potential gain which this approach stands to provide over that of the rectangular approach can actually be realized cannot be predicted at this time. Clearly, this second approach to the split grid model would be more difficult to program.

6.2.3 The Polar Circle Sum

In all forms of the model, the poles are represented by a full latitude circle of points whose values are computed and then averaged. In hardware terms, values from each processor in a partition must be averaged. The standard technique for this is the so-called log sum technique. Progressive shift and add steps produce the sum of 2^N values in 2^N contiguous processors in N-1 steps. In the first step, all values are circularly shifted one place, and the routed value is added to the stationary one. The sum is then routed two places and added to the previous partial sum. Successive routing distances double, until, in the final step, a shift of 2^{N-1} places occurs. In the rectangular and compressed split grid model, the first shift is by one place; in the rectangular split grid model, the first shift is by thirty-two places for the 192x256 grid and by eight places for the 92x128 grid since the initial values are separated by these amounts initially.

6.2.4 A Hardware and Time Comparison of the Clos, Omega and Nearest Neighbor Routing Schemes

The routing network described in section 4.3 requires an assembly-disassembly register in each processor and either two or three crossbar switches for each sixteen processors. Each assembly-disassembly register requires

twenty components, and each crossbar for an eight bit path uses 324 components. The Clos network scheme uses four cables per processor. One of the cables goes from the processor to the routing network, one goes from the routing network back to the processor, and the remaining two cables connect the stages in the three stage Clos network. An omega network uses only three cables per processor.

The nearest neighbor scheme of the SOLOMON and ILLIAC IV requires four cables per processor, assuming - as is true to date - that bi-directional ECL differential cables are not feasible. In any case, four sets of line drivers are required in each processor. To provide the vital broadcast input, a fifth cable and five sets of line receivers are required in each processor. The broadcast operation which permits the control unit to access a value from any of the processors must be included with added hardware if this function is desired. Moreover, some additional hardware is needed to support the input and output needs of the array of processors.

Ignoring anything but the nearest neighbor and broadcast connection, a fully parallel system would use seven six bit registers, four sets of ten quadruple line drivers, five sets of ten quadruple line receivers, and forty eight-to-one data selectors per processor. A byte serial scheme is much more economical. Each processor would have to have an assembly-disassembly register four sets of line drivers, five sets of line receivers, and a byte's width number of eight-to-one data selectors. Table 6.2.4-1 summarizes the component counts and transmission times for the various options.

The nearest neighbor routing network permits only one and sixteen position uniform shifts in a 256 processor circle. Partitions of that circle

Routing Scheme	Components for each Sixteen Processors	Transmission Time in Nanoseconds
Eight Bit Clos Network	1292	574
Eight Bit Omega Network	970	515
Parallel Nearest Neighbor Network	2192	91
Eight Bit Nearest Neighbor Network	736	455

Table 6.2.4-1 Component Counts and Times for the Three Possible Routing Schemes

are not supported. Expansion and contraction for connecting split grid regions stored compactly are not supported. The omega network supports all of the partitions and shifts required by the general circulation models discussed in this paper. Shifts of any distance and direction within the permitted partitions are all accomplished simultaneously in one pass through the routing network. Only shifts of one and sixteen positions take one pass with the nearest neighbor scheme.

It is clear from the above comments that the nearest neighbor routing scheme finishes a distant third in the three way race for inclusion as the routing scheme. Whether the Clos or omega network should be used depends on the control algorithms available when an implementation is undertaken, and the routing requirements on the machine which is being built. The Clos scheme uses thirty-five more components per processor than the nearest neighbor scheme, and the omega network uses only four more components per processor than the nearest neighbor scheme.

6.3 Image Data Processing

Results from the research conducted by a group led by Robert Ray (1974) has shown that the ILLIAC IV is an efficient computer for processing multispectral image data from the Earth Resources Technology Satellite (ERTS) experiment (George, 1971). The initial stages of Ray's work have produced ILLIAC IV implementations of the data clustering (Thomas, 1974b). These algorithms were adapted by the Laboratory for Application of Remote Sensing (LARS) of Purdue University (Wacker, 1970) from the ISODATA algorithm of Ball and Hall (Ball, 1965). These algorithms, originally developed for use with aircraft multispectral scanner image data, have been successfully applied to simi-

lar data collected by the ERTS satellites.

The ERTS satellite measures solar energy reflected from the earth's surface; four different spectral bands of reflected energy are measured for each point. The data is processed in terms of frames which contain $7.7(10)^6$ (3240 times 2340) points each. Since each point is represented by values of reflected energy in four spectral bands, each frame of ERTS data contains almost thirty-one million small integer values.

The LARS technique has two steps. The first step uses manually selected areas to compute "spectral signatures" for known terrain features. The statistical characterizations so determined are then applied to large areas of interest to estimate the extent and amount of terrian with features like those in the training areas. These two steps, called clustering and classification respectively, are described in the following two sections as potential applications of the machine design presented in this paper.

6.3.1 Image Data Clustering

The ERTS data for a given point (an area of approximately 1.1 acres) consists of a vector of four spectral energy measurements. The objective of the clustering algorithm is to partition the data in the test region into M or less spectrally dissimilar classes. Iteration of the steps in the algorithm continues until the M clusters of the initial data are determined. Each cluster is characterized by a mean of its four dimensional spectral data points and a four by four symmetric covariance matrix.

The algorithm is described in detail in the following text together with comments on how the machine design of this paper would be used to implement the algorithm.

The entire set of 256 processors is used in concert during the clustering algorithm. The initialization steps in the algorithm determine initial mean and standard deviation vectors for the set of data points.

A given data point is represented by a four element vector, $X_i = (X_{1,i}, X_{2,i}, X_{3,i}, X_{4,i})$. The initial four means,

$$m_j = \frac{1}{N} \sum_{i=i}^{N} X_{i,j}, \qquad j = 1, 2, 3, 4,$$

are found for the complete set of N data values. The algorithm should distribute the data points uniformly across all 256 processors of the array. The summation process begins with a loop which adds all values within each processor and ends with a log sum step (see section 6.2.3) across all 256 processors. The initial value N is broadcast. The four means, recovered by the control unit through its port to the routing unit, are broadcast to permit computation of four initial standard deviation values:

$$s_j^2 = \frac{1}{N-1} \sum_{i=1}^{N} (X_{i,j} - m_j)^2.$$

The cartesian product of the four real line intervals,

$$\ell_j = [m_j - s_j, m_j + s_j] \quad i = 1, 2, 3, 4,$$

defines a rectangular parallelapiped which should contain most of the sample points. The M initial cluster centers are chosen to be uniformly spaced along a diagonal of this parallelpiped, and all M values are computed and stored by each processor. The algorithm iterates the following two steps to determine M final cluster centers.

Step one determines the eucludian distance between each point and

each of the M cluster centers. Each point is assigned to the cluster with the nearest cluster center. This calculation takes place without any inter-processor communications.

Step two computes new cluster centers by using the means of the vectors in each cluster. If no vector changed clusters in step one, the algorithm terminates. A change of cluster is determined by using the processor mode sensing hardware described in section 4.4.1.

The result of the clustering process is M four element cluster centers and M symmetric four by four variance-covariance matrices. The elements of these matrices,

$$C^2_{i,j} = \frac{1}{P} \sum_{\ell=1}^{P} (X_{\ell,i} - m_i)(X_{\ell,j} - m_j) \quad i,j = 1, 2, 3, 4,$$

and the number of vectors, P, within each cluster are computed by intra-processor summation followed by log sum steps for the entire processor array.

6.3.2 Image Data Classification

The clustering algorithm determines a cluster mean and covariance matrix for each of M clusters which it identifies in the data for a selected set of ERTS data. The classification algorithm uses these two paramters for each of the M classes and, for each point of the data being classified, computes the probability of class membership for each of the M classes, and assigns each point to the class for which its probability of membership is highest. The probability function, based on the assumption that the distribution function is multivariate normal, is

$$P_i(X) = b_i - \tfrac{1}{2}\left[(X-M_i)^T C_i^{-1} (X - M_i)\right], \qquad i = 1, 2, \ldots, M.$$

The terms in the probability function are:

X: a four component vector of ERTS data,

M_i: the four component mean vector for class i,

C_i: the four by four covariant matrix for class i, and

b_i: $-\frac{1}{2} \log| C_i^{-1} |$ (Fu, 1968).

The constants b_i and the covariant matrix inverses are computed by a step intermediate to the clustering and classification steps. These constants may be used in several classification steps.

In the following two sections, we discuss two different ways to organize the execution of the classification process.

6.3.2.1 Classification by Routing Point Values

In this shceme, we partition the array of processors into circles of size M, the number of data clusters or classes. One processor in each partition is loaded with the constants for one data class. Considerable flexibility is provided by this approach. For example, several different sets of data class can be applied to one set of ERTS data by using different input constants in different partitions. The input ERTS data can be distributed across the partitions as desired. If only one set of classification constants is used, the input ERTS data can be uniformly distributed across the array of processor memories. Within each partition, M points at a time (plus a class number and probability value) are routed circularly around the M processors in the circle one step at a time. The probability that a point lies in a class is computed by the processor which stores constants for that class and the class number; the probability of the most likely class and the four spectral values

are forwarded around the circle. When the M steps for each M points have been completed, each of those points has been assigned to its proper class.

This scheme makes full use of the Clos routing network; circular shifts of one position at a time are all that the scheme requires, and arbitrary class sizes are facilitated. Unless M, the number of classes, is a power of two, there will be inactive processors. If M is a power of two, the omega network will support the algorithm.

6.3.2.2 Classification by Broadcasting the Class Constants

In this scheme, the ERTS data is uniformly distributed across the 256 processors and their memories. The sets of constants which describe the classes of interestare broadcast by the control unit for storage in the program memory. Classification with respect to several sets of classification parameters can be performed by broadcasting the several sets of classification constants. In this scheme, there need be no inactive processors. Each cycle in the classification process requires fifteen uses of the routing network to broadcast the ten values for the symmetric covariance matrix, the four class mean values, and the constant "b" term for each class. The previous scheme uses the routing network six times in each step. The degree of independent (that is concurrent) action permitted by the control unit for the processor array and the routing network will determine which of the two schemes is to be preferred.

