A HARDWARE DIVIDER FOR FIXED-INTEGER DIVISORS WHICH USES READ-ONLY MEMORIES

A Thesis

Presented to

The Faculty of the Department of Electrical Engineering

University of Houston

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Electrical Engineering

---

by

Chin-Chuong Ko

December, 1977

ACKNOWLEDGEMENT

The author would like to express his deep gratitude for the guidance and discussion to Dr. T. N. Whitaker, his advisor, as well as to Dr. J. D. Bargainer and Dr. A. H. White who served on the committee. The author also wishes to thank his wife, Yuh-Shang, and his parents for their encouragement.

A HARDWARE DIVIDER FOR FIXED-INTEGER DIVISORS WHICH USES READ-ONLY MEMORIES

An Abstract of a Thesis

Presented to

The Faculty of the Department of Electrical Engineering

University of Houston

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Electrical Engineering

---

by

Chin-Chuong Ko

December, 1977

ABSTRACT

A hardware divider for fixed-integer divisors is designed. It uses a direct table-look-up technique with read-only memory. The dividend is partitioned into several groups, each group is processed by a small ROM. The ROM is designed in a special way so that the final divider is formed by directly linking these small ROM's.

Three examples of the application, including BCD-to-Binary and Binary-to-BCD converters, are given.

TABLE OF CONTENTS

LIST OF FIGURES

FIGURES PAGE

A HARDWARE DIVIDER FOR FIXED-INTEGER DIVISORS WHICH USES READ-ONLY MEMORIES

Introduction

In certain operations, such as code conversion and the location of elements in a multidimensional list, it will be valuable to have a fast hardware divider that uses a fixed-integer divisor and yields both an exact quotient and an exact remainder.

This thesis will describe such a divider which is both fast and economical in the size of the memory required. It uses a direct table-look-up technique with read-only memory in order to achieve its speed and accuracy. The memory needed is kept to a practical size by a new arrangement that is the subject of this thesis.

It will prove helpful to review two prior contributions to the solution of the problem of division by a fixed-integer divisor.

In 1967, IBM[1] presented an efficient division algorithm for use on the IBM 360/Model 91. In this algorithm, the divisor and dividend are considered to be the denominator and numerator of a fraction. On each iteration, the numerator and denominator are both multiplied by a common factor, Rr, so that the denominator converges quadratically toward the quotient which is desired.

A detailed description of this IBM algorithm is given in reference (1). However, a brief outline of this algorithm is given here so that it can be compared with the division technique which is the subject of this thesis. This outline follows:

(1) Bit-normalize the divisor and shift the divident accordingly to get $\frac{1}{2} \leq D < 1$ (D is the divisor).

(2) Determine the first multiplier, R, by a table look-up which inspects the first seven bits of the divisor to guarantee that D x R has seven similar bits to the right of the binary point.

(3) Multiply D by R forming D_1.

(4) Multiply the numerator N by R forming N_1.

(5) Truncate Dr and complement to form multiplier Rr.

(6) Multiply Dr by Rr forming D_{R+1}.

(7) Multiply Nr by Rr forming N_{R+1}.

(8) Iterate on (5), (6), and (7) until $D_{R+n} \rightarrow$ and

(9) This brings $N_{R+n} \rightarrow$ Quotient.

This and other high-speed division algorithms can be applied to the problem of division by fixed-integer divisors. However, in this application, there are at least two points of difficulty when one tries to implement this in a small system. First, these algorithms do not supply the remainder, which might be of great importance in some applications. Second, the cost of the hardware needed to implement these algorithms is high. At a minimum, the hardware required includes a shift counter, a subtractor (or a superfast multiplier if IBM's algorithm is used), and a complex control unit.

In 1972, David H. Jacobsohn presented a cominatorial division algorithm for fixed-integer divisors. In his algorithm, division is performed by multiplying the dividend by the reciprocal of the divisor. The reciprocal is, in all nontrivial cases, a repeating, binary fraction. The quotient and the remainder are then extracted from the

product of an integer and a repeating binary fraction. One disadvantage of this method is that the quotient and the remainder have to be derived from the product which in turn is the result of a time-consuming multiplication procedure.

Table-look-up procedures for division by fixed-integer divisors can use read-only memories efficiently when the dividends are small. For larger dividends, they become impractical because the ROM space size tends to increase exponentially. For example, a system which yields a 16-bit quotient and 5-bit remainder from a 20-bit dividend requires a ROM look-up table with 2^{20} addresses with each address having 21 bits. A single ROM of this capability is, obviously, impractical. If many small memories are interconnected to achieve this size, the cost becomes prohibitive. However, this thesis will present a mthod of building a memory that is much smaller than this but which performs just as well as the larger one.

