A VERY LARGE SCALE INTEGRATION IMPLEMENTATION OF AN ON LINE ARITHMETIC UNIT

Dean Michael Tullsen

June 1986
CSD-860094

ABSTRACT OF THE THESIS

A Very Large Scale Integration Implementation of an On-line Arithmetic Unit

by

Dean Michael Tullsen

Master of Science in Computer Science

University of California, Los Angeles, 1986

Professor Milos D. Ercegovac, Chair

The design of an on-line arithmetic unit to compute AX+B is given here, with *A* and *X* on-line and *B* off-line and the result produced on-line a small delay after the initial inputs. On-line arithmetic has the potential advantages of high concurrency, few inter-connections, and fast operation of complex arithmetic functions. The purpose of this project is to produce performance and cost measurements of an actual VLSI implementation of an on-line arithmetic unit. Other goals of the project include providing a modular design and limiting connections to nearest neighbors in order to be able to use the unit for any size operands. Results of the project include timing and area results for the chip as well as a measure of speedup results for some applications of the on-line arithmetic unit over conventional arithmetic solutions.

UNIVERSITY OF CALIFORNIA

Los Angeles

A Very Large Scale Integration Implementation of an On-line Arithmetic Unit

A thesis submitted in partial satisfaction of the requirements for the degree of Master of Science in Computer Science

by

Dean Michael Tullsen

1986

The thesis of Dean Michael Tullsen is approved.

[illegible]

Vance Tyree

[illegible]

Tomas Lang

[illegible]

Milos D. Ercegovac, Committee Chair

University of California, Los Angeles

1986

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGEMENTS

I would like to express my sincere appreciation to my advisor, Prof. M. D. Ercegovac for his help, guidance and encouragement, without which this work would have been neither begun nor completed.

I am also grateful toward the rest of my Thesis committee members, particularly Dr. Lang for his encouragement and close scrutiny of the manuscript.

Not enough can be said about my loving wife, Nancy, and her support and patience in the completion of this project.

But most of all, I must thank my Lord and Savior Jesus Christ, without whom all successes would be meaningless.

I would also like to thank the Office of Naval Research for their support. This research was supported in part by the ONR Contract N00014-85-K-0159 "On-Line Arithmetic Algorithms and Structures for VLSI."

ABSTRACT OF THE THESIS

A Very Large Scale Integration Implementation of an On-line Arithmetic Unit

by

Dean Michael Tullsen

Master of Science in Computer Science

University of California, Los Angeles, 1986

Professor Milos D. Ercegovac, Chair

The design of an on-line arithmetic unit to compute AX+B is given here, with *A* and *X* on-line and *B* off-line and the result produced on-line a small delay after the initial inputs. On-line arithmetic has the potential advantages of high concurrency, few inter-connections, and fast operation of complex arithmetic functions. The purpose of this project is to produce performance and cost measurements of an actual VLSI implementation of an on-line arithmetic unit. Other goals of the project include providing a modular design and limiting connections to nearest neighbors in order to be able to use the unit for any size operands. Results of the project include timing and area results for the chip as well as a measure of speedup results for some applications of the on-line arithmetic unit over conventional arithmetic solutions.

CHAPTER 1
INTRODUCTION

The purpose of this project is to design an on-line (most significant digit first) arithmetic unit, to be implemented and tested, so as to study the performance and analyze the usefulness of on-line arithmetic in VLSI implementations. For the purposes of this project, we implement the very basic function $Y = AX + B$, with operands A and X on-line and B off-line. The result, Y, is produced in an on-line manner. Although this is a very simple function, it is designed in such a fashion that several levels of these chips may be applied to implement more complex functions [ERCE75].

By on-line arithmetic we mean those algorithms in which the operands as well as the results flow through the arithmetic unit in a digit-by-digit fashion, most significant digit first. These algorithms are such that the jth digit of the result is generated after $j + \delta$ digits of the corresponding operands are input, where δ is the on-line delay or latency [ERCE84].

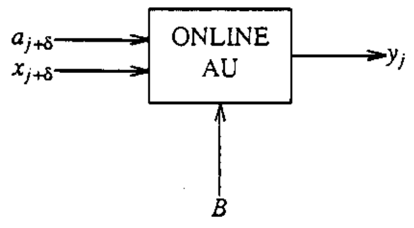

Figure 1.1 -- An On-line Arithmetic Unit

The use of redundant number representations is mandatory for on-line algorithms because with conventional number representations it is usually impossible to know digit i of the result until the entire operation is completed due to the possibility of carry propagation. For this arithmetic unit the inputs and outputs are in binary signed-digit format [AVIZ61] rather than the more conventional two's complement or, more generally, range complement representations. In this (signed digit) representation, each digit has three possible values (1, 0, -1) and is represented by two bits, which could be considered a sign bit and a magnitude bit. (-1) is represented by 11, (0) by 00, and (1) by 01.

This arithmetic unit is designed in modules so that any length operand may be accommodated by putting together the correct number of identical modules. The single module must be able to perform the operations needed by modules at any position within the array of modules (for example first, last, middle). An alternative to this approach would be to use three types of modules. The basic complexity differences between these alternatives are discussed later in the paper.

Basic Theory

The arithmetic unit implements the arithmetic expression AX+B with A and X on-line, most significant digit first, in a radix-two signed-digit format. B is assumed to be available off-line in two's complement form. Outputs are produced on-line in signed-digit form. The derivation of the algorithm follows [ERCE75, TRIV77].

The operands A and X are of the form:

$$X = \sum_{i=0}^{m} x_i 2^{-i}, \qquad x_i \in \left\{ -1,\ 0,\ 1 \right\}$$

$$A = \sum_{i=0}^{m} a_i 2^{-i}, \qquad a_i \in \left\{-1,\ 0,\ 1\right\}$$

In on-line form

$$X_j = \sum_{i=0}^{j} x_i 2^{-i} = X_{j-1} + x_j 2^{-j} \tag{1.1}$$

$$A_j = \sum_{i=0}^{j} a_i 2^{-i} = A_{j-1} + a_j 2^{-j}$$

Notice in the notation that A_j is the vector of digits 0 through j and a_j is the jth digit of A.

Therefore, the product at the jth step is:

$$X_j A_j + B = X_{j-1} A_{j-1} + X_{j-1} a_j 2^{-j} + A_{j-1} x_j 2^{-j} + x_j a_j 2^{-2j} + B$$

$$= X_{j-1} A_{j-1} + (X_j a_j + A_{j-1} x_j) 2^{-j} + B \tag{1.2}$$

Let P_j be the scaled partial product at step j:

$$P_j = X_j A_j 2^j + B \tag{1.3}$$

Then the partial product recurrence is:

$$P_j = X_{j-1} A_{j-1} 2^j + (X_j a_j + A_{j-1} x_j) 2^{-j} 2^j + B \tag{1.4}$$

$$= 2P_{j-1} + (X_j a_j + A_{j-1} x_j)$$

where $P_0 = B$.

Let d_i be the ith computed product digit. Then

$$w_j = P_j - 2^j D_{j-1}\ , \quad D_{j-1} = \sum_{i=0}^{j-1} d_i 2^{-i}, \qquad d_i \in \left\{-1,0,1\right\} \tag{1.5}$$

is the *j*th residual, i.e., the difference between the true and computed partial product. The residual recurrence is:

$$w_j = 2P_{j-1} + X_j a_j + A_{j-1} x_j - 2^j D_{j-2} - 2d_{j-1} \tag{1.6}$$

$$w_j = 2(w_{j-1} - d_{j-1}) + X_j a_j + A_{j-1} x_j \tag{1.7}$$

or, defining $z_j = w_j - d_j$:

$$w_j = 2(z_{j-1}) + X_j a_j + A_{j-1} x_j$$

At step m

$$A_m X_m + B = \sum_{i=0}^{m} d_i 2^{-i} + (z_m) 2^{-m} \tag{1.8}$$

In order to ensure that the output, D_j converges toward the desired value of $A_j X_j + B$, we must choose d_j each step such that z_j, the residual, is no greater than half the value of the current digit position. To accomplish this, we must insure that w_j be less that 3/2 since d is a multiple of 1, and the result digit d_j is selected according to the following rule:

$$d_j = sign(w_j)^* \left\lfloor |w_j| + \frac{1}{2} \right\rfloor, \qquad |w_j| < \frac{3}{2} \tag{1.9}$$

Consequently $\quad |w_j - d_j| \le \frac{1}{2}\ ,\quad$ and $\quad \sum_{i=0}^{m} d_i 2^{-i} \quad$ represents the most significant half of the product $Y = A_m X_m + B$.

In practice we have to introduce some allowable error in the selection of d to be able to use carry-save addition. It is sufficient for convergence that $|w_j|$ always be less than 3/2, and ideally we could always guarantee that $|w_j - d_j| \leq \frac{1}{2}$ but to do that we would have to compute w_j exactly. But we would like to use $\hat{w}_j$, an approximation to w_j, and choose d_j so that $|\hat{w}_j - d_j|$ is less than $\frac{1}{2}$, but have $\hat{w}$ be accurate enough to insure that the actual $|w_j - d_j|$ be less that $\frac{1}{2} + \varepsilon$.

Let $\quad |A| < K, \quad |X| < K$

Since $|w_j| < \frac{3}{2}$ and $|w_j - d_j| < \frac{1}{2} + \varepsilon$

it follows from equation 1.7 that

$$w_{j+1} < 2(\frac{1}{2} + \varepsilon) + K + K \leq 1\frac{1}{2} \tag{1.10}$$

$$or \quad 2\varepsilon + 2K \leq \frac{1}{2}$$

This is most conveniently satisfied with $\varepsilon = \frac{1}{8}$ and K= $\frac{1}{8}$ so that our bounds to insure convergence are:

$$|A|, |X| < \frac{1}{8} \qquad \varepsilon = \frac{1}{8} \tag{1.11}$$

The value of ε means that the fraction part of $\hat{w}$ must be computed to three binary places for selection of d. That is, $|w - \hat{w}| < \frac{1}{8}$. The operands A and X must also be scaled to satisfy the condition (1.11). The only other requirement is that since $P_0 = B$, B must meet the same requirements that govern w. In this implementation, the first bit of B is considered to be the sign bit, so that the first magnitude bit of B is the 1/4 bit, giving B a maximum absolute value of < 1/2.

Figure 1.2 gives a block digram of the arithmetic unit.

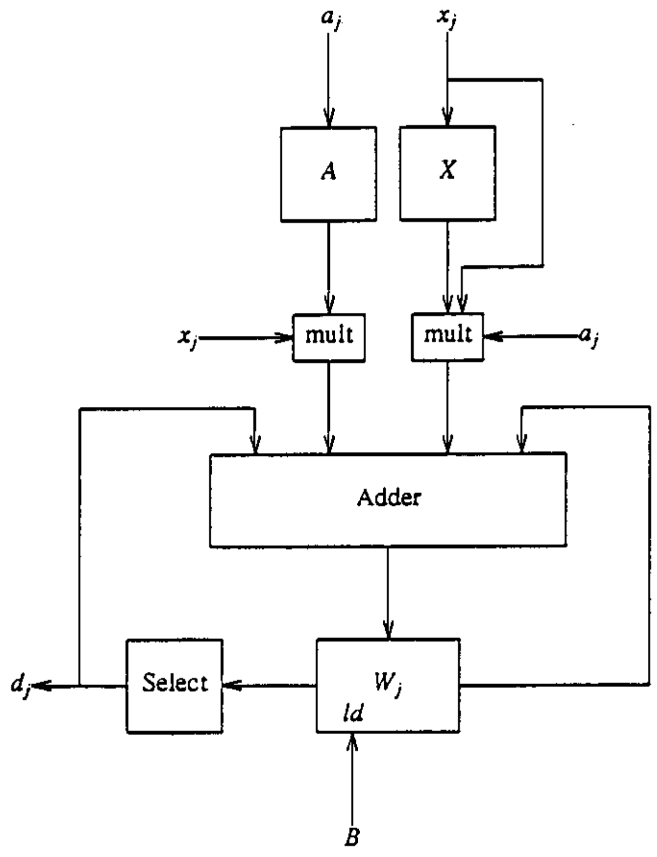

Figure 1.2 -- Block Diagram of the On-line Unit

CHAPTER 2

FUNCTIONAL DESCRIPTION

First we consider the overall operation of the on-line unit, then we will look in more detail at the modules themselves and then the individual bit slices.

2.1 OPERATION OF THE ARITHMETIC UNIT

Each cycle, the unit must evaluate the equation (1.7):

$$w_j = 2(z_{j-1}) + X_j a_j + A_{j-1} x_j$$

$$z_{j-1} = w_{j-1} - d_{j-1}$$

This will be done in several parts. There is the input of a_j and x_j, combining them to A and X, then the multiplication to produce $X_j a_j$ and $A_{j-1} x_j$. Upon adding these to z_{j-1} to determine w_j, all that remains is to select the proper d_j according to formula (1.9), output it, and subtract $w_j - d_j$.

It is not actually necessary that the inputs be in signed-digit (or any other redundant) representation for the algorithm to work, but only that the outputs be in a redundant representation. However, if the on-line unit is to be useful at all, one unit must be able to accept the output of another unit as input. For that reason, it must be allowed for on-line inputs to be in signed-digit form. Although our inputs and outputs are both in signed-digit format, it was found to be simpler to use two's-complement arithmetic for all internal operations as explained in the next section.