6.3.3 Byte Packing and Unpacking

The ERTS data, measured by photosensors and converted to digital data by the satellite, consists of many small integer values: each spectral measurement is converted to a six bit value. Moreover, the classification process assigns each point to a class which can be represented by a small integer. Thus,

for efficient use of the input and output facilities of the machine, it is important to be able to unpack several small integer values from one word of data, and to be able to pack several small computed values into one data work.

Figure 6.3.3-1 illustrates how four ERTS values for one point can be packed into one word for input and unpacked for use by the machine. Part (a) of the figure shows the four bytes packed into the twenty-four bit fraction of a data cord. Part (b) shows the result of an AND operation with a mask which selects value three and assigns it the exponent value plus four.

Because the exponent radix of the machine is sixteen, the binary point can only lie between four bit digit positions; for value three, this means that the binary point is placed within the value, not at its right end where it belongs. A multiplication by 2^2 - that is a shift operation - results in a non-normalized integer value with the correct exponent value and with the binary point in the correct position as shown in Figure 6.3.3-1(c).

Figure 6.3.3-2 illustrates how a small integer value is packed into the desired position of a data word fraction. The initial integer value, a full word as shown in part (a) of the figure, is added to the constant shown in part (b) with a floating point non-normalized addition. The result of the addition is shown in part (c) of the figure. The arrows in part (b) and (c) of the figure indicate the position of the binary point. The value is alligned by a "shift" of two places - division by 2^2 - which yields the result shown in part (d). The final step ANDs the part (d) result with a mask. A final step to OR this result, shown in part (e), into a data word with other packed values is not shown in the figure.

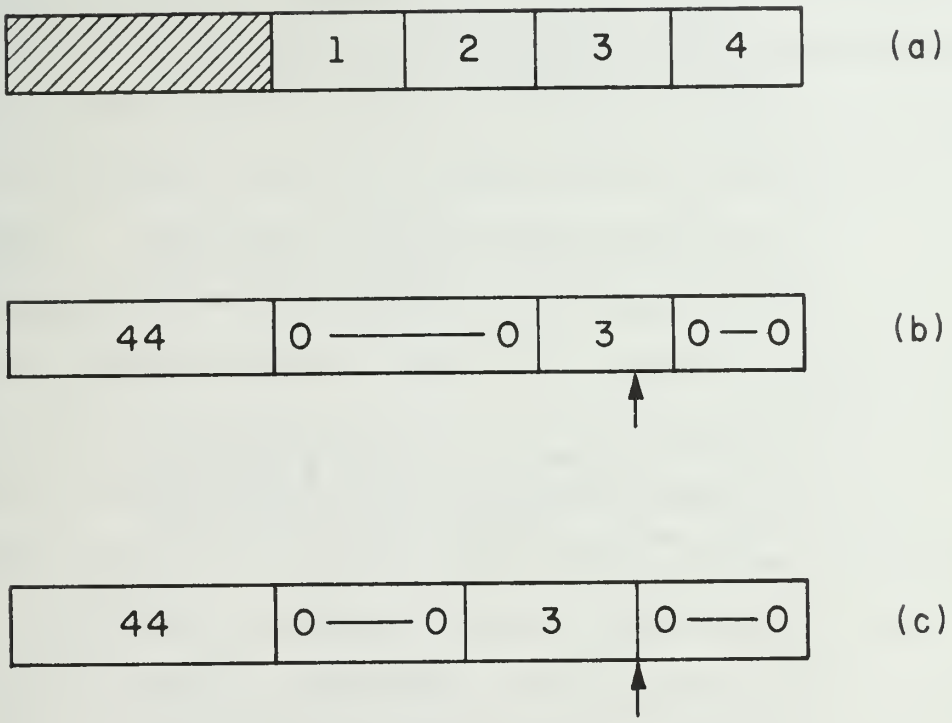
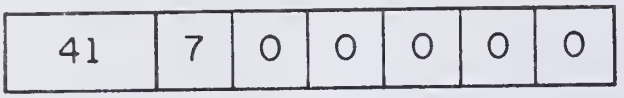
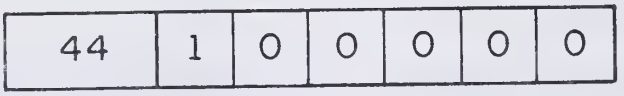


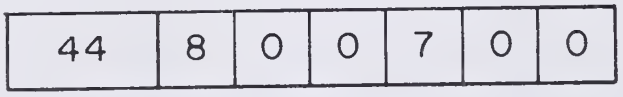
Figure 6.3.3-1 Unpacking Data Values



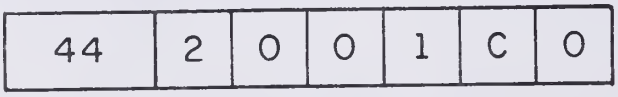
(a)



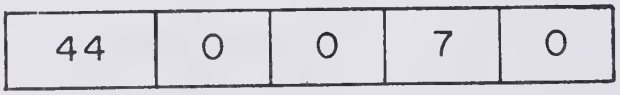
(b)



(c)



(d)



(e)

Figure 6.3.3-2 Packing Data Values

6.4 File Processing and Information Retrieval

In this section, several examples of file processing and information retrieval will illustrate the capabilities of the machine for this class of problems. The first example concerns file comparisons to determine statistics about pairs of similar files including how a large file can be efficiently sorted. A second example shows how information can be retrieved from a file with the machine.

6.4.1 File Statistics

Post processing of weather model data frequently includes comparison of two files of data taken from two model runs with slightly different starting conditions. Average differences between various parameters are sought. Two such files can be read into the memory of the machine and compared 256 points at a time. If the average difference between two temperatures is sought, for example, 256 sums of pointwise differences within the 256 processors can be quickly computed. A final sum of the 256 partial sums can be computed by an eight step "log sum" which adds values routed by one, two, four, eight, . . ., 128 positions. Eight such steps, the log to the base two of 256, produce the sum of all the pointwise differences which was sought. Each one of the 256 processors contains a copy of the same value at the end of the process.

If a distribution for the differences is sought, each processor can compute and sort all differences for the points which it holds. Then a 256 way merge of the 256 sorted lists of differences can be performed by an eight step comparison process which determines the smallest of the 256 locally smallest values, for example. At the end of the process, all 256 processors contain the same smallest value. The number of occurances of the value can be determined

by a log sum of the number of occurances of the value in each of the processors; the log sum result will also be held in each of the 256 processors at the end of the log sum process. Hence, a sorted list of pointwise differences together with a count of their individual frequencies can be easily extracted by the control unit using its connection to one port of the routing network. If an approximate distribution is sought, the interval of interest can be divided into sub-intervals and a log sum of processor computed counts of values which they hold which lie in the broadcast interval can be performed.

6.4.2 Information Retrieval

In this example, we suppose that the files of a computer dating service are stored in the array memory. Since this example is included to illustrate machine functions, no indices for the file are assumed. The raw data records of the file are used. Let us suppose that a young customer wishes to locate all girls which meet the following characteristics:

> EYES: (green or blue) and HAIR: (blonde or red) and RELIGION: (agnostic) and AGE: (22 through 27 years) and EDUCATION: (college graduate) and HEIGHT: (63 through 68 inches) and WEIGHT: (two pounds or less per inch of height).

The mode logic can be used to evaluate 256 records of the file at a time. One status register bit can accumulate the Boolean result while another is used to compute each parenthesized term. After all the tests have been made for each set of records, the 256 MODEOUT values can be ORed together and sampled by the control unit as shown in Figure 4.4.1-1(b). If the sixteen bit result is zero, no match was found. A one bit in any position indicates that one or more of the processors in a sixteen processor group contain matches. With proper bit handling instructions and MODEIN transmissions like those of Figure 4.4.1-2(b),

the control unit can process a sequence of MODEOUT signals of the type shown in Figure 4.4.1-1(c) and locate each match in the array. A control unit specified route can shift the identifying number for the match to the control unit's routing unit port.

6.5 Matrix Inversion by Gaussian Elimination

In this section, we will discuss using the machine to solve systems of equations or invert matrices using the familiar Gaussian elimination technique. The process can be used to solve several systems or invert several matrices simultaneously. Two different situations are described: in the first, a collection of inhomogeneous linear systems are to be solved in the second, the inverses of the given set of matrices are to be found. The algorithms are similar and store the original matrix in skewed form as suggested by Kuck (1968) as illustrated in Figure 6.5-1. In the figure, the matrix and right hand vector of the linear system Ax = b are shown. The A matrix is stored skewed, but the "b" vector is stored all within the memory of one processor. When skewed storage is used, parallel access to all of the elements of any row or any column of the matrix can be achieved. In the figure, the rows are stored across the processors with all elements of a given row having the same word address in the various processor memories. The elements of a column, on the other hand, all occupy different word addresses, so that processor indexing is required to fetch a column.

