LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

Design of an On-line Byte-level Pipelined Arithmetic Processor

Computation Structures Group Memo 162
July 1978

Arif M. Feridun

Thesis submitted in partial fulfillment of the requirements for the degree of Bachelor of Science at M.I.T.

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

DESIGN OF AN ON-LINE BYTE-LEVEL PIPELINED ARITHMETIC PROCESSOR

by

Arif Metin Feridun

Submitted to the Department of Electrical Engineering and Computer Science on April 28, 1978, in partial fulfillment of the requirements for the Degree of Bachelor of Science.

ABSTRACT

Floating point addition-subtraction and multiplication units using signed-digit arithmetic are presented. Signed-digit arithmetic allows algorithms where carry generated during addition propogates only as far as the adjacent digits. Therefore the processors are byte-level pipelined, capable of producing digits of the result of an operation without having to wait for the arrival of the whole operands. The processors described are compatible with the Data Flow Machine concepts.

Thesis supervisor: Jack B. Dennis

Title: Professor of Computer Science and Engineering

TABLE OF CONTENTS

1. Introduction 9

2. Number Representation 11

2.1 Signed Digit Number Representation 11

2.2 Number Representation used in the design 14

2.3 Special Operands 15

3. Algorithms 17

3.1 Normalization 17

3.2 Addition and Subtraction 19

3.3 Multiplication Algorithm 21

4. Floating Point Adder Subtractor 25

4.1 General outline 25

4.2 MPX Module 26

4.3 EXPFIX Module 27

4.4 MODOP Module 30

4.5 ADDOP Module 33

4.6 NORMOP Module 36

4.7 Control of the FPAS 39

5. Floating Point Multiplier 43

5.1 General Description of FPM 43

5.2 EXOP Module 43

5.3 MULTOP Module 46

5.4 PACKOP Module 51

5.5 Control and Timing 55

6. Conclusion 58

Appendix: Petri Net Graphs 60

Bibliography 61

LIST OF ILLUSTRATIONS

Figure 1.1	Data Flow Machine
Figure 2.1	Signed Digit Addition
Figure 2.2	Machine representation used in the design
Figure 3.1	Parallel Signed Digit Adder
Figure 3.2	Double digit parallel adder modified for byte level computation
Figure 3.3	Signed Digit Multiplication
Figure 4.1	General Structure of FPAS
Figure 4.2(a)	MPX module structure
Figure 4.2(b)	DEL structure
Figure 4.3	EXPFIX Module
Figure 4.4	MODOP Module
Figure 4.5	Wrap-around FIFO QUEUE
Figure 4.6	Operation of the "delay" mechanism
Figure 4.7	ADDOP Module
Figure 4.8	NORMOP Module
Figure 4.9	Arrangement of the SOP units
Figure 4.10	STREG Register
Figure 4.11	Control Signals
Figure 4.12(a)	Timing
Figure 4.12(b)	Overlapping
Figure 5.1	General structure of the FPM
Figure 5.2	EXOP Module
Figure 5.3	MULTOP Module

Figure 5.4	OPNDST Structure
Figure 5.5	MULTSEL Structure
Figure 5.6	Configuration to improve size of ROMs
Figure 5.7	PACKOP Module
Figure 5.8	PACK Module
Figure 5.9	Main control signals
Figure 5.10(a)	Operation times
Figure 5.10(b)	Overlapping

CHAPTER 1: INTRODUCTION

The aim of this thesis is to design an arithmetic processor compatible with the architecture of the Data Flow Machine (DFM). However before explaining this "compatability", it is necessary to describe briefly the idea of the DFM.

The DFM is a processor which executes programs written in a data flow language. The latter permits highly parallel, asynchronous processing ordered strictly by the arrival of data. In other words, the only necessary condition for the execution of an operation is the availability of its operands. Description of these concepts and the architecture of the Data Flow Machine can be found in [DM 1], [DML 1] and [DML 2].

An elementary data flow machine[1] is shown in Fig.1.1. It consists of four basic parts as follows:

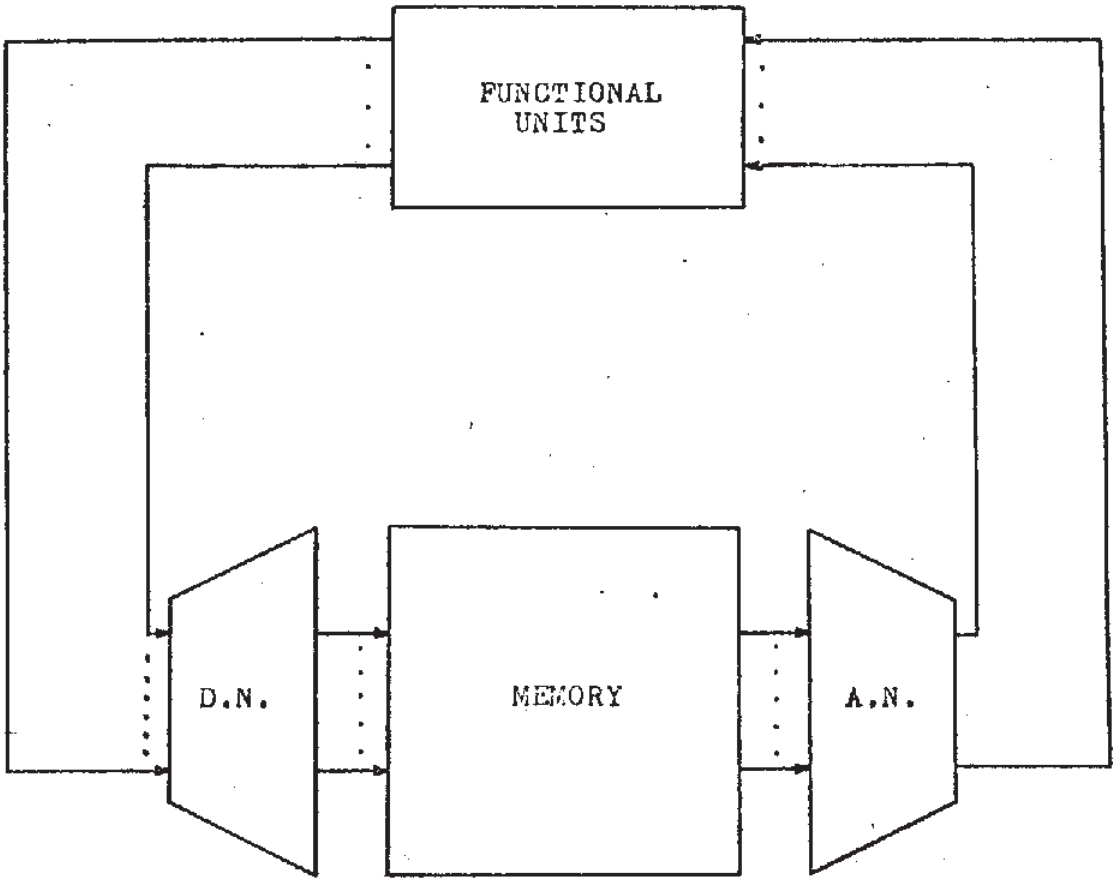
i. MEMORY: This unit is divided into *cell-blocks* which are also further divided into *instruction cells*. An *instruction cell*, in general, contains the operands, information about the operation (i.e. the operation code) and "state" information such as "all operands ready" and the like. A mechanism in the *cell-block* checks the state informations to decide whether a given operation packet (operation code + operands + destination address of the result) should be sent to the processing section of the processor to be evaluated.

ii. TRANSMISSION NETWORKS: These control the inflow (distribution network) and outflow (arbitration network) of packets to/from the memory block. Their basic responsibility is to allow orderly flow of packets.

iii. FUNCTIONAL UNITS: These units perform various operations on the arriving packets. In other words, they are the "processing" units.

As indicated above, the data flow within the processor occurs in terms of packet-flow.

1. *See* [DML 1]



A.N.: Arbitration network
D.N.: Distribution network

Fig.I.1 Data Flow Machine

A proposed packet format consists of a group of bytes (8 bits each) traveling sequentially along byte-width channels. Hence a convenient way to manipulate or examine these packets is to provide byte-serial operation units.

In this thesis, a possibility for byte-serial arithmetic processor is studied. Namely, two different units, floating point adder-subtractor (FPAS) and floating point multiplier (FPM) are described. The designs do not take into consideration issues such as interfacing with DFM, but compatability is assured by their special properties.

FPAS and FPM use signed digit arithmetic which enable algorithms where i) the operation can begin before the operands are available in complete form, and where ii) the first result digits are produced (most significant first) after a certain number of result digits are available. For example, in addition operation, the most significant result digit is available after the first operand digits arrive. This is made possible by the handy property of signed digit arithmetic that limits carry propogation to adjacent digits.[1] As a result, the processors accept bytes as input, and output bytes, consistent with the structure of data packets in the Data Flow Machine. Pipelining allows a high byte processing rate.

The thesis is organized as follows: In chapter 2, signed digit representation and the basic algorithm (addition) is described. A machine representation following this system is proposed, and this is used throughout the thesis.

In chapter 3, three basic algorithms (normalization, addition-subtraction and multiplication) are defined. Modifications for the purposes of this thesis are explained.

Chapters 4 and 5 describe in detail the floating point adder-subtractor and floating point multiplier respectively. In the conclusion section, chapter 6, various improvement possibilities are discussed.

---

1. *See chapter 2*

CHAPTER 2: NUMBER REPRESENTATION

2.1 Signed Digit Number Representation

Various options for number representations are available for fast arithmetic. *Conventional number representations* such as 2's complement are such that for an arbitrary base r, each digit of a number can have r values, chosen from the digit set {0,1,...,r-1}. These representations have the property that carries generated by the summation of digits can propogate from right to left along the whole number, e.g. 999+1→ 1000. This property limits digit-by-digit computations to representations where the least significant digit is available first; otherwise the result can only be obtained as a whole.

Example 1: 9863+0199

i. two digits at a time, right to left:	ii. two digits at a time, left to right:
63+99= 162	98+01= 99
98+01= 99	63+99= 162
100	10062
Result available in parts	*Result available as a whole*

The goal of this thesis is to design a byte-level pipelined processor with on-line properties, i.e. a processor that would receive operands as bytes and output the result also as bytes, in both cases most significant byte first. Such algorithms exist for signed-digit number representation.

Signed digit number system is a redundant system, i.e. each number can have more than one representation. For a chosen base r, this can be achieved by allowing each digit to assume more than r values. For the purposes of this thesis, a symmetric digit set of 2r-1 elements is chosen: {-a,...,-1,0,1,...,a} where a=r-1. This representation is called maximally

redundant [AV 1], and it is the largest possible digit set for the chosen base.

Example 2: For base 8 arithmetic, the maximally redundant signed digit set $S=\{\bar{7},...,\bar{1},0,1,...,7\}$, while the conventional digit set $A=\{0,1,...,7\}$. Hence A is a subset of S. Using the digit set S, redundancy can be shown:

$0.6432_8=0.7\bar{4}32_8=0.7\bar{4}4\bar{6}_8$

Characteristics of signed-digit numbers are as follows:

i. a signed digit number X is represented by n+m+1 digits x_i (i= -n,...,0,...,m) and $X=\sum_{-n}^{m} x_i r^{-i}$ where r=integer base.

ii. X=0 if and only if all $x_i=0$.

iii. sign(X)= sign of the most significant digit, and

iv. inverse of X, i.e. -X is obtained by changing the sign of each x_i in X.

Since fixed format floating point operations will be used here, representation of the number X can be redefined as consisting of m digits $x_i(i=1,...,m)$ so that $X=\sum_{1}^{m} x_i r^{-i}$. This way there are no digits to the left of the radix point. Now definitions for parallel addition and subtraction are given as follows:

1. Addition of digits z_i and y_i is parallel if

i. sum digit s_i is a function of only z_i, y_i and the transfer digit t_i from the (i+1)th position on the right [Fig.2.1], i.e. $s_i=f(z_i, y_i, t_i)$.

ii. the transfer digit t_i is a function of z_{i+1} and y_{i+1} only.

2. Subtraction is done by negating the subtrahend according to property (iv) above and then adding, so that $z_i - y_i = z_i + \bar{y}_i$.

The transfer digit t_i is the carry generated when the digits are added. Since negative sums can be used, there can be negative carry as well. Therefore t_i can assume $\{\bar{1},0,1\}$ as values.

Interim sum digit ,w_i, is defined to be a sub-sum such that

$z_i + y_i = rt_{i-1} + w_i$ (1) and sum digit $s_i = w_i + t_i$ (2)

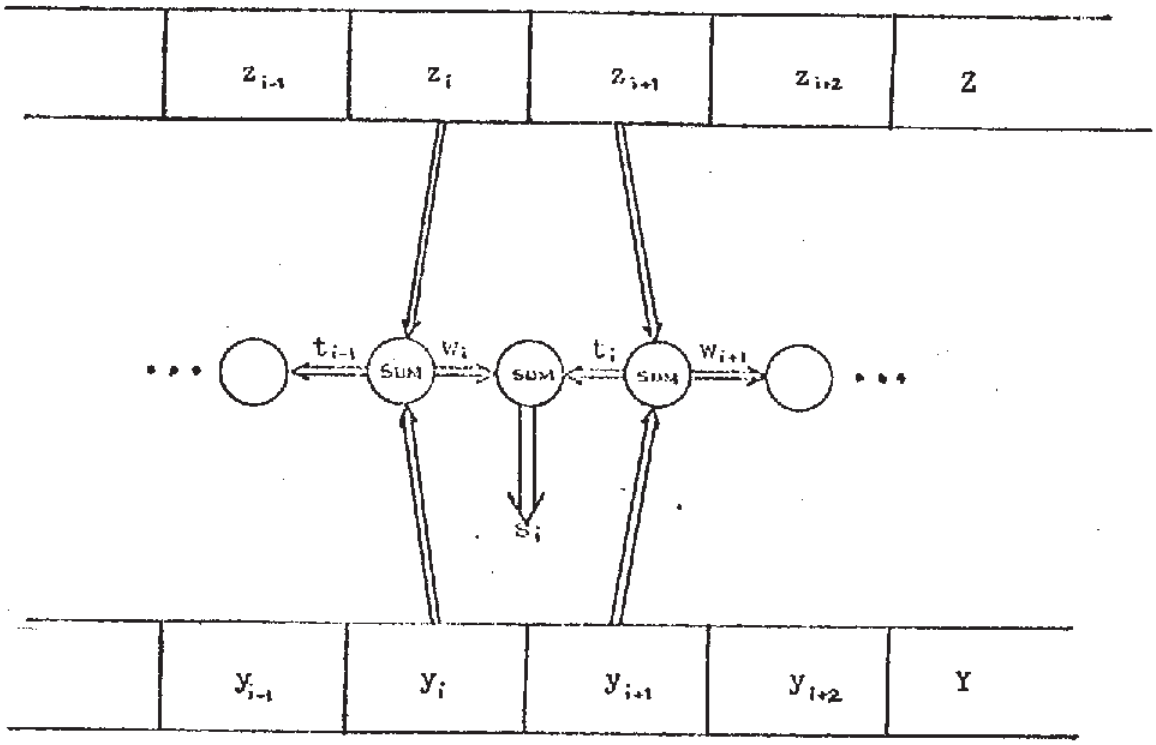


Fig.2.1 Signed Digit Addition

Since $t_i = \{\bar{1},0,1\}$ and since s_i must also be in the same digit set as z_i and y_i (namely $s_i \leq r-1$), then $w_i \leq r-2$, because otherwise (w_i+t_i) will not be in the digit set S (see example 2).

Example 3: Using example 2, $|w_i| \leq r-2 = 8 - 2 = 6$

For $t_i = 1$ and for unallowed value $w_i = 7$,

$s_i = t_i + w_i = 7 + 1 = 10_8$ which is clearly not in S.

So far nothing has been said about the base limit; however because of the restriction on $|w_i|$, it can be seen that r=2 is not allowed. For base 2,

$$|w_i| \leq r-2 = 0$$

If $|w_i|=0$, then there is no t_i to satisfy $z_i + y_i + 1 = 2t_i$ (from eq(1) above). Therefore signed digit representation and algorithms are valid for $r>2$. More is said about S-D addition in section 3.2.

Advantages of using S-D numbers are as follows: 1) since s_i is a function of adjacent digits, carry propogation chains of conventional number representations is eliminated. Since there is no operand width carry, addition and subtraction time is independant of operand precision. 2) Most significant digits can be available before least significant ones and they can be processed further before an operation ends. Hence computations can begin before all of the digits are available, and therefore digit level pipelining is possible for arithmetic operations using S-D number representation.

Disadvantages are seen in the implementation. The adders are more complicated and therefore require more hardware than for example 2's complement addition. S-D adders are discussed further in section 3.2. Because of the digit sets chosen, each digit requires a sign bit and therefore machine representations are larger than in conventional numbers. For example base-8 conventional binary representation would be 0110 where the extra digit stands for the sign. Also roundoff is done by truncation and therefore some accuracy is lost in the computations.

Conversion from S-D representation to conventional form can be done by separating

the positive and negative digits and then subtracting the two numbers:

Example 4: $0.9\bar{7}8\bar{3}4_{10} \rightarrow 0.90804$ (+ digits) $\quad 0.0703$ (- digits)

$0.90804 - 0.0703 = 0.83774_{10}$ (conventional)

Conversion from conventional to S-D form is easier if maximal redundancy is used: in this case, the digit set for conventional number representation is a subset of the signed digit set S (example 2). Therefore for positive numbers no conversion is necessary, while for negative numbers, each digit is negated.

Example 5: Given $.3476_8$, processor uses $.3476_8$

Given $-.3476_8$, processor uses $.\bar{3}\bar{4}\bar{7}\bar{6}_8$

An algorithm for other signed digit sets is suggested in [AV 1].

2.2 Number Representation used in the design

For the arithmetic processor designed here, base-8, fixed format, floating point signed digit representation is used. The digit set chosen is maximally redundant and consists of 15 integers (see example 2). Machine representation is chosen as 16s complement base-8 binary form [AV 2] where each digit occupies 4 bits (Fig.2.2.2). Therefore two digits form an 8-bit byte and the purpose of the design is to achieve a byte-level pipelined, "two-digit-at-a-time" arithmetic processor.

As in all floating point numbers, an exponent and mantissa are required. A sign bit for the whole number is not necessary: the sign of the number is the sign of the most significant digit of the mantissa. The exponent is represented by a binary byte (8 bits): 1 bit is the exponent sign, and 7 bits form the exponent, giving an exponent range of $8^{\pm 127}$ (approx. $5*10^{\pm 114}$). Larger exponents can be obtained by the addition of more bytes as required. Conventional binary representation is used for the exponent because it makes exponent manipulations such as over&underflow detection easier. For example incrementing the exponent can be done using a simple counter. Mantissa format is picked

1. Given base-8 signed digit set

$$S = \{\bar{7},\bar{6},\ldots,\bar{1},0,1,\ldots,6,7\},$$

a possible machine representation (16's complement) :

0	0000		
1	0001	$\bar{1}$	1111
2	0010	$\bar{2}$	1110
3	0011	$\bar{3}$	1101
4	0100	$\bar{4}$	1100
5	0101	$\bar{5}$	1011
6	0110	$\bar{6}$	1010
7	0111	$\bar{7}$	1001

2. Example:

Number to be represented : .7346 E+23
(approx. $+5.387*10^{20}$)

Exponent : 00010111
(sign | exponent)

Mantissa : 0111 | 0011 | 1100 | 1010

Complete operand packet :

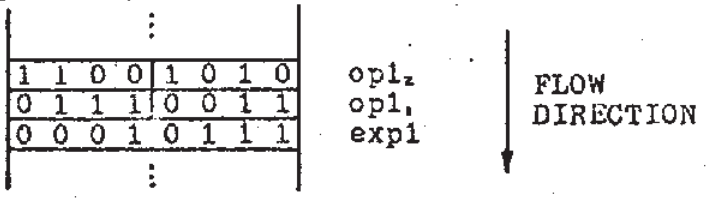


Fig.2.2 Machine representation used in the design

to be 4 digits or 2 bytes. The small number of digits is for clarity; increasing the precision will not change the structure of the processor. Fig.2.2.3 illustrates use of the above format.

As the example shows, operands will follow each other in "packet" form and therefore an operation with two operands will require the delay of the first operand. Also within each packet, the exponent will be the first to arrive. This format has been chosen for simplicity; there are various other formats that can be used for better performance. An efficient arrangement would be to alternate the operand bytes.

Example 6: Given operand-pkt1: exp1 | $op1_1$ | $op1_2$ and operand-pkt2: exp2 | $op2_1$ | $op2_2$ the improved flow arrangement would be

exp1 | exp2 | $op1_1$ | $op2_1$ | $op1_2$ | $op2_2$

In the Data Flow Machine, this format can be achieved by modifying the cell-block structure.

2.3 Special Operands

Various operations result in either error or other special conditions, e.g. exponent overflow, divide by zero. When these are detected, they can either be handled through an error routine, or be unreported and indicated as a special result value (operand). Since the aim is to design a fast processor, error routines are not appropriate due to the fact that in a pipelined asynchronous system, it is hard to find means to report the error [JD 1]. Therefore various special operands are defined:

$\pm\infty$ (infinity for overflow cases)

$\pm\epsilon$ (0^+ & 0^- for underflow cases)

E (error, for indefinite cases)

These operands can be represented by special exponents and since these exponents are processed first, unnecessary operations can be discovered early.

Example 7: For base-8 number format described in section 2.2, one can limit the

exponent range to $8^{\pm 120}$ for example. In this case 8^{+121} would be overflow, while 8^{-121} be underflow. With the remaining 12 possibilities, the following can be done:

exp=+123=$+\infty$

exp=+125=$+\epsilon$

exp= 127= E

When a special operand is detected, normal operation will not be carried out, rather a special operand will be selected and sent out as the result.

Example 8: Let N=normal number. Then

$+\epsilon$ -N= -N 0-(-∞)= $+\epsilon$

E*(-∞)= E 0/(+∞)= 0

This method is used in the Control Data 6600; detailed explanation can be found in [TH 1].

Special operands can also be used or created in case of overflow and underflow occurring after an operation. In such cases the sign of the special operand is chosen to be the sign of the over or underflowed result. More is said about these kinds of errors in section 3.1.

CHAPTER 3: ALGORITHMS

In this chapter normalization, addition-subtraction and multiplication algorithms used in this thesis are described. Modifications done for their compatability with the design aims is explained.

3.1 Normalization

In floating point arithmetic, normalization is basically the adjustment of a result to a specified format. A normalized number is such that the most significant digit of its mantissa is non-zero, i.e. for mantissa m and base r,

$$r^{-1} \leq |m| < 1$$

An exception to this rule is the zero mantissa (the number 0).

Usually, in machine arithmetic involving conventional number representations, the result is ready as a whole and the normalization is done as follows:

i. if there is mantissa overflow then right shift the mantissa 1 digit; increment the exponent, check for overflow. If there is no overflow, pack exponent and mantissa according to format.

ii. if the most significant digit of the mantissa is non-zero, then pack exponent and mantissa according to format.

iii. if the most significant digit of the mantissa is zero then left shift the mantissa; decrement exponent, check for underflow. If there is no underflow, check the new most significant digit; repeat until either most significant digit is non-zero or exponent underflows. Then pack the exponent and mantissa.

The zero case is detected before normalization.

Example 1: i. Given 1.8734 E+72 --mantissa overflow

.18734 E+73 --right shift, increment exponent

.1873 E+73 --pack according to 4 digit mantissa format

ii.Given .0034521 E-6 --zero most significant digit

.034521 E-7 --left shift,decrement exponent

.34521 E-8 -- " " " "

.3452 E-8 –pack according to 4 digit mantissa format

In the design, the result is not available as a whole. Rather, digits are available one-by-one (in the adder-subtractor) and two digits at-a-time (in the multiplier). Since the most significant digits arrive first, this does not change the above algorithm, except that no shifting is done.

Example 2: Given result I.8734 E+72 in an on-line addition-subtraction operation.

.1 E+73 --mantissa overflow, increment exponent

.18 E+73

.187 E+73

.1873 E+73 --done; exponent&mantissa packed

As seen in the above example, normalizing in the design involves also the construction of the mantissa according to the format. In some cases, exponent underflow or underflow may occur during such operation. In the overflow case, $\pm\infty$ is sent out according to the sign of the mantissa overflow digit. If there is underflow, then all result digits have to be examined for the sign until a non-zero digit is found: then $\pm\epsilon$ is sent out according to the sign of this digit.

Example 3: Let E+100 be overlow and E-100 be underflow.

i.Given $\bar{1}.7344$ E+99→ $.\bar{1}7344$ E+100 --negative overflow

Therefore RESULT→ $-\infty$

ii.Given $.000345\bar{6}$ E-98→

.0 E-99 –first digit zero

.0 E-100 –second digit zero, underflow

.0 E-100 --third digit zero

.3 E-100 --non-zero, positive digit

Therefore RESULT→ +€

Unfortunately all zero results cannot be detected easily in a digit-by-digit environment and therefore can cause unnecessary normalizing operations. The proposed method of handling these is to 1) provide mechanisms to check operands pre-operation to discover zero-result cases, e.g. 0 + 0, 10 * 0, and 2) to continue normalizing post-operation until the last result digit is produced. In this case a zero exponent and zero mantissa can be packed and sent.