Signed-Digit to Two´s-Complement Converter

One of the problems that must be overcome is that the on-line inputs come in signed-digit form, but it is more convenient to do internal arithmetic in two's complement form for two reasons: the two's complement representations require fewer bits, and a signed-digit adder is more complex than the equivalent two's-complement adder. In order to take advantage of the simplicity of two's-complement arithmetic, we construct a simple on-line signed digit (SD) to two's complement (RC) converter that is fast and whose time for execution doesn't depend on the length of the operands. This converter would need to make available at each cycle A_{RC} and X_{RC} that are exactly equivalent to that part of A_{SD} and X_{SD} that has been input up to step i. The algorithm used is functionally equivalent and derived from the algorithm in [ERCE85]. The differences were not fundamental but made interconnections and layout simpler.

Because there is no redundancy in the RC form, it is not always possible to tell whether bit i will change after its value is determined at step i. For instance the number $10\bar{1}1$ (where $\bar{1}$ is -1) will be translated as 0111 after four steps, but if the next digit is -1 making our number $10\bar{1}1\bar{1}$, our new number will be translated as 01101. Notice that bit four was changed when digit five appeared, but bits one through three weren't. Another more extreme example: 1000000 translates to 1000000, but $1000000\bar{1}$ translates to 01111111. In general, whenever a -1 appears every bit up to and including the last nonzero bit is complemented, and a 1 is saved in that last position. To implement this, with each bit is associated a flag indicating whether it is confirmed or unconfirmed. Then, when a 1 appears, every previous bit is confirmed and will not change. When a 0 appears, all previous bits remain as they were, either confirmed or unconfirmed. When a -1 appears, all previous unconfirmed bits are complemented, then confirmed and will not change. The current bit is always marked unconfirmed and won't become confirmed until the next nonzero digit is input.

Table 2.1.1 specifies the converter when applied to each bit individually. The **init** signal applies to all bits at the same time, but the **ld** signal is applied to each bit in succession (i.e., the **ld** signal is applied to bit *i* when the *i*th digit is available). In other words, the **init** signal is a one-time signal that begins the process, and the **ld** signal is applied to a bit position when the digit corresponding to that location is available.

Table 2.1.1 -- Converter Transition Table

previous	0	1	-1
0c	0c	0c	0c
0u	0u	0c	1c
1c	1c	1c	1c
1u	1u	1c	0c
init	0c	0c	0c
ld	0u	1u	1u

The following algorithm illustrates how this conversion works across the entire length of the operands. In this algorithm *n* is the length of the operand, S and D are the sign and digit part of the current signed-digit (SD) input (for example, if the current input is $\bar{1}$, then S=1 and D=1), respectively. A[i] and F[i] are the digit part of the two's complement number at bit position i, and the corresponding confirmed/unconfirmed (c/u) flag for that bit. A[0] is the single bit to the left of the radix point.

```
1. A[0,1,...,n] = (0,0,...,0);          /* initialization
   F[0,1,...,n] = (u,c,...,c);

2. For ld = 1 to n                      /*  For each input digit,
   { Input(S,D)                         /*  adjust previous bit
     For all i=1 to n                   /*  according to table 2.1
       { if F[i] == u and D == 1
           then { F[i] = c;
                  A[i] = A[i] - S }
       }
     A[ld] = D;                         /*  Load the current digit
     F[ld] = u;
   }

3. End
```

Conversion Algorithm

Of course, in the actual circuit, the operation at each bit slice is done in parallel, since the operation at each bit position never depends on the operation or values at another bit position, but only on its current value and the latest input.

This scheme, then, requires two registers to store each bit of our converted number, one to store the digit value (0/1) and one to store the confirmed flag (u/c). One bit slice of our arithmetic unit will require four registers, two for A and two for X. As shown later the logic can be implemented quite easily with pass transistors and some flip-flops.

Here is an example of how the conversion would work on the input

$101100\bar{1}\bar{1}1$:

Table 2.1.2 -- Conversion of $101100\bar{1}\bar{1}1$

k	A_k	$A[k]_{RC}$									Value Used
1	1	1u									0.100000000
2	0	1u	0u								0.100000000
3	1	1c	0c	1u							0.101000000
4	1	1c	0c	1c	1u						0.101100000
5	0	1c	0c	1c	1u	0u					0.101100000
6	0	1c	0c	1c	1u	0u	0u				0.101100000
7	-1	1c	0c	1c	0c	1c	1c	1u			0.101011100
8	-1	1c	0c	1c	0c	1c	1c	0c	1u		0.101011010
9	1	1c	0c	1c	0c	1c	1c	0c	1c	1u	0.101011011

In order to insure that the sign of the number is correct, it is necessary to initialize the leading bit (which will be the sign bit) to 0u, as was done in the algorithm given previously. In that case, if the first nonzero bit is negative, then the sign bit will be set to 1, otherwise it will be set to 0, as desired. In fact, any number of leading bits can be added in the same manner without affecting the value of the number. Let's look at a shorter example with the leading nonzero bit equal to -1. Consider the number $0\bar{1}0\bar{1}1$.

Table 2.1.3 -- Conversion of $0\bar{1}0\bar{1}1$

k	A_k	$A[k]_{RC}$						Value Used
0		0u						0.00000
1	0	0u	0u					0.00000
2	-1	1c	1c	1u				1.11000
3	0	1c	1c	1u	0u			1.11000
4	-1	1c	1c	0c	1c	1u		1.10110
5	1	1c	1c	0c	1c	1c	1u	1.10111

It is important to note that although bits of the SD number may change at step $i + n$ after being used at step i, there is no error introduced. This is because at each step the value of the converted number, call it A_{RC_i}, is exactly equivalent to the value of the incomplete SD number, A_i, as defined by equation (1.1). This follows directly from the definition of the conversion algorithm. Therefore, using A_{RC_i} at step i, even when some of the bits may change at step $i+1$ or later, is exactly the same as adding the current SD number, shifting, then adding the SD number again with the next digit added on. Since the signed digits never change, then as long as the two's complement number is equal to the current part of the SD inputs the recurrence relation is not violated.

Addition and Selection

To avoid full-precision computation of w in the recurrence equation (1.7), it will be represented as the sum of two values, the carry part (C) and the sum part (S) so that carry-free addition using carry-save adders may be used. Because we allow an error in the selection of d, we only need to compute an estimate of the actual w to make the selection. Since the absolute value of w is guaranteed to be less than $\frac{3}{2}$, it is sufficient to represent all numbers with two bits to the left of the radix point. We also use two's complement form. With this in mind, we only need five bits of C and S to compute w and select the digit. In fact, we only need add the first 3 bits (up to the 1/2 place, positions i= -1, 0, 1) --since the digit being selected is an integer, additional precision would be useless-- of C and S plus a carry bit from the rest (up to the 1/8 place, positions 2 and 3).

$$\begin{array}{r l} & C_{-1}\,C_0 . C_1\,C_2C_3 \\ + & \underline{S_{-1}\,S_0 . S_1\,S_2S_3} \\ & W_{-1}W_0 . W_1 \quad\;\; \\ & c_{in} \qquad\quad \end{array}$$

where $c_{in} = C_2S_2 + (C_2 + S_2)C_3S_3$ and $\hat{w}$, the truncated value of w ($W_{-1}W_0W_1$), is simply the sum of the three most significant bits of C and S.

The recurrence formula (1.7) requires that the chosen value of d each cycle must be subtracted from w. Since the highest three bits of w must be calculated for selection of d, it is not necessary to subtract d from the carry-sum representation of w. Instead calculate $\hat{w}-d$ (since d is 0, 1, or -1 subtracting it from the most significant portion of w will not affect other bits), keep track of that value separate from C and S and set the most significant three bits of C ($C_{-1,0,1}$) and S equal to 0. The result is a different representation of w that is equal to z or w-d.

So far, the basic cycle of operation which limits the maximum speed of the chip is defined by the recurrence formula and involves two steps of carry-save adders, the computation of $W_{-1,0,1}$ and the c_{in} bit, the selection of d, then finally the computation of $\hat{w}-d$. It will soon be seen, however, that it will be much simpler than that, inasmuch as 1) it is actually possible to calculate $\hat{w}-d$ before knowing d, and 2) surprisingly enough, the information needed to compute $\hat{w}-d$ at step i does not depend on the result of $\hat{w}-d$ from step $i-1$.

Let's see just why this occurs while looking more closely at our selection mechanism. Here are the possible ranges of w and the corresponding values of d we want to select.

$$d = \begin{cases} 1 & \text{if } \frac{1}{2} \le w < 1\frac{1}{2} \\ 0 & \text{if } -\frac{1}{2} \le w < \frac{1}{2} \\ -1 & \text{if } -1\frac{1}{2} < w < -\frac{1}{2} \end{cases} \tag{2.1.1}$$

Based on these requirements, we come up with the following table for all possible values of $\hat{w}$ and c_{in} and the resulting values of d and $\hat{w}-d$ (which is $\hat{z}$).

Table 2.1.4 -- Digit Selection

$\hat{w}$	c_{in}	d	$\hat{w}-d$
00.0	0	0	00.0
	1	1	11.0
00.1	0	1	11.1
	1	1	11.1
01.0	0	1	00.0
	1	-	----
01.1	0	-	----
	1	-	----
10.0	0	-	----
	1	-1	11.0
10.1	0	-1	11.1
	1	-1	11.1
11.0	0	-1	00.0
	1	0	11.0
11.1	0	0	11.1
	0	0	11.1

This table also facilitates a proof of the fact that we are considering a sufficient number of bits of w. Recalling that our conditions for convergence were that $|w_j - d_j| < \frac{1}{2} + \varepsilon$ where $\varepsilon = \frac{1}{8}$, our condition is $|w_j - d_j| < \frac{1}{2} + \frac{1}{8}$. From the above table it can be seen that in each possible value of w, the value of $\hat{w} - d + \frac{c_{in}}{2}$, or w^*, is either 00.0 or 11.1, so that:

$$\frac{-1}{2} \le w^* = \hat{w} - d + \frac{c_{in}}{2} \le 0 \tag{2.1.2}$$

Now from here, additional possible errors come from two sources, the difference between w^* and $\hat{C} + \hat{S}$, the truncated values of C and S, and the error incurred by truncating C and S. The error in w^* results when using c_{in} for the last two bits of $\hat{C} + \hat{S}$ instead of calculating a five-bit $\hat{w}$. This error will always be positive since we always underestimate c_{in} (in other words, if c_{in} is 1 then $C_{2,3} + S_{2,3}$ is 1 or greater). Since the possible values of $C_{2,3} + S_{2,3}$ range from 0 to (.011 + .011 =) 6/8 by eighths and c_{in} is in the 1/2 position, the maximum difference between $(C + S)_{2,3}$ and c_{in} and therefore between w^* and $\hat{C} + \hat{S} - d$ is 3/8. Therefore:

$$0 \le (\hat{C} + \hat{S} - d) - w^* \le \frac{3}{8} \tag{2.1.3}$$

The last error is that caused by truncating both C and S. This error will also always be positive since both positive and negative range-complement numbers are reduced by truncation. The greatest error possible here occurs when all truncated bits are 1's. Since we are using three bits to the right of the radix point, the maximum possible value of the truncated portion is 1/8, so the maximum error from truncating both is 1/4. Therefore:

$$0 \le w - (\hat{C} + \hat{S}) < \frac{1}{4} \tag{2.1.4}$$

Combining all three of these results produces:

$$-\frac{1}{2} \le w - d < \frac{5}{8} \tag{2.1.5}$$.

which satisfies our convergence requirement, $|w - d| < \frac{1}{2} + \frac{1}{8}$.

The simplicity of this recurrence does not become clear until we look at the equations for the individual bits of $\hat{z} = \hat{w} - d$ from table 2.1.4. The resulting expressions are:

$$\hat{z}_1 = W_1 \tag{2.1.6}$$

$$\hat{z}_0 = \hat{z}_{-1} = W_1 + c_{in}$$

Now we see that $\hat{z}=\hat{w}-d$ is actually simpler to compute than d, and since only $\hat{z}$ is critical to the recurrence formula and thus the principal cycle, we can even delay the selection of d until later without affecting the speed of the main cycle. What isn't so obvious is that the computation of $\hat{z}$ does not depend on the result of computing $\hat{z}$ in the previous cycle. To see this, we must look at the actual computation of $\hat{w}$.

$$\begin{array}{lllllll} & z_{-1} & z_0 & & & & \\ & C_{-1} & C_0 . & C_1 & C_2 & C_3 & \\ + & S_{-1} & S_0 . & S_1 & S_2 & S_3 & \\ \hline & W_{-1} & W_0 . & W_1 & & & \\ & & & c_{in} & & & \end{array}$$

In order for this to accurately represent w, the most significant two bits of C and S must not duplicate the information in $\hat{z}$. For that reason, in those bits, the previous C and S are not added in because they are include in $\hat{z}$. For example, $2C_0 + S_0 = a_0 x_i + x_0 a_i + c_{int(1)}$, where $c_{int(1)}$ is just the intermediate carry from bit position 1. Now $\hat{z}$, as seen in equation 2.1.6, only depends on W_1 and c_{in}, which only depend on the previous C_1 and S_1, C_2 and S_2, C_3 and S_3. What this really means, then, is that once we have produced C and S from the carry save adders at step i, we have all the information needed to begin computing $\hat{z}$ in step $i+1$. Theoretically, it could be done without ever computing $\hat{z}$ at step i.

To illustrate this, suppose after the additions at step i we had the results C=11.0101100... and S=00.1100100... and $\hat{z}=00$ The computation of $\hat{z}$ would proceed like this:

$$\begin{array}{l} 00 \\ 11.0\ 10\ 1100... \\ \underline{00.1\ 10\ 0100...} \\ 11.1 \\ \ c_{in}=1 \end{array}$$

On the other hand, if in the previous cycle, we had delayed the computation of $\hat{z}$, then the first two bits of w could not immediately be computed. In that case, the computation of $\hat{z}$ would start out like this:

$$\begin{array}{l} ?? \\ 11.0\ 10\ 1100... \\ \underline{00.1\ 10\ 0100...} \\ ??.1 \\ \ c_{in}=1 \end{array}$$

But based on the knowledge that $\hat{w}_1 = 1$ and c_{in}=1, we can still compute:

$$\hat{z}_{-1} = 1, \qquad \hat{z}_0 = 1, \qquad \hat{z}_1 = 1$$

which when shifted in the next cycle would become (z_{-1} is shifted out) $\hat{z}$ = 11.