6.5.1 Solution of Inhomogeneous Systems

Up to thirty-two seven-by-seven inhomogeneous systems can be solved simultaneously if their coefficients are stored as shown in Figure 6.5-1. The Gaussian elimination procedure has two phases. In the first phase, the matrix

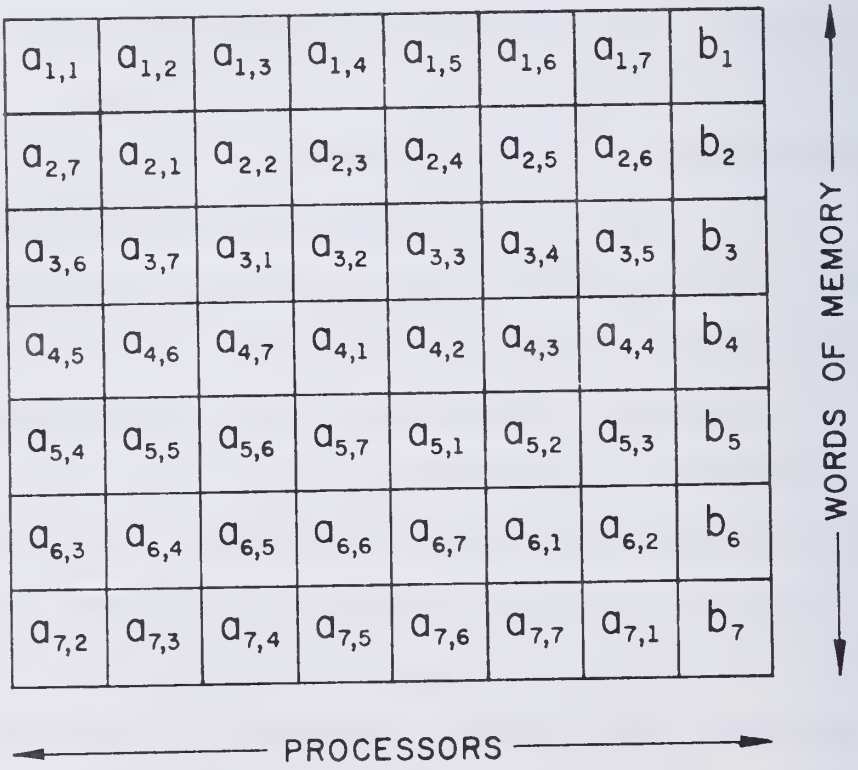


Figure 6.5-1 Storage Map for a Seven by Seven Inhomogeneous System

of the original system is reduced to upper triangular form with ones on the main diagonal. In the second phase, the solution is found by back-substitution, reducing the matrix to the identity matrix and the right hand side to the solution. The technique processes the columns one by one, beginning with column one and proceeding through the columns in turn to the rightmost (or highest numbered) column. The matrix under consideration is gradually reduced one column (and one row) at a time until an upper triangular system remains.

The steps in the algorithm, described in detail in the following sections, are:

1a) Find the element with the largest absolute value in the lowest numbered column which remains under consideration, and call it column i.

1b) Find the smallest row number of the several rows which may contain elements with the value identified in step (1a).

1c) Exchange the row identified in step (1b) with row i. Both rows must be shifted so that they are properly skewed in their new positions.

1d) Divide all the elements of the new row i by element $A_{i,i}$. Divide the new b_i by $A_{i,i}$ also.

1e) For each of the rows i+1 through seven, multiply row i by element $A_{j,i}$ and subtract from row j.

At the completion of steps (1a) through (1e), the matrix will be in the upper triangular form. The back substitution steps proceed from the last row's right hand side element, b, back through that of the first row. They operate on the columns of the upper triangular matrix from the highest numbered back through to the first. The steps are:

2a) Distribute b_j for use with all rows from 1 to j-1.

2b) Multiply row j by element $A_{i,j}$ for each row i from 1 to j-1, and subtract the resulting multiple of row j from row i.

the result of the back substitution steps is to reduce A to the identity matrix and the column of b's to the sought solution vector.

The seven steps outlined above are described in detail in the following seven sections.

6.5.1.1 Find the Pivot Element in the Leftmost Remaining Column

The matrix was stored in skewed form as shown in Figure 6.5-1 so that all elements of any desired column would be available in parallel. The element in the leftmost remaining column with the largest absolute value is found by a process which resembles the log sum process described in section 6.2.3. In that section, however, the number of cooperating processors was always a power of two in number, while here, the number of processors varies from step to step all the way from two up to the size of the system being solved. In section 6.2.3, the processors which were cooperating were contiguous; here, because the matrix is stored in skewed form, the elements which must be considered together may not be stored in contiguous processors. We will ignore the noncontiguity and describe the algorithm as though the processors were contiguous. The Clos routing network, which can perform every permutation, can be used to facilitate the desired connections.

For a collection of processors which are a power of two in number, the steps are the same as in a log sum, except that each processor selects the larger of the two elements it considers at each step rather than producing their sum. The number of comparison steps is the logarithm of the number of processors to the base two. When the total number of processors is not a power of two, subsets of the total number which each contain a power of two processors form partial results which are then combined pairwise until the

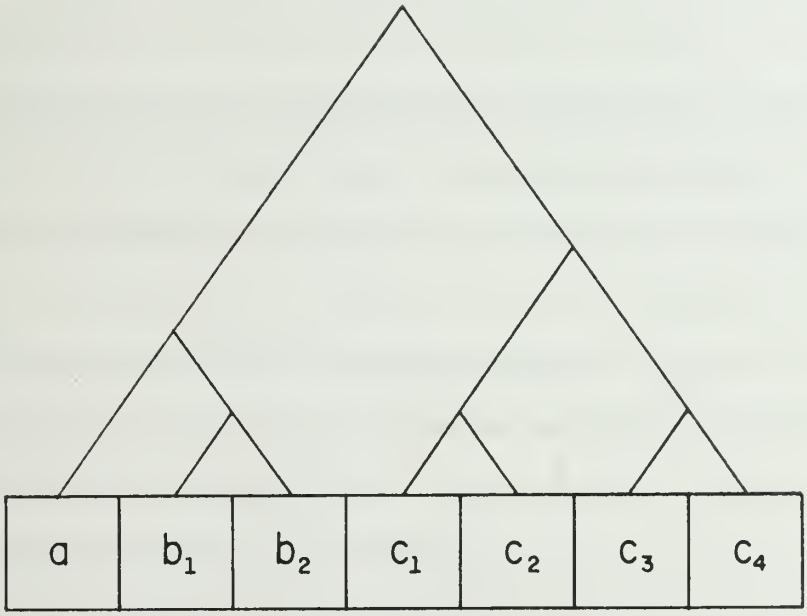


Figure 6.5.1.1-1 The Log Combination Process for a Collection of Processors not a Power of Two in Number

final result is produced. There is one such subset for each one bit in the binary representation of the number of processors. Figure 6.5.1.1-1 illustrates the process for seven processors. Three comparison steps are required; in general, the number of comparison steps is the logarithm to the base two of the smallest power of two which is greater than or equal to the number of processors.

6.5.1.2 Find the Smallest Numbered Row which Contains the Pivot Element

Once the pivot element value is identified, each processor which stores that element submits its row number for a minimum seeking comparison process. Processors which do not store the pivot value - by far the majority - submit a value which exceeds the number of rows in the matrix. A log minimum process determines the row number of the row to be exchanged with the lowest numbered currently considered row. At the completion of this step, every active processor contains the number of the row which contains the pivot element.

6.5.1.3 Exchange of the Pivot Row with the First Active Row

The number of the pivot row is available to all active processors as the result of the previous step. The first active row number is available by broadcast from the control unit. The difference of the two values is the amount that the pivot row must be shifted left and the first row shifted left to retain the correct skewed storage relationships. This shifting process goes on in parallel for each of the systems being solved by the 256 processor array. The shifting algorithm proceeds as follows:

1. Each processor puts the shift distance - a binary integer of eight or less bits - in its eight bit status register within the mode logic. The number of bits to be considered is the same as the number of steps in the log comparison process which identified the pivot element.
2. For each bit to be considered, the mode of the processor is set from the proper status register bit. The pivot row elements are shifted left by the amount specified by the selected bit; the shifted values are stored under mode control so that the shift takes place only in those processors - that is only in those equation systems - for which a shift by that distance is required.
3. The first row still under consideration is shifted right by a process similar to that described in step two above. The only difference is that right shifts are used instead of left shifts.

6.5.1.4 Divide the Pivot Row by the Pivot Element

The pivot element was distributed among all active processors by the steps described in section 6.5.1.1. This value is divided into each element of the pivot row. This step leaves the pivot element exactly one in value.

6.5.1.5 Reduce the Leftmost Column to Lower Triangular Form

The pivot row is the lowest numbered remaining row, and it has been normalized by the previous step so that the pivot element is one. For all rows below the pivot row, we

1. distribute the element in the pivot column to all active processors by a log distribution process, and
2. multiply a temporary copy of the pivot row by the distributed element and subtract from the subject row.

The completion of the above two steps for all rows beyond the pivot row reduces the lowest numbered remaining column to lower triangular form.

6.5.1.6 Back Substitution

At the completion of the previous steps, the matrix is in upper triangular form with ones on the main diagonal. Back substitution reduces this upper triangular form to the diagonal identity matrix. The last row of the upper triangular form contains only a one in the last column and all the rest zero elements. The back substitution process uses successive main diagonal ones from right to left as follows.

1. For each row above the row which contains the current main diagonal one, distribute the element in the column which contains that main diagonal one by a log distribution process.
2. Multiply a temporary copy of the row with the main diagonal one by the distributed element and subtract from the row from which the distributed element was taken. Include the right hand side vector in the multiplication and subtraction process.

At the completion of the above two steps for all main diagonal elements from right to left, the original matrix is reduced to the identity matrix and the right hand side vector becomes the solution to the given set of equations.

6.5.1.6 Efficiency and Routing Requirements of the Gaussian Elimination Proces

The Gaussian elimination process described in the preceeding sections clearly requires routing operations beyond the capabilities of the omega network. The Clos network is necessary to support this algorithm, but we do not currently have algorithms to compute the necessary control patterns.

As we have seen, the technique described in this section begins with

all processors in productive use, proceeds until only a fraction of the processors are contributing, and returns to the condition where all processors are in productive use. On the average, approximately half of the processors are productive. When a great many matrices are to be processed, they should be handled 256 at a time by a conventional program with one matrix (or system) stored in each of the 256 processors. No inter-processor communication is required. A collection of 128 or more matrices (or systems) can be processed in this way with a processor efficiency at least as good as for the parallel technique described above.