Example 4: For case 1 above, 789 * 0 = 0 can be detected before operation.For addition and subtraction, there can be pre-operation detection of all zero operands only, i.e. 0 + 0 .

3.2 Addition and Subtraction[1]

Signed digit addition and subtraction has been described in section 2.1. What follows is an algorithmic description; the various terms used below can be found in 2.1.

Given operands Z and Y, S-D (signed-digit) addition is done at two levels. First

$$z_i + y_i = rt_{i-1} + w_i$$

where z_i&y_i are *i'th* digits of Z&Y respectively (i digits right of the radix point, t_{i-1} is the transfer digit and w_i is the interim sum digit).

The second level produces the *i'th* sum digit:

$$s_i = w_i + t_i$$

Since $|w_i| \leq r-2$ (sect. 2.2), a value for $|w_{max}|$, the largest magnitude, has to be selected. In this design, w_{max} is chosen to be r-2. Now a stepwise description of addition can be made:

1. A. Avizienis, [AV 1&2]

i.add z_i to y_i to obtain x_i, i.e. $x_i = z_i + y_i$

ii.generate the transfer digit t_i using s_i and w_{max} where $w_{max} \leq r-2$:

a. if $x_i > w_{max}$, there is positive carry; i.e. $t_{i-1} = 1$

b. if $-w_{max} \leq x_i \leq w_{max}$, there is no carry; i.e. $t_{i-1} = 0$

c. if $x_i < w_{max}$, there is negative carry; i.e. $t_{i-1} = -1$

iii.obtain *i'th* interim sum digit w_i:

$w_i = x_i - rt_{i-1}$

iv. finally, compute *i'th* sum digit:

$s_i = w_i + t_i$

Figure 3.1 summarizes the above. It should be noted that using this algorithm, given *i'th* operand digits z_i and y_i, *i'th* sum digit s_i is produced when t_i is available, which is to say when *(i+1)'st* digits are available. Once s_i is produced, it can be used up in another process before s_{i+1} is available. Initially w_0 is zero so that t_0 produced by the first most significant digits z_1 and y_1 indicates overflow; i.e. if $s_0 \neq 0$, then there is overflow. Subtraction is done by negating the subtrahend as explained in 2.2.

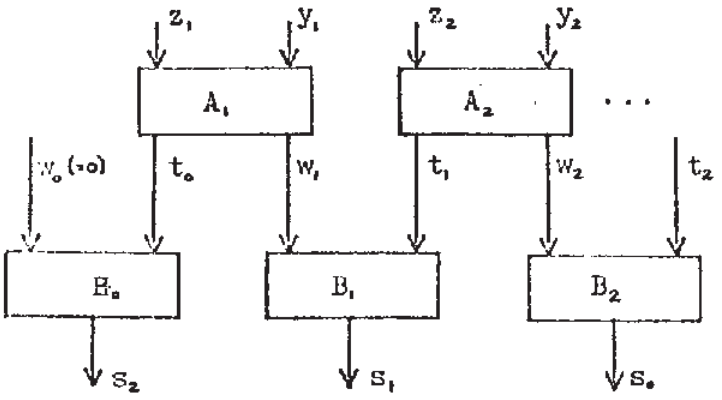
In the adder-subtractor designed here, bytes will be processed. Since a byte is two digits, a two digit parallel adder can be used as shown in Fig.3.2. Only variation here is the extension of the transfer digit of A_2 to B_1 to enable sequential byte-level addition. Computation sequence is indicated next to each port in Fig.3.2.

Example 5: S-D addition using base-8 arithmetic:

Let $|w_{max}| = 6$. Also let the digit set be maximally redundant, $S=\{\bar{7},..,\bar{1},0,1,..,7\}$ as in examples 2&3 in section 2.1.

Given $Z=.65\bar{1}\bar{3}$ and $Y=0.4\bar{7}\bar{1}\bar{4}$, the sum is:

	s_0	s_1
1) $6 + 4 = 12$	$w_0=0$	
	$t_0=1$	1
2) $5 + \bar{7} = -2$	$w_1=2$	



Let k=digit position to the right of the radix point. Then :

A_k : $z_k + y_k = x_k$
if $x_k > w_{max} \Rightarrow t_{k-1} = 1$
if $-w_{max} \leq x_k \leq w_{max} \Rightarrow t_{k-1} = 0$
if $x_k < -w_{max} \Rightarrow t_{k-1} = -1$

B_k : $s_k = w_k + t_k$

Fig.3.1 Parallel Signed Digit Adder

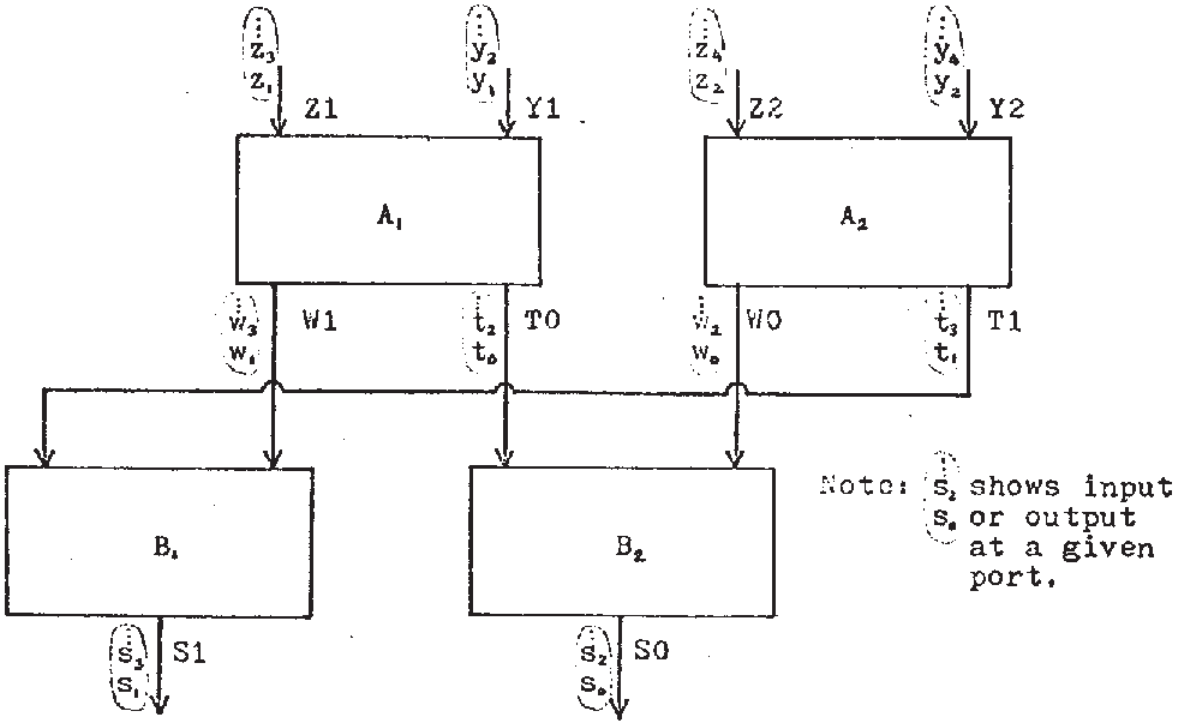
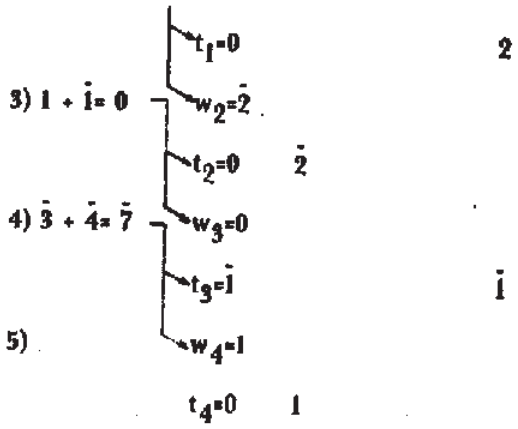


Fig.3.2 Double digit parallel adder modified for byte level computation



RESULT: $0.65\bar{1}\bar{3} + 0.4\bar{7}\bar{1}\bar{4} = 1.2\bar{2}\bar{1}1$ where the 1 to the *left* of the radix point indicates overflow.

More digits can be added in parallel according to need. Due to the elimination of carry propagation chains, parallel addition of more digits will not change the time of computation. In other words, a two digit parallel adder and a four digit parallel adder will have the same time of computation t.

3.3 Multiplication Algorithm

An efficient algorithm for the signed digit multiplication is used here.[1] Following is a description of the algorithm:

Operands are defined as in section 2.1, namely

$$X = \sum_{1}^{n} x_i r^{-i} \quad \text{and} \quad Y = \sum_{1}^{n} y_i r^{-i}$$

As explained previously, this representation has no digits to the left of the radix point.

Let X_j and Y_j be the j-digit representation of X and Y respectively. In other words, let

$$X_j = \sum_{1}^{j} x_i r^{-i} = X_{j-1} + x_j r^{-j} \quad \text{and} \quad Y_j = \sum_{1}^{j} y_i r^{-i} = Y_{j-1} + y_j r^{-j}$$

In an on-line environment, X_j and Y_j are considered as the available parts of X and Y respectively on the j^{th} step. Now the partial product

---

1. Trivedi&Ercegovac [TE 1].

$$X_jY_j = (X_{j-1} + x_jr^{-j})(Y_{j-1} + y_jr^{-j}) = X_{j-1}Y_{j-1} + X_{j-1}y_jr^{-j} + x_jy_jr^{-2j} + x_jY_{j-1}r^{-j}$$
$$= X_{j-1}Y_{j-1} + r^{-j}(X_jy_j + Y_{j-1}x_j) \quad (1)$$

Defining P_j to be the scaled partial product, i.e. $P_j = X_jY_jr^j$, then

$$P_j = P_{j-1} + X_jy_j + Y_{j-1}x_j \quad (2)$$

from equation (1) above. Using this and the fact that $P_0=0$, the desired result can be obtained by

$$P_n = X.Y.r^n \quad (3)$$

This algorithm can be used for non-redundant numbers where the result digits are available *least* significant <u>first</u> in order to cope with carry propogation requirements. Since the interest is on-line computation, a new algorithm can be derived for signed digit multiplication with the on-line property, where input and outputs are obtained most significant digit first.

Using the symmetric and maximally redundant digit set S as defined in example 2, section 2.1, the following new algorithm can be written using (2):

$w_j = r(w_{j-1} - d_{j-1}) + X_jy_j + Y_{j-1}x_j$ (4) where digits d_j are in S, and

$d_j = \text{sign}(w_j).\left\lfloor |w_j| + \frac{1}{2} \right\rfloor$

The result of the multiplication can be expressed as

$$X.Y = r^{-n}(w_n - d_n) + \sum_{1}^{n} d_ir^{-i}$$

In order to meet the restriction that d_j be in S, the operand bounds are limited so that for maximal redundancy,

$$|X|, |Y| < \frac{1}{4}$$

The derivation of the algorithm and the bounds can be found in [TE 1]. Fig.3.3 illustrates the algorithm.

What has been described so far is a digit-at-a-time multiplication algorithm. For the design in this thesis, a two-digit-at-a-time algorithm is required and this can be made possible by slightly modifying eq.(4). Since digits arrive as pairs, the partial operands are redefined as follows:

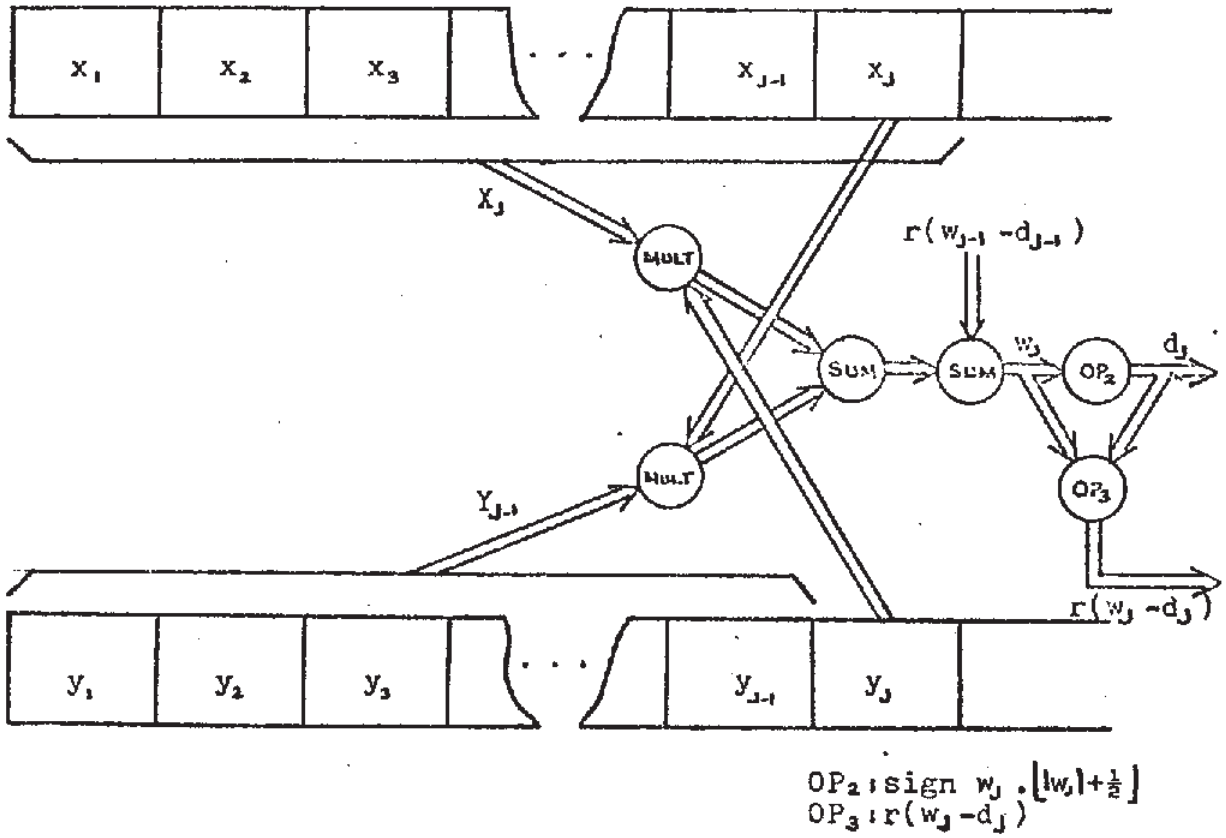


Fig.3.3 Signed Digit Multiplication

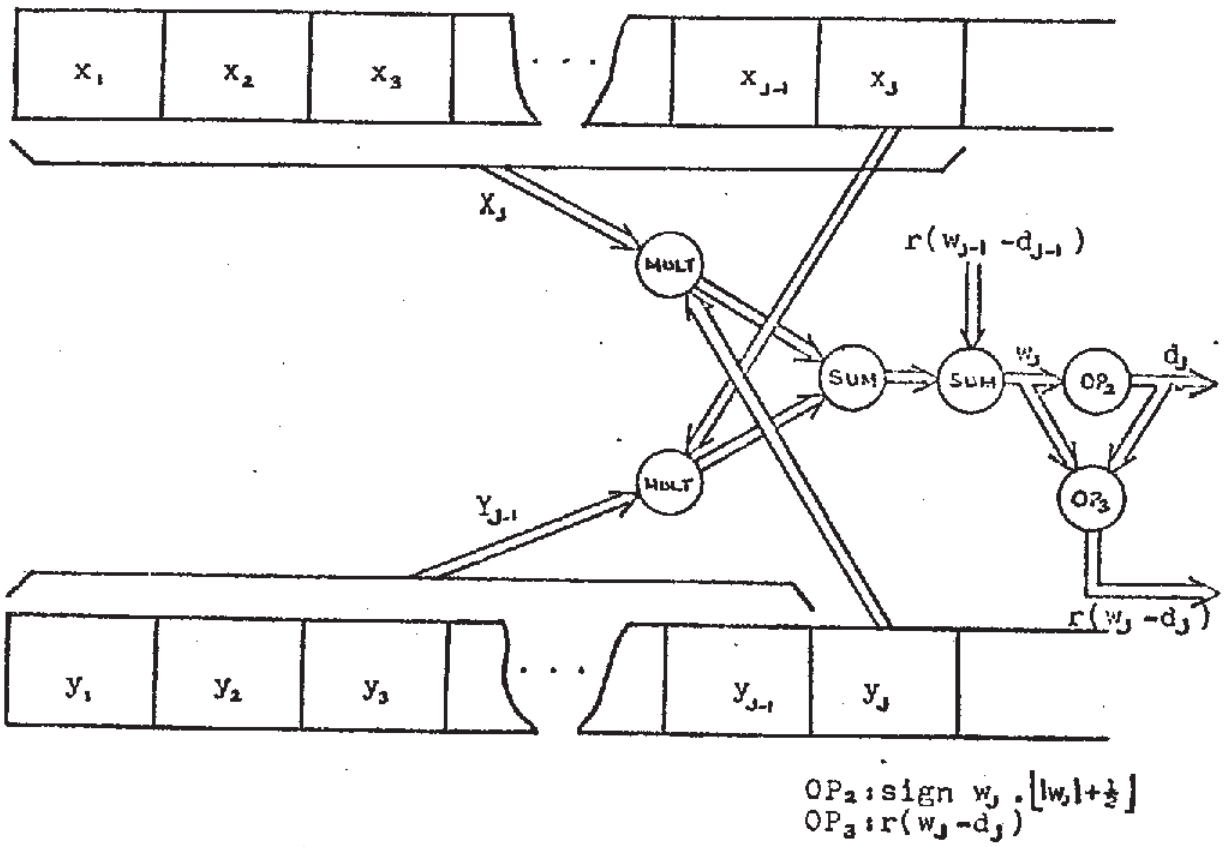


Fig.3.3 Signed Digit Multiplication

$$X_j = \sum_{1}^{j} x_i r^{-2i} = X_{j-1} + r^{-2j}x_j \quad \text{and}$$

$$Y_j = \sum_{1}^{j} y_i r^{-2i} = Y_{j-1} + r^{-2j}y_j$$

Using the same derivation method as before, the new algorithm is defined as follows:

$$w_j = r^2(w_{j-1}-d_{j-1}) + X_j y_j + Y_{j-1}x_j \text{ and}$$

$$X.Y = r^{-n}(w_n-d_n) + \sum_{1}^{n} d_i r^{-2i}$$

The new algorithm produces d_j's that are digit pairs where <u>each</u> digit is in S. Operand bounds still apply, i.e. $|X|, |Y| < \frac{1}{4}$.

<u>Example 6</u>: 1.Signed digit multiplication using single digits:

Let $X = 0.025_{10}$ and $Y = 0.0129_{10}$

j	x_j	y_j	X_j	Y_{j-1}	$X_j y_j$	$Y_{j-1}x_j$	SUM	w_j	d_j	$10(w_j-d_j)$
0	0	0	.0	.0	.0	.0	.0	0.	0	0.
1	0	1	.0	.0	.0	.0	.0	0.	0	0.
2	2	2	.02	.1	.04	.2	.24	0.24	0	2.4
3	5	9	.025	.12	.225	.6	.825	3.225	3	-

$w_3-d_3=0.225$

Result can be obtained as digit pairs, i.e. by d_1, d_2, d_3 and (w_3-d_3).

Therefore, X.Y=0.003225

2.Signed digit multiplication using double digits:

Let $X=0.0234_{10}$ and $Y=0.2463_{10}$

j	x_j	y_j	X_j	Y_{j-1}	$X_j y_j$	$Y_{j-1}x_j$	SUM	w_j	d_j	$100(w_j-d_j)$
0	00	00	.0	.0	0.0	0.0	0.0	00.	00	00
1	02	24	.02	.00	0.48	0.00	0.48	00.48	00	48.
2	34	63	.0234	.24	1.4742	8.16	9.6342	57.6342	58	-

$w_2-d_2=0.\overline{3658}$

Result, $X.Y=0.00\overline{5836}58$ (=0.00576342)

A problem that arises in the use of this algorithm is that normalized numbers can be greater than $\frac{1}{4}$ and therefore if directly used, the algorithm will not work. The cure for this is to design the multiplier such that internally X and Y are treated as $r^{-2}.X$ and $r^{-2}.Y$ respectively. This way a) X and Y are always less than $\frac{1}{4}$ internally, b) since the shift or the zero padding is even, the input digit pairs need not be broken apart and therefore c) the result is produced as it would be without the shift, but with two extra zeros.

Example 7:Multiplication for normalized arithmetic:

Let $X=0.9734_{10}$ and $Y=0.9999_{10}$

Internally, these will be seen as

$X=0.009734$ and $Y=0.009999$

j	x_j	y_j	X_j	Y_{j-1}	X_jy_j	$Y_{j-1}x_j$	SUM	w_j	d_j	$100(w_j-d_j)$
0	00	00	.0	.0	.0	.0	0.	00.	00	0.
1	00	00	.00	.00	.00	.00	0.	00.	00	0.
2	97	99	.0097	.00	.9603	.00	0.9603	00.9603	01	$\bar{3}.\bar{9}\bar{7}$
3	34	99	.009734	.0099	.963666	.3366	1.300266	-2.669734	$0\bar{3}$	-

$w_3-d_3=0.330266$

Result: $0001.0\bar{3}330266$ $(.97330266)^1$

As noted in the above example, step j=1 can be omitted. In fact, in the machine implementation, the radix point is not used so that a number entering the multiplier can be treated as if zeros have been placed at its beginning. The multiplication unit is described in section 5.3.

---

1. Note that there is mantissa overflow as indicated by 1 in d_2.

CHAPTER 4: FLOATING POINT ADDER SUBTRACTOR

In this chapter, a fixed format floating point adder-subtractor (FPAS) using signed-digit arithmetic is described. General outline of the FPAS is followed by a detailed description of each module. The operand format used is the same as that of Fig.3.2.