This allows us to also take the computation of $\hat{z}$ completely out of our main cycle (as well as the computation of $\hat{w}$ and c_{in} along the way). Therefore, the operation that limits the speed of our step cycle is just two parallel carry-save adder steps (to reduce four summands, $A_{j-1}x_j$, X_ja_j, C, and S to two, C and S). The most significant bits of w are important, however, for the selection of d at each step. In fact, here are the formulas for computing d, based on $\hat{w}$ and c_{in} derived from Table 2.1.4. Since d is in signed-digit format, there are two components of d, a sign component, d_s, and a magnitude component, d_d.

$$d_d = \overline{w}_0\overline{w}_1c_{in} + \overline{w}_0w_1 + \overline{w}_{-1}w_0 + w_0\overline{w}_1\overline{c}_{in} + w_1\overline{w}_0 \tag{2.1.7}$$

$$d_s = w_{-1}\overline{w}_1\overline{c}_{in} + w_{-1}\overline{w}_0$$

What can be done, then, is at each step produce the least significant bits (all those to the right of the radix point) of the results C and S by carry-save addition, and then send those to the next step to be added again to Ax and Xa. At the same time produce incomplete values of the most significant bits of C and S (by assuming that the most significant bits of the last w (C and S) were zero, since they'll be added again later). All this information is passed on to the next stage of the pipeline which computes $\hat{z}$ and d from the information available (that information being the incomplete C and S, the previous $\hat{z} = z_{-1}$ and z_0), while the previous pipeline stage produces another C and S.

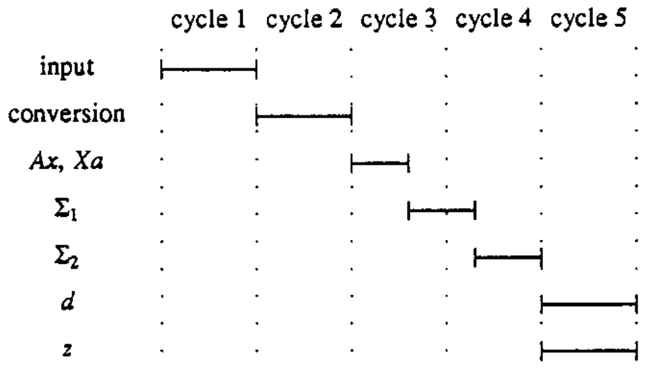

Figure 2.1.1 -- Pipelined Operation of the Unit

Figure 2.1.1 shows the pipelined operation of the unit, showing when each of the important operations take place: input of the operands, conversion to two's complement representation, multiplication of the vectors by the current digit, the two adders, and finally the computation of z and d, the next output. New digits are input every cycle and similarly a result digit is output every cycle. The first adder step spans two stages. This is necessary because in order to complete the addition it needs C produced by the second adder from the previous trip through the pipeline. This is not available until the beginning of cycle four; therefore, as much of the addition as possible is done in the previous cycle (generating all possible results from the other two operands so that when C is available it need only choose between them).

2.2 MODULAR IMPLEMENTATION

One of the goals of this project is to produce the arithmetic unit in a modular fashion so that modules of the unit can be connected so as to handle inputs of any size without significant degradation in speed. The module produced will be an 8-bit module which can serve as the highest byte, lowest byte or any intermediate byte. This means that all modules will produce output digits, but only the digit output by the highest order byte will be meaningful. There will also be communication between modules, which is basically the same as the communication between bit slices. There is shifting of data (w in each step) and the passing of an intermediate carry bit to the next slice in each carry-save addition step.

Figure 2.2.1 shows communication between two neighbor modules, the left one being the highest order byte. The X's and A's represent the computed values of $X_j a_j$ and $A_{j-1} x_j$, respectively. The capital C and S are the final components of w, and the lower case c and s represent the intermediate carry and sum results from the first addition step.

```
XX.XXXXXXXX       XXXXXXXX
AA.AAAAAAAA       AAAAAAAA
CC.CCCCCC       CCCCCCCC
--------------------------
s s. s s s s s s s s      s s s s s s s s
c c. c c c c c c c     c c c c c c c c
SS.SSSSSSS       SSSSSSSS
--------------------------
SS.SSSSSSSS       SSSSSSSS
CC.CCCCCCC       CCCCCCCC
```

Figure 2.2.1 -- Intermodule Relationships

At each step exactly four bits from module 2 are needed by module 1: two bits of C, one bit of S and one bit of the intermediate carry must be passed between the two modules, from less significant module to more significant. The passing of these bits is shown in Figure 2.2.2.

```
XX.XXXXXXXX                    XXXXXXXX
AA.AAAAAAAA                    AAAAAAAA
CC.CCCCCC       CC   <--       CCCCCC       CC   <--
s s. s s s s s s s s           s s s s s s s s
c c. c c c c c c c    c   <--  c c c c c c c    c   <--
SS.SSSSSSS       S   <--       SSSSSSS       S   <--
SS.SSSSSSSSS                   SSSSSSSS
CC.CCCCCCC                    CCCCCCCC
```

Figure 2.2.2 -- Intermodule Communication

Since all modules are identical, each module must be able to do the following: 1) Compute w in two parts (C and S) in parallel across the whole module, 2) keep track of the bits to the left of the radix point for sign extension purposes, and 3) select d and keep track of the most significant bits of w. It also must be able to pass and accept the necessary information between modules.

Each module, then, must perform two different sets of functions-- those performed by each chip in parallel each cycle, and those specialized functions only needed by particular chips like digit selection in the highest byte or adjustment for complementation in the lowest byte. The reason the adjustment for complementation can always be done in the lowest byte is because the operands are always assumed to be as wide as the number of modules allow. In other words, all operands are filled with zeroes until those bits receive a value as more of the inputs are received. When a number is complemented, then, all the zeroed bits are complemented and become ones so that by just adding one to the lowest digit position in the lowest module, the equivalent of the complement of the number will be obtained. For example, if the *load* signal is being applied to bit i, then all bits beyond i have both a_{i+n} and x_{i+n} equal to zero. So for instance, $A_i = a_0 a_1 ... a_i 00...0$. The complement of A_i, then, would be:

$$\bar{A}_i = \begin{array}{r} \overline{a_0 a_1 ... a_i} 11...1 \\ + \quad\quad\quad\quad 1 \end{array}$$

with the 1 added later to avoid the carry propagation.

Nearest Neighbor Organization

One problem that on-line arithmetic units must overcome, particularly when dealing with modular units, is the problem of high fanout in off-the-chip connections for the input operands. The chief reason for this is the need to broadcast the inputs across all modules of the unit in parallel, as is the case here.

Assume, for instance, we are dealing with more than one arithmetic unit and 64-bit operands from 8-bit modules arranged such that the output of one multi-module unit was the input of another. In this case the result of one unit would be sent as input to the next on-line unit which would consist of 8 chips. Since the inputs are needed by each module in the unit at the same time, the result of the first unit must be broadcast across 8 chips, and thus our chip's outputs would have to be able to accommodate such a load and more, which would be quite difficult.

One way to achieve nearest-neighbor connections in this case would be to send the inputs only to the first module and then have that module send them to the next on the following cycle and let them ripple through all the modules in that way, rather than broadcasting them. If such a scheme is to be used, the question arises as to whether to begin with the highest order byte or the lowest. It turns out that both have their respective advantages and disadvantages as discussed next.

If the inputs begin with the highest module, then the second module (and likewise on down the line) will be operating one cycle behind the first. This means that when the adders in bits 7 and 8 (past the radix point) are ready for inputs, the inputs shifted from the second module will not be ready. This does not present a major problem, however, since bit-values are not critical until they migrate as far as the 3rd binary place. This means that they can be added in later after they do become available. As shown before, there are four bits passed from the lower module to the higher each cycle, one to the seventh binary place, and three to the eighth. If we delay adding these bits in for, say, one cycle, then they double in value, and must be added in at a different place. Consequently, they would no longer fit in conveniently to our two-level carry-save adder; in fact, there would be 6 operands to add at the seventh binary place bit, rather than four. There is no way to add this in, then, without adding at least one more level to our carry-save adder, so as to accommodate five operands. This could be achieved by adding the four bits to get one 3-bit value in one cycle, allow another cycle to transfer the values across chips, and then add that value in at binary places four through six in the next cycle. Remember that as long as the bits are added in before they affect the third binary place, we will not lose any accuracy.

The obvious drawback of this scheme is the fact that it adds an extra adder step to our main cycle, and would thus add more than 50 percent to our cycle time, a considerable cost. Of course, one other option would be to add another stage to the pipeline in such a way that there would be three levels of carry save adders in three pipeline stages but the critical cycle remains the same. Letting p be the three-bit sum of the passed bits, this could be done by adding Ax, Xa and p to get the the first intermediate carry save result, small c' and s', which would be added to C to get the second intermediate result, small c and s, which are finally added to S to get C and S.

The entire operation would look like this:

```
 XXXXXXXX
 AAAAAAAA
+     ppp
 s's's's's's's's'
 c'c'c'c'c'c'c'
+CCCCCC
 s s s s s s s s
 c c c c c c c
+S S S S S S S
 SSSSSSSS
 CCCCCCC
```

This timing for this scheme, using three adders rather than two, is illustrated in Figure 2.2.3.

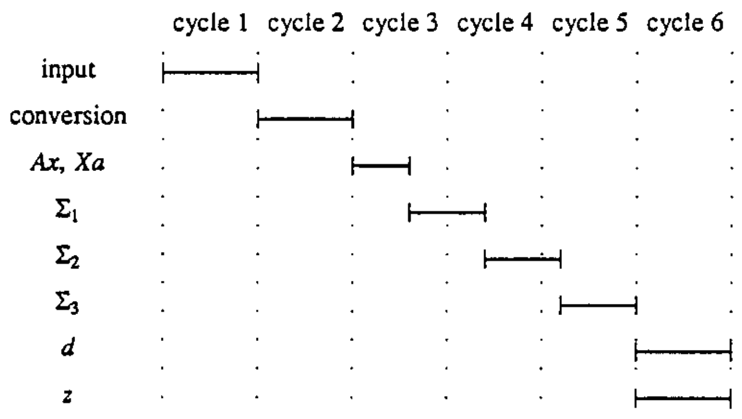

Figure 2.2.3 -- Timing for High-to-Low Scheme With Extra Pipeline Stage

With this scheme the delay between productions of C and S in one cycle and the next is exactly the same as in the original scheme, so the cycle speed remains the same, but the single module latency is increased by one cycle. Unfortunately this scheme is not possible in an 8-bit module. In order to not introduce any errors with this scheme, p must be added in before it would influence the computed bits of w, namely bits -1 to bit 3, so that the latest it could be added is when it would span bits 5-7. This is because there are two levels of carry save adders between p and w (C and S), allowing the carry bits affected by p to migrate two positions. If added in immediately, p would be added in at bits 6-8. It is delayed two cycles, however, one to reduce the four bits to three, and one to pass the data chip-to-chip. At this point it could be added in at bits 4-6 if added in at the same pipeline stage, but according to this scheme it must be added in at a point one or, more likely, two pipeline stages earlier. Therefore, p would have to be added in at bits 2-4 or 3-5 and our value of w each step would be inaccurate, causing a possible overflow in our recurrence relation.

This scheme would, however, be quite possible for a larger chip like a 16-bit module. The cost involved would be 1) increase of latency time by one cycle and 2) departure from a bit-slice architecture, since bit slices are no longer equivalent, making it more difficult to extend the basic design to different sizes (even impossible in the 8-bit case), although the byte- or module-slice architecture would still be preserved and the bit slices themselves would not be very different from each other.

The other option (illustrated in Figure 2.2.4) is to migrate the inputs right to left, or from the least significant module, then on one module at a time to higher order modules. In this case, the higher of two neighbor modules will be one cycle behind the lower, which means that the bits needed to complete the add step at the lower bit places in each module will be ready one cycle early.

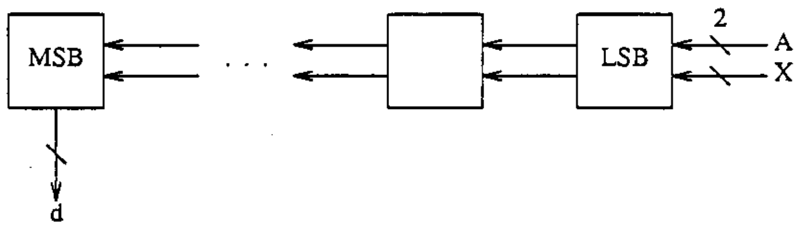

Figure 2.2.4 -- Module Utilizing Nearest-Neighbor Connections

This gives us the luxury of taking one full cycle to transfer the bits and still have them ready to be added at their proper place. The penalty in this scheme is the latency time. In other words, where in the first scheme, or without nearest neighbor connections, the first digit of the result would be ready five cycles after the arrival of the first digit (that means a latency of five cycles) of the inputs, in this scheme, the first digit of the result would arrive five cycles after the first digit arrives at the highest module, or 5 + M - 1 (where M is the number of modules) cycles after the inputs first arrive at the lowest module. As an example, if we have 64-bit operands and 8-bit modules, then the latency would increase from 5 to 12 cycles. If we implemented 16-bit modules, the latency would improve to 8 cycles for the low-to-high scheme because there would be 4 modules instead of 8. On the other hand, if we were working with 16-bit operands, then the latency would be 6 for 8-bit modules and 5 again for 16-bit modules.