6.5.2 Inversion of a Matrix

To invert an N by N matrix with the Gaussian elimination technique, one begins with an N by 2N matrix which includes an identity matrix appended to the right of the given matrix, extending each row to twice its original size. In a parallel processor, the best approach is to store the given matrix in skewed form and the appended identity in non-skewed form in the same set of processors with the given matrix. The operations performed on the given matrix under the Gaussian technique are also performed on the appended identity matrix (except for the shifts to reskew the identity). At the completion of the process, the given matrix has been transformed to an identity matrix and the appended identity matrix is transformed to the inverse of the given matrix.

7. Operating Parameters of the System

This section summarizes the cost, reliability, and power consumption of the system. The calculations are based on the component counts shown in Figures 7-1 through 7-4 which give detailed component counts, prices and power requirements for the processor, memory module, sixteen by sixteen crossbar and table look up hardware. Table 7-1 summarizes these figures and gives total parts counts and costs for these units; total costs are calculated including the spares indicated, and power and parts counts include only the units needed to form a complete operating system. These costs were derived from data taken from competitive bids, parts orders for parts for the multiplier prototype which was built and telephone calls to suppliers. Assuming that assembly costs will be approximately equal to integrated circuit costs, the total cost for a 256 processor system with eight million words of data memory and 128,000 words of program memory is approximately $3,000,000 if a Clos three stage routing network is built.

A system with an omega routing network would be approximatley $100,000 less expensive. The cost figures do not include the costs of air conditioning equipment.

The operating life of an integrated circuit component depends on the operating temperature. The prices quoted for parts in Figures 7-1 through 7-4 assume that the lower cost SN7400 series parts, whose operating temperatures must lie between zero and seventy degrees Celcius, are used. Figure 7-5 is a graph of the expected component failure rates versus temperature. The failure rate data were taken from a Signetics Corporation report supplied to the author by a supplier (Signetics Corporation, 1974b), and refer to that

COMPONENT	NUMBER OF UNITS	WATTS PER UNIT	COST PER UNIT
10124	2	0.468	$ 4.50
10125	2	0.540	$ 4.50
AM25S10	16	0.467	$ 2.60
AM9309	10	0.240	$ 6.00
AM9334	1	0.240	$ 5.20
NAT8551	1	0.360	$ 1.00
74S02	2	0.050	$ 0.54
74S04	3	0.050	$ 0.47
74S11	3	0.050	$ 0.52
74S20	2	0.050	$ 0.50
74LS32	1	0.049	$ 0.34
74S51	4	0.110	$ 0.23
74H52	10	0.275	$ 0.23
74H61	1	0.080	$ 0.22
74S64	2	0.250	$ 0.38
74S74	4	0.250	$ 0.75
74S85	8	0.250	$ 3.93
74S86	1	0.250	$ 0.71
74S133	3	0.300	$ 0.42
74148	2	0.190	$ 1.50
74150	2	0.340	$ 1.41
75S151	4	0.225	$ 2.25
74S153	14	0.225	$ 4.50
74S157	22	0.390	$ 3.76
74S158	1	0.305	$ 3.76
74S172	40	0.500	$ 5.99
74S175	1	0.480	$ 1.68
74S181	7	1.100	$ 3.15
74S182	6	0.260	$ 4.86
74S195	16	0.545	$ 1.68
74S257	12	0.495	$ 3.76
74S260	13	0.300	$ 0.42
74S274	36	0.500	$ 12.50
74S283	12	0.500	$ 2.76
74S299	1	0.500	$ 1.50
74S381	21	0.800	$ 3.15
SIG8204	4	0.850	$ 27.20
SIG8205	2	0.850	$ 33.40
SIG8228	33	0.512	$ 21.87
SIG8243	12	0.500	$ 4.95
SIG8263	5	0.475	$ 4.50

TOTAL NUMBER OF COMPONENTS: 342

TOTAL POWER DISSIPATION: 154.923 WATTS.

TOTAL COST: $ 2236.04

Figure 7-1 Component Statistics for the Processor

COMPONENT	NUMBER OF UNITS	WATTS PER UNIT	COST PER UNIT
AMS	304	0.400	$ 6.12
74S04	1	0.270	$ 0.47
74LS138	1	0.055	$ 1.43
74154	2	0.280	$ 1.35
74S157	2	0.390	$ 1.43
74S280	12	0.525	$ 0.40
SIG82S42	10	0.290	$ 0.71

TOTAL NUMBER OF COMPONENTS: 332

TOTAL POWER DISSIPATION: 132.465 WATTS.

TOTAL COST: $ 1879.84

Figure 7-2 Component Statistics for One Processor Memory of 32,768 Words with Thirty-eight Bits Each

COMPONENT	NUMBER OF UNITS	WATTS PER UNIT	COST PER UNIT
10101	36	0.135	$ 0.47
10115	32	0.135	$ 0.47
10133	32	0.390	$ 2.95
10145	16	0.754	$ 13.00
10158	16	0.200	$ 1.55
10164	256	0.390	$ 1.65

TOTAL NUMBER OF COMPONENTS: 388

TOTAL POWER DISSIPATION: 136.764 WATTS.

TOTAL COST: $ 781.56

Figure 7-3 Component Statistics for One Sixteen by Sixteen Crossbar

COMPONENT	NUMBER OF UNITS	WATTS PER UNIT	COST PER UNIT
10124	2	0.468	$ 4.50
10125	2	0.540	$ 4.50
74S157	4	0.390	$ 3.76
74LS193	4	0.155	$ 2.12
74S195	16	0.545	$ 1.68

TOTAL NUMBER OF COMPONENTS: 28

TOTAL POWER DISSIPATION: 12.916 WATTS.

TOTAL COST: $ 68.40

Figure 7-4 Component Statistics for One Table Look Up Unit Exclusive of the Memory

ITEM		BUILD		RUN		
Name	Parts	Number	Cost	Number	Parts	Power (watts)
Processor	342	300	670,800	256	87,552	46,500
Memory	332	320	601,600	276	91,632	36,708
Crossbar	388	---	---	---	---	---
Clos	---	60	126,480	48	18,642	6,576
Omega	---	40	84,320	32	12,416	4,384
Table Look Up	28	20	1,360	16	448	208
TOTALS						
Clos	---	---	$1,400,240	---	198,256	89,992
Omega	---	---	$1,358,080	---	192,048	87,800

Table 7-1 System Component, Component Counts, and Power Consumption

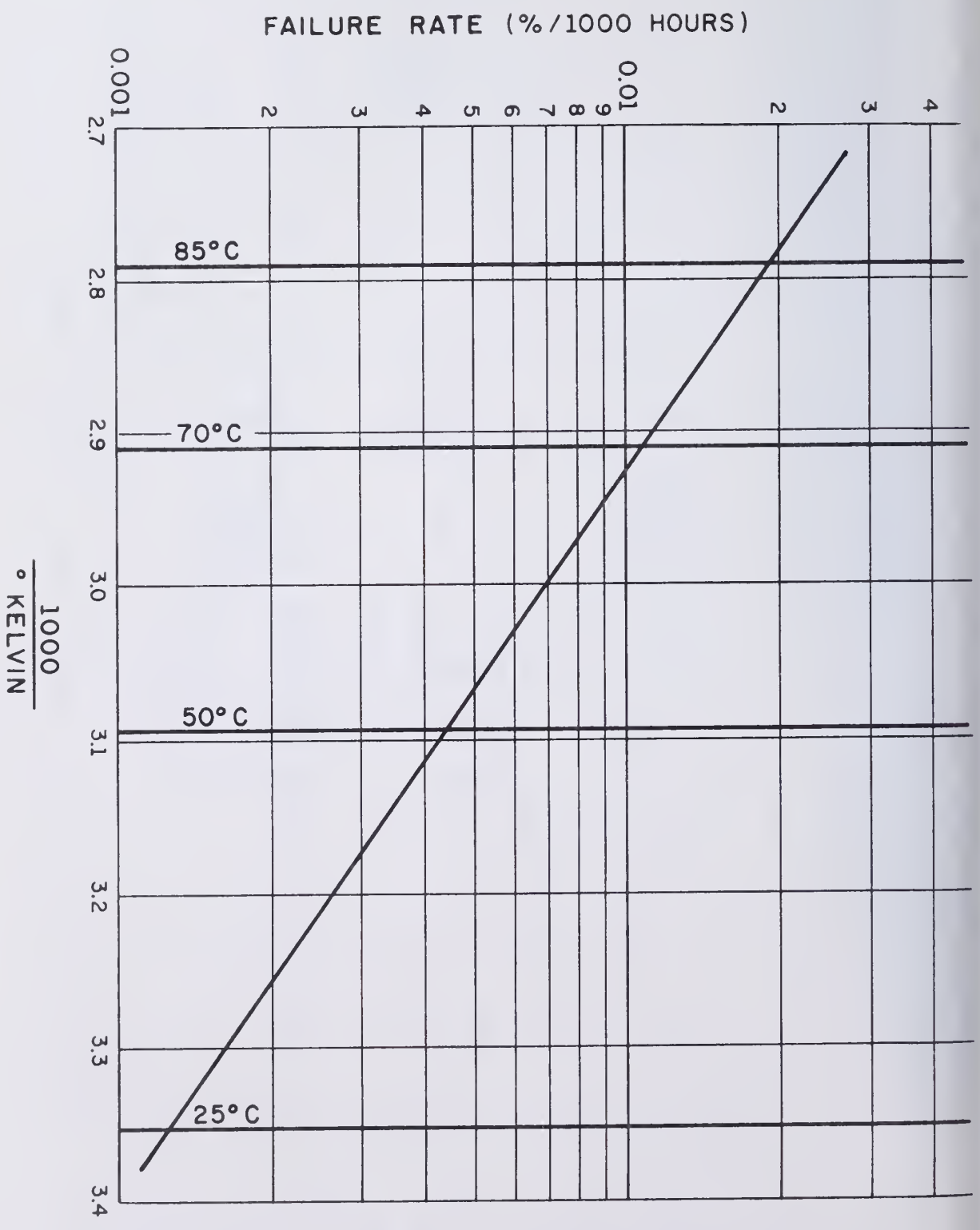


Figure 7-5 Graph of Component Failure Rate Versus Temperature

company's SN7400 line. This report presented the most comprehensive review of failure rate data which the author was able to obtain. The data in the report pertain to the low power Shotty devices in the Signetics 7400 lines, not to the regular (non-low power) devices used in this design. Table 7-2 gives the failure rate data for a 200,000 integrated circuit component system using values taken from the graph in Figure 7-5. As the table indicates, we should expect the system to operate for twenty-six to forty-five hours between failures. Several spare processors, crossbars and memory modules will be available to replace a unit which fails. No design for the control unit was included since work came to end before that was possible. However, because of its critical role in the system, it could well be the best policy to build two complete control units so that a spare one would be available in the event of a control unit failure.