4.1 General outline

FPAS has been divided into five functional modules as shown in Fig.4.1. The following briefly describes each module.

i. MPX : This module controls the input data flow. Since the operands arrive as a series of bytes, it is necessary 1) to delay the first operand until the second arrives, and 2) to separate the exponents and the mantissa. Therefore given an input sequence (exp1 | op1 | exp2 | op2) , MPX outputs (exp1, exp2) followed by (op1, op2), each from separate ports.

ii.EXPFIX: This module is required for a) comparing exponents, and b) detecting special operands (section 2.3). Exponent comparison is done to adjust the operands as required in floating point addition and subtraction. Possible outcomes of this operation are as follows:

Given two operands, opnd1 and opnd2, let exp1 - exp2 = x be the exponent difference. Also let ∂ = the number of digits in a mantissa (i.e. in op1 and op2). Then:

1. if $x \geq \partial$, then opnd1>>opnd2; operation is unnnecessary; result ← exp1 | op1
2. if $x \leq -\partial$, then opnd1>>opnd2; operation is unnnecessary; result ← exp2 | op2
3. if x=0, then exponents are equal; normal operation; result ← exp1 | (op1+op2)
4. if $0 < x < \partial$, then opnd1>opnd2; normal operation, delay op2 by x;
   result ← exp1 | (op1+op2)
5. if $-\partial < x < 0$, then opnd2>opnd1; normal operation, delay op1 by x;

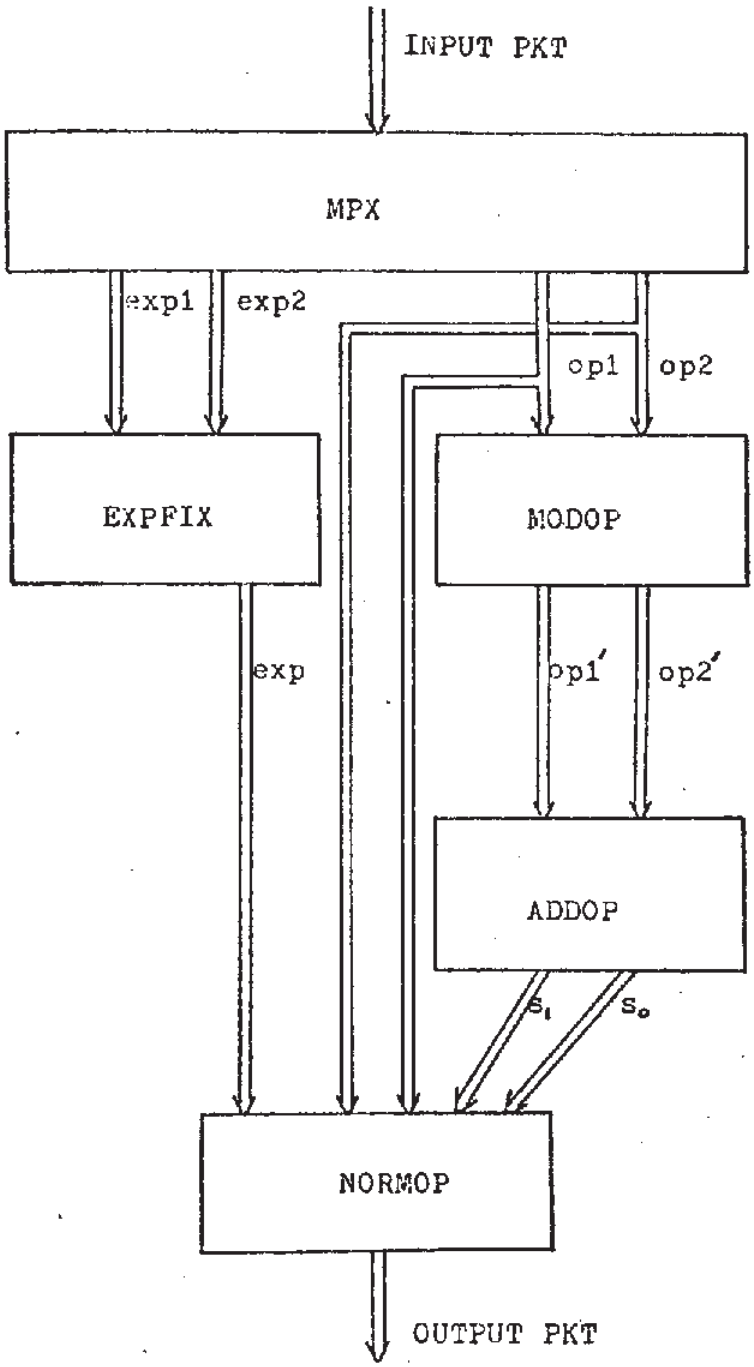


Fig.4.1 General Structure of FPAS

result ← exp2 | (op1+op2)

Delay means reducing the value of an operand by padding zeros to the left of the most significant digit. This is equivalent of "right shift" in processors where the number is available as a whole. In cases 3,4 and 5, the result is unnormalized (sect. 3.1). Special operands are detected by testing the exponents: if any is found, rest of the system is informed.

iii. MODOP: This module is responsible for 1) adjusting the mantissas op1 and op2 according to their exponent difference and 2) for negating during subtraction. Exponent difference is obtained from EXPFIX through a control signal. Op1' and op2' are the modified mantissas which are sent to ADDOP module for addition.

iv. ADDOP: This is the signed digit adder. It is as described in section 3.2. It receives operands as bytes (2 digits long) and outputs the sum digits through two ports.

v. NORMOP: This module 1) normalizes the result and packs it according to the format in Fig.2.2, 2) in case of special operand involving operations or exponent overflow and underflow after the operation, it outputs the appropriate special operand, and 3) in case of "bypass" operations due to the unallowed exponent difference, it passes out the selected operand.

FPAS communicates with the outside through MPX and NORMOP modules. Control and timing is explained in section 4.7.

The adder-subtractor described here can only handle single precision numbers. Multiple precision can be easily implemented with minor modifications as explained in chapter 6.

4.2 MPX Module

MPX or "multiplex" module is basically a serial to parallel converter. Its internal structure is shown in Fig.4.2.

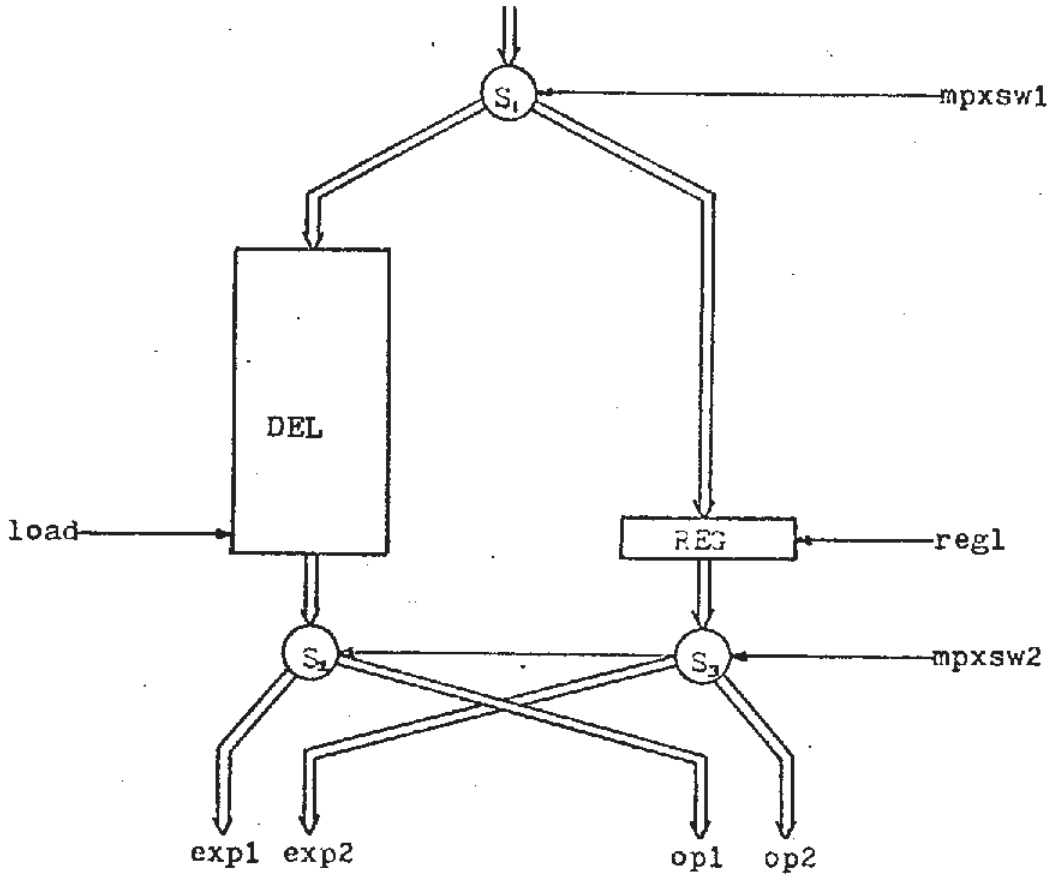


Fig.4.2(a) MPX module structure

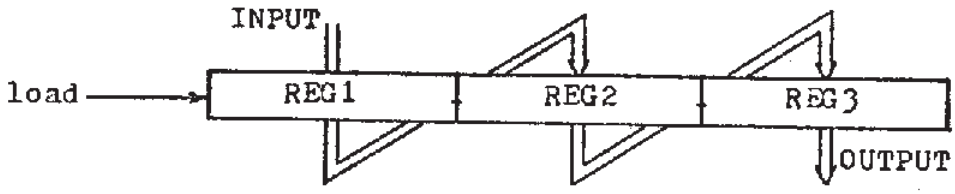


Fig.4.2(b) DEL structure

For single precision processor, DEL is a FIFO queue of length d bytes, where d is the number of bytes in an operand packet; in this example, d=3 . Fig.4.2 (b) shows the structure of DEL. It consists of d registers (of byte width) with common load and clear control signal.

REG is a byte width register used as a buffer to help in pipelining. S1, S2 and S3 are switches; they pass their input to one of the output channels.

Since the operands arrive successively as packets, it is necessary to delay operand-pkt1 (opnd1) so that both operands can be sent into the rest of the processor in parallel. It is also necessary that the the exponents and the mantissas be separated so that they can be handled in different modules. On these lines, the operation of MPX is as follows:

1. The first operand-pkt (opnd1) is delayed by routing it to DEL FIFO where it is stored. When all bytes in opnd1 are loaded, DEL will have exp1 as output.

2. When the second operand-packet (opnd2) arrives in MPX, the bytes are directed to REG register so that both operands are now in parallel form.

3. First exp1 and exp2 are sent out through EXPO1&2 ports to EXPFIX; then the mantissa bytes are made available through op1&2 ports. When last of the mantissa bytes are absorbed by the processor, MPX can receive new input.

Various control signals have been provided to enable the above operation and to make timing analysis easier.

i . *MPXSW1* is used to control S1 which sends incoming bytes to DEL or to REG.

ii . *LOAD* is the common "load" signal to the registers consisting DEL FIFO. Apart from loading, it is also used to change the output of DEL.

iii. *REGL* is used to "load" register REG.

iv . *MPXSW2* controls switches S2 and S3; it allows the outputs of DEL and REG to be available either through EXPO1&2 or op1&2 ports.

4.3 EXPFIX Module

This module is responsible for adjusting the exponents and for preventing

unnecessary operations which may occur if i) exponents differ by an amount equal to or greater than the mantissa precision, or if ii) if one or both exponents indicate special operands. Fig. 4.3 shows the structure of EXPFIX.

Input exponents are stored in REG1 and REG2, both byte width registers.

CEXP is a combinational logic circuit that checks both exponents for special operand indication. There are five possibilities as explained in section 2.3. Possible configuration for CEXP is a series of comparators, the results of which are combined to produce a signal indicating the presence of special operands.

SUBTRACTOR is a binary adder modified to subtract. Since the exponent difference is required, exp2 will be negated prior to addition. To prevent overflow, an adder of input size greater than that of the exponent size is chosen, i.e. if the exponents are 7 bits, adder should be able to handle 8 bits.

COMPARATOR checks the difference between exponents, x (=exp1 - exp2). It compares x with ∂, number of digits in the mantissa. As a result of this comparison, two control variables are produced: 1) *SEL*, which selects the resulting unnormalized exponent using multiplexer M and 2) *EXOP*, which indicates which mantissa is to be "delayed" (if any).

ABS-VAL unit outputs the absolute value of its input. In other words it receives x and produces |x| which is used as the *SFD* variable indicating by how much to delay.

M is a multiplexer.

Mode of operation as follows: once the new exponents exp1&2 arrive, CEXP checks them for the presence of special operands:

i. If there are special operands, then there is no need to carry out the operation. Therefore a result is chosen in accordance with section 2.3; modules MPX and NORMOP are informed so that they can handle this case.

ii. If there are no special operands, then normal operations resume.

In floating point addition and subtraction, operand adjustment is required before

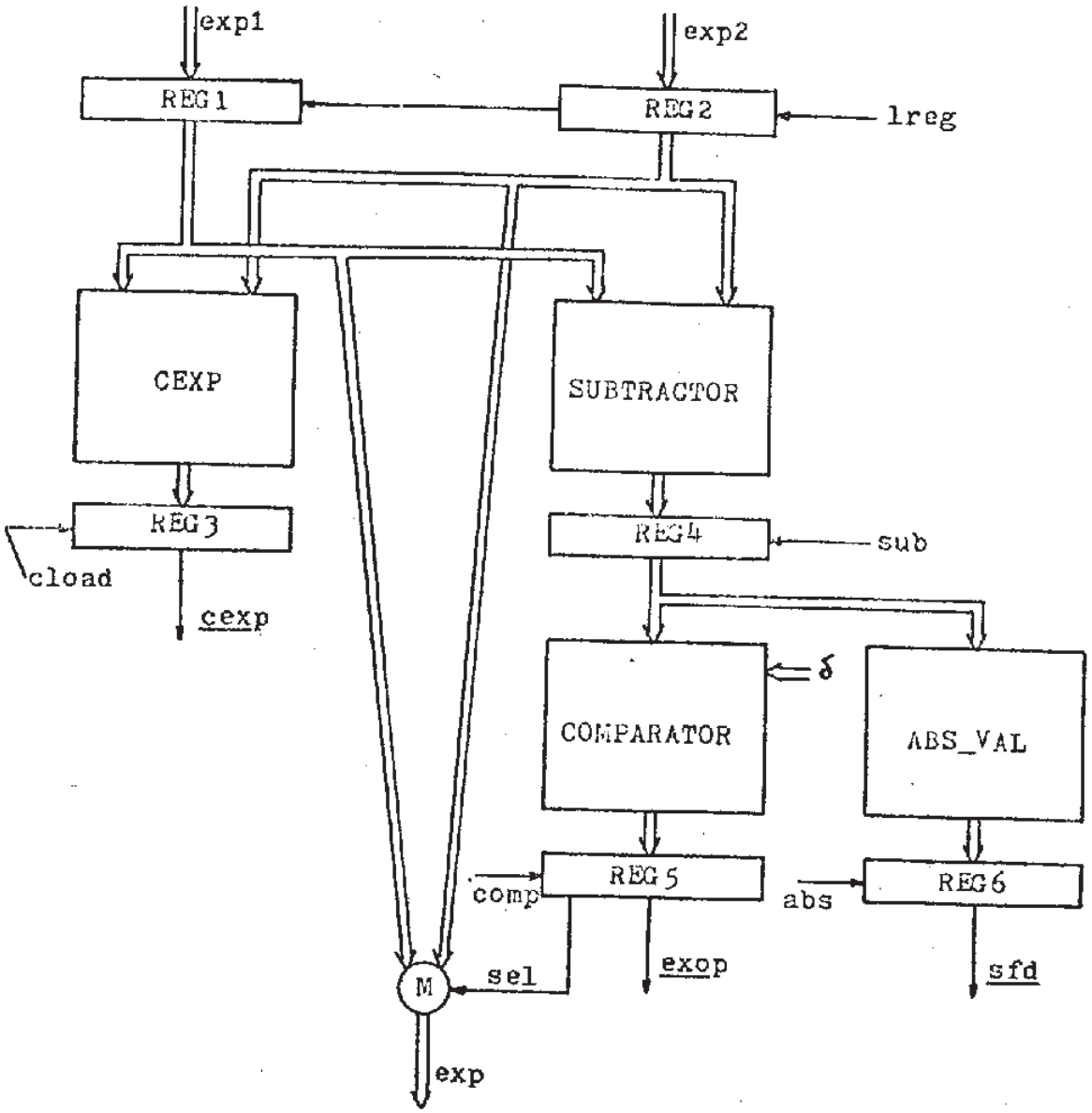


Fig.4.3 EXPFIX Module

operation. That is, the operands are required to have the same exponent value at the time of computation. This adjustment means the shifting of the mantissa with the smaller exponent to the right of the radix point.

Example 1: Given $X=0.3451_8$ E+13 and $Y=0.5766_8$ E+11 , then

$$X+Y= 0.3451 \text{ E+13} + 0.005766 \text{ E+13} = 0.353066 \text{ E+13}$$

This may cause unnecessary operations if the exponents differ by an amount equal to or greater than ∂ (no. of digits in the mantissa); in other words, the result would be same as if the operation was not done:

Example 2: Given $X=0.3\bar{7}67$ E+69 and $Y=0.2\bar{5}\bar{5}7$ E+65, then

$$X+Y= 0.3\bar{7}67 \text{ E+69} + 0.00002\bar{5}\bar{5}7 \text{ E+69} = 0.37672\bar{5}\bar{5}7 \text{ E+69}$$

Packing the result according to format, i.e. after normalizing,

$$X+Y= 0.3\bar{7}67 \text{ E+69} = X$$

Since truncation is used for rounding up, the result digits beyond the format are not necessary and therefore for $x = exp1 - exp2$, operation is carried out iff $-\partial < x < \partial$. Otherwise one of the operands is chosen as the result. Five possibilities for EXPFIX operation are listed in section 4.1 (ii). These can be expressed as a signal triplet to the control: (*EXOP, SFD, SEL*). For example, for possibility 4, the triplet would be

(delay op2, by |x| digits, choose exp1 as result exponent)

When this signal is ready, necessary operations can begin elsewhere in the processor.

Various signals and registers have been shown in Fig.4.3 . The registers act as buffers for various interim results so as to provide an orderly operation which is a necessity during timing analysis. So, for example, if *CLOAD* signal is sent, this loads the outcome of special operand check into REG3. When this is acknowledged, one can assume that for example special operands have been detected.

operation. That is, the operands are required to have the same exponent value at the time of computation. This adjustment means the shifting of the mantissa with the smaller exponent to the right of the radix point.

Example 1: Given $X=0.3451_8$ E+13 and $Y=0.5766_8$ E+11 , then

$$X+Y= 0.3451 \text{ E+13} + 0.005766 \text{ E+13} = 0.353066 \text{ E+13}$$

This may cause unnecessary operations if the exponents differ by an amount equal to or greater than ∂ (no. of digits in the mantissa); in other words, the result would be same as if the operation was not done:

Example 2: Given $X=0.3\dot{7}6\dot{7}$ E+69 and $Y=0.2\ddot{5}5\ddot{7}$ E+65, then

$$X+Y= 0.3\dot{7}6\dot{7} \text{ E+69} + 0.00002\ddot{5}5\ddot{7} \text{ E+69} = 0.37\dot{6}72\ddot{5}5\ddot{7} \text{ E+69}$$

Packing the result according to format, i.e. after normalizing,

$$X+Y= 0.3\dot{7}6\dot{7} \text{ E+69} = X$$

Since truncation is used for rounding up, the result digits beyond the format are not necessary and therefore for x = exp1 - exp2, operation is carried out iff $-\partial < x < \partial$. Otherwise one of the operands is chosen as the result. Five possibilities for EXPFIX operation are listed in section 4.1 (ii). These can be expressed as a signal triplet to the control: (*EXOP, SFD, SEL*). For example, for possibility 4, the triplet would be

(delay op2, by |x| digits, choose exp1 as result exponent)

When this signal is ready, necessary operations can begin elsewhere in the processor.

Various signals and registers have been shown in Fig.4.3 . The registers act as buffers for various interim results so as to provide an orderly operation which is a necessity during timing analysis. So, for example, if *CLOAD* signal is sent, this loads the outcome of special operand check into REG3. When this is acknowledged, one can assume that for example special operands have been detected.

4.4 MODOP Module

MODOP (modify operand) delays operands - if necessary - and negates for subtraction. Basic idea behind the operation of MODOP is to i) manipulate the mantissa bytes if necessary and then ii) load them into buffer registers REG1 and REG2 where they will be used up by ADDOP. Possible outcomes of the operation of MODOP are as follows:

Let Δ=delay. Also let output (X) indicate the output of operator X. Given that (i) op1 enters via OPIN1 and (ii) op2 enters via OPIN2, then:

	output (REG)	output (REG2)	
1.	op2	op1	;no delay
2.	-op2	op1	;no delay-subtraction
3.	op1+Δ	op2	;op1 delayed
4.	-op2	op1+Δ	;op1 delayed-subtraction
5.	op2+Δ	op1	;op2 delayed
6.	-(op2+Δ)	op1	;op2 delayed / subtraction

What "delay" means here is as follows: since the complete operand is not fully available for shifting while adjusting for exponent difference, digits have to be "delayed" by padding a specified number of zeros in front of the mantissa. This process is explained later. Fig.4.4 shows detailed structure of MODOP.

Q1 and Q2 are each n+1 digit size, wrap-around FIFO queues where n is the number of mantissa bytes in the operand-packet. They are provided to store the delayed operand. In this design, Q1 and Q2 have the representative structure shown in Fig.4.5 . There are two signals: *LOAD* places new data in the queues and *SENAB* enables the digit width output. It should be noted that an acknowledged output is "lost". Wrap-around property eliminates the need for pointer resetting when the queue is full. The problem of overwriting is avoided by providing maximum size.

NEG is the negation unit. Due to the properties of signed-digit numbers (sect.2.1),

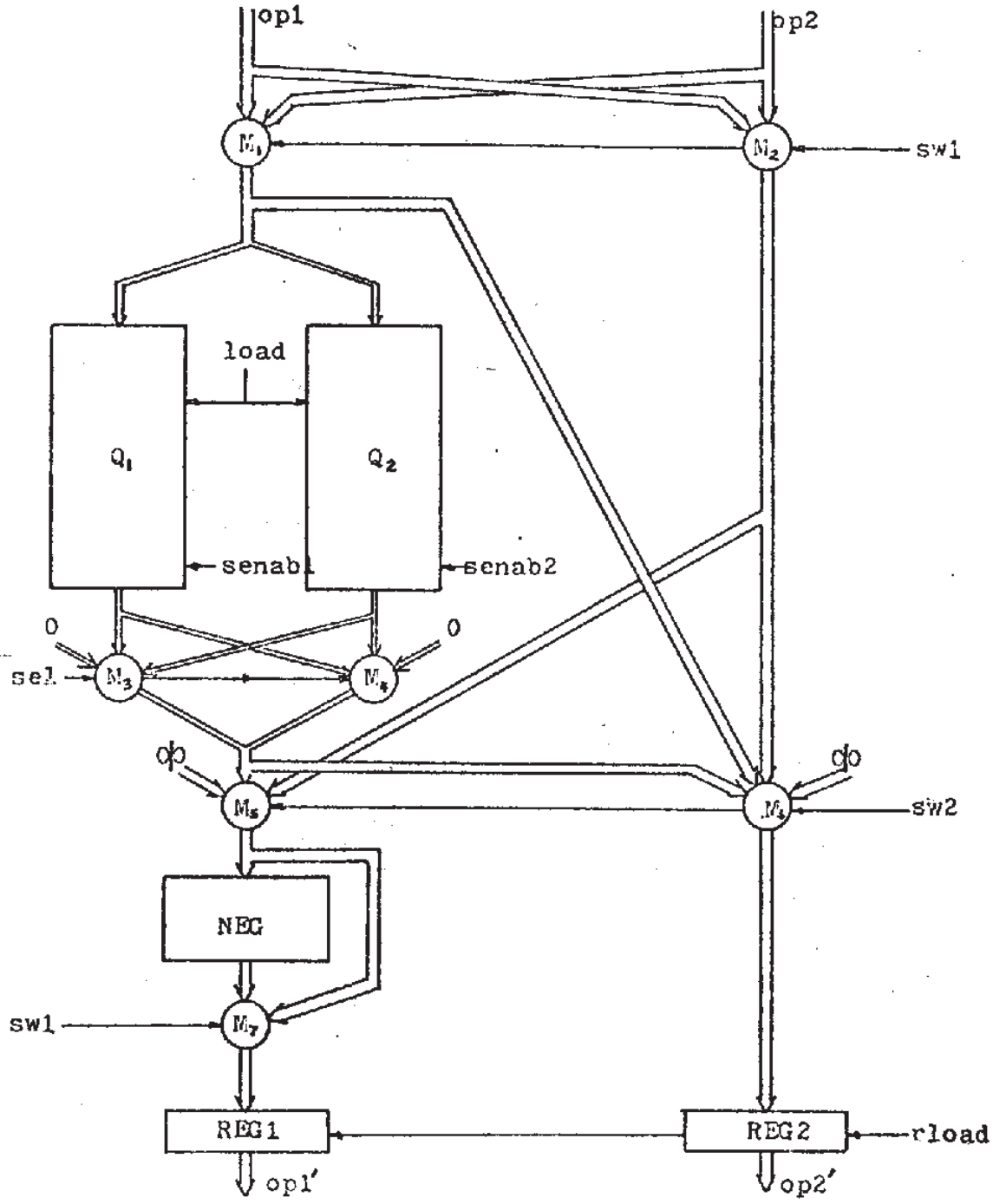
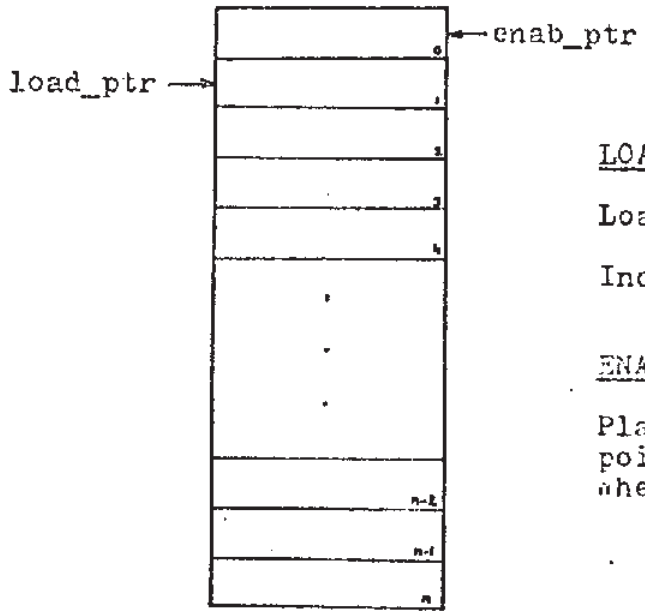


Fig.4.4 MODOP Module



wrap-around FIFO
(modulo n+1 counter)

LOAD INSTRUCTION

Load into register pointed by load_ptr;
Increment load_ptr

ENAB INSTRUCTION

Place contents of register pointed by enab_ptr on the output; when acknowledged, increment enab_ptr

Fig.4.5 Wrap-around FIFO QUEUE

negation requires the sign of each digit to be changed. In the machine representation used here (base-8, 16's complement), this can be done by complementing each digit and then adding 1, i.e. two's complement.