The two schemes which are possible for 8-bit modules are compared here. The third scheme (adding another carry-save adder in a separate pipeline stage to accommodate high-to-low connections) would certainly need to be considered for a particular application where size of operands are known and the size of the modules does not need to be as flexible. For the purposes of this project, however, we desire a module that is small enough to be tested easily, but has a design which could be easily extended to larger modules. Another consideration is that this scheme does not have an advantage over the low-to-high scheme until the unit is three or more modules wide. Thus, for the large modules that are needed to implement the scheme, this only happens with very wide operands.

We need to determine, then, which cost is greater, the cost in cycle time for the high-to-low scheme, or the cost in latency for the low-to-high scheme. In the case of the low-to-high scheme the total time of operation for one level with operands of length N would be $4 + \frac{N}{8} + N - 1$ ($\frac{N}{8}$ is the number of modules needed, assuming 8-bit modules) and for L levels of arithmetic units it would be $L(4 + \frac{N}{8}) + N - 1$ clock cycles. For the high-to-low scheme or for units without nearest-neighbor restrictions, the delay would come to $5L + N - 1$ clock cycles. We introduce the factor K which represents the increase in cycle time caused by the high-to-low scheme. Then the question to be answered is under what conditions is:

$$4L + \frac{LN}{8} + N < K(5L + N) \tag{2.2.1}$$

Assuming, as we did before, that K is greater than 1.5, then we find that the left hand of the equation is smaller for all cases when $L \leq 4$. It would still be smaller in some cases for even more than 4 levels. If we had been assuming 16-bit modules, in fact, then the low-to-high scheme would be faster in all cases where $L \leq 8$. This means that for configurations of 4 levels or less for 8-bit modules or 8 levels or less for 16-bit modules, it would be more advantageous to use the low-to-high scheme for nearest-neighbor connections.

Consequently, if we are to implement nearest neighbor connections to the modules, then in most configurations it would be less costly to send the inputs first to the lowest byte and let them propagate one module at a time toward the highest module. This is the method that has been chosen for this implementation.

Another cost in implementing nearest neighbor connections would be an increase in the number of module-to-module interconnections and thus pins. Namely, there would be five more pins: A_{out} (2), X_{out} (2), and $init_{out}$. This increases the total pin count from 29 to 34 for 8-bit modules and from 37 to 42 for 16-bit modules. Figure 2.2.5 shows what a single arithmetic unit, made up of several modules (the most significant on the left) would look like, with the corresponding interconnections shown. Some of the control signals shown are *init*, which signals the beginning of operation for the entire unit, *ld*, the load signal sent to each module when it should begin accepting inputs (the first module stores A,X_{0-7}, the second stores A,X_{8-15}, etc.) and *lob* which is only high to mark the least significant module.

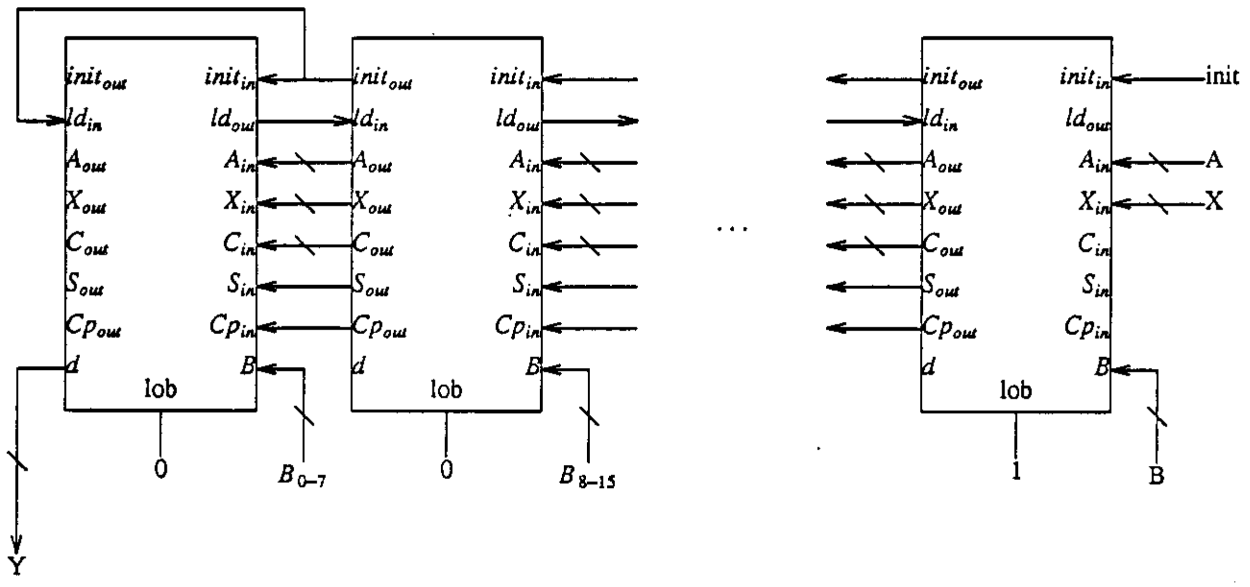

Figure 2.2.5 -- Inter-Module Connections

The speed of operation of the unit is now somewhat dependent on the length of the operands and the width of the modules we have chosen to implement, but the cycle time is still completely independent of the size of the operands. This will also change the thinking in determining the size of the module to implement. Obviously, we can speed up the operation of the unit as a whole by increasing the width of the modules. For our purposes the 8-bit chip will be sufficient, but for commercial purposes, it would certainly be more advantageous to implement much wider chips of say, 16 or 32 bits. Since the chip is designed in bit slices, the conversion of the VLSI layout to a wider chip would be trivial.

2.3 BIT SLICE OPERATION

The module implements equation (1.7),

$$w_j = 2(w_{j-1} - d_{j-1}) + X_j a_j + A_{j-1} x_j$$

including the selection of d_j each step.

This will be done in two parts, in different pipelined stages. The first will do the part that is performed across all bit slices--the adding of $A_{j-1}x_j + X_j a_j + 2w$. The rest, which only concerns the most significant bits, is done in the second part, namely the selection of d and the computation of $w-d$. It was shown before that these operations can be done independently.

The principal section of the module will be made up of 8 bit slices so that the design could be easily expanded to implement 16-bit modules, etc. Each slice will implement several functions, namely those functions that need be carried out across all bits. Each bit slice will be comprised of five principal sections as shown in Figure 2.3.1.

The first section contains the A and X registers. This section will not only hold the values of A and X but will also contain the signed-digit-to-two's-complement conversion logic. It must contain, as mentioned before, not only a register to store the current values of x_i and a_i, but also registers to store a flag bit for each showing whether that bit position is confirmed or not. There must also be a traveling load signal across the slices to control the loading of the current a_j and x_j at the correct slice. Because of the nature of the recurrence relation, it is important that a_j be loaded one cycle after x_j so that A is actually A_{j-1} as in the recurrence equation.

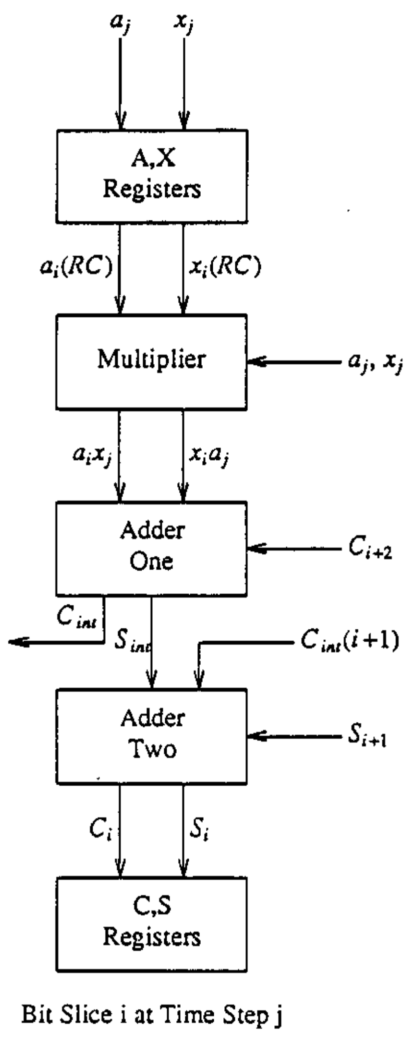

Figure 2.3.1 -- Operation of a Bit Slice

The next section of the bit slice is the multiplier. This section implements the multiplications X_ja_j and $A_{j-1}x_j$. This is simply a multiplexor which can select the proper multiple of each bit (e.g. it selects x_i, 0, or the complement of x_i depending on whether a_j is 1, 0, or -1). There must also be some logic in the low order module to compensate for the complementation in two's-complement arithmetic.

The next section is the first of the two carry save adders. This one adds the two outputs of the multiplier (*Ax* and *Xa* for short) with *C*, the carry part of the output of the final carry save adder from the last cycle. These produce two intermediate outputs, S_{int} and C_{int}. These two are added (with C_{int} shifted) in the second carry save adder array to *S* and produce the two components of *w*, *C* and *S*.

The last section of the bit slice is the *C* and *S* registers. The outputs of the adders are shifted to the correct slice (at which they will be used in the next cycle) and stored in registers. Initially the *S* registers are set to 0 and the *C* registers are set to the value of *B* (*B* is available off-line and in two's complement form).

2.4 THE REST OF THE CHIP

In addition to the eight bit slices contained in each chip, there are also several other sections of logic important to the operation of the unit. Each chip has routing of several control signals, like initialize and the load signal at each bit position. There is also that circuitry that is specialized for one module in each unit, for example, the selection logic for the output in the most significant module, the adjustment for complementation in the lowest, and two bits of sign extension in the highest, all of which are contained in all modules.

Other Bits

Each module contains 8 bit slices operating in parallel. However, each also contains two other bit slices that are only used if the module is in the most significant position. These are the two bits to the left of the radix point. These two bit positions are necessary to accommodate the maximum possible absolute value of *w*, 1 1/2, which can be represented in two's complement with two bits to the left of the radix point. They are similar to the other bit slices, except for a few minor changes.

Since the first two bits of C and S (w) are not completely calculated until the next stage in the pipeline, zeroes are added in where C and S would be. So the outputs of the first two bit slices are simply the values of $a_i x_j + x_i a_j$ in each place plus any carries from other bit positions. Obviously, since neither C nor S are inputs to these slices, there is no need for those registers in these bit positions.

The outputs of these bits are stored elsewhere for future use, namely in the selection of d and the updating of $w-d$.

Selection Logic

The last section of the chip is the selection logic. This selection chooses d and subtracts it from the most significant bits of w. This logic, although it will of course be repeated in every module of an arithmetic unit, will only be meaningful in the most significant module. This logic must take the inputs C_{-1} to C_3, S_{-1} to S_3 (10 inputs) and z_{-1}, z_0 from the previous cycle and then produce d, the current output, and z to be used next cycle. To do this, the first three bits of C and S are added to the two bits of z to get $\hat{w}$ as previously defined. Meanwhile the next two bits of C and S are added to determine the carry bit, c_{in}. Then, with those results, d and z can be determined according to equations 2.1.7 and 2.1.6. These operations are shown in Figure 2.4.1.

This figure illustrates the reason that selection can be pipelined after the recurrence formula implemented in the bit slices. Although results of the bit slice operation are needed for selection, the bit slice operation is not dependent upon any results of the selection process as illustrated in Figure 2.4.1. The adder in this figure could more accurately be considered two adders in parallel, one for $\hat{w}$ and one for c_{in}.

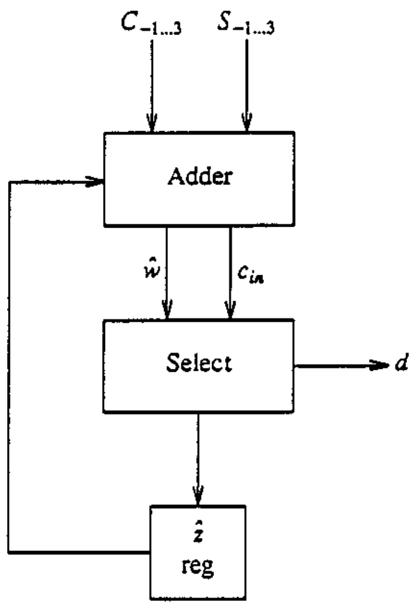

Figure 2.4.1 -- Selection Operation

CHAPTER 3
CIRCUIT LEVEL DESCRIPTION

Following is a circuit-level description of the chip in the 4-micron nMOS technology. Although part of the purpose of this project is to have a design that could also be implemented in better and faster technologies, like CMOS, at the level of circuit design full advantage was taken of nMOS circuit characteristics. A great deal of pass transistor logic is used which, in particular, doesn't port well to CMOS. On the other hand, the logic description itself, at a level one step above the circuit design does not change according to implementation. The following descriptions are of the five basic elements of each bit slice.

A and X Registers

The first cell in the bit slice is where the *A* and *X* operands are stored. It also contains the signed-digit to two's complement conversion logic. As mentioned before, this section contains four register flip-flops: one each to store 0/1 and another to store the u/c flag for both *A* and *X*.

The circuit for the two u/c registers is shown in Figure 3.1. It implements Table 2.1.1. These diagrams are for *X*, but the diagrams for *A* will be nearly identical. Notice that x_d is the digit part of x (0 or 1) and x_s is the sign part (1 corresponds to -, 0 corresponds to +). Also notice that the value of U does not change until ϕ_2. This insures that the changing of U will not affect the value of x until the next cycle.