Temperature °C	Number of Failures per 1000 hours	Failures per 1000 hours for a 200,000 component system	Mean Time Between System Failures (hours)
85°C	0.00019	38	26
70°C	0.00011	22	45
50°C	0.000044	9	111

Table 7-2 System Reliability

8. Conclusion

The author believes that the forgoing sections - mainly section 4, section 6.2, and section 7 - show that a computer with roughly 100 times the computing capacity of the IBM 360/95 can be built for significantly less than other computers with similar capability.

Another result of the work described here is the simulation methodology described in section 5 and illustrated in the appendix.

Considerable work remains to be done on the routing system. Although we believe that an omega network is sufficient to support the intercommunication needs of the general circulation model, the matrix manipulation example of section 6.5 shows that the three stage Clos network would provide support for a wider class of problems at a modest increase in cost. However, we have no algorithm to produce control patterns for the Clos network.

References

Advanced Micro Devices Incorporated, 1974
Advanced Micro Devices Data Book, Advanced Micro Devices Incorporated, 1974.

Arakawa, 1972
Arakawa, A., Design of the UCLA General Circulation Model, Technical Report Number 7 of the Department of Meteorology, University of California at Los Angeles, July 1972.

Benes, 1965
Benes, V. E., Mathematical Theory of Connecting Networks and Telephone Traffic, Academic Press, 1965.

Breuer, 1972
Breuer, Melvin A. (Editor), Design Automation of Digital Systems, Volume One, Theory and Techniques, Prentice Hall, 1972, pp. 101-172.

Carroll, 1967
Carroll, Arthur B. and Wetherald, Richard T., "Application of Parallel Processing to Numerical Weather Prediction," Journal of the Association for Computing Machinery, Volume 14, number 3, July 1967, pp. 591-614.

Clos, 1953
Clos, Charles, "A Study of Non-blocking Switching Networks," Bell System Technical Journal, Volume 32, number 2, March 1953, pp. 406-424.

Dietmeyer, 1975
Dietmeyer, Donald L., Chairman of the 1975 workshop on computer hardware description languages and their applications, verbal communication, 1975.

Downing, 1974
Downing, Robert, Physics Department, University of Illinois at Urbana-Champaign, verbal communication, 1974.

Fox, 1961
Fox, L., Numerical Solution of Ordinary and Parital Differential Equations, Pergamon Press, 1962, p. 348.

Fu, 1968
Fu, K. S., Sequential Methods in Pattern Recognition and Machine Learning, Academic Press, 1968.

Garcia, 1974
Carcia, G., Department of Computer Science, University of Illinois at Urbana-Champaign, unpublished communication, 1974.

Gates, 1975
Gates, W. L., RAND Corporation, oral communication, 1975.

George, 1971
George, Theodore A., "ERTS A and B - The Engineering Systems," Astronautics and Aeronautics, Volume 9, number 4, April 1971, pp. 41-51.

Halem, 1974
Halem, M., Goddard Institute for Space Studies, oral communication, 1974.

Hamming, 1950
Hamming, Richard W., "Error Detecting and Correcting Codes," Bell System Technical Journal, Volume 29, April 1950, pp. 147-160.

Hnatek, 1973
Hnatek, Eugene R., A User's Handbook of Integrated Circuits, John Wiley and Sons, 1974.

IBM, 1970
IBM System/360 Principles of Operation, file number S360-01, order number GA22-6821, version 8, 1970, pp. 41-42.

Karn, 1974
Karn, Ronald, Goddard Institute for Space Studies, oral communication, 1974.

Karn, 1975
Karn, Ronald, GISS Model ILLIAC Implementation, Computer Science Corporation, 1975.

Kasahara, 1967
Kasahara, A. and Washington, W. M., "NCAR Global General Circulation Model of the Atmosphere", Monthly Weather Review, Volume 95, number 7, July 1967, pp. 389-402.

Knuth, 1968
Knuth, Donald E., The Art of Computer Programming, Volume 1: Fundamental Algorithms, Addison-Wesley, 1968.

Kuck, 1968
Kuck, David J., "ILLIAC IV Software and Application Programming," IEEE Transation on Computers, Volume 17, August 1968, pp. 758-770.

Lawrie, 1973
Lawrie, D. H., "Memory-Processor Connection Network," PhD Thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, report number UIUCDCS-R-73-557, February 1973.

Ledley, 1960
Ledley, Robert S., Digital Computer and Control Engineering, McGraw-Hill, 1960, pp. 519-525.

Lorentz, 1963
Lorentz, E. N., "The Predictability of Hydrodynamic Flow," Transactions of the New York Academy of Science, 1963, serial 2, pp. 409-432.

Manabe, 1969
Manabe, S. and Bryan, K., "Climate Calculations with a Combined Ocean-Atmospheric Model," Journal of the Atmospheric Sciences, Volume 26, number 4, July 1969, pp. 786-789.

Mintz, 1974
Mintz, Y. and Arakawa, A., Notes distributed at the second workshop on the UCLS general circulation model, March 25-April 4, 1974, Department of Meteorology, Univeristy of California at Los Angeles.

National Semiconductor Corporation, 1974
Digital Integrated Circuits, National Semiconductor Corporation, 1974.

Ray, 1974
Ray, Robert M., Thomas, John and Donovan, Walter E., Implementation of ILLIAC IV Algorithms for Multispectral Image Interpretation, Center for Advanced Computation document number 112, Center for Advanced Computation, University of Illinois at Urbana-Champaign, June 1974.

Semptner, 1974
Semptner, A. J., Department of Meteorology, University of California at Los Angeles, oral communication, 1974.

Signetics Corporation, 1974A
Signetics Digital, Linear, and MOS Data Book, Signetics Corporation, 1974.

Signetics Corporation, 1974B
Signetics Bipolar Junction Isolated TTL Low Power Shottky Integrated Circuit Failure Rates, Signetics Corporation, November 1974.

Slotnick, 1962
Slotnick, Daniel L., "The SOLOMON Computer," Proceedings of the 1962 Fall Joint Computer Conference, Spartan Books, 1962, pp. 97-107.

Slotnick, 1968
Slotnick, Daniel L, et al., "The ILLIAC IV Computer," IEEE Transactions on Computers, Volume 17, August 1968, pp. 746-757.

Smagorinsky, 1963
Smagorinsky, J., "General Circulation Experiments with the Primitive Equations: I. The Basic Experiment," Monthly Weather Review, Volume 91, number 3, March 1963, pp. 99-164.

Somerville, 1974
Somerville, R. J. C., et al., "The GISS Model of the Global Atmosphere," Journal of the Atmospheric Sciences, Volume 31, number 1, January 1974, pp. 84-117.

Stenzel, 1975
Stenzel, William, "A Class of Compact High Speed Parallel Multiplication Schemes," Masters Thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 1975.

Tessler, 1968
Tessler, Larry G. and Enea, H. J., "A Language Design for Concurrent Processes," Proceedings of the 1968 Spring Joint Computer Conference, Thompson Book Company, 1968, pp. 403-408.

Thomas, 1974A
Thomas, John, An ILLIAC IV Algorithm for Cluster Analysis of ERTS-1 Data, Center for Advanced Computation Technical Memorandum Number 17, Center for Advanced Computation, University of Illinois at Urbana-Champaign, May 1974.

Thomas, 1974B
Thomas, John, An ILLIAC IV Algorithm for Statistical Classification of ERTS-1 Data, Center for Advanced Computation Technical Memorandum Number 18, Center for Advanced Computation, University of Illinois at Urbana-Champaign, May 1974.

Texas Instrument Corporation, 1973
The TTL Data Book for Design Engineers, first edition, document number CC-411, Texas Instruments Incorporated, 1973.

Texas Instrument Corporation, 1974
Supplement to the TTL Data Book for Design Engineers, first edition, document number CC-416, Texas Instruments Incorporated, 1974.

Tsang, 1973
Tsang, L. C. and Karn, R., A Documentation of the GISS Nine-Level Atmospheric General Circulation Model, Computer Sciences Corporation, October 1973.

Wacker, 1970
Wacker, Arthur G. and Landgrebe, David A., "Boundaries in Multispectral Imagery by Clustering," presented at the 1970 IEEE Symposium on Adaptive Processes, December 1970.

Williamson, 1973
Williamson, D. L. and Washington W. M., "On the Importance of Precision for Short Range Forecasting and Climate Simulation," Journal of Applied Meteorology, Volume 12, 1973, pp. 1254-1258.