Example 3: Given digit pair $\bar{4}3$,

$$\bar{4}3 = 1100\ |\ 0011 \rightarrow 0011\ |\ 1100 \rightarrow 0011 + 0001\ |\ 1100 + 0001 \rightarrow 0100\ |\ 1101 = 4\bar{3}$$

NEG can either be a selection circuit where given two digits it would output their negated form, or it can be a combinational logic circuit , i.e. a complement circuit plus adder. Since the number of possibilities is small, the first method seems to be the appropriate solution if enough selection speed can be achieved.

M1 through M6 are multiplexers which enable routing and selection of bytes.

Delaying can cause problems if the exponent difference leads to padding by an odd number of zeros, whereby bytes (containing two digits each) have to be broken up and then reconstructed with each digit "shifted". An even number delay is of no problem because all-zero bytes can be used for non-zero padding in this case. Now, the problem of odd number delay can be solved efficiently by using two queues of (each) one digit width with common *LOAD* but separate *ENAB* signal inputs. This way each queue will contain a digit of each byte entered and therefore using multiplexers M3 and M4, a new byte of desired form can be produced. In the design example, the following routines are used to cope with the delay problems:

1. If the delay is even, then the output of Q1 is passed through M3 and the output of Q2 through M4 until queues are empty.
2. If the delay is odd, then:

   i. output(M3) = 0 and output(M4) = output(Q1)

   ii. output(M3) = output(Q2) and output(M4) = output(M4)

   iii. output(M3) = output(Q2) and output(M4) = 0

These instructions are for the number representation used here; more digits will mean the

repetition of step(ii). Fig.4.6 shows examples of the above routines.

With the given mantissa format, possible odd number delays are 1 and 3. When the delay is 3 digits, the first 2 zeros (i.e. a zero byte) are handled through multiplexer M5 - the rest of the digits are sent using routine 2. In case of even delay (2 digits), the zeros are made available through M5 again, followed by routine 1.

Multiplexers M1, M2 and M7 control the main byte flow. These will be set at the beginning of a MODOP operation and reset when done. According to the outcome of exponent comparison in EXPFIX and the kind of operation (addition and subtraction) there are six possibilities for the use of these multiplexers:

1. pass op1 to Q1/Q2, op2 to M6 :negate (subtraction) if opnd2>opnd1
   or :pass (addition) (delay op1)
2. pass op2 to Q1/Q2, op1 to M6 :negate (subtraction) if opnd1>opnd2
   or :pass (addition) (delay op2)
3. pass op1→M1, op2→M2 :negate (subtraction) if exp1=exp2
   or :pass (addition)

M7 has a choice between the negated or unnegated output of M5.

As explained previously, multiplexers M3 and M4 are used to "reconstruct" delayed bytes. Zero inputs are used in the case of odd delays, as shown in Fig.4.6 (ii).

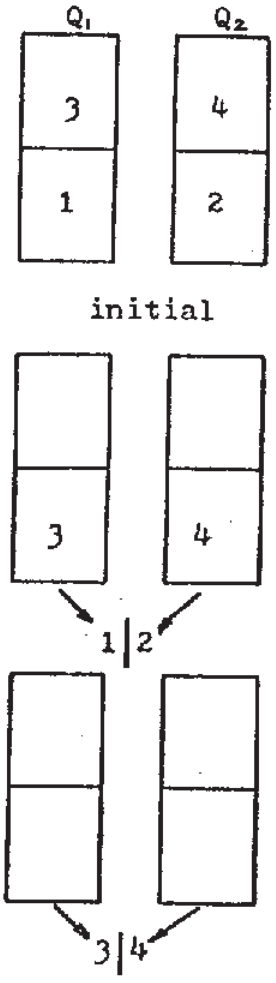
Multiplexers M5 and M6 have the following functions:

a. They route the operand to be negated if there is a subtraction operation. For example, in case 1 above, op1 is delayed, so it is sent to Q1/Q2. By assumption, op2 is <u>always</u> the subtrahend, therefore the delayed op1 bytes coming out of M3&4 are passed through M6 to REG2 while op2 bytes are sent through M7.

b. They provide zero bytes during delay operation. In this design, this would occur in delays of 2 and 3 only.

c. They provide zero bytes when MPX "runs out" of mantissa bytes. This occurs for

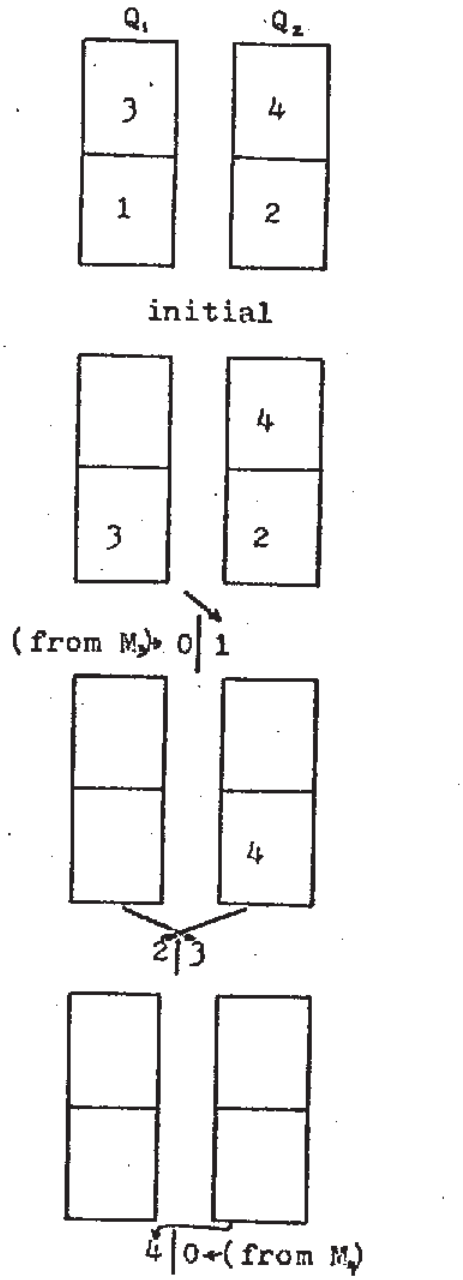
Delayed operand: .1234



Possible delayed output:
.001234

i. even delay

Q_1 & Q_2 queues of size 2 bytes



Poss. delayed outputs:
.012340
.00012340

ii. odd delay

Fig.4.6 Operation of the "delay" mechanism

example, when op1 is delayed by two digits. In this case, op1 becomes a 6 digit number, but since the format used is 4 digits for the mantissa, the undelayed operand op2 will require two zero digits (a zero byte) tagged at the end so that it can also consist of 6 digits.

To achieve all of these operations, various control signals have been provided. A list appears below:

1. *SW1* controls multiplexers M1, M2 and M7. This is set once per MODOP operation.
2. *LOAD* loads bytes into Q1 and Q2 (combined).
3. *SENAB1* and *SENAB2* brings out contents of the queues. They are separate (one each queue) to enable the construction of various bytes.
4. *SEL* controls M3 and M4 to construct a delayed byte.
5. *SW2* controls M5 and M6; it is responsible for producing the final form of the mantissa bytes sent to ADDOP.
6. *RLOAD* loads the modified bytes into REG1 and REG2 when they are ready. Actually this signal indicates the availability of a new mantissa byte pair to ADDOP.

4.5 ADDOP Module

ADDOP is a two digit S-D adder as described in section 3.2. It has been modified to be able to add bytes of operands. Fig.4.7 shows its detailed structure.

As in Fig.3.1 and 3.2, it is necessary to divide the adder into two units A_k and B_k. The A_k unit which produces w_k and t_{k-1} for given operand digits z_k and y_k can be constructed in two ways:

1. It can either be implemented by a combinational logic circuit whereby given two digit input z_k and y_k, the appropriate w_k and t_{k-1} would be directly produced, or
2. it can be a table, such as a ROM,[1] whereby the digit-pair input would be used as an

1. ROM: *Read only memory*

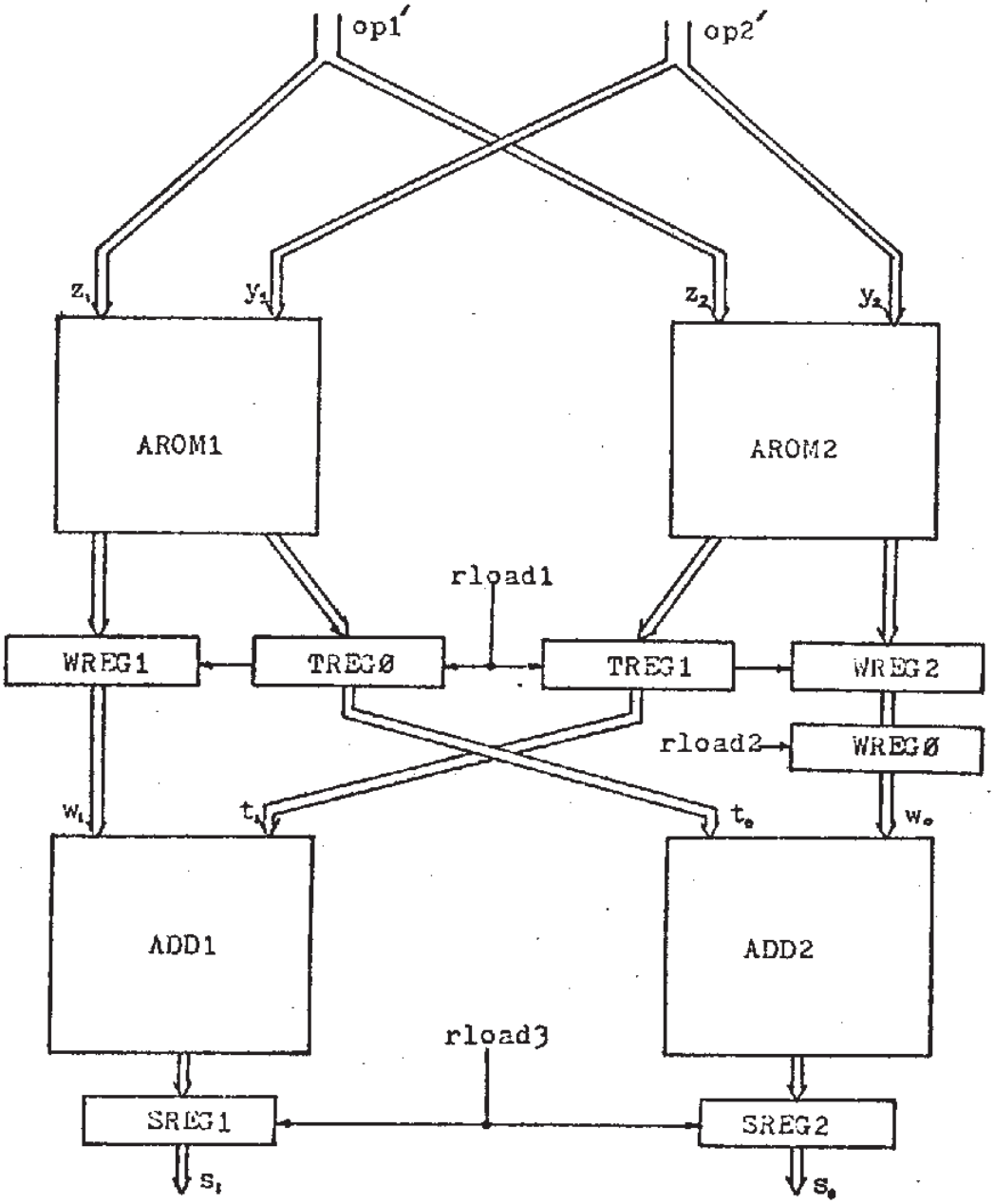


Fig.4.7 ADDOP Module

address to "read" the required (w_k, t_{k-1}) pair.

A third possibility is to break up A_k into smaller units whereby each step of the signed-digit algorithm is implemented. As it turns out, this method fails with the number representation used here due to ambiguity.

Example 4: If the intention is to obtain $x_k\ (z_k+y_k)$, then using the machine representation of Fig.2.2, the following ambiguities can occur:

$\bar{4} + 0 = 1100 + 0000 = 1100 = \bar{4}$

$\bar{2} + \bar{2} = 1110 + 1110 = 11100 \neq \bar{4}$

$6 + 6 = 0110 + 0110 = 1100 = \bar{4} \neq$ correct answer

Between the first two possibilities, the second one is the most preferable because the complexity of producing a combinational logic circuit here is a disadvantage that outweighs the deficiency of ROMs in terms of increased power consumption and slower speed. Therefore ROM tables are used for A_k units.

AROM1 and AROM2 are ROMs of size 225*8 bits each. The size is computed as follows: since the address pair will consist of two digits z and y, and since each digit can assume 15 values in the machine representation used in this design, there are 15*15 possible addresses. Since the intention is also to generate w and t as digits each of length 4 bits, each ROM is required to have at least 225*8 bit size, and this can be obtained through the use of a 256*8 ROM. Now, this number can be decreased by addition of extra hardware to avoid similar addresses such as (3,2) and (2,3). A suggested solution is shown in section 5.3. Contents of the ROMs are arranged according to the signed digit addition algorithm and the table thus formed is invariable.

ADD1 and ADD2 are each 4 bit binary adders. They are used to compute the sum digits. Since the sum digits will be in the digit set S (sect. 2.1), addition of w and t always yields a digit result. In other words, no special decoding is necessary due to the handiness of the number representation used here.

Example 5: $6 + \bar{1} = 0110 + 1111 = 10101$ (disregard carry) $\rightarrow 0101 = 5$

$\bar{6} + 1 = 1010 + 0001 = 01011 \rightarrow 1011 = \bar{5}$

WREG0, 1&2 are digit width buffer registers provided to store the interim sum digits w. Similarly, TREG0&1 and SREG0&1 store the transfer digits t and the sum digits s respectively.

The operation of ADDOP is straight forward. Operation begins as the operand bytes arrive from MODOP. The only "trick" comes in producing the first extra sum digit, s_0. As explained in section 3.2, the first sum digit s_0 indicates mantissa overflow and it is formed by the addition of t_0 and w_0, where w_0=0 and t_0 is the transfer digit formed by the addition of z_1 and y_1, the first operand digits. What this means is that an interim sum digit produced by AROM2 has to be delayed one "cycle-time" where a "cycle" is the arrival time of a new digit pair. Therefore WREG0 keeps the previous w value. An example will clarify the point:

Example 6: Let X="don't care". Now, given operands $.z_1z_2$ and $.y_1y_2$, then

WREG0	TREG0	WREG1	TREG1	WREG2	SREG1	SREG0
0	0	0	0	0	0	0
w_0(=0)	t_0	w_1	t_1	w_2	0	0
w_0	t_0	w_1	t_1	w_2	s_0	s_0
w_2	t_2(=0)	X	X	X	s_1	s_0
w_2	t_2	X	X	X	X	s_2

The sequence of operation is shown in Fig.3.2.

Various control signals have been provided for sequencing and timing. They are as follows:

1. *RLOAD1* which places the outputs of the AROM1 and AROM2 into WREG1&2 and TREG0&1 registers. In other words it is a "load" signal.

2. *RLOAD2* loads the output of WREG2 into WREG0 for storage of *w*.

3. *RLOAD3* signal places the computed sum digits in SREG1&2.

4.8 NORMOP Module

NORMOP takes care of special operands, "bypass" operations and normalization of the result. Its structure is shown in Fig.4.8.

As mentioned in related sections, special operands are detected in the EXPFIX module, or they can be created in NORMOP as a result of normalizing operations, namely in cases of exponent overflow and underflow. In all cases, a special operand output is selected according to the operands and operation.

For speed considerations, special operands are kept in a series of registers called SOP. Each such SOP unit consists of n byte-width registers where n is the number of bytes in an operand packet (opnd); in this case, n=3. They are connected as shown in Fig.4.9. In the FPAS, there will be 5 SOP units, namely one each for E, $+\infty$, $-\infty$, $+\epsilon$, and $-\epsilon$. Special operand bytes will be available through the use of multiplexers M1 to M5; the required special operand will be selected using the multiplexer M6.

EXPREG is a register/counter of byte width that stores the unnormalized exponent received from EXPFIX. It can count-up (*CTU*) or count-down (*CTD*) the exponent while normalizing.

O/UFLOW is a comparator which detects if the exponent over or underflows during the process of counting up or down.

ZERO checks whether the incoming digits from ADDOP are zero or not. It can be implemented as a combinational logic circuit.

STREG is a "two-digit" register with single digit input. It is provided to construct bytes out of single digits arriving from ADDOP (Fig.4.10). Note that there is a single "load" signal. Operation of NORMOP can be separated into three:

1. If a special operand case has been detected in EXPFIX, then special operand bytes are

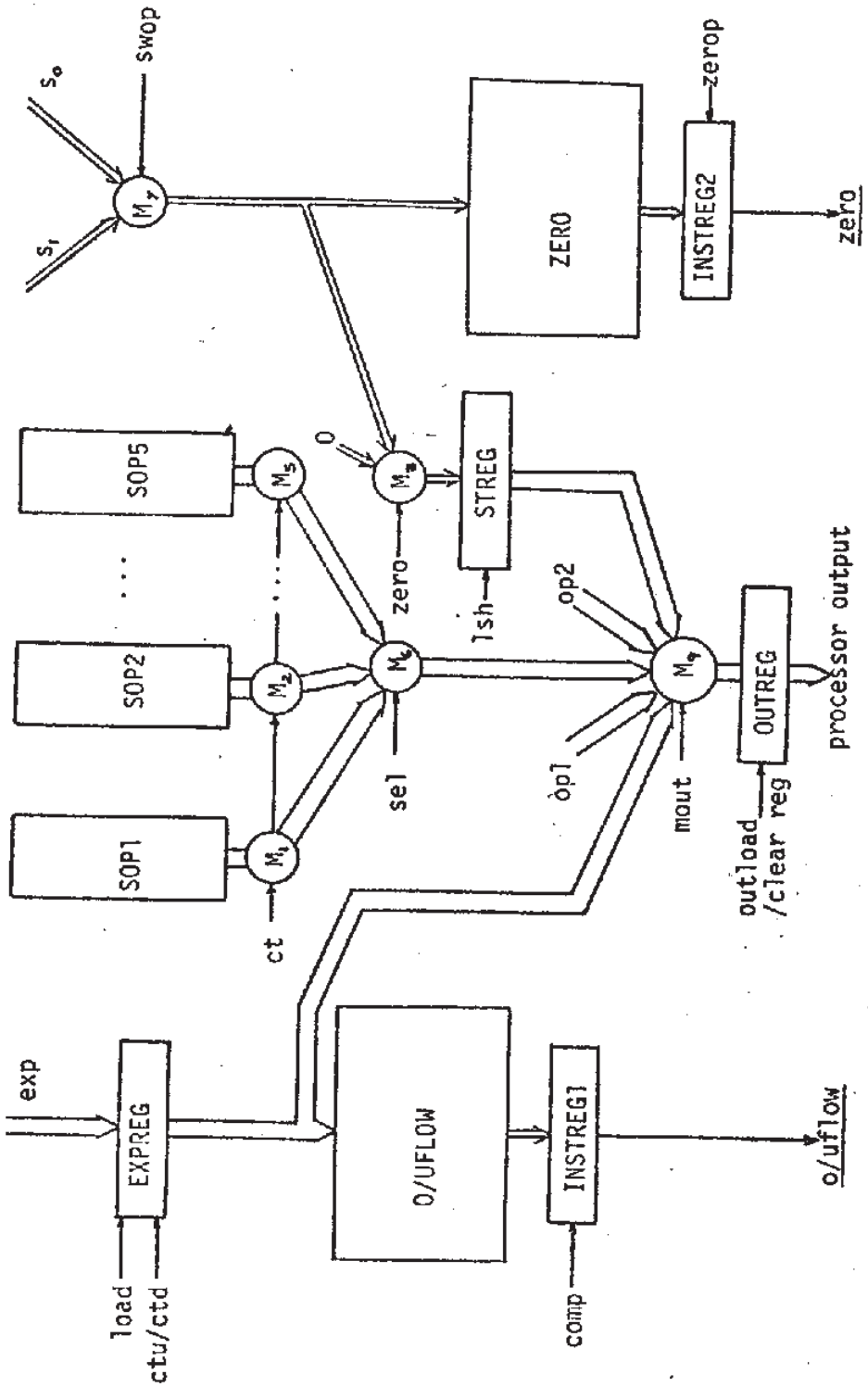


Fig.4.8 NORMOP Module

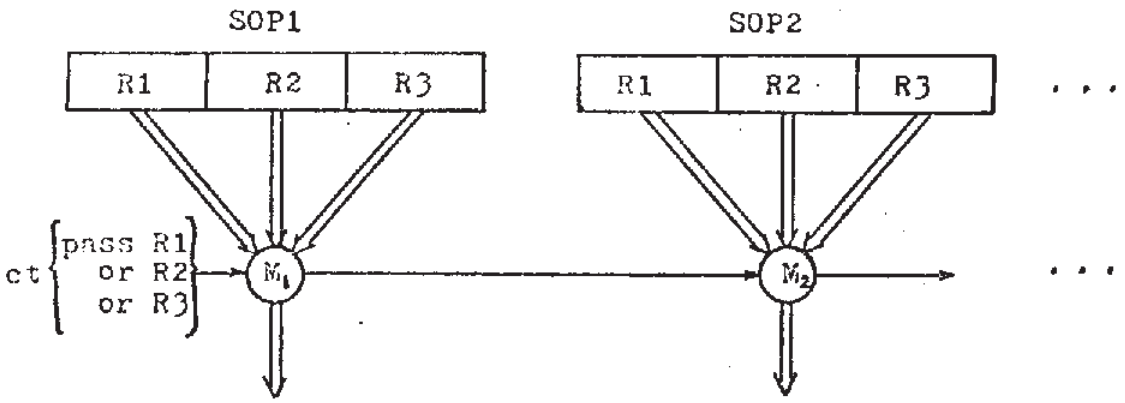


Fig.4.9 Arrangement of the SOP units

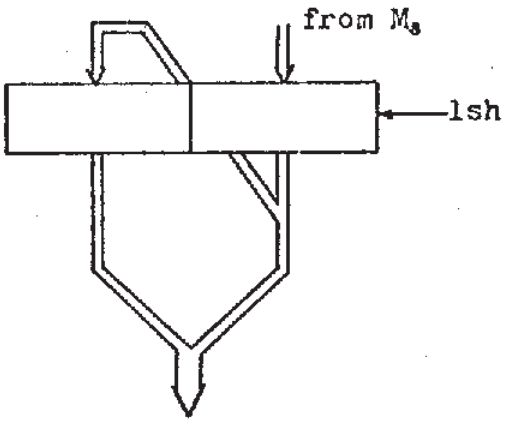


Fig.4.10 STREG Register

made available through M1 thru M5, and the special operand is chosen using multiplexer M6. The passage to output is achieved by the use of M9.

2. If the operation is "by-pass" as explained in section 4.3, then using M9, the exponent in EXPREG is sent out, followed by op1 or op2, received directly from MPX.

3. If the operation is normal, i.e. no special operands or "by-pass" operations, then NORMOP receives result digits from ADDOP. Number of digits produced by the latter range from 5 (no delay) to 8 (3 delay).[1] For the normalization and packing, the following routine is carried out:

   i. if the first digit is non-zero, then ther is mantissa overflow. In this case the exponent is incremented and checked for exponent overflow.

      a. if there is overflow, then an appropriate special operand, $\pm\infty$ is sent out according to the sign of the overflow digit using step 1 above.

      b. if there is no overflow, then through M9 the exponent and 4 digits are sent out. i.e.

         | exp |+| s_0 s_1 |+| s_2 s_3 |

         Rest of the result digits arriving from ADDOP are not used.

   ii. if the first digit, s_0, is zero, then each of following result digit will be checked until a non-zero digit is detected by the ZERO unit. The exponent is decremented for each zero digit found after s_0 and checked for underflow in each case. Again, if underflow is detected, then $\pm\epsilon$ is sent out (section 3.1); otherwise the adjusted exponent and 4 digits (consisting of the first non-zero digit and the following ones) is sent out.

Above algorithm fails to consider a few special problems. First of all, if exponent overflow and underflow is detected in NORMOP, then one of the special operand packets is sent out. None of the computed result digits show up in the packets. Secondly there is the problem of number of digits available: since NORMOP has to send out a certain number of digits (in this case 4), there may be two problems: i) NORMOP may not require all the

1. *Given figures apply to this design; algebraically, it ranges from ∂+1 to 2∂, where ∂= no. of digits in mantissa.*

digits produced by ADDOP, or ii) NORMOP may request more than what ADDOP has produced.

Example 7: Given ADDOP output .1234567, NORMOP uses only .1234 ; also given ADDOP output .000023, NORMOP requires .23__ ; the empty places are filled with zeros.

In order to cope with this problem, a signalling protocol will be made between these modules so that ADDOP will signal the availability of the last digit with a "last data" signal, and NORMOP will dummy acknowledge receipt of excess digits if it does not require more digits. This will be further explained in section 4.7. In the former case where NORMOP requests more digits, zero digits will be packed and sent out. These zeros can either come from M8 (single zeros) or OUTREG (zero bytes). If all result digits come out to be zero, then an all zero operand packet is sent out using OUTREG.