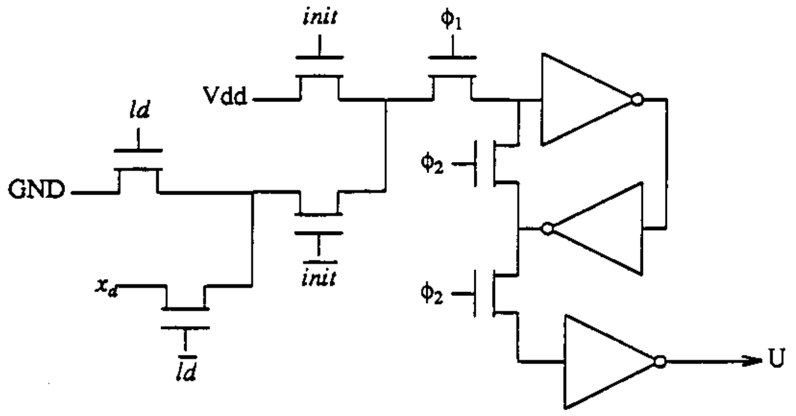

Figure 3.1 -- X Register Flag Circuit

The circuitry for the bit value of x (shown in Figure 3.2) is more critical with respect to time, because it is x and $\bar{x}$ that get carried on to the next level. That is why ld and U are generated in the ϕ_2 cycle so that when ϕ_1 comes high, the next value of x (if it is to change) will have settled waiting just on the other side of the ϕ_1 pass transistor. Of course, x_d and x_s must settle during ϕ_2 as well for this to be the case.

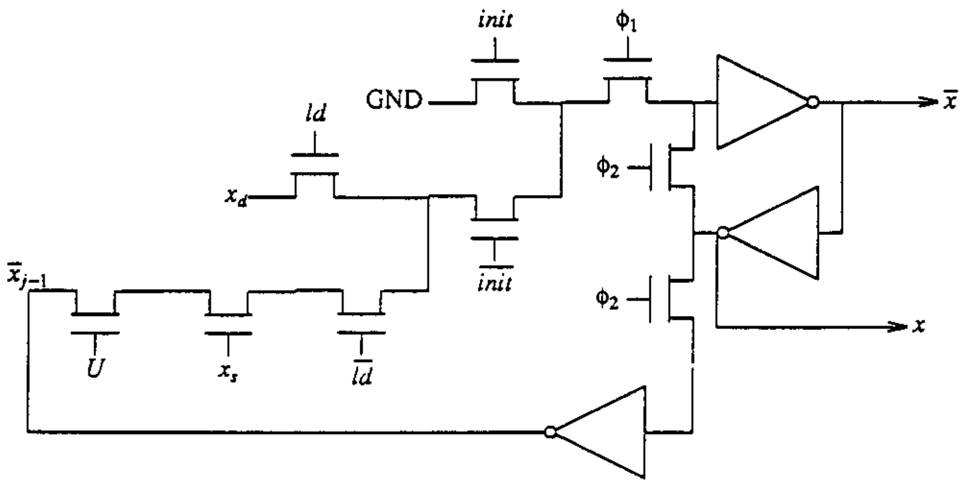

Figure 3.2 -- X Register Circuit

Notice that it is necessary to have access to the previous value of x (actually $\bar{x}$), since x is complemented in some cases. Also, both x and $\bar{x}$ are passed on to the next stage.

The traveling *ld* signal is generated by a shift register (Figure 3.3) that travels across the bit slices from high to low. The extra inverter is used to make sure *ld* and $\overline{ld}$ are consistent at both clock phases.

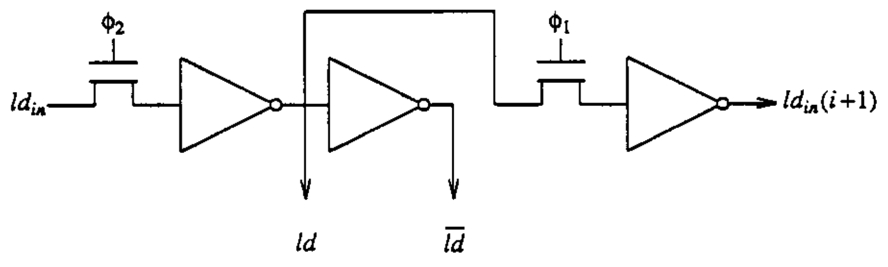

Figure 3.3 -- Load Signal Circuit

This is the case for X, but A must be loaded one cycle later, so the *ld* signal from the next bit slice must be brought back to be used for A. Also, the values for a_d and a_s must be one cycle old so that the value used in the next step is A_{j-1} rather than A_j to be consistent with the recurrence equation (1.7).

Multiplier

The next cell of the bit slice is the multiplier. This section implements the multiplications $X_j a_j$ and $A_{j-1} x_j$. We will just look at the computation of $X_j a_j$, since the other will be identical. In a single bit slice (bit i in this case) the actual computation implemented will be $x_i a_j$, where x_i is the ith bit of the two's complement representation of X, and a_j is the current on-line digit of the signed-digit value of A.

This simple multiplication is implemented by a multiplexor as shown in Figure 3.4. There are three possible values of a_j (-1, 0, 1), so it needs more than just an AND gate. Since we have both x and $\overline{x}$ from the previous section, it is easy to generate all multiples of x. Whenever possible, the effort was made to generate both the result and its complement in parallel, since in almost all cases both were needed by the next section and it saved much time against just complementing the result.

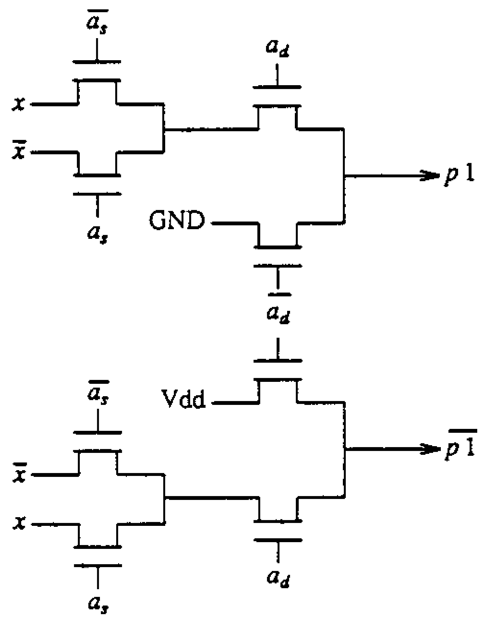

Figure 3.4 -- Multiplier

At the end of this section, then, we have two products, $p\,1$ (a_ix_j) and $p\,2$ (x_ia_j), as well as their complements, $\overline{p\,1}$ and $\overline{p\,2}$.

First Adder

This next cell is the first level carry-save adder (Figures 3.5 and 3.6). It will add the products $p\,1$ and $p\,2$ with the carry half of w, which is C. It will produce two outputs, an intermediate C and S, call them C_{int} and S_{int}. We add C in first rather than S because C is calculated a bit earlier, as we'll see later on.

This is the beginning of the "critical cycle", which is that part of the computation that must be done every cycle and cannot be overlapped with any part of itself. Thus, this is also the division between two different stages of the pipeline. The conversion of A and X and the multiplication, etc. can all be done beforehand, because none of it depends on the computation of w (C and S). In fact, it can be seen that this adder was actually designed into two parts so as to make as little of the adder dependent on C as possible. This allows us to minimize the second pipeline stage, which is our critical one as far as design decisions go. The division between these two pipeline stages is the ϕ_1 signal which is applied to the outputs of this first adder.

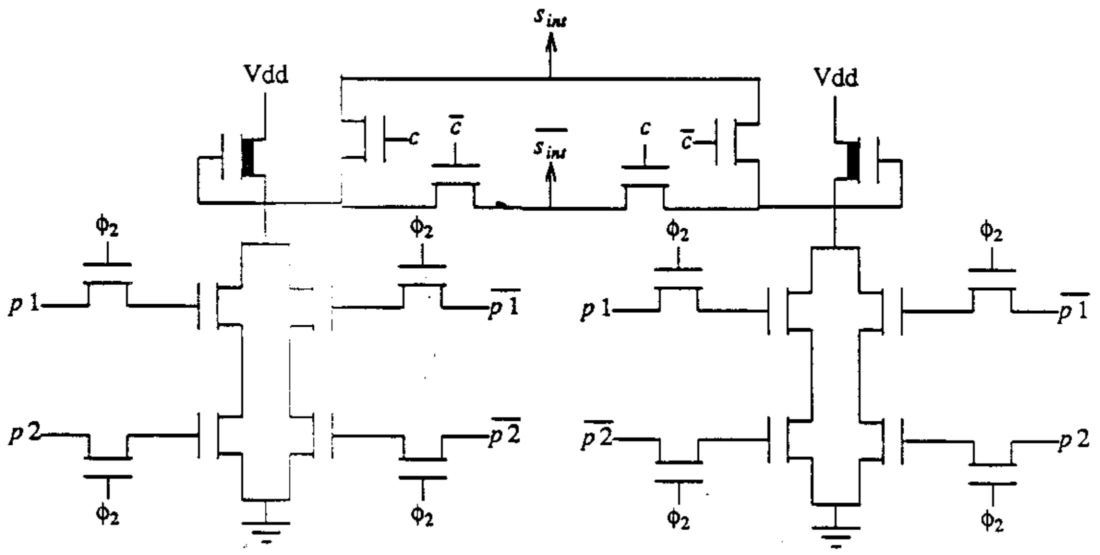

Figure 3.5 -- First Adder Sum Generator

Notice again that all possible values of c_{int} and s_{int} are computed in parallel, so that when c becomes available, it need only choose between the options.

The outputs c_{int} and s_{int} are passed to the next adder stage with c_{int} shifted one place to the next bit position. Before that, however, so that all necessary inputs are available to the next adder, all outputs are clocked by ϕ_1 and c_{int} is inverted, which isn't shown here.

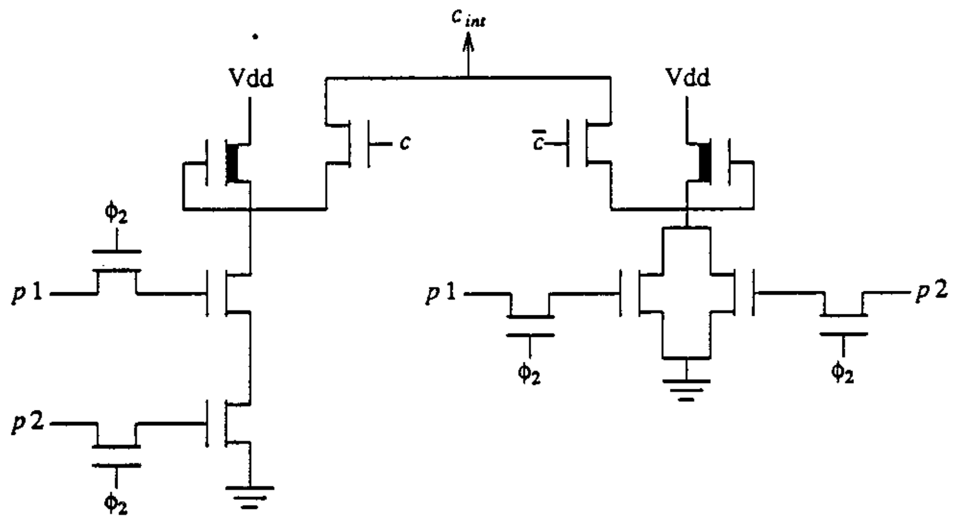

Figure 3.6 -- First Adder Carry Generator

Second Adder and Final Registers

This adder (Figure 3.7) is a bit more conventional, as all inputs are available at approximately the same time. This adder takes the inputs C_{int}, S_{int}, and S and produces the final output w, which has two components, C and S.

In designing this, the assumption is made that c_{int} is the latest arriving of the three inputs.

The outputs of this adder, c and s, are shifted before being stored in flip-flops to be used for the next cycle. c is shifted two places to the left, and s one. The c register looks like Figure 3.8. The s register is identical except that the input is actually $\bar{s}$ instead of s, and it is thus initialized to 0 (all 1's, instead of B_i in the figure). Thus w $(C + S)$ is initialized to B. Notice that the output of the second adder was actually the complement of s and thus it makes sense that s is initialized to all 1's.