Appendix

The material in this appendix is a sequence of computer printout which gives the complete set of control cards, logic description and control data (STEPs) which were used to test the floating point addition subset of the array processor.

```
//COMPEL EXEC PGM=COMPEL,REGION=154K,PARM='R,ISA(74K)'                  01/00100
//SYSPRINT DD SYSOUT=A                                                  01/00200
//DECK DD DSN=&DECKFOG,UNIT=DISK,DCB=(BLKSIZE=3120,RECFM=FB),           01/00300
// SPACE=(TRK,(5,1)),DISP=(NEW,PASS)                                    01/00400
//*DECK DD SYSOUT=A,DCB=(BLKSIZE=800,RECFM=FB)                          01/00500
//MICRO DD DSN=&MICFOG,UNIT=DISK,DCB=(BLKSIZE=3120,RECFM=FB),           01/00600
// SPACE=(TRK,(5,1)),DISP=(NEW,PASS)                                    01/00700
//*MICRO DD SYSOUT=A,DCB=(BLKSIZE=800,RECFM=FB)                         01/00800
//PLIDUMP DD SYSOUT=A                                                   01/00900
```

```
$ THIS LOGIC TESTS THE "A" FRACTION FOR ZERO                    02/00100
ATEST(1) : S260 A(1,4) ;                                        02/00200
ATEST(2) : S260 A(5,4) ;                                        02/00300
ATEST(3) : S260 A(9,4) ;                                        02/00400
ATEST(4) : S260 A(13,4) ;                                       02/00500
ATEST(5) : S260 A(17,4) ;                                       02/00600
ATEST(6) : S260 A(21,4) ;                                       02/00700
ATEST(7) : S260 A(25,4) ;                                       02/00800
ATEST(8) : S260 A(29,4) ;                                       02/00900
AZERO : S133 ATEST(1,8) ;                                       02/01000
10 : OUTPUT AZERO BZERO ;                                       02/01100
20 : OUTPUT ATEST(1,8) ;                                        02/01200
```

```
$ THIS LOGIC TESTS THE "B" FRACTION FOR ZERO                    03/00100
BTEST(1) : S260 B(1,4) ;                                        03/00200
BTEST(2) : S260 B(5,4) ;                                        03/00300
BTEST(3) : S260 B(9,4) ;                                        03/00400
BTEST(4) : S260 B(13,4) ;                                       03/00500
BTEST(5) : S260 B(17,4) ;                                       03/00600
BTEST(6) : S260 B(21,4) ;                                       03/00700
BTEST(7) : S260 B(25,4) ;                                       03/00800
BTEST(8) : S260 B(29,4) ;                                       03/00900
BZERO : S133 BTEST(1,8) ;                                       03/01000
20 : OUTPUT BTEST(1,8) ;                                        03/01100
```

```
$ THIS LOGIC CONTROLS THE ALIGNMENT SHIFTING                          04/00100
ASHSEL : S20 EXC2 AZERO BZERO SHZERO ;                                04/00200
BSHSEL : S20 EXC2BAR AZERO BZERO SHZERO ;                             04/00300
: OUTPUT SHZERO ;                                                     04/00400
ASHIFT(1,3) : S157 ABS(5,3) ZEROS(1,3) ASHSEL ZERO ;                  04/00500
BSHIFT(1,3) : S157 ABS(5,3) ZEROS(1,3) BSHSEL ZERO ;                  04/00600
                                                                      04/00700
GTR8 : S260 ABS(1,4) ;                                                04/00800
ENASH : S51 GTR8 AINH ZERO ZERO ;                                     04/00900
ENBSH : S51 GTR8 BINH ZERO ZERO ;                                     04/01000
20 : OUTPUT GTR8 ENASH ENBSH ;
```

```
$ THIS LOGIC COMPARES THE "A" AND "B" FRACTIONS                          05/00100
ALESS1(1) UNUSED AGTR1(1) : S85 A(4,4) B(4,4) B(8) ZERO A(8) ;          05/00200
ALESS1(2) UNUSED AGTR1(2) : S85 A(9,4) B(9,4) B(13) ZERO A(13) ;        05/00300
ALESS1(3) UNUSED AGTR1(3) : S85 A(14,4) B(14,4) B(18) ZERO A(18) ;      05/00400
ALESS1(4) UNUSED AGTR1(4) : S85 A(19,4) B(19,4) B(23) ZERO A(23) ;      05/00500
ALESS1(5) UNUSED AGTR1(5) : S85 A(24,4) B(24,4) B(28) ZERO A(28) ;      05/00600
ALESS1(6) ABEQ1 AGTR1(6) : S85 A(29,4) B(29,4) ZERO ONE ZERO ;          05/00700
ALESS2 ABEQ2 AGTR2 : S85 AGTR1(2,4) ALESS1(2,4)                         05/00800
                ALESS1(6) ABEQ1 AGTR1(6) ;                              05/00900
AHIGH(1,4) : FORM A(1,3) AGTR1(1) ;                                     05/01000
BHIGH(1,4) : FORM B(1,3) ALESS1(1) ;                                    05/01100
ALESS ABEQ AGTR : S85 AHIGH(1,4) BHIGH(1,4) ALESS2 ABEQ2 AGTR2 ;        05/01200
: OUTPUT A(1,32) B(1,32) ;                                              05/01300
10 : OUTPUT ALESS ABEQ AGTR ;                                           05/01400
20 : OUTPUT AHIGH(1,4) BHIGH(1,4) ;                                     05/01500
20 : OUTPUT ALESS1(1,6) ABEQ1 AGTR1(1,6) ;                              05/01600
20 : OUTPUT ALESS2 ABEQ2 AGTR2 ;                                        05/01700
```

```
$ THIS LOGIC PRODUCES THE ADDER FUNCTION FOR ADD AND SUBTRACT          06/00100
$   THE PRIMARY MEANS FOR THIS THE THE SIG8205 ROM                     06/00200
JUNK(1,5) : FORM AZERO BZERO EXC2BAR CUADD CUSUB ;                     06/00300
JUNK(6,4) : FORM EXPA(1) EXPB(1) AGTR ABEQ ;                           06/00400
ADDADDR(1,9) : FORM JUNK(1,5) JUNK(6,4) ;                              06/00500
XX(1,4) : S02 ABS(1,4) ABS(4,4) ;                                      06/00600
ABEXEQ : S20 XX(1) XX(2) XX(3) XX(4) ;                                 06/00700
ADDCNTL(1,8) : SIG8205 ADDADDR(1,9) ;                                  06/00800
AFUNC1(1,4) : S257 ADDCNTL(1,4) ADDCNTL(5,4) ABEXEQ ENABADD ;          06/00900
AFUNC(1,3) : WOR AFUNC1(1,3) CUAFUNC(1,3) ;                            06/01000
10 : OUTPUT ADDADDR(1,9) ;                                             06/01100
5 : OUTPUT ADDCNTL(1,8) ;                                              06/01200
20 : OUTPUT ABEXEQ ;                                                   06/01300
10 : OUTPUT AFUNC1(1,3) ;                                              06/01400
: OUTPUT CUAFUNC(1,3) CUADD CUSUB ;                                    06/01500
SIGN : S157 EXPB(1) AFUNC1(4) NINH ZERO ;                              06/01600
: OUTPUT SIGN ;                                                        06/01700
```

```
$ THIS IS THE "A" ALIGNMENT SHIFTING LOGIC                          07/00100
: OUTPUT AINH ;                                                     07/00200
LEFT(1,8,4) : SIG8243 A(1,8,4) ASHIFT(1,3)                          07/00300
                ENASH ONE ONE ;                                     07/00400
LEFT(2,8,4) : SIG8243 A(2,8,4) ASHIFT(1,3)                          07/00500
                ENASH ONE ONE ;                                     07/00600
LEFT(3,8,4) : SIG8243 A(3,8,4) ASHIFT(1,3)                          07/00700
                ENASH ONE ONE ;                                     07/00800
LEFT(4,8,4) : SIG8243 A(4,8,4) ASHIFT(1,3)                          07/00900
                ENASH ONE ONE ;                                     07/01000
5 : OUTPUT LEFT(1,32) ASHIFT(1,3) ;                                 07/01100
```

```
$ THIS IS THE "B" ALIGNMENT SHIFTING LOGIC                          08/00100
ARIGHT(1,8,4) : SIG8243 B(1,8,4) BSHIFT(1,3)                        08/00200
                ENBSH ONE ONE ;                                     08/00300
ARIGHT(2,8,4) : SIG8243 B(2,8,4) BSHIFT(1,3)                        08/00400
                ENBSH ONE ONE ;                                     08/00500
ARIGHT(3,8,4) : SIG8243 B(3,8,4) BSHIFT(1,3)                        08/00600
                ENBSH ONE ONE ;                                     08/00700
ARIGHT(4,8,4) : SIG8243 B(4,8,4) BSHIFT(1,3)                        08/00800
                ENBSH ONE ONE ;                                     08/00900
5 : OUTPUT ARIGHT(1,32) BSHIFT(1,3) ;                               08/01000
```

```
$ THIS IS THE LEFT SHIFT LOGIC USED IN NORMALIZATION                    09/00100
: OUTPUT NSHIFT(1,3) NINH ;                                             09/00200
NSHIFT(1,3) : S157 NSH(1,3) ZEROS(1,3) ZFF ZERO ;                       09/00300
NSH(1,3) : TI148 BTEST(1,8) ;                                           09/00400
NORM(1,8,4) : SIG8243L B(1,8,4) NSHIFT(1,3)                             09/00500
                NINH ONE ONE ;                                          09/00600
NORM(2,8,4) : SIG8243L B(2,8,4) NSHIFT(1,3)                             09/00700
                NINH ONE ONE ;                                          09/00800
NORM(3,8,4) : SIG8243L B(3,8,4) NSHIFT(1,3)                             09/00900
                NINH ONE ONE ;                                          09/01000
NORM(4,8,4) : SIG8243L B(4,8,4) NSHIFT(1,3)                             09/01100
                NINH ONE ONE ;                                          09/01200
10 : OUTPUT NORM(1,32) NSHIFT(1,3) ;                                    09/01300
10 : OUTPUT NSH(1,3) ;                                                  09/01400
```

```
RIGHT(1,32) : WAND ARIGHT(1,32) NORM(1,32) ;                    10/00100
5 :OUTPUT RIGHT(1,32) ;                                         10/00200
```

```
$ THIS IS THE PRIMARY EXPONENT ADDER                                                    11/00100
: OUTPUT EXPA(1,8) EXPB(1,8) ;                                                          11/00200
AEXPO(1,8) : FORM ZERO EXPA(2,7) ;                                                      11/00300
AEXSTR : S11 ZFF AEXSTRC ONE ;                                                          11/00400
AEXP(1,5) : S157 AEXPO(1,5) ZEROS(1,5) EX157 ZERO ;                                     11/00500
AEXP(6,3) : S157 AEXPO(6,3) NSHIFT(1,3) EX157 AEXSTR ;                                  11/00600
10 : OUTPUT AEXPO(1,8) ;                                                                11/00700
: OUTPUT EX157 ;                                                                        11/00800
BEXP(1,8) : FORM ZERO EXPB(2,7) ;                                                       11/00900
XORSIGN : S86 EXPA(1) EXPB(1) ;                                                         11/01000
: OUTPUT EXCARRY ;                                                                      11/01100
BAFUNC(1,3) : FORM ZEROS(1,2) ONE ;                                                     11/01200
ABG(2) ABP(2) : S381GP AEXP(5,4) BEXP(5,4) ABFUNC(1,3) ;                                11/01300
ABG(1) ABP(1) : S381GP AEXP(1,4) BEXP(1,4) ABFUNC(1,3) ;                                11/01400
BAG(2) BAP(2) : S381GP AEXP(5,4) BEXP(5,4) BAFUNC(1,3) ;                                11/01500
BAG(1) BAP(1) : S381GP AEXP(1,4) BEXP(1,4) BAFUNC(1,3) ;                                11/01600
EX1(1,4) : S381 AEXP(1,4) BEXP(1,4) ABFUNC(1,3) FABC4 ;                                 11/01700
EX1(5,4) : S381 AEXP(5,4) BEXP(5,4) ABFUNC(1,3) EXCARRY ;                               11/01800
EXBA(1,4) : S381 AEXP(1,4) BEXP(1,4) BAFUNC(1,3) FBAC4 ;                                11/01900
EXBA(5,4) : S381 AEXP(5,4) BEXP(5,4) BAFUNC(1,3) ONE ;                                  11/02000
ABS(1,7) : S157 EXBA(2,7) EX1(2,7) EXC2BAR ZERO ;                                       11/02100
UNUSED UNUSED FBAC4 EXC2 UNUSED : S182 ONE FBAG(1,4) FBAP(1,4) ;                        11/02200
UNUSED UNUSED FABC4 EXC2BAR UNUSED : S182 EXCARRY FABG(1,4) FABP(1,4) ;                 11/02300
FABG(1,4) : FORM ONES(1,2) ABG(1,2) ;                                                   11/02400
FABP(1,4) : FORM ONES(1,2) ABP(1,2) ;                                                   11/02500
FBAG(1,4) : FORM ONES(1,2) BAG(1,2) ;                                                   11/02600
FBAP(1,4) : FORM ONES(1,2) BAP(1,2) ;                                                   11/02700
5 : OUTPUT ABS(1,7) EX1(1,8) EXBA(1,8) ;                                                11/02800
: OUTPUT ABFUNC(1,3) ;                                                                  11/02900
5 : OUTPUT AEXP(1,8) BEXP(1,8) ;                                                        11/03000
10 : OUTPUT FABC4 EXC2BAR EXC2 ;                                                        11/03100
10 : OUTPUT FBAC4 ;                                                                     11/03200
15 : OUTPUT ABG(1,2) ABP(1,2) ;                                                         11/03300
15 : OUTPUT BAG(1,2) BAP(1,2) ;                                                         11/03400
: OUTPUT EXCARRY ;                                                                      11/03500
: OUTPUT XORSIGN ;                                                                      11/03600
```

```
$ THIS IS THE 32-BIT FRACTION ADDER                                    12/00100
ENABBAR : S04 ENABADD ;                                                12/00200
AC : H52 ENABADD CUAC ENABBAR AFUNC1(2) AFUNC1(3) ;                    12/00300
: OUTPUT CUAC ;                                                        12/00400
AGH(1) APH(1) : S381GP LEFT(1,4) RIGHT(1,4) AFUNC(1,3) ;               12/00500