Now a description of various signals can be given:

1. *LOAD* places the unnormalized exponent into EXPREG register/counter. *CTU/CTD* is for counting up or down of the contents of EXPREG during normalization.
2. *SWOP* signal controls multiplexer M7 to provide digit input to NORMOP. It basically selects s_0 or s_1 outputs of ADDOP.
3. *ZERO* controls multiplexer M8 so that input to STREG can be a result digit, s, or a zero digit.
4. *LSH* loads the output of multiplexer M8 into STREG.
5. *CT* is a three mode signal that controls M1 thru M5. Possible modes can be seen in Fig.4.9.
6. *SEL* picks up the special operand-pkt required using multiplexer M6.
7. *MOUT* selects the output of the processor using M9. It has a choice between the exponent(in EXPREG), result mantissa(in STREG), operand mantissas op1&op2(from MPX) and special operand(from M6).

8. *OUTLOAD* places the output of M9 into OUTREG. *CLEAR* "clears" OUTREG and it is used to create zero output bytes if necessary.
9. *ZEROP* places the result of ZERO check in INSTREG2.
10. *COMP* is used to load signal u/oflow into INSTREG1.

4.7 Control of the FPAS

The detailed description of the individual modules so far touched lightly on the subject of control. Algorithms defined within each module did not specify explicitly the communication between the modules. This is intended to clarify these points.

FPAS begins operations by the receipt of "operand_pkt ready" signal from the "outside", i.e. the rest of the system. MPX begins to load the incoming bytes and when the exponents are ready[1] , EXPFIX is signalled to load the exponents and start the necessary operations. When EXPFIX acknowledges receipt of the operands, MPX prepares $op1_1$ & $op2_1$ and waits for EXPFIX to finish its operation which may result in the following:

1. If there is a special operand, then EXPFIX signals MPX and NORMOP through *CEXP* to scrap the normal operation, or
2. if the operation is "by-pass", it signals NORMOP, or
3. it signals NORMOP and MODOP to operate normally.

In case (1), MPX can get rid of the operation packet it has to receive a new one, while NORMOP sends out a special operand pkt. In case (2), NORMOP loads the chosen exponent from EXPFIX and links with MPX to send out the chosen mantissa (op1 or op2). In the last case, MODOP begins operating on mantissa bytes arriving from MPX while NORMOP loads the chosen exponent.

MODOP loads and passes the operand bytes from MPX at a pace dependant on ADDOP. When all bytes from MPX are received, MPX can receive a new operand-pkt.

1. *after the arrival of 4 bytes in this case*

EXPFIX will be free when NORMOP acknowledges the receipt of exponent.

ADDOP signals NORMOP when the digits are ready. Since NORMOP has to examine each of the sum digits, speed of ADDOP will depend on how fast normalization process can take place. This dependency extends all the way to MPX and therefore one can conclude that the speed of FPAS is proportional to the speed of NORMOP.

As discussed in 4.6, a signal protocol is arranged between ADDOP and NORMOP. When ADDOP produces the first result digit, it sends a ready to NORMOP which acknowledges when it is through with it. When ADDOP produces the last digit, it then signals "last data" so that NORMOP can send zero digits if it requests for more. It may be the case that ADDOP produces more digits than NORMOP -- when this happens, the proposed solution is to provide "dummy" acknowledges to the ADDOP so that the uncomputed result digits can be "drained out" without abruptly stopping rest of the processor. Otherwise one has to know the exact state of each module: for example if MPX has just began processing a new operand packet while MODOP and ADDOP are working on the old one, than the system will have to know which module to clear. By "draining" the excess digits, this problem is avoided.

Finally, when NORMOP starts sending operand-pkts, it signals the "outside". Fig.4.11 summarizes the main control signals.

Control sequence for FPAS can be shown using timed petri-nets.[1] They are easy to derive and due to their similarity to state machines, they can specify the control requirements quite clearly. Using the various control signals defined in the explanations, the petri-net for the MPX module is derived here (Appendix; A1).[2] Section 4.2 and Fig. 4.2(a) will help in the derivation.

When an operand packet is available, MPX can begin operations. Therefore the first transition sets the input switch. Then the incoming bytes are loaded into DEL; once that is

1. ref. [RAM 1]
2. *Petri-net graphs for the FPAS can be found in the* Appendix (A1-A7).

MODULE NAME	OUTSIDE	INTERMODULAR
MPX	opnd-pkt ready/ack	exp-ready/ack(to/from EXPFIX) op-ready/ack (to/from MODOP and NORMOP)
EXPFIX	none	cexp(special op.)(to MPX and NORMOP) op-start (to MODOP) sfd, exop } (to MODOP&NORMOP) res-exp ready/ack(to/from NORMOP)
MODOP	none	oper-ready/ack (to/from ADDOP)
ADDOP	none	res-ready/ack (to/from NORMOP) last-data (to NORMOP)
NORMOP	res-pkt ready/ack	dummy-ack (to ADDOP)

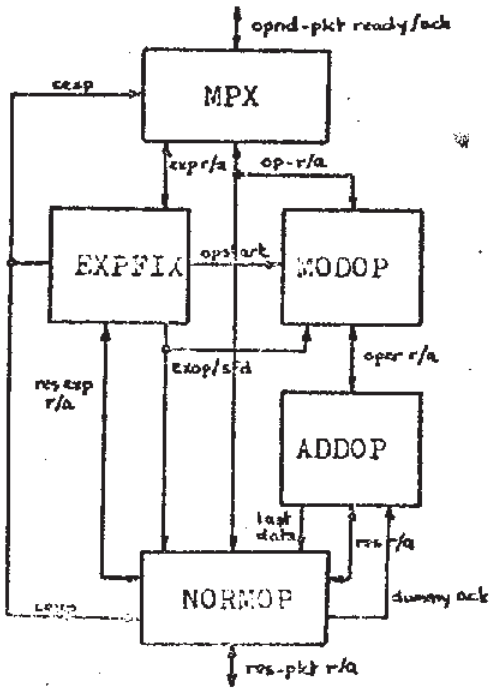


Fig.4.11 Control Signals

done, the second operand set is available, and therefore the switches are set (*MPXSW1/SW2*). *REGL* loads exp2 into REG and now a "ready" signal can be sent to EXPFIX. Two responses are possible: 1) *CEXP*, if there is a special operand, in which case MPX "dummy" acknowledges the "outside" for the operand bytes yet to be received; or 2) *EXP-ACK* whereby op1 and op2 can be sent to MODOP. Once either of these is done, MPX can start again. Note the ready and acknowledge signals which control the flow.

A series of such petri-nets have been used in performance analysis of FPAS: they are all not given here, but they are easy to obtain through the use of the above example and the descriptions of the modules.

Due to nature of the signed digit numbers, various worst case situations of maximum computation time are possible. Following is a typical example:

base-8 addition: 0.1777 E63 + 0.1001 E60

Exponent difference = 3, therefore second operand is delayed by 3;

Using S-D addition, 0.1777 + 0.0001001 = .0000001 E63 → 0.1000 E57

The result obtained here requires NORMOP to examine 8 digits. The first is the overflow digit which is checked in all cases, but the following six zeros each require 1) zero check (*ZEROP*), 2) exponent decrement (*CTD*) and 3)n underflow check (*COMP*). This consumes lots of time and can be considered a worst-case. However the occurence of such an operation does not have a high probability, but as it represents the worst case possible, it is worthwhile to analyze the total time of computation.

Worst case time estimate is found to be close to 800 ns; this is the time passed from the arrival of the first operand byte to the sending out of the last result byte. This estimate is obtained by assigning a time value to each transition of the petri-net. These values are also estimated by modelling each transition with a TTL logic component(s).[1] Some examples are as follows:

1. *values used here are from* [TI 1]

Register load	(*RLOAD, LSH, RLOADI...*)	20 ns.
Add	(*ADD1, ADD2...*)	15 ns.

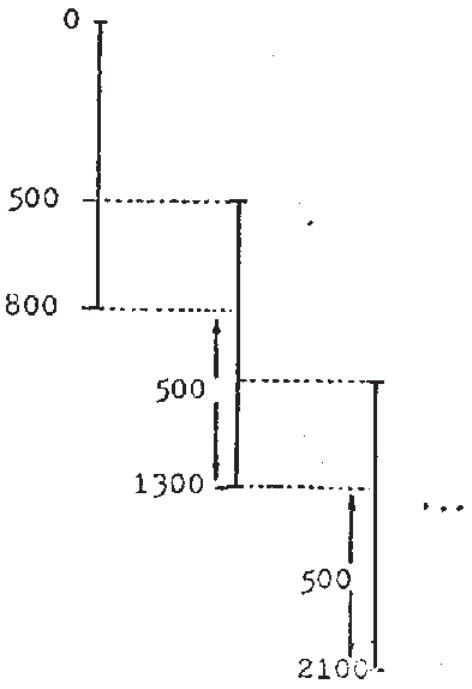
Some of the transitions are composed of various logic levels: *SUB* of EXPFIX has an operation time of (negation circuit) + (adder).

Let the operand-pkt arrive at time zero. MPX then does 4 register loads and at 80 ns, EXPFIX can begin. It produces its result in about 50 ns. MODOP begins at 130 ns, and loads at 150 ns when MPX has an operand byte pair available. ADDOP can start at 200 ns time point. First number for NORMOP is available at 275 ns. Start and end times for each module is shown in Fig.4.12(a). A new operation can begin 130 ns before ADDOP ends so that when the sum digit s_7 is produced, a new byte is available for the next operation. In other words, MPX can begin at 670-130≈500 ns (approx.) after the first operation began. Fig.4.12(b) shows how the computation times can overlap. This gives an overall worst-case computation rate of 2 MHz.

The main delaying element here is NORMOP; the three operations described above require about 70 ns each time, and this prevents a faster rate of operation for ADDOP. A possible cure is to use faster logic. However, on the average the 70 ns will not effect the computation adversely because their will be less zeros to compare. It should be also noted that the time rates obtained here are for the format used in this design; in the very likely case when more precision will be used, this time will increase.

Module	Start	End	Total
MPX	0 ns	260 ns	260 ns
EXPFIX	80 ns	130 ns	50 ns
MODOP	130 ns	475 ns	345 ns
ADDOP	200 ns	670 ns	470 ns
NORMOP	275 ns	800 ns	525 ns

Fig.4.12(a)Timing



Worst case throughput:
1 result in 500 ns,
:.2 MHz rate

Fig.4.12(b)Overlapping

CHAPTER 5: FLOATING POINT MULTIPLIER

In this chapter, a byte-serial, fixed format, floating point multiplier (FPM) is described. A general discussion is followed by detailed description of the multiplier.

5.1 General Description of FPM

As in the case of the floating point adder-subtractor, FPM is also divided into functional modules. General outline is shown in Fig.5.1; only the data flow lines have been shown. Following are brief descriptions of each module:

i. MPX : This module is the same as its namesake described for the floating point adder subtractor. A description can be found in sections 4.1(i) and 4.2.

ii. EXOP: This module manipulates exponents before the actual mantissa operation. Basically it is responsible for 1) detecting special operands and for 2) adding the exponents to obtain the unnormalized result exponent. This addition may result in exponent overflow or underflow which is detected also in EXOP.

iii.MULTOP: This is a two digit at-a-time signed-digit multiplier. The algorithm for its operation is explained in section 3.3. It receives, processes and outputs bytes.

iv.PACKOP: In this module, normalization of the result and, if necessary, special operand output preparation are done. The operation is similar to that of NORMOP module in FPAS wih minor structural differences.

5.2 EXOP Module

In floating point multiplication, the unnormalized result-exponent (resexp) is obtained by adding the exponents of the operands, e.g.

(2.5 E-4) * (3 E+7) = 7.5 E((-4) + (+7)) = 7.5 E+3

This operation can result in either

1. a resexp that is within the allowed range, in which case normal operations can be

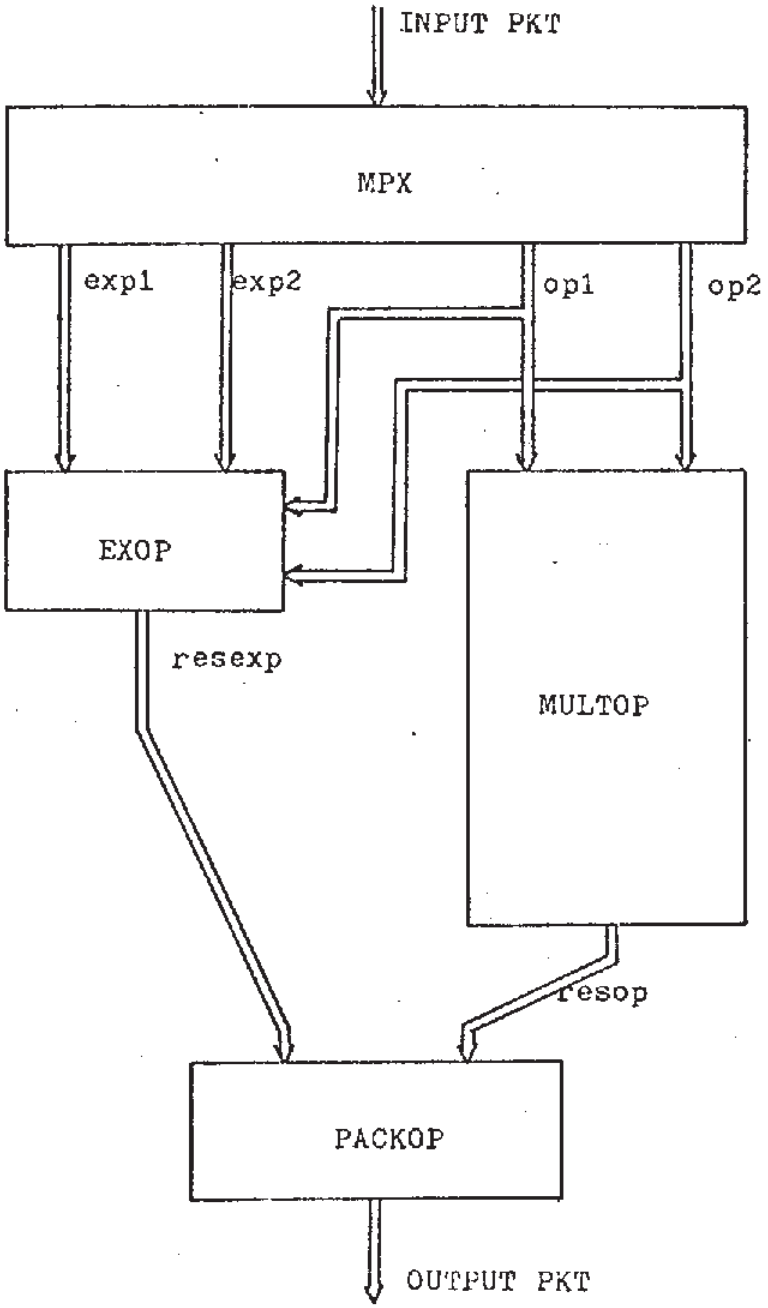


Fig.5.1 General structure of the FPM

carried out; or

2. a resexp that overflows or underflows by 1, in which case normal operations can be carried out, but a final check on overflow or underflow is done in PACKOP; or
3. a resexp which overflows or underflows by more than 1, in which case an appropriate special operand is sent out.

Use of normalized numbers necessitate case 2. Underflow by one may not lead to a final underflow after the operation if there is mantissa overflow only; although this cannot happen in conventional normalized arithmetic,[1] signed digit multiplication may result in mantissa overflow.[2] Initial overflow by 1 which may not be a final overflow can occur easily:

Example 1: Using base-10 arithmetic, let the exponents >99 be overflow. Then

0.1 E 65 * 0.1 E 35	→	.01 E 100	(overflow)
normalize	→	.1 E 99	(no overflow)

To handle all these cases, a final check on the result is required. The actual procedure is explained later in the section after a description of the units of EXOP (Fig. 5.2).

EXPREG1&2 are byte long registers for storing the input exponents, exp1 and exp2.

CEXP is the same as that of EXPFIX in the FPAS (section 4.3). It is a combinational logic circuit that detects special operands using the input exponents as explained in section 2.3. Outcome of the operation of this unit is available as a signal, *CEXP*.

ADD is an 8-bit binary adder that adds input exponent bytes from EXPREG1&2. Since exponents are represented by a sign bit plus 7 bits, this unit will be a conventional sign-magnitude adder. The output, (exp1+exp2), will be obtained as a sign bit plus 8 bits so as to cope with possible magnitude overflow.

---

1. *For mantissa* m *and base* r, *normalized numbers require* $r^{-1} \leq |m| < 1$. *Therefore given any* m_1 *and* m_2, m_1+m_2 *is always* < 1. *See section 3.1.*
2. *See section 3.3, example 7*

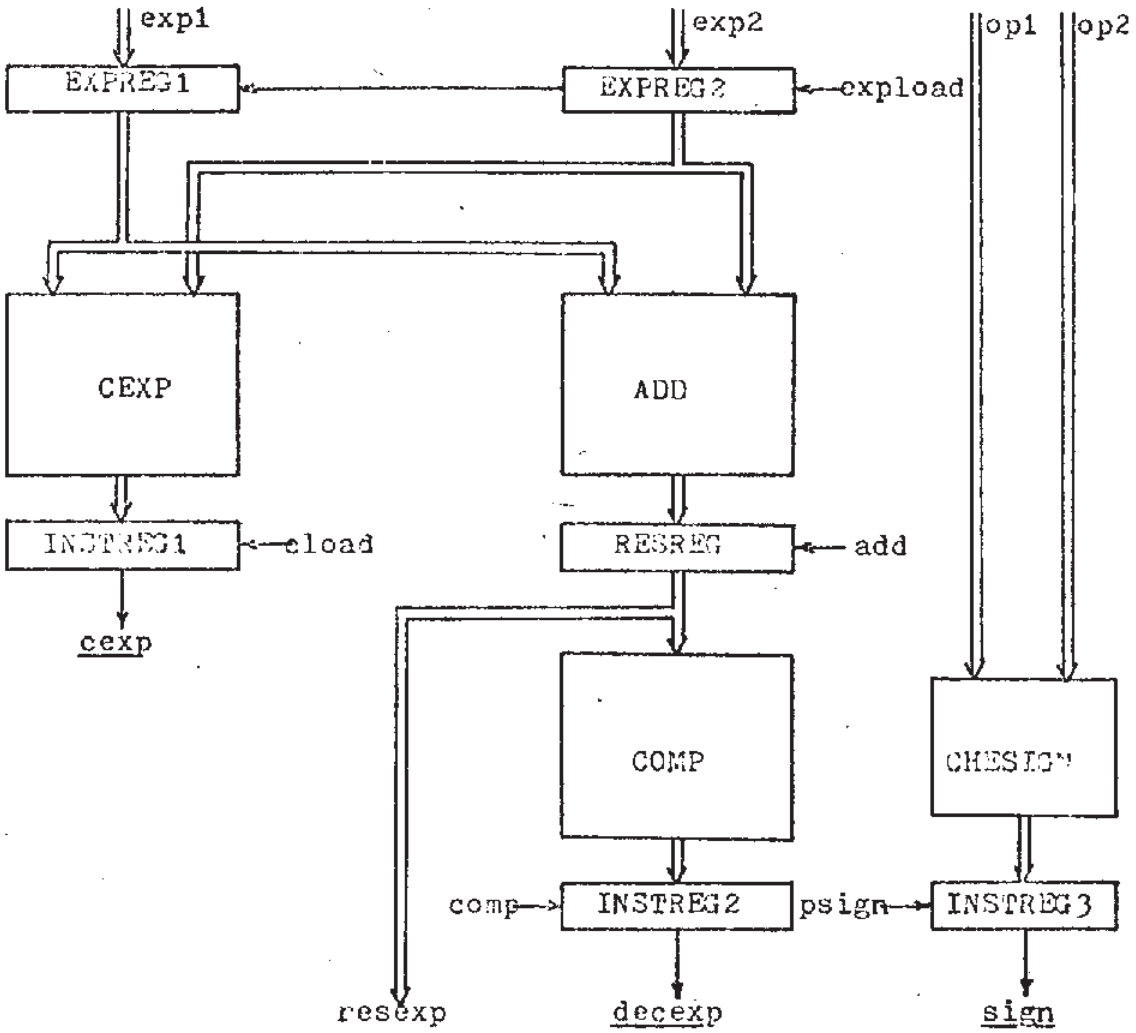


Fig.5.2 EXOP Module

COMP is a comparator which is used to detect overflow and underflow situations by inspecting the output of ADD. It can havefour possible output modes: a) overflow by 1, b) overflow, c) underflow by 1 and d) underflow. These results are expressed as *DECEXP*.

Finally, CHESIGN checks the sign of the first digits of each operand to decide on the sign of the output of FPM if the output is a special operand. For example if the result-exponent is found to overflow, then CHESIGN decides whether the result packet to be sent out $+\infty$ or $-\infty$, or if only one of the operands is a special operand detected in EXOP, then CHESIGN decides on the output sign, e.g. given $+\infty$ and .1000 E 1, the output is decided to be $-\infty$.

Operation begins when MPX signals the availability of new exponents which are loaded into registers EXPREG1&2. Then CEXP unit checks the exponents for special operands. Possible outcomes are:

1. If there are special operands, then MPX and PACKOP are informed so that while MPX prepares to receive a new operand-packet, PACKOP sends out an appropriate special operand.
2. if there are no special operands, EXOP continues operation. The two exponents are then added (using ADD) and the result is checked in COMP. This checking can result in the following:

   Let x= exp1 + exp2, ov = overflow limit, and u = underflow limit.

   1) if $u \leq x \leq ov$, then normal operations are carried out. When MPX prepares the first mantissa bytes, MULTOP is signalled to start mantissa multiplication, while PACKOP loads x, the unnormalized result-exponent. Also CHESIGN examines the first bytes to to come up with the sign of the sign of the product (the result).

   2) if $x > ov$, then there is overflow.

      i. if $x - ov = 1$, then there is "limited" overflow, in which case PACKOP is informed of this special condition, but MULTOP and MPX proceed normally as in (1). If after computation and normalization the exponent still overflows, then + or - ∞ is sent

out from PACKOP, the sign being obtained from CHESIGN.

ii. if x - ov > l, then MPX is signalled to prepare the first operand bytes. When they are ready, CHESIGN examines the most significant digit of each byte to determine the sign of the special operand to be sent out. After this operation, MPX is signalled to receive a new operand packet while PACKOP sends out + or - ∞.

3) if x < u, then there is underflow.

i. if u - x = l, then there is "limited" underflow. The same procedure as (2) is followed, but the underflow situation can only be avoided if there is mantissa overflow. If this does not happen, then + or - ϵ is sent out: again the sign is obtained from CHESIGN.

ii. if u - x > l, then a procedure similar to 2(ii) is followed. The only difference is that in this case, ϵ is sent out as the special operand.

As shown in Fig.5.2, various control signals have been provided to represent the possible control structure:

1. *EXPLOAD* loads the exponents from MPX into registers EXPREG1&2.
2. *CLOAD* is used to load the outcome of special operand check into INSTREG1.
3. *ADD* places the unnormalized result exponent (resexp) into register RESREG.
4. *COMP* loads the result of overflow-underflow check on computed exponent into INSTREG2.
5. *PSIGN* places the result of CHESIGN into output register INSTREG3.

5.3 MULTOP Module

This module is a two digit at-a-time signed digit multiplier based on the algorithm defined in section 3.3. The structure of the module is shown in Fig.5.3.

OPREG1&2 are byte long registers that store operand bytes arriving from MPX. They act as buffers to speed up operations.