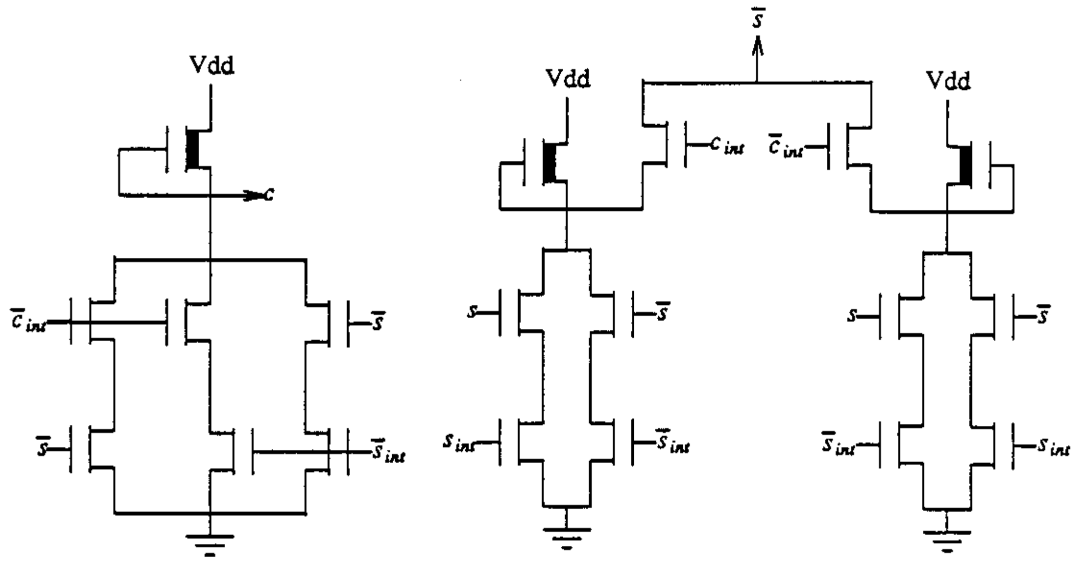

Figure 3.7 -- Second Adder

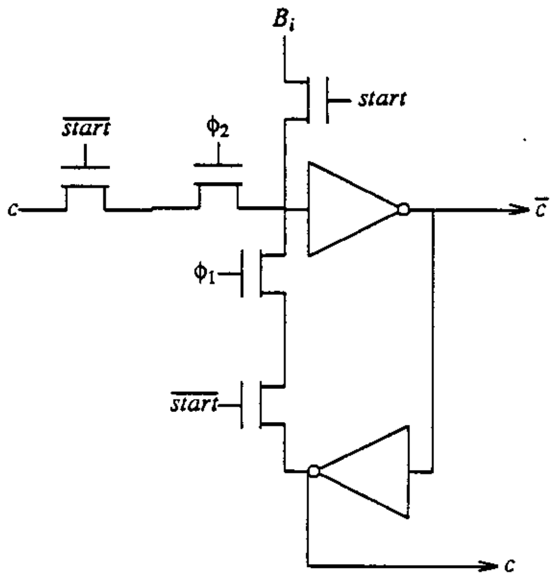

Figure 3.8 -- C Register

With B available off-line in two's complement form, it is easiest to let B be the initial value of w (in fact, the initial value of C), and then just add the first products to it. As long as $|B|$ is bounded by the same bounds that limit $|w-d|$ ($< 1/2$), this will work. As mentioned before, B is guaranteed to be within these bounds because bit 0 is considered the sign bit, giving the most significant magnitude bit a value of 1/4.

Selection Logic

The selection logic was implemented using two programmable logic arrays implementing the computation of $\hat{w}$ and c_{in} and then d and $\hat{z}$ from the equations 2.1.6 and 2.1.7 as illustrated in Figure 2.4.1. The motivation behind this implementation is simply the speedup available using PLA's. The reason the PLA's provide a faster implementation is that in nMOS conventional logic gates are generally limited to two transistors in series between Ground and the depletion device, thus limiting the size of a product in a logic equation realizable by a single gate to two terms. Consequently, several levels of logic gates would have been necessary to implement the logic equations. The result of this being that more than one clock cycle would have been required to do the selection. If the equations for $\hat{w}$ and c_{in} were to be inserted into equations 2.1.6 and 2.1.7, it is immediately seen that the logic is far too complex both in the number of gates required (number of products) and the levels of logic (size of largest product).

The PLA's were generated from logic equations by VLSI PLA-generation tools after first having the logic minimized. The equations were too complex to be implemented by a single PLA, but by splitting the logic into two parts PLA's of reasonable size were produced. The splitting was done to try to produce PLA's of approximately the same size and to produce products from the first PLA that are used most frequently by the second. This was accomplished quite well with the first producing partial sums from the different operands, and the second finishing the addition and doing the selection)

CHAPTER 4

EVALUATION

4.1 AREA ANALYSIS

The chip implemented in 4-micron nMOS (the minimum feature size is 4 microns) measures 1866 microns by 1838 microns without pads for the 8-bit chip. It is composed of 1957 transistors. With the input and output pads the area is increased to 2646 microns by 2568 microns (shown in Figures 4.2 and 4.3).

The corresponding measurements for the 16-bit module chip are 3002 microns by 1838 microns without pads, 3694 microns by 2568 microns with pads, and a total of 3165 transistors.

What this corresponds to, then, for any size chip is an overhead of 730 by 1838 microns plus 142 by 1838 microns for each bit slice. In other words, the size of a *b*-bit module would be 730+*b**142 microns by 1838 microns. With pads, it is less exact, but it comes out to an overhead of about 1598 microns by 2568 microns plus about 132 microns by 2568 for each bit slice. Looking at transistors, there is an overhead of 749 transistors plus an additional 151 transistors per bit slice. The layout of a bit slice is shown in Figure 4.1.

Of those 151 transistors in each bit slice, they are broken up as follows: the largest portion, 60 transistors, are used for the signed-digit to range complement conversion and A and X registers. The multiplier uses 16 transistors, all enhancement pass transistors. The two carry save adders combined represent 55 transistors. Finally, the C and S registers are formed from 20 transistors. Looking at it another way, each bit slice is comprised of 27 logic "gates," determined by counting the number of depletion mode transistors. The rest of the transistors are enhancement mode transistors that are either part of a logic gate or are pass transistors. Of the large overhead of 749 transistors, a large part of that are the two PLA's that make up the selection logic. They combine for 168 transistors, or more than 20%. The two extra bit slices to the left of the radix point account for another 212 transistors, or another 30% of the overhead. The rest of the transistors are used for input and output buffering, and control signal generating and routing.

Of the area actually used (i.e. ignoring empty spaces), approximately 43% of it was used by the 8 bit slices, 28% was used for the various control signals and on- and off-chip communication, 19% for the digit selection logic (this is rather high because it was done with PLA's), and 9% was used by the two bits of sign extension. Of course, that is for the 8-bit module. For the 16-bit modules, the 16 bit slices represent 60% of the useful area.

In light of the fact that such a large proportion of the chip area is taken up by the overhead needed for the most significant module of the on-line unit, it would seem that it might have been more advisable to implement two different types of modules, one that could operate in the most significant place and one that could operate in any other place (least significant or in the middle). In this case, however, the size of the chip is completely determined by the input and output pads as can be seen from Figure 4.2. Therefore, a simple 8-bit slice without digit selection logic would take up nearly the same area (it would require two fewer pads -- d_d and d_s). The most significant module would require several fewer pads but would still be about the same area due to the size of the logic. Therefore, the only advantage gained in implement-