AGH(2) APH(2) : S381GP LEFT(5,4) RIGHT(5,4) AFUNC(1,3) ;               12/00600
AGH(3) APH(3) : S381GP LEFT(9,4) RIGHT(9,4) AFUNC(1,3) ;               12/00700
AGH(4) APH(4) : S381GP LEFT(13,4) RIGHT(13,4) AFUNC(1,3) ;             12/00800
AGL(1) APL(1) : S381GP LEFT(17,4) RIGHT(17,4) AFUNC(1,3) ;             12/00900
AGL(2) APL(2) : S381GP LEFT(21,4) RIGHT(21,4) AFUNC(1,3) ;             12/01000
AGL(3) APL(3) : S381GP LEFT(25,4) RIGHT(25,4) AFUNC(1,3) ;             12/01100
AGL(4) APL(4) : S381GP LEFT(29,4) RIGHT(29,4) AFUNC(1,3) ;             12/01200
AG2 AP2 AC4H AC8H AC12H : S182 AC16 AGH(1,4) APH(1,4) ;                12/01300
AG1 AP1 AC4L AC8L AC12L : S182 AC AGL(1,4) APL(1,4) ;                  12/01400
AC16 : S182X AC AG1 AP1 ;                                              12/01500
ACOUT : S182X AC16 AG2 AP2 ;                                           12/01600
SUM(1,4) : S381 LEFT(1,4) RIGHT(1,4) AFUNC(1,3) AC12H ;                12/01700
SUM(5,4) : S381 LEFT(5,4) RIGHT(5,4) AFUNC(1,3) AC8H ;                 12/01800
SUM(9,4) : S381 LEFT(9,4) RIGHT(9,4) AFUNC(1,3) AC4H ;                 12/01900
SUM(13,4) : S381 LEFT(13,4) RIGHT(13,4) AFUNC(1,3) AC16 ;              12/02000
SUM(17,4) : S381 LEFT(17,4) RIGHT(17,4) AFUNC(1,3) AC12L ;             12/02100
SUM(21,4) : S381 LEFT(21,4) RIGHT(21,4) AFUNC(1,3) AC8L ;              12/02200
SUM(25,4) : S381 LEFT(25,4) RIGHT(25,4) AFUNC(1,3) AC4L ;              12/02300
SUM(29,4) : S381 LEFT(29,4) RIGHT(29,4) AFUNC(1,3) AC ;                12/02400
: OUTPUT AC ;                                                          12/02500
5 : OUTPUT LEFT(1,32) RIGHT(1,32) AFUNC(1,3) ;                         12/02600
15 : OUTPUT AGH(1,4) APH(1,4) ;                                        12/02700
15 : OUTPUT AGL(1,4) APL(1,4) ;                                        12/02800
5 : OUTPUT SUM(1,32) ;                                                 12/02900
15 : OUTPUT AC4H AC8H AC12H ;                                          12/03000
15 : OUTPUT AC4L AC8L AC12L ;                                          12/03100
5 : OUTPUT ACOUT ;                                                     12/03200
15 : OUTPUT AC16 AG1 AG2 AP1 AP2 ;                                     12/03300
```

```
$ THIS LOGIC RESPONDS TO FRACTION ADDITION OVERFLOWS                    13/00100
$ IT "SHIFTS" THE FRACTION ONE DIGIT TO THE RIGHT ON OVERFLOW           13/00200
OVFL(1,32) : FORM ONES(1,3) ZERO SUM(1,28) ;                            13/00300
FRACT(1,32) : S158 OVFL(1,32) SUM(1,32) OVFLSEL ZERO ;                  13/00400
OVFLCON(1,8) : FORM ONES(1,3) ACOUT ONES(1,4) ;                         13/00500
OVFLSEL : S151 OVFLCON(1,8) AFUNC1(1,3) ;                               13/00600
5 : OUTPUT OVFLSEL ;                                                    13/00700
5 : OUTPUT OVFL(1,32) ;                                                 13/00800
: OUTPUT FRACT(1,32) ;                                                  13/00900
```

```
$ THIS IS THE EXPONENT CORRECTION ADDER, WHICH INCREMENTS                14/00100
$ THE EXPONENT BY ONE WHEN A FRACTION OVERFLOW OCCURS                    14/00200
EXSELIN(1,8) : FORM ONE ZERO ONES(1,2) ZEROS(1,2) ONE ZERO ;             14/00300
EXCNTRL(1,3) : FORM EXC2 AZERO BZERO ;                                   14/00400
EXSEL : S151 EXSELIN(1,8) EXCNTRL(1,3) ;                                 14/00500
10 : OUTPUT EXSEL ;                                                      14/00600
100 : OUTPUT EXCNTRL(1,3) EXSELIN(1,8) ;                                 14/00700
EX3TO1L : S20 EXSEL EXP1 ONE ONE ;                                       14/00800
EX3TO1C(1,2) : FORM EX3TO1H EX3TO1L ;                                    14/00900
EX3TO1(1,8) : SIG8263 EX1(1,8) AEXPO(1,8) BEXP(1,8)                      14/01000
                EX3TO1C(1,2) ZERO ;                                      14/01100
20 : OUTPUT EX3TO1(1,8) EXPSUM(1,8) ;                                    14/01200
: OUTPUT EX3TO1H EXP1 ;                                                  14/01300
10 : OUTPUT EX3TO1C(1,2) ;                                               14/01400
20 : OUTPUT EX3TO1L ;                                                    14/01500
EXPSUM(1,4) UNUSED : S181 EX3TO1(1,4) ZEROS(1,4) CORRCRY                 14/01600
                ZEROS(1,4) ZERO ;                                        14/01700
EXPSUM(5,4) CORRCRY : S181 EX3TO1(5,4) ZEROS(1,4) ZERO                   14/01800
                ZEROS(1,4) ZERO ;                                        14/01900
EXP(1,8) : S157 EXPSUM(1,8) EX3TO1(1,8) OVFLSEL ZERO ;                   14/02000
30 : OUTPUT CORRCRY ;                                                    14/02100
: OUTPUT EXP(2,7) ;                                                      14/02200
```

```
$ THIS LOGIC TESTS THE RESULT FRACTION FOR ZERO, AND SETS                15/00100
$ THE ZERO FLIP-FLOP ACCORDING TO THE RESULT OF THE TEST                 15/00200
ZFFBITS(1) : S260 FRACT(1,4) ;                                           15/00300
ZFFBITS(2) : S260 FRACT(5,4) ;                                           15/00400
ZFFBITS(3) : S260 FRACT(9,4) ;                                           15/00500
ZFFBITS(4) : S260 FRACT(13,4) ;                                          15/00600
ZFFBITS(5) : S260 FRACT(17,4) ;                                          15/00700
ZFFBITS(6) : S260 FRACT(21,4) ;                                          15/00800
ZFFBITS(7) : S260 FRACT(25,4) ;                                          15/00900
ZFFBITS(8) : S260 FRACT(29,4) ;                                          15/01000
ZFFINBAR : S133 ZFFBITS(1,8) ;                                           15/01100
ZFFIN : S04 ZFFINBAR ;                                                   15/01200
*ZFF *ZFFBAR : S74 ZFFIN CLOCK ;                                         15/01300
10 : OUTPUT ZFFIN ;                                                      15/01400
10 : OUTPUT ZFFINBAR ;                                                   15/01500
20 : OUTPUT ZFFBITS(1,8) ;                                               15/01600
: OUTPUT ZFF ZFFBAR CLOCK ZFFIN ;                                        15/01700
```

```
//DECK EXEC ASSEMBLY,PARM='FX,ESD,LSETC=12',REGION=180K                16/00100
//SYSLIB DD DSN=USER.P4293.SUPPORT,DISP=SHR                            16/00200
// DD DSN=USER.P4293.PACKAGES,DISP=SHR                                 16/00300
// DD DSN=SYS1.MACLIB,DISP=SHR                                         16/00400
//SYSIN DD DSN=&DECKFOG,DISP=(OLD,DELETE)                              16/00500
```

```
//LINKDECK EXEC LINKEDIT,PARM='LIST,MAP,NCAL,LET',REGION=102K,          17/00100
// LOADSET='USER.P4293.LINKOUT(LOGFOG)'                                 17/00200
//SYSLIB DD DSN=USER.P4293.LINKOUT,DISP=SHR                             17/00300
//SYSLMOD DD DISP=OLD,SPACE=(TRK,(10,3,10))                             17/00400
```

```
//MICRO EXEC ASSEMBLY,PARM='NOXREF,NOLREF,ESD',REGION=180K                18/00100
//SYSLIB DD DSN=USER.P4293.SPECIAL,DISP=SHR                               13/00200
// DD DSN=USER.P4293.MICRO,DISP=SHR                                       18/00300
// DD DSN=USER.P4293.SUPPORT,DISP=SHR                                     18/00400
// DD DSN=USER.P4293.PACKAGES,DISP=SHR                                    18/00500
// DD DSN=SYS1.MACLIB,DISP=SHR                                            18/00600
//SYSIN DD DSN=&MICFOG,DISP=(OLD,DELETE)                                  18/00700
// DD *                                                                   18/00800
```

```
PRINT NOGEN                                                                    19/00100
STEP AEXSTRC=0,CUAC=0,EX3TO1H=1,CLOCK=1,AINH=1,BINH=1                          19/00200
STEP SHZERO=1,NINH=1,CUAFUNC=000,CUADD=1,CUSUB=0,EX157=0                       19/00300
STEP ENABADD=0,EXP1=1,ABFUNC=010,EXCARRY=1                                     19/00400
STEP EXPA=01001000,EXPB=01001000                                               19/00500
STEP A=X0,B=X0                                                                 19/00600
RUN                                                                            19/00700
STEP AEXSTRC=1,CUAC=1,EX3TO1H=0,CLOCK=0,NINH=0                                 19/00800
STEP CUAFUNC=011,ENABADD=1,B=X0,EXP1=0,ABFUNC=011                              19/00900
STEP EXPB=*EXP,AINH=0,BINH=0,EX157=1,EXCARRY=0                                 19/01000
RUN                                                                            19/01100
STEP AEXSTRC=0,CUAC=0,EX3TO1H=1,CLOCK=1,AINH=1,BINH=1                          19/01200
STEP SHZERO=1,NINH=1,CUAFUNC=000,CUADD=1,CUSUB=0,EX157=0                       19/01300
STEP ENABADD=0,EXP1=1,ABFUNC=010,EXCARRY=1                                     19/01400
STEP CUADD=0                                                                   19/01500
RUN                                                                            19/01600
STEP CUSUB=1                                                                   19/01700
RUN                                                                            19/01800
STEP A=X80000000                                                               19/01900
RUN                                                                            19/02000
STEP B=X50000000                                                               19/02100
RUN                                                                            19/02200
STEP EXPA=01000010                                                             19/02300
RUN                                                                            19/02400
STEP EXPA=01001000                                                             19/02500
STEP CUSUB=0                                                                   19/02600
RUN                                                                            19/02700
STEP B=X80000058                                                               19/02800
RUN                                                                            19/02900
STEP AEXSTRC=1,CUAC=1,EX3TO1H=0,CLOCK=0,NINH=0                                 19/03000
STEP CUAFUNC=011,ENABADD=1,B=X58,EXP1=0,ABFUNC=011                             19/03100
STEP EXPB=*EXP,AINH=0,BINH=0,EX157=1,EXCARRY=0                                 19/03200
RUN                                                                            19/03300
STEP AEXSTRC=0,CUAC=0,EX3TO1H=1,CLOCK=1,AINH=1,BINH=1                          19/03400
STEP SHZERO=1,NINH=1,CUAFUNC=000,CUADD=1,CUSUB=0,EX157=0                       19/03500
STEP ENABADD=0,EXP1=1,ABFUNC=010,EXCARRY=1                                     19/03600
STEP EXPB=01000111                                                             19/03700
RUN                                                                            19/03800
STEP CUSUB=1                                                                   19/03900
RUN                                                                            19/04000
STEP CUADD=1                                                                   19/04100
RUN                                                                            19/04200
STEP EXPB=01001000                                                             19/04300
RUN                                                                            19/04400
STOP                                                                           19/04500
END                                                                            19/04600
```

```
//LINKMIC EXEC LINKEDIT,PARM='LIST,MAP,LET',REGION=102K,                20/00100
// LOADSET='USER.P4293.LINKOUT(MICFOG)'                                 20/00200
//SYSLIB DD DSN=USER.P4293.LINKOUT,DISP=SHR                             20/00300
//SYSLMOD DD DISP=OLD,SPACE=(TRK,(10,3,10))                             20/00400
```

```
//LINKSIM EXEC LINKEDIT,PARM='LIST,MAP,LET',REGION=102K,                21/00100
// LOADSET='USER.P4293.LINKOUT(TESTFOG)'                                21/00200
//SYSLIB DD DSN=USER.P4293.LINKOUT,DISP=SHR                             21/00300
//SYSLIN DD *                                                           21/00400
 ENTRY PROGRAM                                                         21/00500
 INCLUDE SYSLIB(MICFOG,LOGFOG)                                         21/00600
//SYSLMOD DD DISP=OLD,SPACE=(TRK,(10,3,10))                             21/00700
```

```
//RUN EXEC PGM=TESTFOG,REGION=32K,TIME=(,10),PARM='255'                22/00100
//SYSPRINT DD SYSOUT=A                                                 22/00200
//SYSUDUMP DD SYSOUT=A                                                 22/00300
```

```
//RUN EXEC PGM=TESTFOG,REGION=32K,TIME=(,10),PARM='0'                23/00100
//SYSPRINT DD SYSOUT=A                                               23/00200
//SYSUDUMP DD SYSOUT=A                                               23/00300
```