OPNDST1&2 are used to construct the complete operands according to the algorithm,

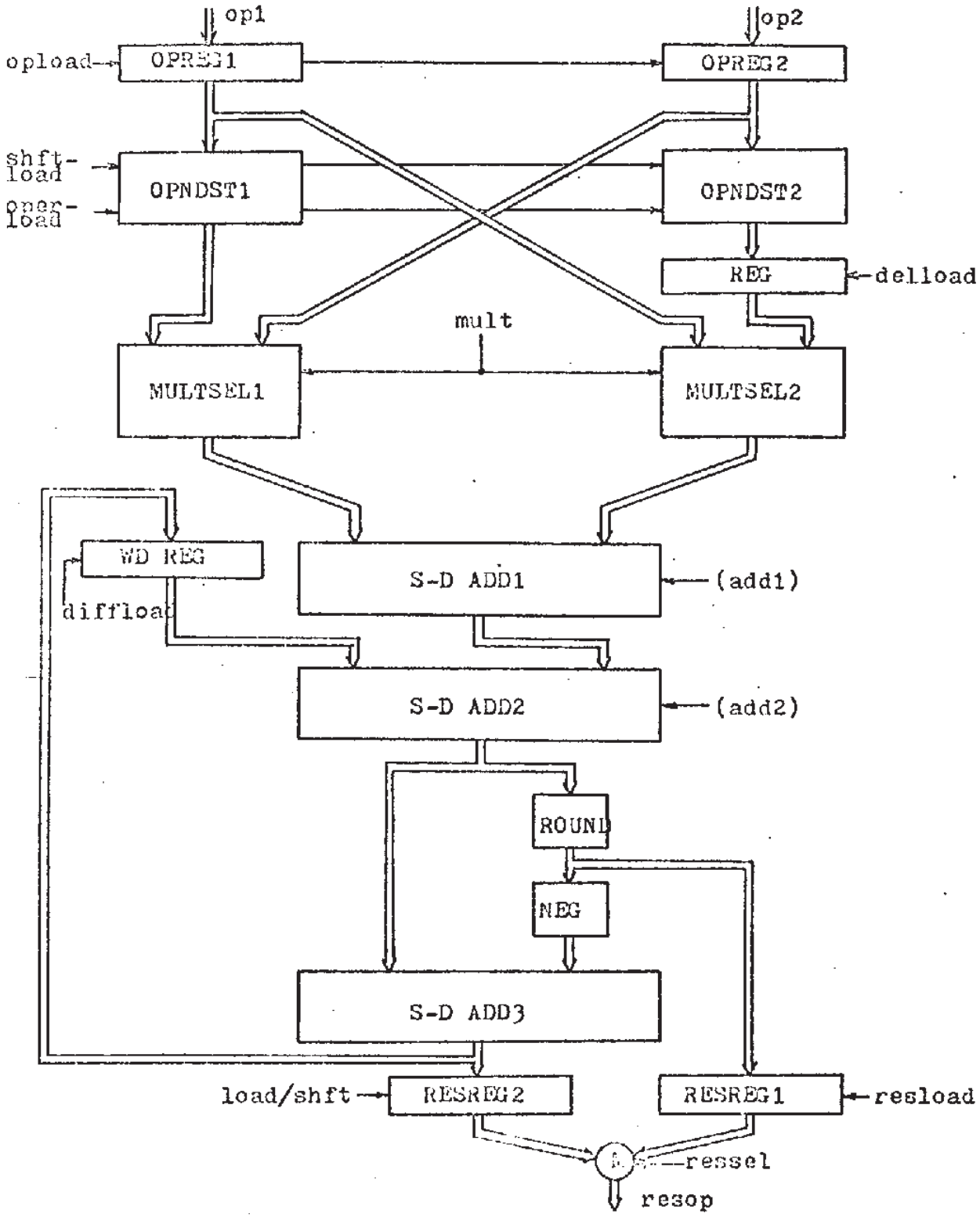


Fig.5.3 MULTOP Module

namely

$$Z_j = Z_{j-1} + 8^2 z_j \quad \text{and} \quad Y_j = Y_{j-1} + 8^2 y_j$$

Instead of adding, a series of shift and load operations are used to speed up the formation of the partial operands. A structure capable of doing this is shown in Fig.5.4. In each OPNDST unit there are two switches SW1 and SW2, and two registers (of byte-length) OPND REG1&2. SW1 is used to control the flow of "load" signal to the registers, while SW2 is responsible for input flow to the registers. This way, the most significant byte from MPX would be loaded into OPND REG1 followed by the least significant into OPND REG2. To summarize, the two modes possible are as follows:

Let C(R) be the contents of register R. Then

1. send C(OPREG) to OPND REG1 (using SW2)
   enable "load" of OPND REG1 (using SW1)
2. send C(OPREG) to OPND REG2 (using SW2)
   enable "load" of OPND REG2 (using SW1)

For the general case where each operand is n bytes, SW1 and SW2 are required to be 1-to-n switches, while the number of registers is n.

OPND REG3 is a two byte-width register provided to store Y_{j-1} which is needed in the multiplication algorithm. This allows the formation of Y_j without the loss of Y_{j-1}.

MULTSEL1&2 compute the products $Z_j y_j$ and $Y_{j-1} x_j$ respectively. Signed digit number representation used here prevents direct binary multiplication to obtain these products because of the special coding necessary. To obtain these products, two methods can be used:

1. A cheap but slow solution is to multiply the individual digits using binary parallel adders and then to add the adjacent digits together. This will be slow because elaborate decoding will be required to obtain these products.
2. A less cheap but faster solution is to replace the multiplication and decoding above by a ROM table. Such an implementation is shown in Fig. 5.5 as used in the design here.

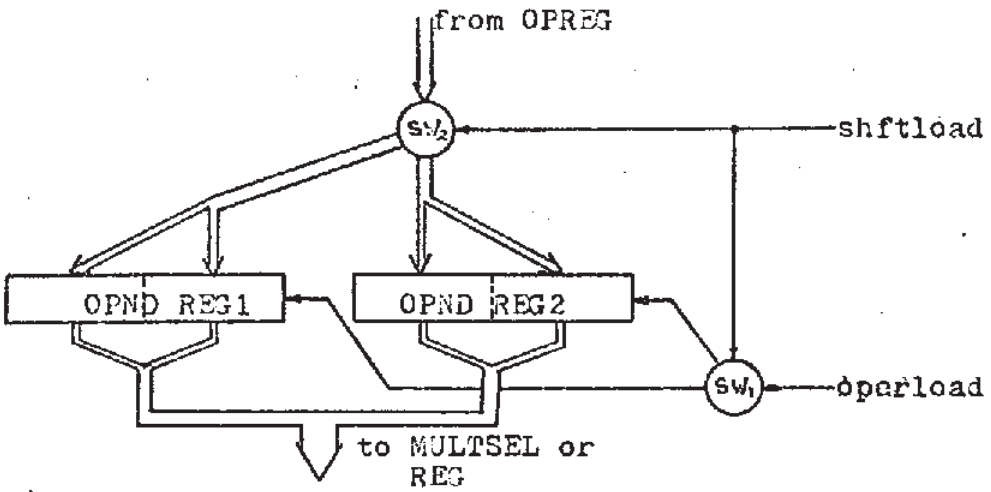


Fig.5.4 OPNDST Structure

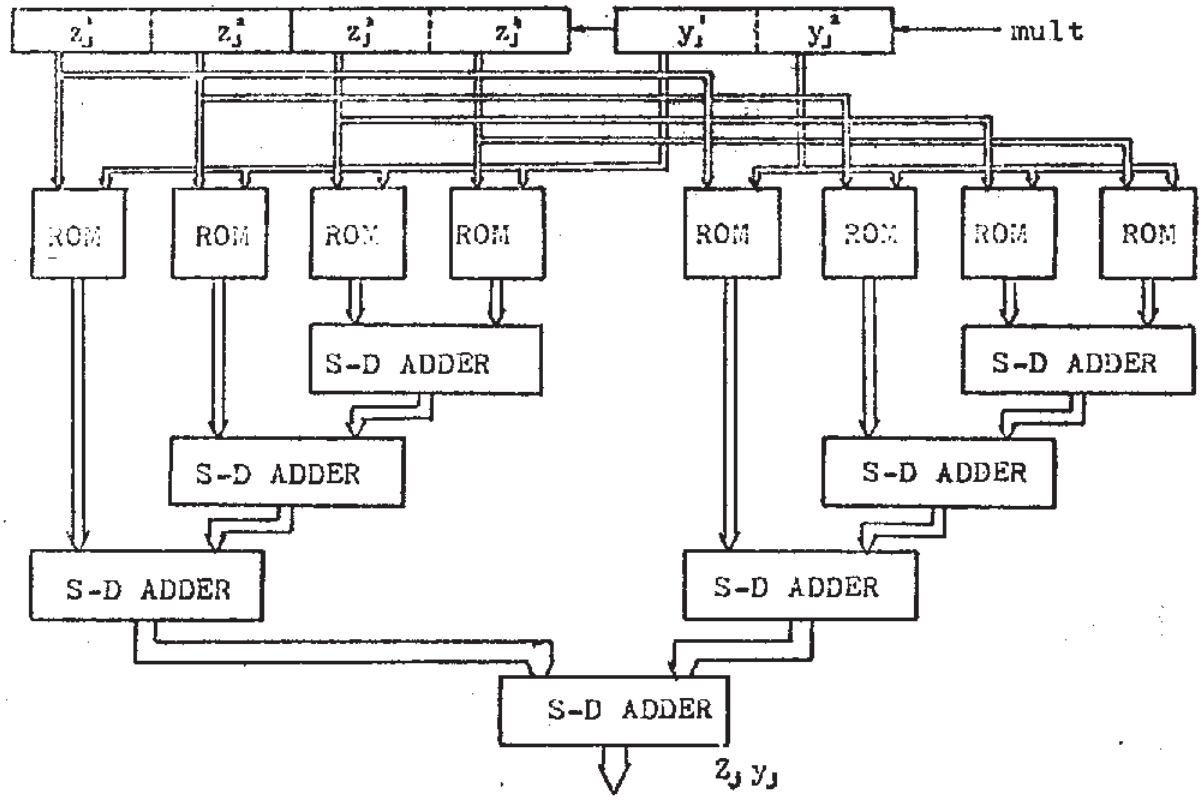


Fig.5.5 MULTSEL Structure

Basic idea is to break each of the numbers to be multiplied into digits. Then a digit of each, combined, will form an adress to a ROM which outputs their product. The size of the ROM is determined as follows: for a given base r, there are (2r-1) distinct digits possible for maximal redundancy (sect.2.1); since an address consists of two digits, and since each address has to reference a two-digit word (i.e. a byte), it can be concluded that the size of the ROM is to be $(2r-1)^2$ bytes. For base-8 representation, the number of bytes required is 15^2, or 225. This requires a standard 256*8 bit ROM. The size can be reduced if identical addresses such as (3,1) and (1,3) can be avoided, and if reduced, power consumption as well as access time of the ROM will go down. An algorithm enabling this reduction is as follows:

Let (x,y) be an address pair. Then

```
if x<y      then    temp=x
                    x=y      --- swopping operation ---
                    y=temp
            else    continue/direct address entry
```

Using this algorithm, the number of bytes required is significantly decreased. Whereas 225 bytes were required before, the size is now reduced to $\sum_{1}^{15} i = 120$. In general, for a given digit set of n elements, the number of spaces or slots required is computed to be $\sum_{1}^{n} i$. The implementation of this algorithm can be done using a single comparator which controls a multiplexer as shown in Fig. 5.6. Some additional time will be lost in comparison, but reduction in power consumption and access time will offset this loss.

Using table multiplication, the products are obtained as follows:

Example 2: Given $Z_j=7332_8$ and $Y_j=16_8$. Then

2*1	2	2*6	14
3*1	3	3*6	22
sum	32		234
3*1	3	3*6	22
sum	332		2434

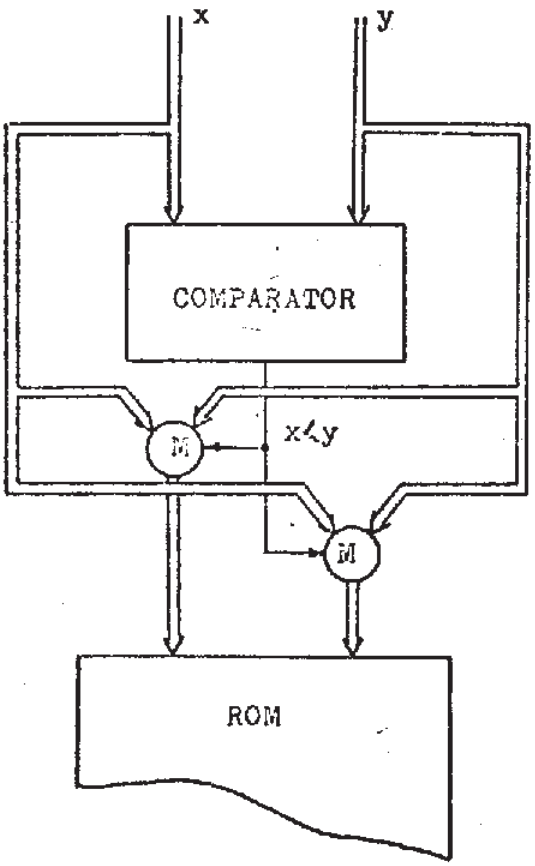
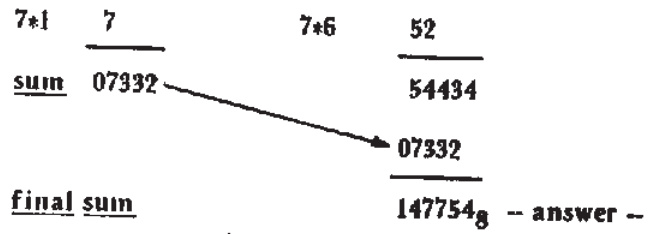


Fig.5.6 Configuration to improve size of ROMs



Therefore $Z_j y_j$ and $Y_{j-1} z_j$ are obtained using MULTSEL1&2 respectively, each product being 6 digits long (24 bits).

S-D ADD1 is a six digit parallel signed-digit adder; it computes the sum $Z_j y_j + Y_{j-1} z_j$ according to the addition algorithm described in section 3.2.

As mentioned in section 3.3, the multiplication algorithm places bounds on the magnitude of the operands that can be used here. In order to meet this requirement, it was also mentioned that the operands could be reduced during computation and then "renormalized" at the end of an operation. Since this "reduction" is done by the movement of the radix point, normal computation can be done by placing zeros where necessary. For this reason, two zeros are placed as most significant digits of the input arriving from S-D ADD1 to S-D ADD2 (Fig. 5.3), and the computation becomes as if each operand was shifted by two zeros, i.e. as if Z and Y were less than $\frac{1}{64}$. S-D ADD2 is therefore a 10 digit, parallel signed-digit adder, and it computes w_j.[1]

ROUND is a combinational logic circuit which performs a kind of rounding operation to obtain d_j from w_j.[2] It basically adds $\frac{1}{2}$ to the magnitude of w_j, truncates the digits to the left of the representative radix point and places the sign of w_j to the resultant number to obtain d_j. In this design, this can be done as follows:

a) if the output of S-D ADD2, w_j, is negative, then $-\frac{1}{2}$ is added to it. The digits to the right of the radix point are truncated, and the result is d_j.

b) if the output of S-D ADD2 , w_j, is positive, then $+\frac{1}{2}$ is added to it. The digits to the

---

1. $w_j = 64(w_{j-1} - d_{j-1}) + Z_j y_j + Y_{j-1} z_j$
2. $d_j = \text{sign} w_j \cdot \left\lfloor |w_j| + \frac{1}{2} \right\rfloor$

right of the radix point are truncated, and again the result is d_j.

It should be noted that in all operations, the location of the radix point is representative, but fixed and known.

The output of ROUND, d_j, is negated in NEG. Since d_j is constrained to be at the most two digits, i.e. a byte, the structure of this unit is the same as that described in the MODOP module of FPAS (sect. 4.4). The output of NEG is used to obtain the difference $(w_j - d_j)$ in S-D ADD3.

W-D REG is used to store the shifted difference, $64(w_{j-1} - d_{j-1})$. Initially, the content of this register has to be zero according to the multiplication algorithm. RESREG1&2 are registers for storing the result digits d_j. The latter is provided to keep the "last batch" of three result bytes that are produced at the same time. The output of this register is byte width, and through shifting all the contents are made available to the multiplexer M. The output of MULTOP is controlled by this multiplexer.

Example 3: Given the operation of section 3.3, example 7, then M will first pass d_2 and d_3 from RESREG1, followed by the output of RESREG2, giving the following sequence:

01 | 03 | 33 | 02 | 66

The description of the individual units have also explained the mode of operation of MULTOP. The control sequence can be seen in Fig.5.10.

The control signals of MULTOP are as follows:

1. *OPLOAD* places new operand bytes into OPREG1&2.
2. *SHFTLOAD* controls the switches SW1 and SW2 of each OPNDST unit. *OPERLOAD* is the load signal to OPND REGs.
3. *DEL-LOAD* is the load signal to OPND REG3 which stores Y_{j-1}.
4. *MULT* loads the "addresses" into MULT SEL1&2 as shown in Fig.5.5.
5. *ADD1* and *ADD2* are signals to the buffer registers (not shown) that store sums between

operations.

6. *DIFFLOAD* loads the shifted difference[1] into WDREG.
7. *LOADISHFT* can either load RESREG2 or shift its contents to the left bytewise. This is necessary to allow all contents of RESREG2 to appear on its byte-width output.
8. *RESLOAD* loads output of ROUND into RESREG1.
9. *RES-SEL* controls multiplexer M.

5.4 PACKOP Module

This module is similar to NORMOP of the floating point adder subtractor. It is responsible for preparing result-packets either from the computed result or from special operands. The structure of PACKOP is shown in Fig.5.7.

ZERO is a detection circuit which checks the byte input from MULTOP for zero digits. Given a byte of two digits where the left digit is the most significant one, there are 4 possible outcomes expressed as *ZERO* signal:

(1) only the left digit is zero,

(2) only the right digit is zero,

(3) both digits are zero, and

(4) none are zero.

The outcome of this detection basically control the normalization operation as will be explained later.

PACK is the mantissa-packing unit. Its internal structure is shown in Fig.5.8. In here, bytes arriving from REGIN are broken up if necessary (e.g. if there is a zero at the most significant digit) and then reconstructed. REG1 and REG2 are each 2 digits, i.e. 1 byte width registers. The mantissa in "final" form is added into these registers. Multiplexers M_a and M_b control the input to the left digit slot of REG1&2 respectively. On the other hand, M1

---

1. $64(w_{j-1}-d_{j-1})$

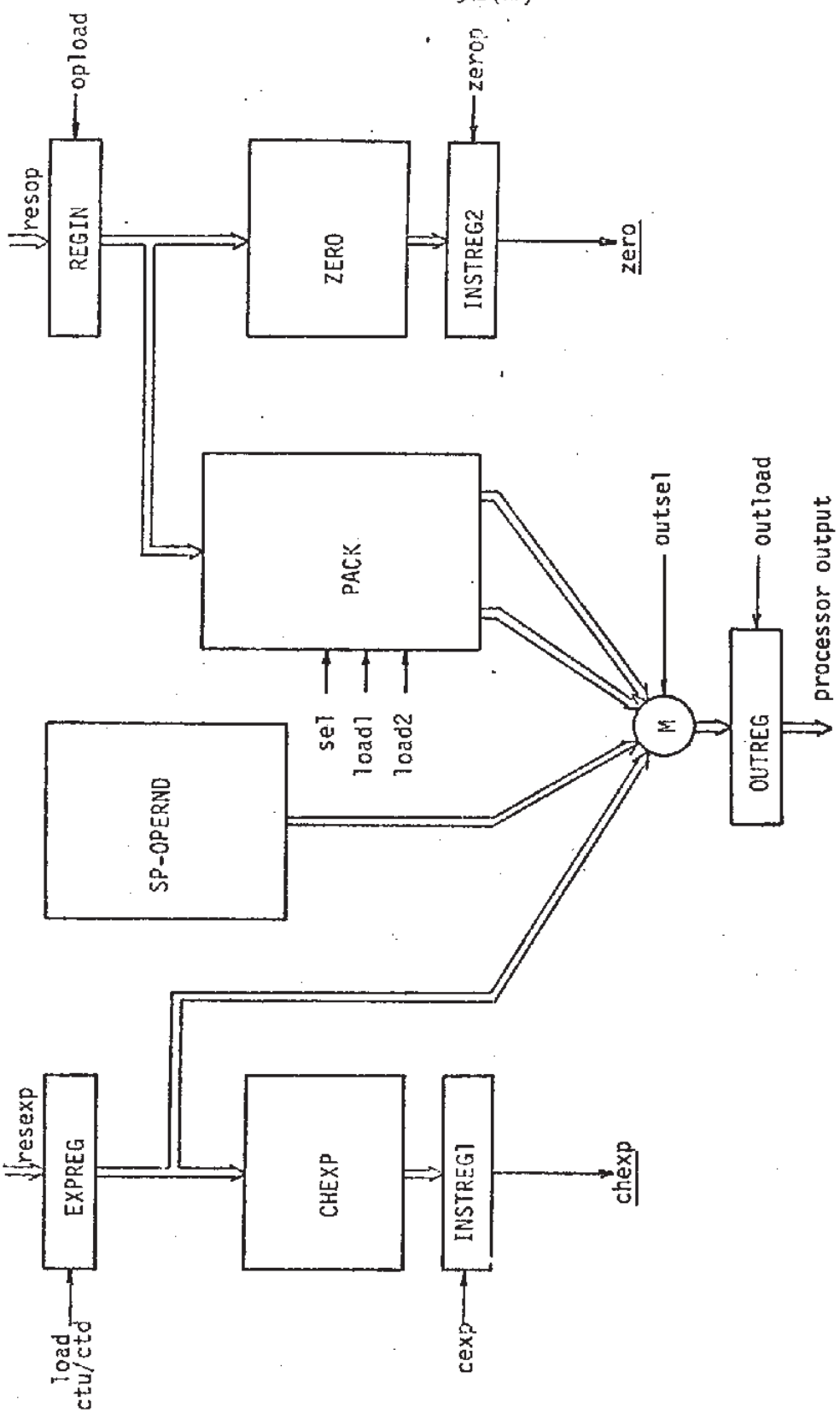


Fig.5.7 PACKOP Module

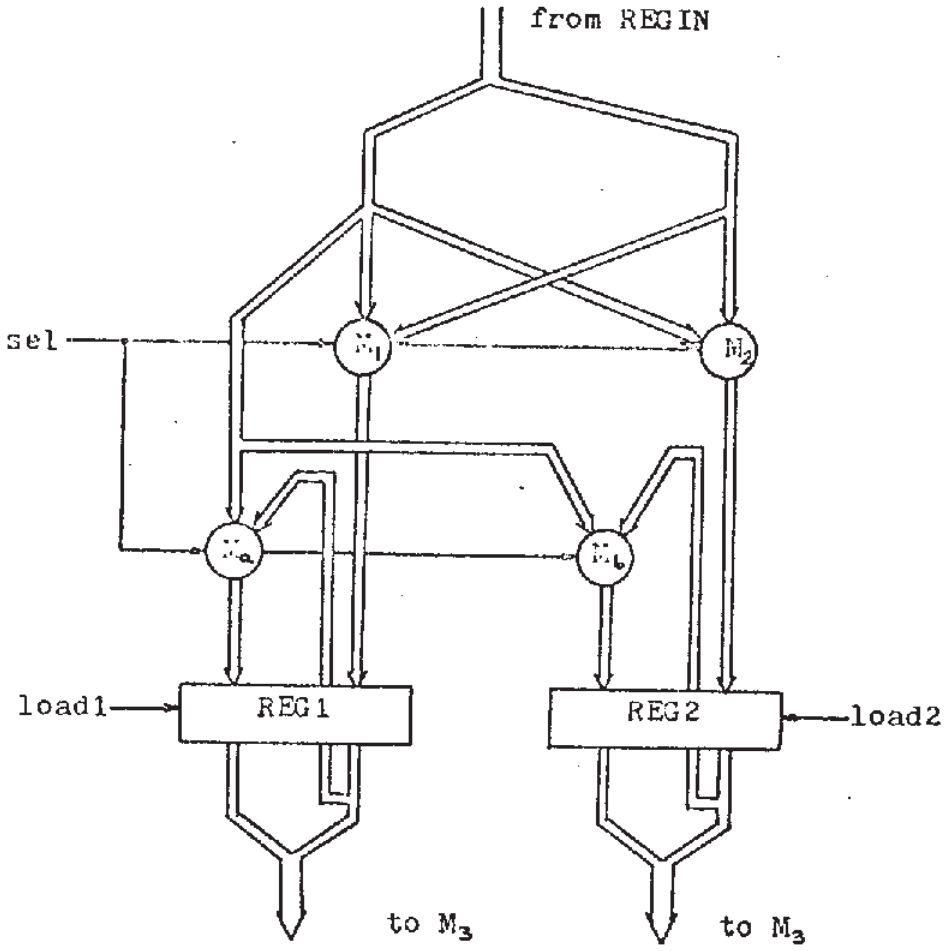


Fig.5.8 PACK Module

and M2 control the input to the right digit slot of these registers.

Example 4: Given a byte of two digits, let left and right digits of the j^{th} byte be l_j and r_j respectively. Then for the input sequence

$| l_1 r_1 | | l_2 r_2 | | l_3 r_3 | \ldots$,

the following initial REG1&2 configurations are possible:

	REG1	REG2	
1.	$\| l_1 r_1 \|$	$\| l_2 r_2 \|$	:no normalization necessary
2.	$\| r_1 l_2 \|$	$\| r_2 l_3 \|$	:most significant digit zero
3.	$\| l_2 r_2 \|$	$\| l_3 r_3 \|$	:first two digits zero

Various other forms are possible based on the above pattern (such as if the first two input bytes are all zeros, etc.).