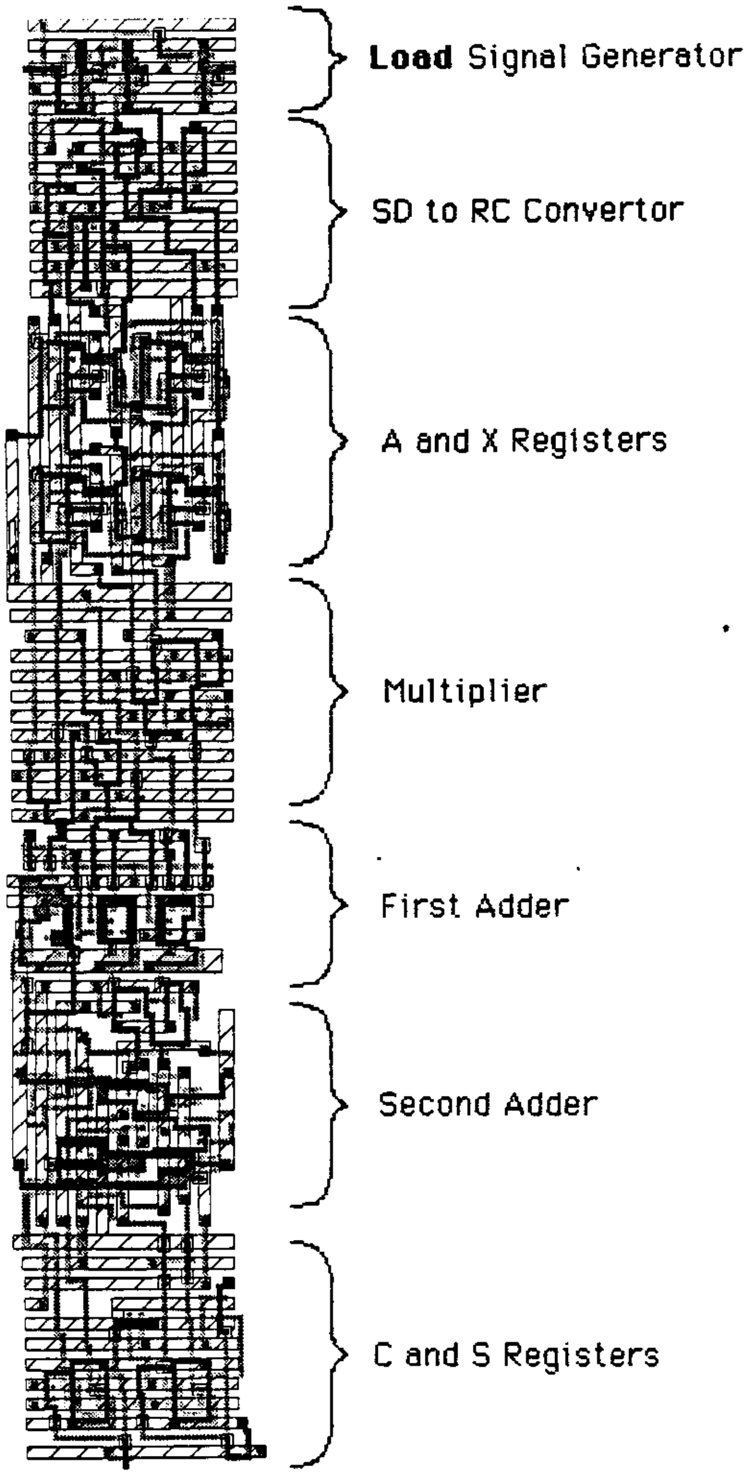

Figure 4.1 -- Layout of a Bit Slice

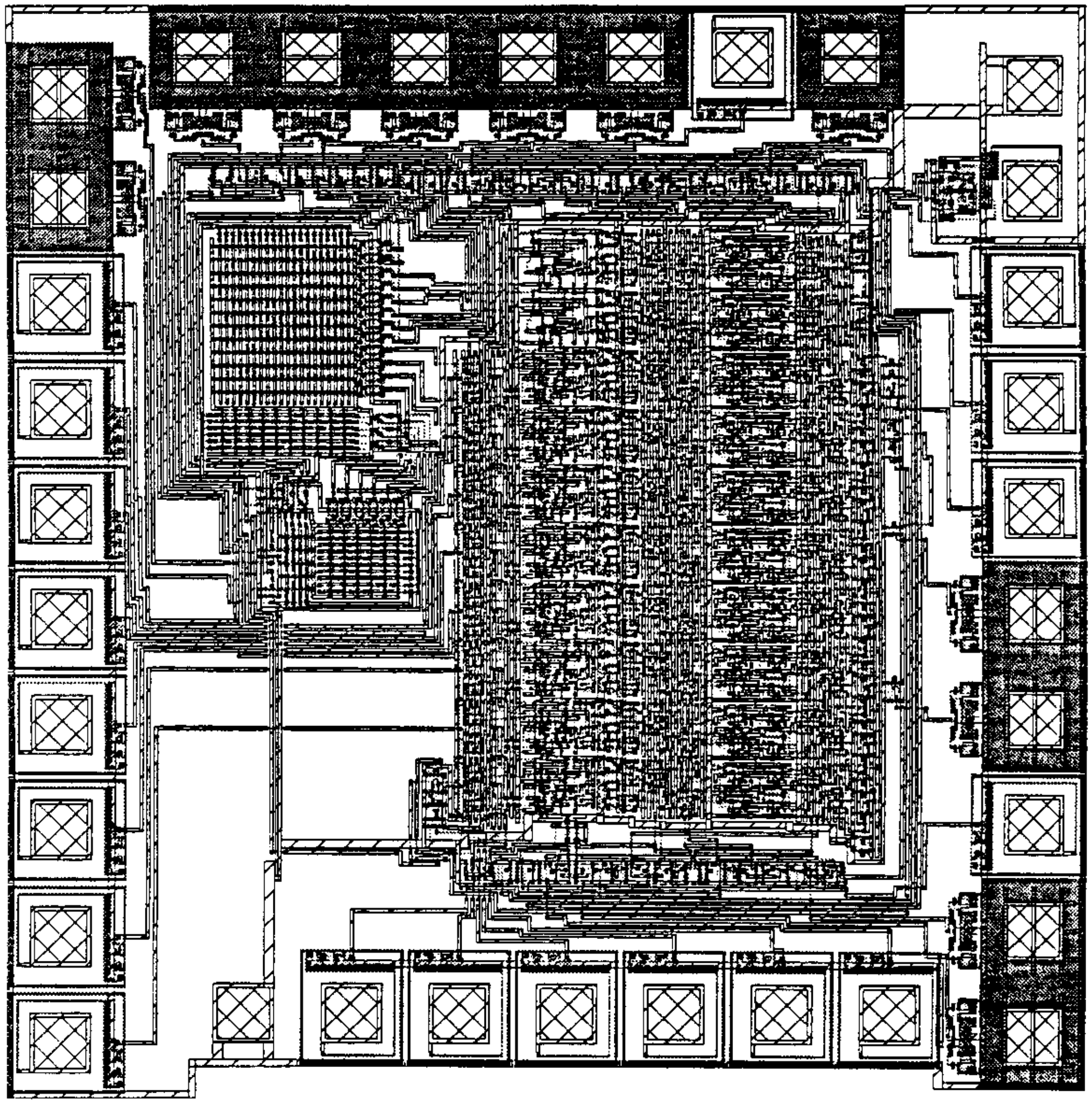

Figure 4.2 -- Layout of the Chip

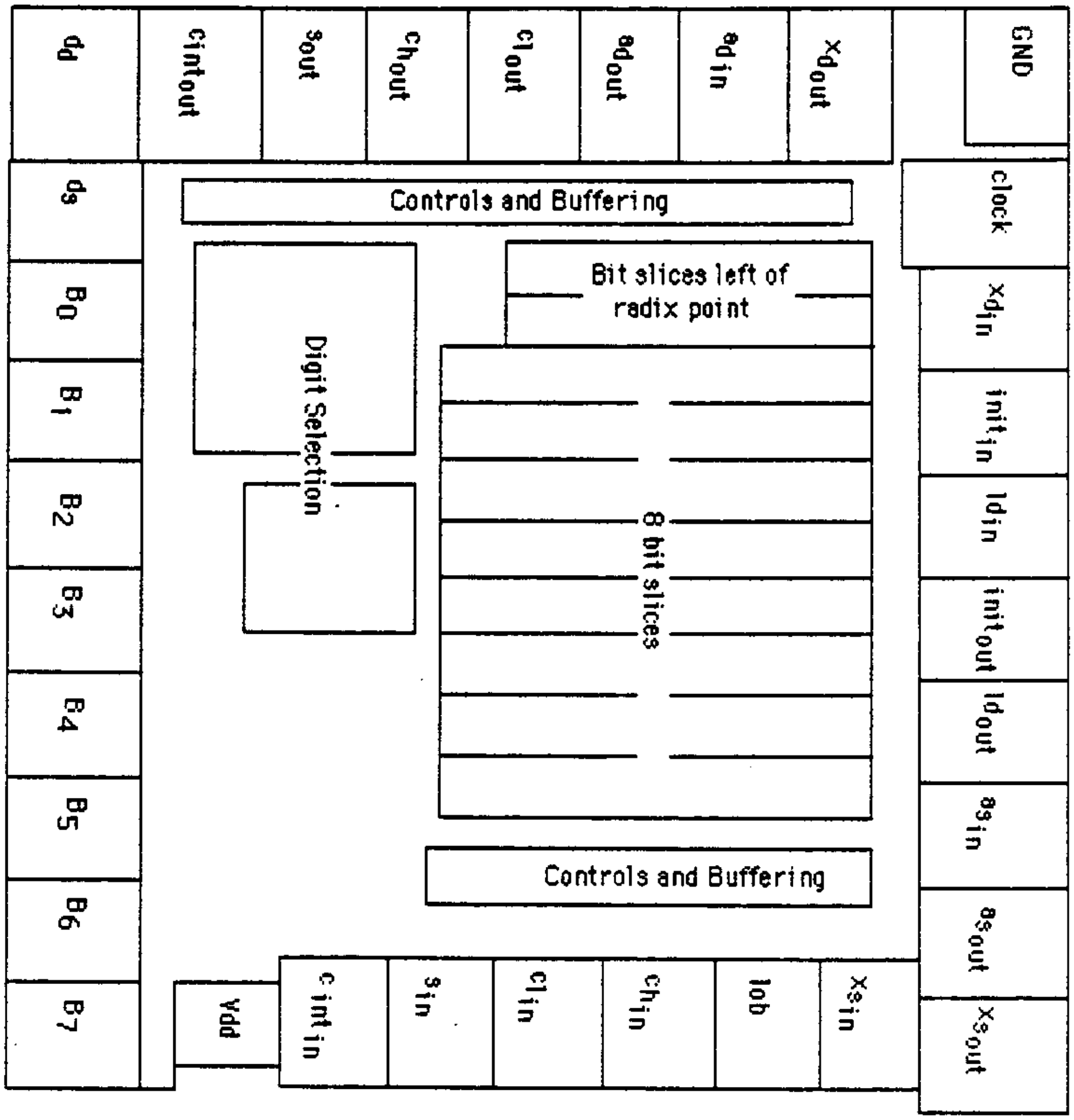

Figure 4.3 -- Floorplan of the Chip

ing separate module types would be a slightly increased chip yield due to less surface area used. The same would be true for 16-bit modules, as their size is also determined almost completely by the number of pads.

4.2 TIMING ANALYSIS

The chip will operate in a pipelined manner with a maximum clock cycle of about 110 nanoseconds in the 4-micron nMOS implementation. For a less technology-dependent comparison, that figure corresponds to three gate delays per clock cycle, plus a few pass transistors that contribute a small percentage of that delay. These results were obtained by using "mextra" [MAYO83], a circuit extractor, and Crystal [OUST83], a timing analysis program which finds the worst case path in a circuit. The longest pipeline stage, which identifies the maximum speed of the circuit is found by identifying the longest path between two successive ϕ_1 or two successive ϕ_2 signals, which may be the same signal in circular circuitry. In this case the worst case path travels through the C register (loaded at ϕ_2), enabling S_{int}, which is input to the adder gate that produces S to be loaded in the S register at ϕ_2. The signal path travels through three gates, two for the register and one for the adder, plus a few pass transistors.

Although the chip was much too large to use SPICE [NAGE73] for simulation purposes, the following method was used to facilitate the use of SPICE to verify maximum cycle speed results. The importance of this method is that Crystal is a generic program for predicting delays, but with SPICE we were able to use transistor models that more accurately reflected the actual fabrication process to be used. Using Crystal to find single slowest paths and modifying the circuit to speed these up, it was assured that the slowest path during each clock cycle (and in different pipeline stages) was through repeated circuitry (in other words, the slowest path was through the bit slice, rather than through irregular circuitry). It was then possible to extract a single bit slice, modified to accurately reflect inter-slice connections and routing, and simulate it through SPICE. As it turned out, the worst case path occurred in one particular slice, bit slice 1, because several of its outputs had larger loads, since they were routed not only to the digit selection logic, but also to super buffers to be sent off-chip the next clock cycle. To accommodate this, Crystal was used to find that the worst path through a regular digit slice was about 100ns. Since the digit slice ran under SPICE perfectly with a variety of inputs at this speed, it is safe to assume that the slowest bit slice would run at the speed Crystal predicted, 110 ns, possibly less. In order to verify that the results of the simulation were correct, the bit slice was simulated at 100ns and a much longer clock cycle to insure that all signals had plenty of time to become stable, then the two results were compared.

One other way that we double-checked Crystal results was to run SPICE on Crystal output of just the worst case paths, to see if they stabilized within the time that Crystal claimed. Using this method, Crystal results were verified by SPICE to be accurate to within about 1 nanosecond.

To again get results that are less dependent upon the implementation technology, the chip was simulated in 1-micron technology, which is about the state-of-the-art in nMOS at the moment. Although no accurate SPICE parameters were known, again the worst case delays were found through the circuit extractor and Crystal. In this technology it was found that the chip would be able to operate at a cycle time of less than 10 nanoseconds. This means that, for instance, a 64-bit operand would completely cycle through the unit (assuming four 16-bit modules) in 720 ns, or with 8-bit modules, 760 ns. Of course, in such a technology the assumption that a signal can be sent off-chip and received in one clock cycle may no longer be valid, which would mean that the speed of a multi-module unit would be limited by the chip-to-chip transfer time.

4.3 FUNCTIONAL VERIFICATION

In order to verify that the chip operates correctly functionally and logically, a couple of tools were used. These were 1) a program in C to simulate the desired operation of the chip, implementing the bit-level algorithm of the on-line module and 2) the switch-level logic simulation program "esim" [MAYO83] which takes as input the results of the circuit extractor "mextra." In other words, esim simulates the actual circuit as defined by the VLSI artwork.

The first step was to test the C program to ensure that it produced the desired correct results in all cases. After this was done, this program produced results that could be checked against other simulations for both debugging purposes and verification. The next step, then, involved comparing esim results with those predicted by the C program for particular inputs. This was done quite carefully, checking not just outputs, but many intermediate results within the circuit. After this, the circuit was again simulated by esim, but now paying attention primarily to the inputs and outputs and checking them for accuracy. This was done for a large variety of inputs.

All simulation up until this point had been assuming that the unit consisted of only one module, in other words that the operands were only eight bits wide and therefore there was no concern about intermodule communication. The next step was to modify the C program to operate with the same algorithm, but now 16 bits wide. Then the circuit was simulated, first acting as the low order byte and keeping track of all the outputs to the higher order byte (also again checking intermediate values as well against the correct bits in the C simulation). Next the high byte was simulated with the outputs of the low byte as inputs. The outputs of this module were then checked to be accurate. In this way the correctness of the module was verified in all special cases and positions that it could occupy within a unit.

Several of these results are included in [TULL86]: the C simulation program, results of this simulation for three different inputs, and esim outputs on the identical inputs. The results of the two simulations can therefore be cross-referenced.

Also included are esim outputs for three different inputs, as well as the 16-bit simulation results for both the C simulation and and the esim 2-module simulation. In addition, [TULL86] includes Crystal and SPICE outputs to verify timing results.

CHAPTER 5

EXAMPLES OF APPLICATIONS

In order to understand more fully how this arithmetic unit could be used, and also to see more clearly some of its advantages, we would like to look at how the on-line arithmetic unit could be applied to solve a particular class of equations.

Although certainly the unit could be used to solve equations like $G = ((A*B + C)*(D*E) + F)$ that fit into a tree-like structure with several inputs and one output, a much more useful application would be polynomial evaluation, which could be solved using the following method, given in [ERCE75, ERCE77]. Assume, for instance, that we were interested in evaluating the following third-degree polynomial $P_3(x)$, with p_{0-3} being constant coefficients:

$$P_3(x) = p_3x^3 + p_2x^2 + p_1x + p_0 \tag{5.1}$$

This equation could be translated into the following system of linear equations:

$$P_3(x) = y_0 = xy_1 + p_0 \tag{5.2}$$

$$y_1 = xy_2 + p_1$$

$$y_2 = p_3x + p_2$$

Equations 5.2, then, could be solved iteratively, the most significant digit first, with three on-line arithmetic units arranged linearly in the manner illustrated by Figure 5.1. Each of the arithmetic units shown in Figure 5.1 would consist of one or

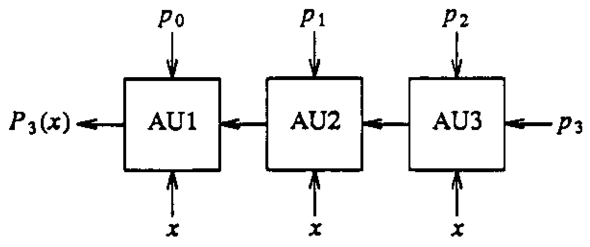

Figure 5.1 -- On-line Network for Polynomial Evaluation

more of the on-line modules. In a similar manner, then, these arithmetic units could be used to solve a number of polynomial equations as well as series approximations to more complex functions.

Assume that the units are being used to compute a third degree polynomial in the manner of Figure 5.1 and that the operands (p_{0-3} and x) are each 32 digits wide. Also assume that 32 digits of the result are required and 16-bit modules are used. In this case, the timing would be as given in Figure 5.2, where AU3 inputs x and p_3 and outputs y_2, which is input by AU2, and so on.

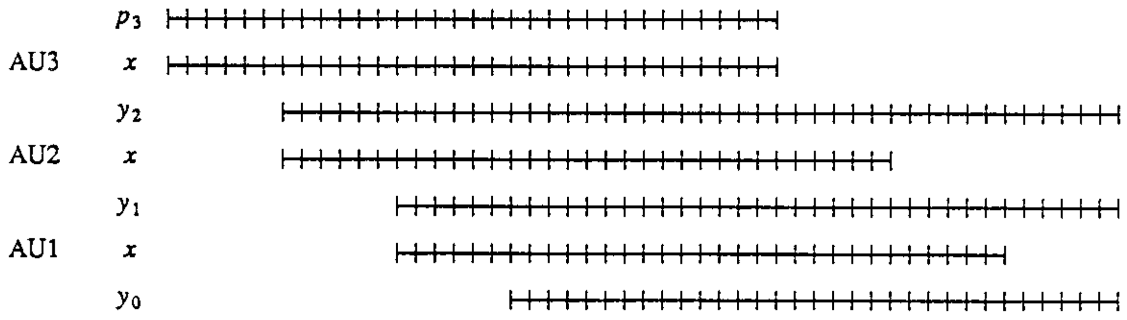

Figure 5.2 -- Timing of Polynomial Evaluator

Off-line inputs are ignored in this timing diagram. x (and p_3) can be in either signed-digit or two's complement representation because any two's complement number is equivalent to the same signed-digit number with the sign bit always zero. In fact, if both the inputs and the final outputs need to be in range complement form, the only requirement is an extra device similar to the signed-digit to two's-complement converter described previously which would produce the output in the desired form after only one additional step.

With these parameters the polynomial evaluator would require 49 steps after the first inputs are available (3 levels, 6 steps latency each, and 32 digits of results minus 1) to produce the final product using 16-bit modules (55 steps for 8-bit modules). It is assumed that the coefficients and the independent variable x are in the range given in [ERCE75] and do not violate the bounds given in Chapter 1. With the timing limits from Chapter 4, this would amount to 5500 nanoseconds (5.5 microseconds) to evaluate a third degree polynomial to a precision of 32 digits. This configuration would require two on-line modules per arithmetic unit connected as in Figure 2.2.4 and thus a total of six on-line modules.