BIBLIOGRAPHIC DATA SHEET	1. Report No. UIUCDCS-R-75-761	2.	3. Recipient's Accession No.

4. Title and Subtitle

AN ARRAY COMPUTER FOR THE CLASS OF PROBLEMS TYPIFIED BY THE GENERAL CIRCULATION MODEL OF THE ATMOSPHERE

5. Report Date: December 1975

6.

7. Author(s)

MARVIN L. GRAHAM and D. L. SLOTNICK

8. Performing Organization Rept. No. UIUCDCS-R-75-761

9. Performing Organization Name and Address

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801

10. Project/Task/Work Unit No.

11. Contract/Grant No. US NASA NAS-5-23334

12. Sponsoring Organization Name and Address

NASA Goddard Space Flight Center
2880 Broadway
New York, NY 10025

13. Type of Report & Period Covered Doctoral and Final Report 1975

14.

15. Supplementary Notes

16. Abstracts

The goal of this research was the design of a computer suited to the class of problems typified by the general circulation model of the atmosphere. The needs that prompted the research imposed several constraints on the design which was sought. Primary among these was that the new machine has roughly 100 times the computing capability of the IBM 360/95 which is now used in general circulation model research. Of equal importance, the cost of the machine was to be significantly less than that of extant machines with similar computing capability.

The design which is presented is that of an array processor similar in architecture to ILLIAC IV. The design is in terms of commercially available TTL and ECL small and medium scale integrated circuits. We believe that it achieves the cost and performance goals set for it.

17. Key Words and Document Analysis. 17a. Descriptors

array computer, parallel computer, general circulation model, switching network, Clos network, omega network, logic simulation, simulation of computer logic

17b. Identifiers/Open-Ended Terms

17c. COSATI Field/Group

18. Availability Statement: Unlimited

19. Security Class (This Report) UNCLASSIFIED

20. Security Class (This Page) UNCLASSIFIED

21. No. of Pages 298

22. Price