SP-OPERAND is the special operand unit and it is the same as that in NORMOP.[1] Along the same lines, EXPREC and CHEXP are also the same as in NORMOP except that EXPREC here can handle exponents of size 1 byte + 1 bit due to the special "overflow" and "underflow" cases.

The operation of PACKOP is as follows:

1. If EXOP signals a special operand case, the multiplexer M3 chooses the input from SP-OPERAND as its output, and therefore a special operand packet is sent out.
2. Otherwise, in the normal and "limited" overflow&underflow case, MULTOP generates the result bytes. The computed result consists of 10 digits (5 bytes), and the first byte is used for mantissa overflow detection.

As explained in section 5.1, "limited" overflow and underflow occurs if the exponent is one more than the allowed limits. In order for such cases to end "normally", the computed mantissa must be less than r^{-1} for "limited" overflow, and there should be mantissa overflow for "limited" underflow. Therefore, the following is done:

---

1. *Section 4.6*

a. If there is "limited" underflow, then ZERO checks the first byte that arrives into PACKOP. If the result, *ZERO*, is (1) [1] then there is mantissa overflow; therefore the exponent is decremented, and normalization routine is carried out. Otherwise, the operation results in underflow and a special operand, ϵ, is sent out. The sign is obtained from the CHESIGN unit of EXOP.

b. If there is "limited" overflow, then ZERO checks the first two bytes to decide whether it is limited or not. The requirement is that the first 3 digits arriving at PACKOP be zero showing that the computed mantissa is less than r^{-1}. So the "minimal" requirement is to obtain possibilities (3) and (1) as *ZERO* values for the first two bytes respectively. If any of the first three digits is non-zero, then a special operand, ∞, is sent out with the sign of the first non-zero digit or from the CHESIGN unit of EXOP. Otherwise, if the result is less than r^{-1}, the rest of the digits are put into a normalized form and sent out.

The normalization procedure is the same as that of NORMOP, except that now the result digits from MULTOP arrive as bytes. Namely,

1. If there is mantissa overflow as indicated by non-zero second digit, then the exponent is incremented, checked for overflow. Now, a) if there is overflow, then + or - ∞ is sent out according to the sign of the overflow digit, else b) if there is no overflow, then the adjusted exponent plus the first 4 digits (including the overflow digit) are sent out.
2. If there is no mantissa overflow, then the third digit is checked. This time, a) if the digit is non-zero, then the exponent plus four digits including the digit being checked is sent out, otherwise b) if the digit is zero, then the exponent is decremented and checked for underflow. If there is no underflow, then an appropriate ϵ is chosen with the sign obtained from EXOP. If there is no underflow, then the fourth digit is checked, and the procedure recurses.

Example 3: Given computed mantissa

---

1. *See beginning of this section*

| 0 0 | | 0 0 | | 0 5 | | 4 4 | | 3 5 | and exponent E+72, then

First byte arrives	0 0	:both zero, hence no mantissa overflow
		ask for second byte
Second byte arrives	0 0	:3*rd* and 4*th* digits zero, decrement exp twice
		E+72 → E+70 ⇒ no underflow
		ask for third byte
Third byte arrives	0 5	:5*th* digit zero, decrement exp once
		E+70 → E+69 ⇒ no underflow
		6*th* digit non-zero → send next four digits
		Place 5 in REG1. Ask for fourth byte.
		Send exp out.
Fourth byte arrives	4 4	:Place 4 in REG1 ⇒ send out contents (i.e. 54)
		Place 4 in REG2. Ask for fifth byte.
Fifth byte arrives	3 5	:Place 3 in REG2 ⇒ send out contents (i.e. 35)
		OPERATION FINISHED

Operand-pkt sent out: .5443 E+69

It should be noted that when the second byte arrives, both digits are checked at the same time, and the next two steps can be taken "at-one-go".

The control signals as shown in Fig.5.7 are as follows:

1. *OPLOAD* controls buffer register REGIN.
2. *LOAD* places the unnormalized exponent arriving from EXOP into register/counter EXPREG. *CTU/CTD* are signals that can increment or decrement the contents of EXPREG.
3. *SEL* controls multiplexers M1, M2, M_a and M_b.
4. *LOAD1* and *LOAD2* are the load signals to REG1 and REG2 (of PACK unit) respectively.

5. *OUTSEL* is used to select the output of FPM using multiplexer M3. As mentioned before, it can select its output from the following inputs: 1) normalized exponent from EXPREG, 2) special operand bytes from SP-OPERAND, and 3) normalized mantissa bytes from PACK.
6. *ZEROP* places the result of the ZERO test into INSTREG2.
7. *CEXP* loads chexp signal into INSTREG1.

5.5 Control and Timing

Operation of FPM begins with the arrival of "ready" signal from the external circuit indicating the availability of a new operand packet. MPX begins processing the incoming bytes, and when the exponents are ready,[1] it signals EXOP to start its operations. EXOP checks for special operands, and if any is found, it signals MPX to prepare the first mantissa bytes so that the sign of the special operand to be sent out can be decided. Therefore when SIGN unit of EXOP finishes its operation, MPX is signalled to prepare for a new operand packet, while PACKOP starts sending out an appropriate special operand. In other words, normal operations are not done.

If EXOP cannot find special operands, then it proceeds to manipulate the exponents as described in section 5.2. When the exponents are added, it may be the case that there will be definite overflow or underflow. If this happens, the same procedure as the special operand case is followed, except that the special operand to be sent out is either ∞ or ϵ.

If the exponent overflow or underflow is "limited", then EXOP informs PACKOP of this condition, but the rest of the system operates normally. In normal mode, MPX is signalled to prepare the mantissa bytes to MULTOP, while PACKOP is sent a "ready" to load the unnormalized exponent. The multiplication proceeds according to ready/acknowledge signals between MPX & MULTOP, and MULTOP & PACKOP. So

1. *See section 4.2*

MULTOP begins when MPX has the first mantissa bytes available. When the first result byte, i.e. d_1, is produced, PACKOP is signalled to start its normaliztion operation. MULTOP is finished with a given operand packet when register RESREG 2 is loaded with the last three result bytes.

Finally, when PACKOP prepares the output of the FPM, it signals the external "world" that the result packets are ready. There is also a signal protocol between MULTOP and PACKOP to cope with the problem of excess or insuufficient number of digits which may occur during normalization. This topic is discussed in detail in section 4.6. Fig.5.9 summarizes the control signals of FPM.

Analysis of the computation time is done with petri nets with timed transitions. These petri nets can be used to model the control structure as well as to estimate the time of computations. Petri Nets for the FPM can be found in the Appendix (A1, A8, A9). It should be noted that this petri-net could easily be extended to handle multiplication of mantissas of length n bytes; the section between the "stars" in the figure is basically the main part of the algorithm and therefore it is repeated for each byte input.

A typical computation time has been computed as an example. The operation is assumed to produce a result mantissa that requires 1 digit left shift while normalizing. Various times assigned to operations are typical TTL logic values.

Computation begins at time zero when MPX receives an operand_packet. As in the FPAS, the exponents are ready to be processed by EXOP after 80ns. EXOP takes approximately 75 ns to process the exponents. For example this value is obtained as follows:

EXPLOAD	register load	20 ns
CLOAD	comparison	20 ns
ADD	binary additon	15 ns
COMP	comparison	20 ns
	TOTAL	75 ns

Now, MPX begins preparing the next operand bytes at about 155 ns, and MULTOP is ready

MODULE	EXTERNAL	INTERMODULAR
MPX	opnd-pkt ready/ack	exp ready/ack(to/from EXOP) op ready/ack (to/from MULTOP & EXOP)
EXOP	none	cexp (special opnd)(to/from MPX&PACKOP) decexp(over/underflow after addition)(to PACKOP) sign(to PACKOP) res-exp ready/ack(to/from PACKOP)
MULTOP	none	res-ready ready/ack (to/from PACKOP) oper-end(last data)(to PACKOP)
PACKOP	res-pack ready/ack	dummy-ack (to MULTOP)

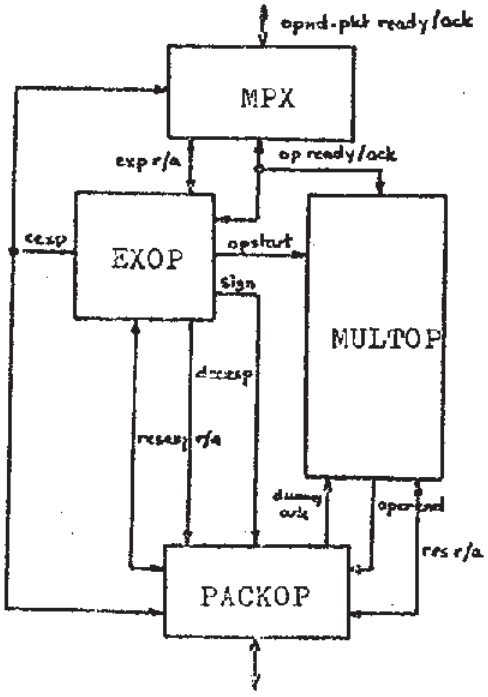


Fig.5.9 Main control signals

to start at 175 ns. Various operation times associated with this module can be seen in Appendix, A9. This analysis then continues down to PACKOP. The start and end times for each module can be seen in Fig.5.10(a).

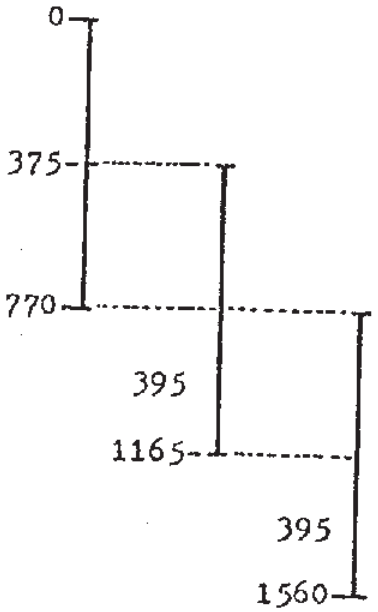
The conclusion is that the multiplication of two digit numbers takes approximately 770 ns with the given TTL values. This is the time that elapses between the arrival of the first byte, and the sending out of the last result byte. With pipelinig, overlapping of operations is possible as shown in Fig.5.10(b). This means that for continuous operations, FPM can produce a result packet every 395 ns, or at a rate of 2.5 MHz; stated in a different way, the input byte "consumption" rate of approx. 15 MHz can be achieved.

These figures can be lowered with the use of faster logic. As the precision will increase, the time of computation will definitely go up. As in the case of the FPAS, the trick lies in providing efficient hardware to speed up normalization process.

MODULE	START	END	TOTAL
MPX	0 ns	375 ns	375 ns
EXOP	80 ns	155 ns	75 ns
MULTOP	175 ns	700 ns	525 ns
PACKOP	430 ns	770 ns	340 ns

Fig.5.10(a) Operation times

MPX is free at 375 ns. If a new operand-pkt is processed RESREG1 is loaded with result after 430 ns. Starting at time 375, that means the first result byte is available at 805 ns, when the first operand-pkt is already processed; thus conflict is avoided.



time for n computations:

$t = 770 + (n-1)395$ ns

Throughput: 1 result/395 ns

2.5 MHz rate

Fig.5.10(b) Overlapping

CHAPTER 6: CONCLUSION

In this thesis, two processors capable of byte-serial addition-subtraction and multiplication have been described. As the design shows, using signed digit arithmetic and pipelining methods, a high byte processing rate can be achieved; although the computation times mentioned in sections 4.7 and 5.5 do not look very impressing, better performance can be obtained by the use of faster logic and methods that can eliminate some of the "bottle-neck" units. For example if such processors are actually to be implemented, an LSI chip would promise reasonable improvement over the given figures.

In the description of the processors, various points have been left out. One of these is the handling of multiple-precision. The operand packets and operations described in this thesis are for single precision only, i.e. the operand format is fixed. However adapting to a fixed format or variable multiple precision environment is easy. Modifications involve the algorithms (certain steps have to be repeated according to the precision used) and expansion of the delay buffers such as DEL of the MPX module.[1] Provisions have to be made to inform modules such as EXPFIX of the FPAS of the length of the mantissa used to determine the result exponent. For example, ∂ input to COMP unit of EXPFIX can be connected to an external source thus informing the module of the new constraints on the exponent difference magnitude.

Certain operations can be avoided by checking the operands before computation, as mentioned in section 3.1. Structures can be provided to detect these cases if feasible.

Basically, there are two possibilities for further work. Division has not been mentioned in this thesis. An algorithm utilising signed digit arithmetic is described in [TE 1]. The division algorithm defined produces the first result digits after 4 operand digits are available to the processor.

---

1. *See section* 4.2

Another issue that requires further work is on how such individual processors can be connected together to form an arithmetic unit. For example both FPAS and FPM are fixed operators, e.g. an operand arriving at FPM can only be multiplied. Therefore a mechanism has to be provided to route arriving operand packets to the appropriate unit according to the operation indicated by the packet. Also it is necessary to put the result packet into the proposed form in the Data Flow Machine, i.e. append destination addresses to the result, etc.[1] Of course these issues involve the actual design of the DFM, and therefore depend heavily on how the latter is implemented.

1. *See* [DML 1]

APPENDIX: Petri Net Graphs

A1.	MPX
A2.	EXPFIX
A3.	MODOP
A4.	ADDOP
A5.(a)	NORMOP
A5.(b)	NORMOP
A6.	Formatting procedure of NORMOP(out-proc)
A7.	Procedures for NORMOP operations
A8.	EXOP
A9.	MULTOP

BIBLIOGRAPHY

[AV 1] Avizienis, A.: Signed-Digit Number Representations for Fast Parallel Arithmetic, IRE Transactions on Electronic Computers, EC-10 (1961), 389-400.

[AV 2] Avizienis, A.: Binary Compatible Signed-Digit Arithmetic, Proceedings, Fall Joint Conference, 1964, 663-671.

[DM 1] Dennis, J.B., and D.P. Misunas: A Preliminary Architecture for a basic Data-Flow Processor, Computation Structures Group Memo 102, Project MAC, MIT, August 1974.

[DML 1] Dennis, J.B., C. Leung, and D.P. Misunas: Specification of the Instruction Cell Block for a Data Flow Processor, Data Flow Design Note 1, LCS, MIT, December 16, 1975.

[DML 2] Dennis, J.B. , D.P. Misunas, and C. Leung: A Highly Parallel Processor Using a Data Flow Machine Language, Computation Structures Group Memo 134, LCS, MIT, January 1977.

[JD 1] Dennis, J.B.: private discussion, January 1978.

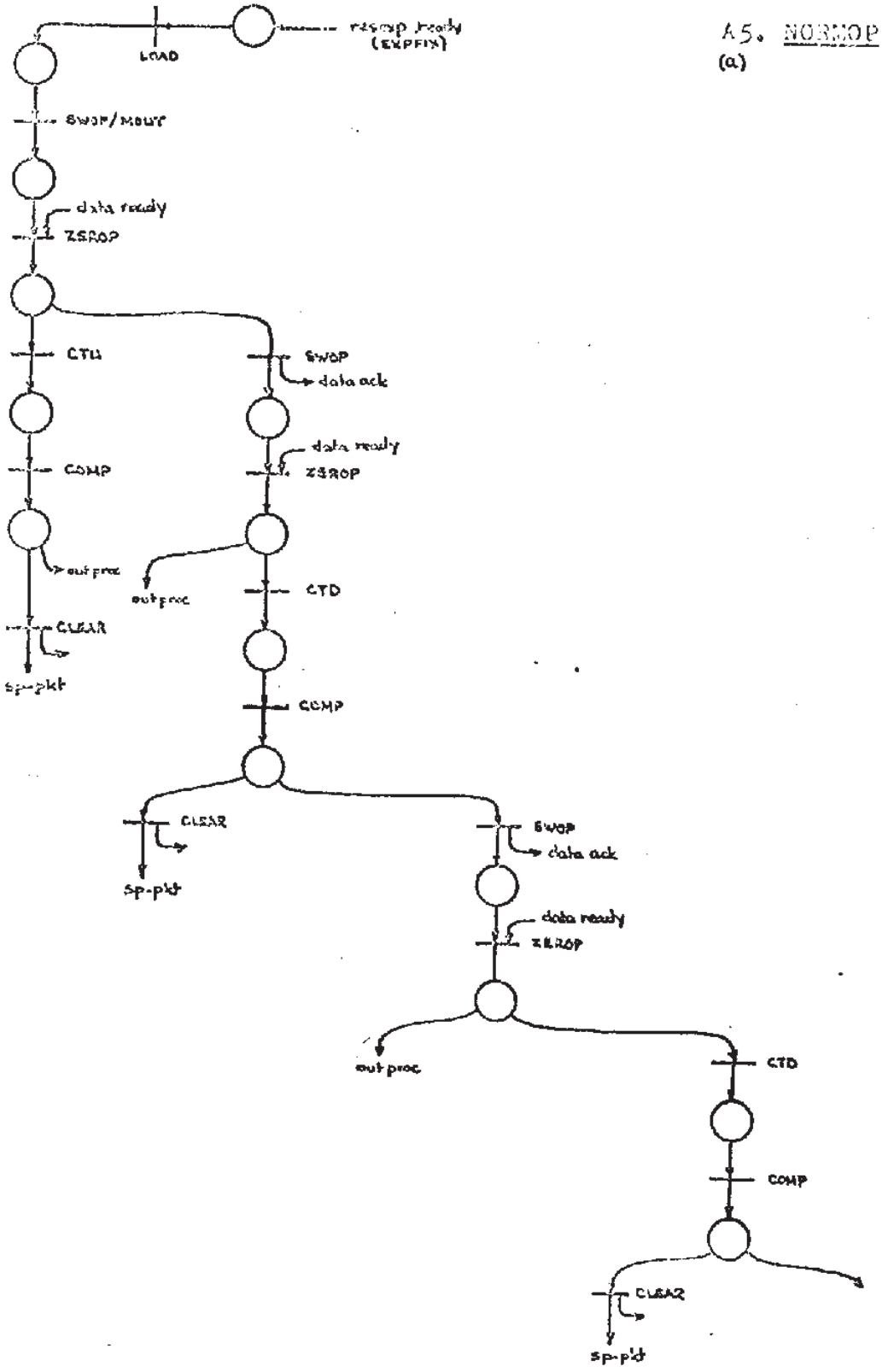
[RAM 1] Ramchandani, Chander: Analysis of Asynchronous Concurrent Systems by Petri Nets, Technical Report TR-120, LCS, MIT, February 1974.

[TE 1] Trivedi, K., and Ercegovac, M.: On-line Algorithms for Division and Multiplication, University of Illinois at Urbana-Champaign, 1976.

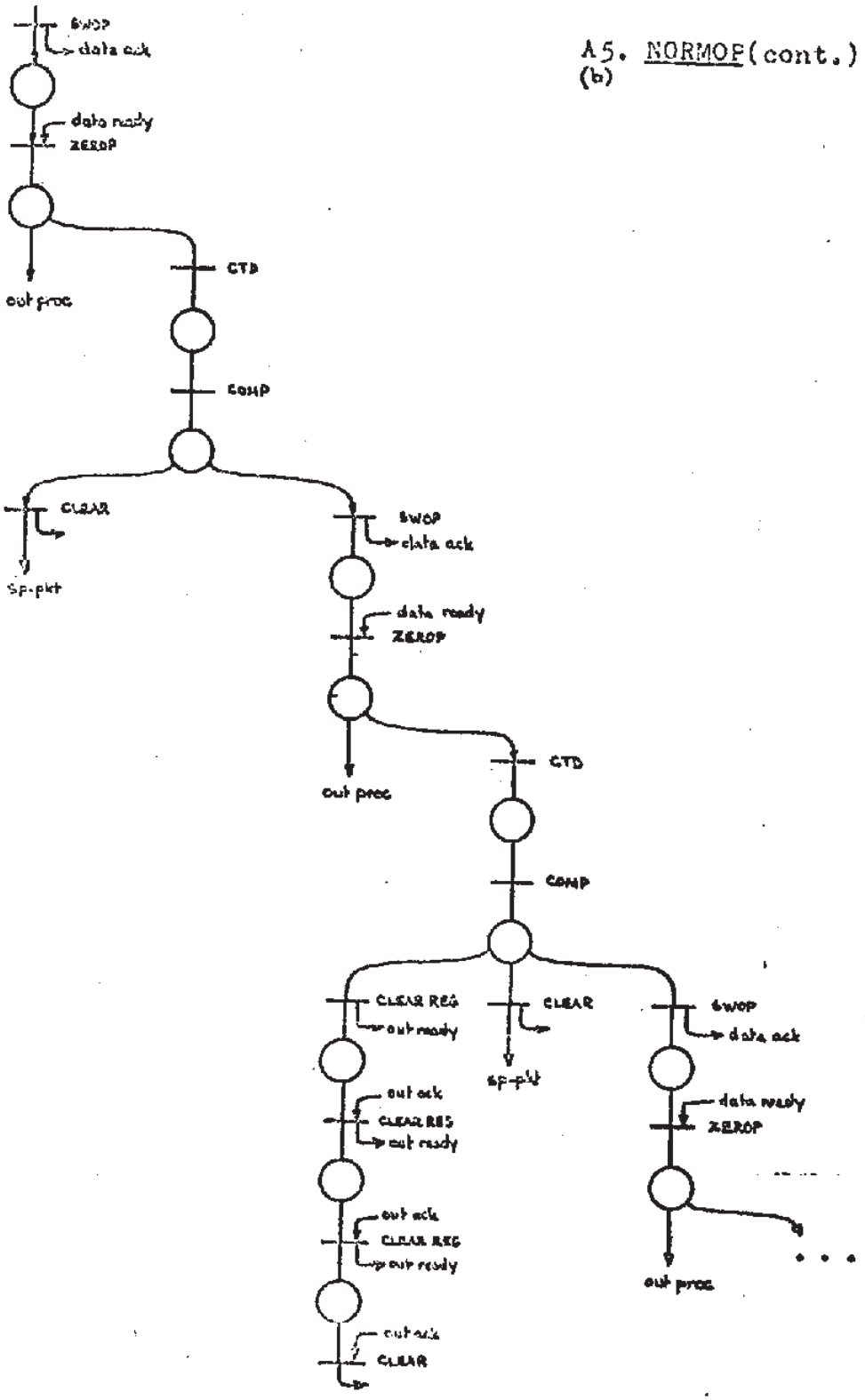
[TH 1] Thornton, J.E.: Design of a Computer--The Control Data 6600, Scott, Foresman and Company, Glenview, Illinois, 1970.

[TI 1] Texas Instruments: The TTL Data Book for Design Engineers, Texas Instruments Inc., Dallas, Texas, 1979.