Table 5.1 gives an example of this scheme using 16-bit operands and 8-bit modules, so that the latencies are the same as the above discussion. The operands are:

$$p_0 = 1.1111010110100010$$

$$p_1 = 1.1111000010110111$$

$$p_2 = 0.0010111011011101$$

$$p_3 = 0.0010111011011101$$

$$x = 0.000\bar{1}110\overline{111}000101\bar{1}$$

and the 16-bit result is

$$P_3(x) = 0.0000\bar{1}10\bar{1}01\bar{1}10001$$

This result translates to -0.0187836, which differs from the actual value by a rounding

w1	d1	w2	d2	w3	d3
00.0110111110111010	0				
00.1101111101110100	1				
11.1011111011101000	0				
11.0101110111010000	-1				
00.1101101110100000	1				
11.1101011101000000	0				
11.1010111010000000	0	11.1110000101101110	0		
11.0011110100000000	-1	11.1100001011011100	0		
00.0101010010000000	0	11.1000010110111000	0		
00.1000100010000000	0	10.1101101101110000	-1		
01.0000101101000000	1	11.1101111011100000	0		
00.0001000011000000	0	11.1111010111000000	0		
00.0010000110000000	0	11.1110101110000000	0	11.1111010110100100	0
00.0101110111110100	0	11.1010010000000000	0	11.1110101101001000	0
00.1011011000101100	1	11.0001000100000000	-1	11.1101011010010000	0
11.0100010111100101	-1	11.1110101100000000	0	11.1011110100100000	0
00.1000101111001010	1	11.1101000001000000	0	11.0110101001000000	-1
11.0001011110010100	-1	11.1010000010000000	0	00.1100010010000000	1
00.0010111100101000	0	11.0100000100000000	-1	11.1000100100000000	0
00.0101111001010000	0	00.1011100100100000	1	11.0010001000000000	-1
00.1011110010100000	1	11.0110110010000100	-1	00.0101100110000000	0
11.0111100101000000	-1	00.1010011110100011	1	00.1100001110000000	1
00.1111001010000000	1	11.0100111101000110	-1	11.1000011100000000	-1
11.1110010100000000	0	00.1001111010001100	1	01.0000111000000000	1
11.1100101000000000	0	11.0011110100011000	-1	00.0010000111000000	0
11.1001010000000000	0	00.0111101000110000	0	00.0010110100111100	0
11.0010100000000000	-1	00.1111010001100000	1	00.0110000000110100	0
00.0101000000000000	0	11.1110100011000000	0	00.1100101100110001	1

Table 5.1 -- Example of Polynomial Evaluation

error of 0.016%. Most of that is the rounding error from considering only 16 bits of the final result. The accumulated rounding error if the 32-bit result is used is only 0.0006%.

The first result of AU3, d_1^1, is actually output 5 cycles after the initial inputs, which isn't obvious from the table, and the final output begins 12 cycles later and finishes after a total delay of 33 clock cycles.

Similarly the unit could be used to find a root of a polynomial equation by an iterative method. For instance, the root of a fourth degree polynomial like equation 5.3 could be found by solving for x iteratively.

$$x = p_4 x^4 + p_3 x^3 + p_2 x^2 + p_1 x + p_0 \tag{5.3}$$

Of course, this equation and the initial value of x would have to satisfy certain convergence conditions that are well documented in numerical analysis texts. The configuration in Figure 5.3 utilizing buffers for x would solve equation 5.3 iteratively and continuously, once every n cycles, where n is the length of operands (or desired output). In Figure 5.3, n is assumed to be 32, again using 16-bit modules, resulting in a six cycle latency between one unit and the next. In this configuration, the new x would be produced beginning 24 cycles after the old is input to module AU4. After eight more cycles AU4 would be ready to begin to input the new x, which it could do without influencing the outputs at AU1 before it finished producing all 32 digits of the current x. The timing of the four units in terms of their inputs is shown in Figure 5.4.

This would be possible because when a new *init* signal arrives at a 2-module unit, the next five outputs will remain unaffected and the sixth will be the first digit of the next output. This means that even if x_{32} and $p_{4_{32}}$ are followed immediately by *init* and x_1 and p_{4_1} (which are always input simultaneously), y_{32} of AU4 will be pro-

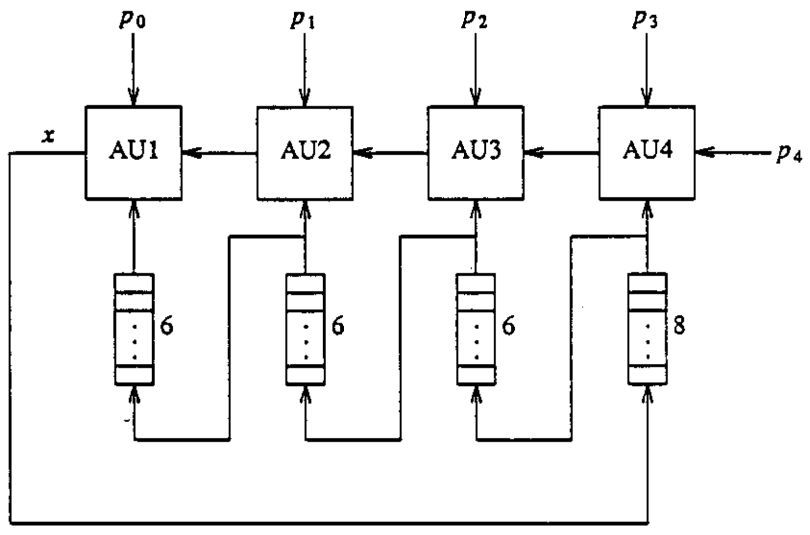

Figure 5.3 -- Solution for the Root of a Polynomial Equation

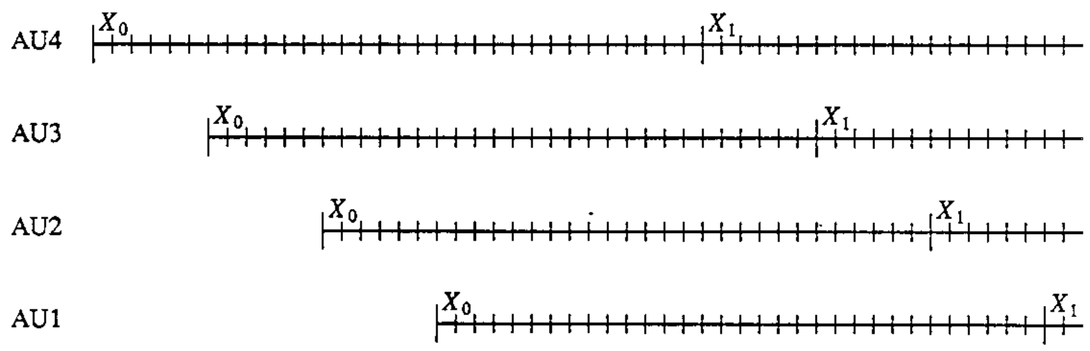

Figure 5.4 -- Timing of Root Finder

duced five cycles later unaffected, followed by the next y_1 on the sixth. The same is true for each module so that AU1 will produce the new x_1 immediately following the last x_{32}. In this manner, this configuration after an initial setup time of 23 cycles (four times the latency minus 1), will then complete an iteration of the fourth degree polynomial every 32 cycles. In that case, for instance, 10 iterations would require 343 clock cycles. In comparing this scheme with one using conventional arithmetic chips several differences are seen. One is that although four different units are operating in a pipelined manner on the output of the pipeline, they are all operating 100% of the time. With conventional methods, none of the units could operate until the previous one completed, and the whole process could not be restarted until AU1 completed. Therefore, each of the modules would be operating only 25% of the time.

Another difference is the small number of connections between units (and modules within each unit). There are only two lines between each unit in this configuration, and there is never any single line having to broadcast signals. With conventional chips, there would have to be 32 lines between each two chips for the pipeline to run as efficiently as possibly.

Each of the on-line clock cycles corresponds to approximately one carry-save adder step and a register. A 32-bit module to compute $AX + B$ in the conventional manner would require 32 similar clock cycles, or if some simple radix-4 recoding was used, it would require 16 add-store steps. If computed with a similar configuration, these modules would then require 64 clock cycles, because no overlap would be possible. To minimize delay, a tree-like structure could be used to evaluate the polynomial in three steps rather than four as shown in Figure 5.5 (Estrin's method). In this case, the time to complete one iteration would be 48 steps. Since the next iteration could not be started until the previous one completed, the time for 10 iterations would be 480 clock cycles.

In addition, the performance of the on-line method can be improved by adding more hardware. If we duplicated the hardware of Figure 5.3, the output of the first polynomial evaluator could be sent as input to the second, to avoid the 8-step delay as x waits in the buffer for the pipeline to be ready for new inputs. In this way, the second could begin immediately after 24 cycles accepting the new x, and 24 cycles later, it would produce the first bit of x_2 to be input to the free first evaluator. Thus, with multiple on-line evaluators, an iteration could be begun every Δ steps, where Δ is

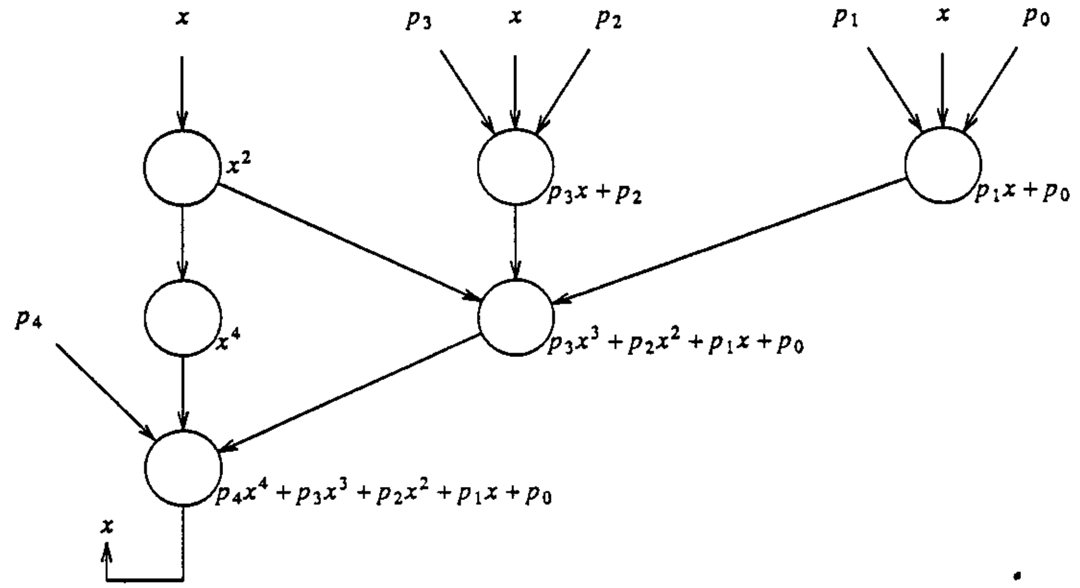

Figure 5.5 -- Conventional Root-Finding Solution of $P_4(x)$

the total latency of one evaluator. Therefore, with this scheme, 10 iterations could be completed in 271 cycles. This improvement is even greater when n, the width of the operands, is much larger than Δ, as is the case when $n = 64$, resulting in $\Delta = 32$. With 64-bit operands, it could still be done at maximum speed with two evaluators (because Δ is exactly half of n), but with any larger operands, more than two evaluators would be necessary). The total delays for N iterations of operands M bits wide for the conventional (conv), on-line (ol), and on-line with multiple evaluators (olm) schemes are:

$$D_{conv} = \frac{3NM}{2}, \qquad D_{ol} = NM + \Delta - 1, \qquad D_{olm} = N\Delta + M - 1 \tag{5.4}$$

Table 5.2 summarizes the time to complete the evaluation for the three different schemes varying the number of iterations and the size of the operands. From this it can be seen that with multiple on-line configurations, the speedup approaches a maximum of 3 in the 64-bit case.

Table 5.2 -- Comparison of Root Solving Implementations

Number of Iterations	32-bit Operands			64-bit Operands		
	Conv	On-line	Multiple On-line	Conv	On-line	Multiple On-line
1	48	55	55	96	91	91
10	480	343	271	960	667	383
100	4800	3223	2431	9600	6427	3263

REFERENCES

[AVIZ61] A. Avizienis, "Signed Digit Number Representations for Fast Parallel Arithmetic," IRE Transactions on Electronic Computers, pp. 389-400 (1961)

[ERCE75] M. D. Ercegovac, "A General Method for Evaluation of Functions and Computations in a Digital Computer," Ph.D. Thesis, Report No. 750, Department of Computer Science, University of Illinois, Urbana, August 1978

[ERCE77] M.D. Ercegovac, "A General Hardware-Oriented Method for Evaluation of Functions and Computations in a Digital Computer," IEEE Transactions on Computers, Vol. C-26(7), pp. 667-680 (July 1977)

[ERCE84] M.D. Ercegovac, "On-Line Arithmetic: An Overview," Proceedings SPIE Conference on Real-Time Signal Processing, San Diego, 1984

[ERCE85] M.D. Ercegovac and T. Lang, "On-the-Fly Conversion of Redundant into Conventional Representations," Report No. CSD-850026, UCLA Computer Science Department, August 1985

[MAYO83] R. N. Mayo, J. K. Ousterhout, W. S. Scott, editors, "1983 VLSI Tools" Report No. UCB/CSD 83/115, Computer Science Department, University of California, Berkeley, March, 1983

[MEAD80] C. Mead and L. Conway, "Introduction to VLSI Systems," Addison-Wesley, 1980

[NAGE73] L. Nagel, D. Pederson, "Simulation Program with Integrated Ciruit Emphasis (SPICE)," 16th Midwest Symposium on Circuit Theory, Waterloo, Ontario, April 12, 1973

[OUST81] J. K. Ousterhout, "Caesar: An Interactive Editor for VLSI", VLSI Design, Vol II, No. 4, Fourth Quarter, 1981, pp. 34-38

[OUST83] J. K. Ousterhout, "Crystal: A Timing Analyzer for nMOS VLSI Circuits", Third Cal Tech Conference on Very Large Scale Integration, 1983, pp. 57-69

[TULL86] D.M. Tullsen, "Simulations of an On-Line Arithmetic Unit Design," Internal Memorandum, UCLA Computer Science Department, May 1986

[TRIV77] K.S. Trivedi and M.D. Ercegovac, "On-Line Algorithms for Division and Multiplication" IEEE Transactions on Computers, Vol. C-26, No. 7, July 1977