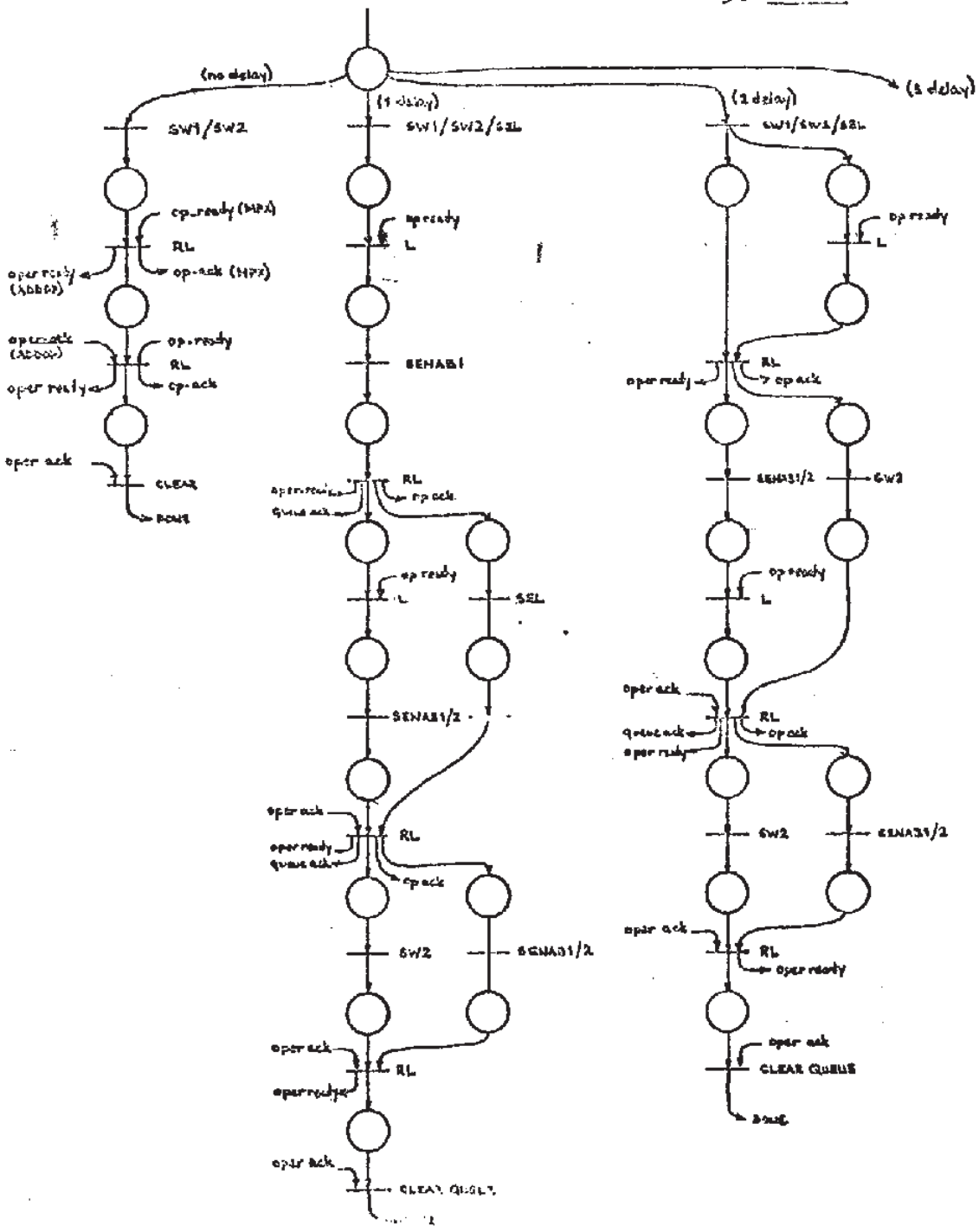
A5. NORMOP
(a)


LOAD
resop ready (EXPFIX)
SWOP/MOUT
data ready
ZSROP
CTU
SWOP
data ack
COMP
data ready
ZSROP
out proc
out proc
CTD
CLEAR
sp-pkt
COMP
CLEAR
sp-pkt
SWOP
data ack
data ready
ZSROP
out proc
CTD
COMP
CLEAR
sp-pkt

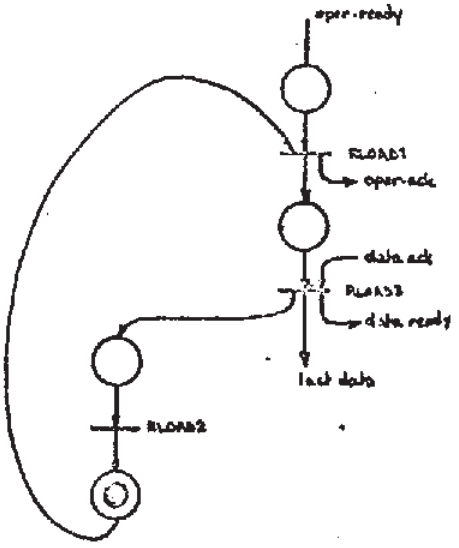
A5. NORMOP(cont.)
(b)



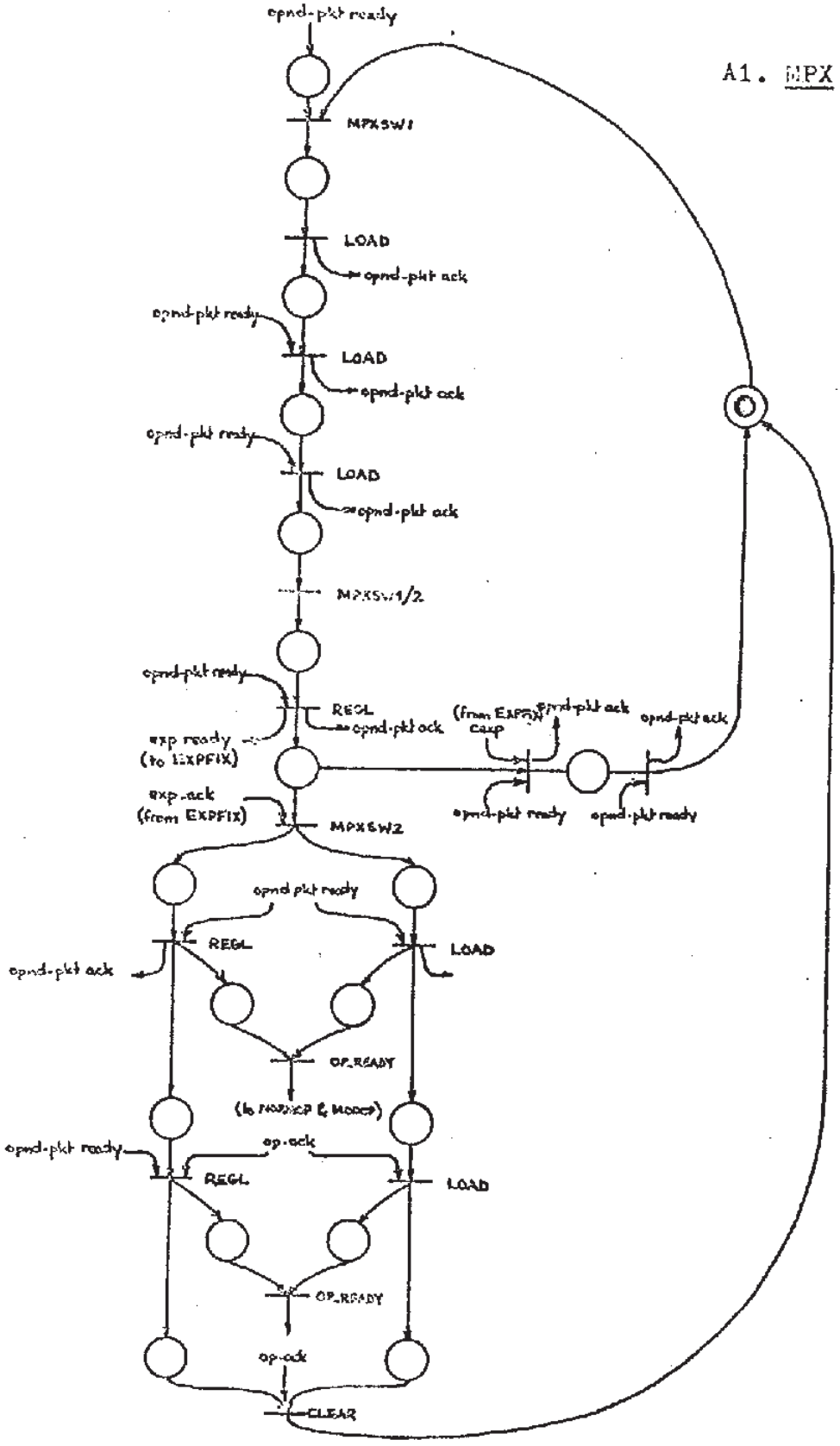
A3. MODOP


(no delay)
(1 delay)
(2 delay)
(3 delay)
SW1/SW2
SW1/SW2/SEL
SW1/SW2/SEL
op_ready (MPX)
RL
op-ack (MPX)
oper ready (ADDER)
op_ready
L
op-ack
oper ack (ADDER)
op_ready
RL
cp-ack
oper ready
oper ack
CLEAR
ENABLE
RL
oper ready
queue ack
op ack
op ready
L
SEL
SENABLE/2
oper ack
oper ready
queue ack
RL
op ack
SW2
SENABLE/2
oper ack
RL
oper ready
oper ack
CLEAR QUEUE
op ready
L
RL
op ack
oper ready
SENABLE/2
SW2
op ready
L
oper ack
RL
op ack
queue ack
oper ready
SW2
SENABLE/2
oper ack
RL
oper ready
oper ack
CLEAR QUEUE

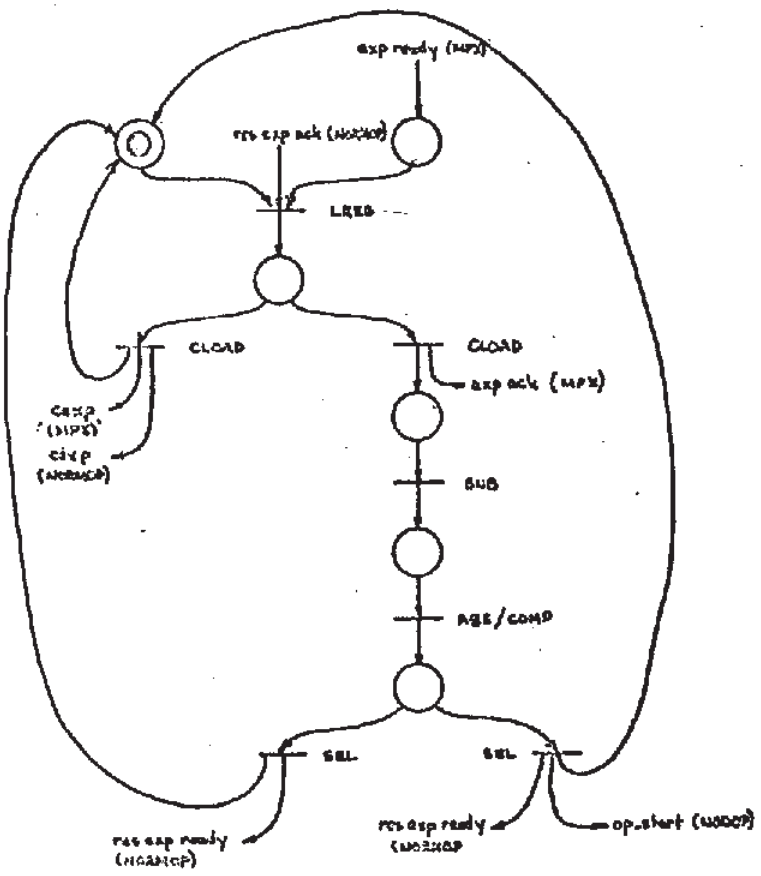
A4. ADDOP


oper-ready
FLOAD1
oper-ack
data-ack
FLOAD3
data-ready
last data
FLOAD2

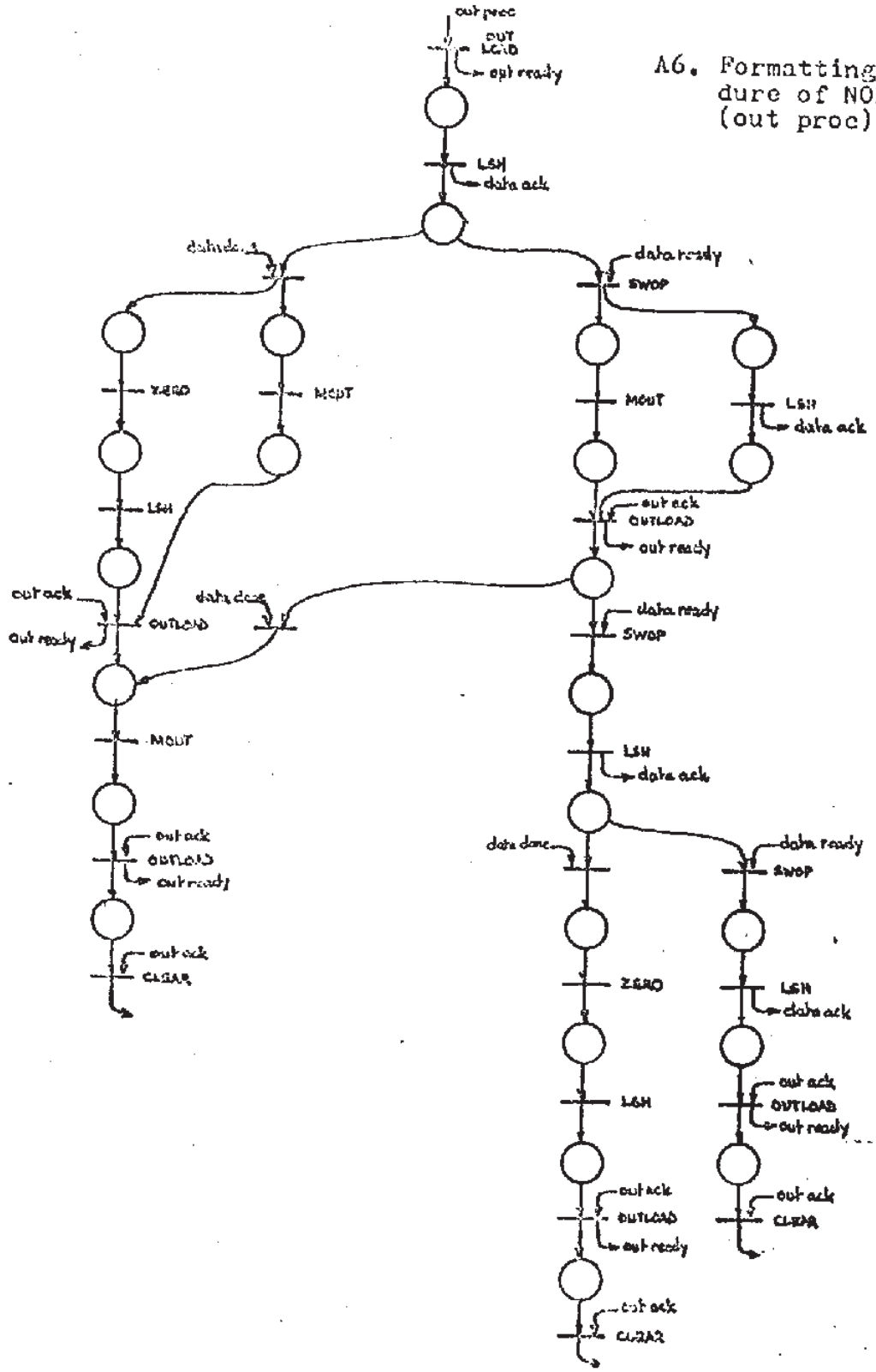
A1. MPX


opnd-pkt ready
MPXSW1
LOAD
opnd-pkt ack
opnd-pkt ready
LOAD
opnd-pkt ack
opnd-pkt ready
LOAD
opnd-pkt ack
MPXSW1/2
opnd-pkt ready
REGL
opnd-pkt ack
exp ready
(to EXPFIX)
(from EXPFIX)
exp
opnd-pkt ack
opnd-pkt ack
opnd-pkt ready
opnd-pkt ready
exp-ack
(from EXPFIX)
MPXSW2
opnd pkt ready
REGL
LOAD
opnd-pkt ack
OP_READY
(to MPXSEL & MODE)
opnd-pkt ready
op-ack
REGL
LOAD
OP_READY
op-ack
CLEAR

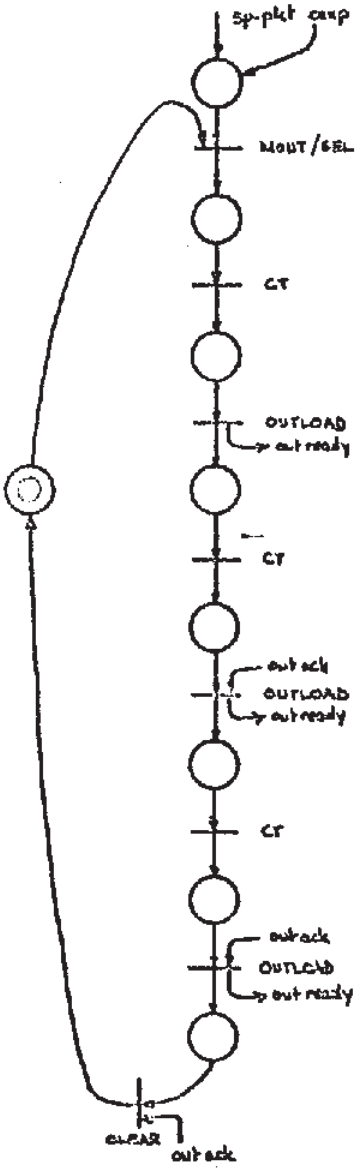
A2. EXPFIX


exp ready (MPX)
res exp ack (MOSAIC)
LREQ
CLOAD
CLOAD
exp ack (MPX)
SUB
AGE/COMP
SEL
SEL
op_start (MOSAIC)

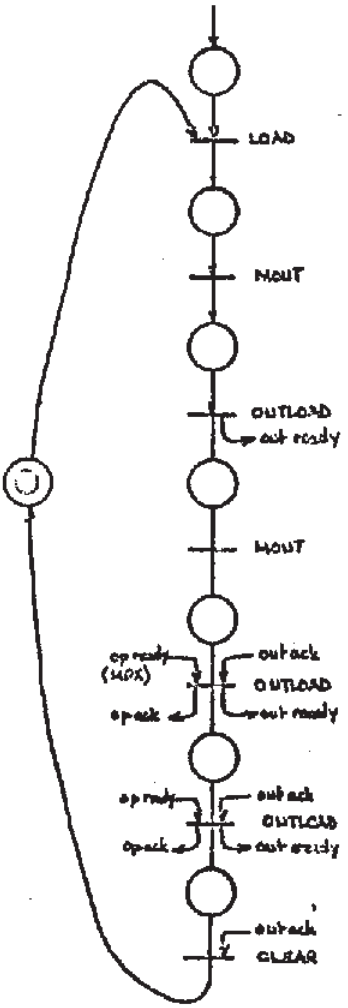
A6. Formatting procedure of NORMOP (out proc)



A7. Procedures for
NORMOP operations

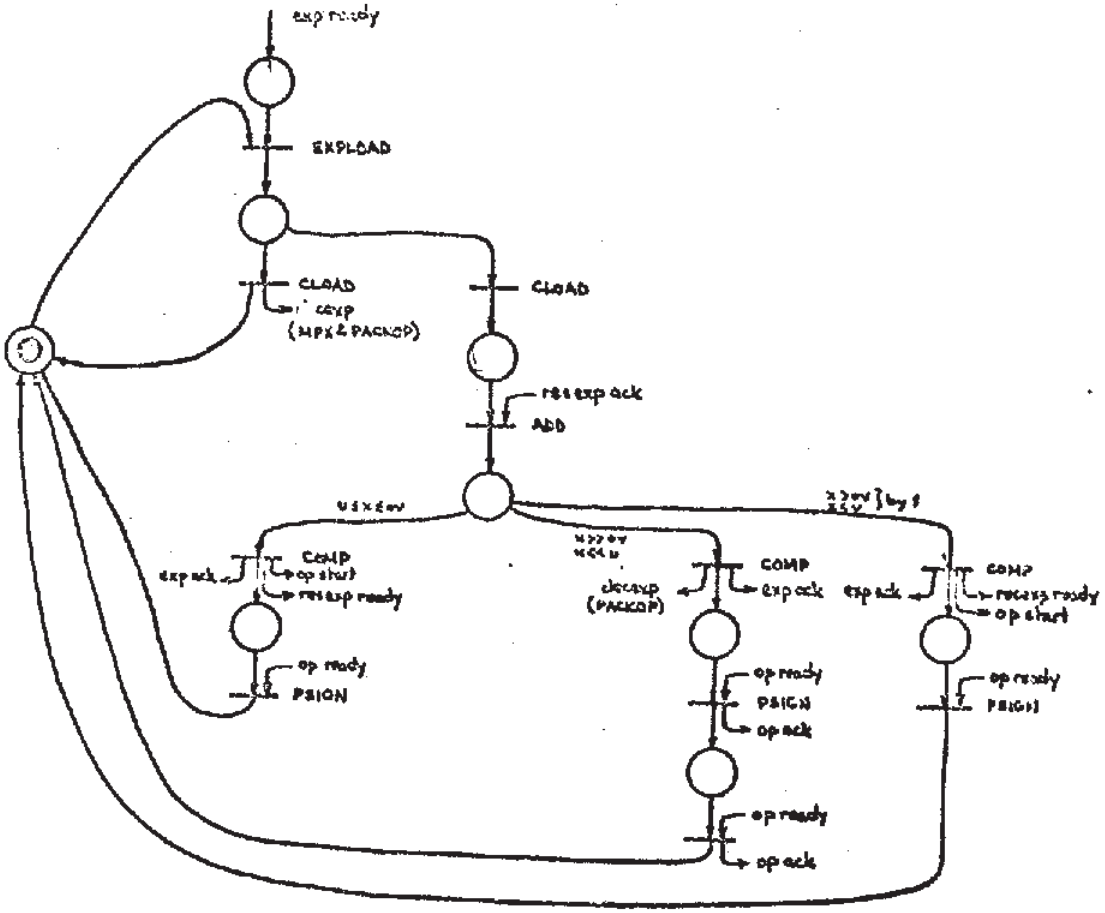


Special operand procedure
(sp-pkt)

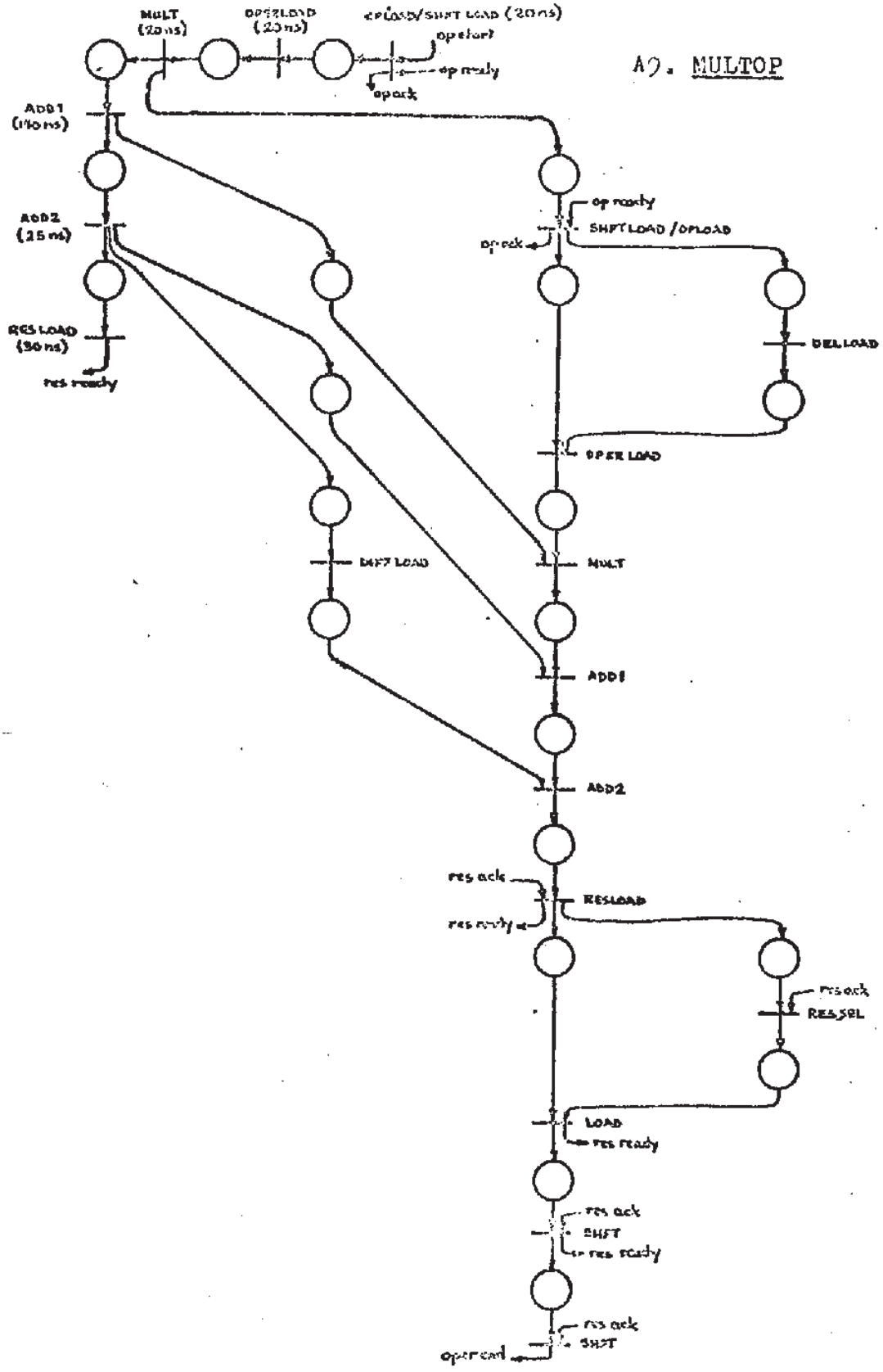


"Bypass" procedure

A8. EXOP


exp ready
EXPLOAD
CLOAD
cexp
(MPX & PACKOP)
CLOAD
res exp ack
ADD
COMP
op start
res exp ready
exp ack
op ready
PSIGN
COMP
exp ack
clear exp
(PACKOP)
op ready
PSIGN
op ack
op ready
op ack
COMP
exp ack
res exp ready
op start
op ready
PSIGN

A9. MULTOP


MULT (20 ns)
OPER LOAD (20 ns)
OPER LOAD / SHIFT LOAD (20 ns)
op start
op ready
op ack
ADD1 (140 ns)
ADD2 (25 ns)
RES LOAD (30 ns)
res ready
op ready
SHIFT LOAD / DPLOAD
op ack
DEL LOAD
OPER LOAD
SHIFT LOAD
MULT
ADD1
ADD2
res ack
RESLOAD
res ready
res ack
RESSEL
LOAD
res ready
res ack
SHIFT
res ready
res ack
SHIFT
operand