A New Approach

If the dividend can be partitioned into several groups and the highest-order group is regarded as a partial-dividend, a much smaller ROM can be built and used as a look-up table for the highest-order partial-dividend. If this ROM contains enough information to link with the next-order group as another partial-dividend, an equivalent table-look-up divider can be built by cascading these smaller ROM's. The following section outlines this approach for N/D, assuming that D is a nontrivial m-bit fixed integer divisor and N is any ℓ-bit integer dividend.

The Algorithm for N/D

(1) Choose a proper number j. (j is selected by the user; it can be larger than, equal to or smaller than m, the basic ROM size will be determined by m + j)

(2) Starting from the least significant bit, partition the dividend into j-bit groups. Example: 1 100 011 110 for j = 3.

(3) Add at least one leading zero bit to the dividend to make an m-bit most significant group. Call this group m_N. Example: $\underbrace{01\ 100}_{m_N}$ 011 110 for the same example as in (2) for j = 3 and m = 5.

(4) Using m_N and the highest order j-bit group as the partial dividend, compute the j-bit partial quotient and the m-bit partial remainder. The partial quotient will be the first j-bit group of the final quotient. Example: 01 100 011 is the first partial-dividend for the same example as in (3).

(5) Write j-bit group next to the right of the partial remainder. Call this new number M.

(6) Using M as the new partial dividend, compute the new partial remainder by the same method as in step (4).

(7) Repeat step (5) and (6) to compute the next partial quotient and partial remainder.

(8) The last partial remainder will be the final remainder, and the quotient is obtained by directly assembling all the partial quotients in order.

Proof of the Algorithm

The above algorithm can be proved mathematically as follows:

Given an m-bit fixed-integer divisor

$$D = d_{m-1}\, d_{m-2}\text{-----}d_o,\ d_{m-1} = 1$$

where d_{m-1} is the most significant bit of D, and any ℓ-bit dividend

$$N = n_{\ell-1}\, n_{\ell-2}\text{-----}n_o$$

where $n_i = 1$ or 0 for $i = \ell-1, \ell-2, \text{----- } 0$

(i is a subscript).

By partitioning the dividend into j-bit groups as selected by the user, N can be represented as:

$$\underbrace{\underbrace{0\text{----}0}_{e \text{ bits}}\, n_{\ell-1} \text{ ---- } n_{pj-1}\, n_{pj-2}\text{---- } n_{(p-1)j}}_{m \text{ bits}}\text{----}$$

$$\text{----}\ \underbrace{n_{ij-1}\, n_{ij-2}\text{---- } n_{(i-1)j}}_{i\text{-th j-bit group}}\ \text{----}\ \underbrace{n_{j-1}\, n_{j-2}\text{----- } n_1 n_0}_{\text{first j-bit group}}$$

Since each group has j bits and the LSB of the first group is n_0, the LSB of the second group is n_j and the LSB of the i-th group is $n_{(i-1)j}$. The p-th group is the highest order j-bit group which is not included in the m-bit group.

e is the smallest integer that can satisfy both of the following two conditions:

(1) $\ell + e - m$ can be evenly divided by j.

(2) $e \geq 1$.

Let the i-th j-bit group be denoted by ${}^{j}_{R}N$, i.e., ${}^{j}_{i}N = n_{ij-1}\, n_{ij-2}\text{---- } n_{(i-1)j}$ and the one m-bit group denoted by m_N that is

$$m_N = \underbrace{00\text{---}0n_{n-1}\text{----}}_{m \text{ bits}}$$

where the superscripts j and m denote the maximum bit number of the groups represented; the subscript i denotes the i-th j-bit group; and $^{m}_{N}$ denotes the most significant m-bit group. With this notation, we have

$$N = {}^{m}_{N}\cdot 2^{pj} + {}^{j}_{p}N\cdot 2^{(p-1)j} + {}^{j}_{p-1}N\cdot 2^{(p-2)j} + \text{---}$$
$$+ {}^{j}_{k}N\cdot 2^{(k-1)j} + \text{---} + {}^{j}_{2}N\cdot 2^{j} + {}^{j}_{1}N$$

and $D = {}^{m}_{D}$.

Hence

$$\frac{N}{D} = \frac{{}^{m}_{N}\cdot 2^{j} + {}^{j}_{p}N}{{}^{m}_{D}} \cdot 2^{(p-1)j} + \frac{{}^{j}_{p-1}N\cdot 2^{(p-2)j} + \text{---} + {}^{j}_{1}N}{{}^{m}_{D}} \quad \text{-----} \quad (1)$$

Since $^{m}_{N}\cdot 2^{j} + {}^{j}_{p}N$ is an m+j bit number and $^{m}_{N} < {}^{m}_{D}$ ($^{m}_{N}$ has at least one leading zero bit)

we have

$$\frac{{}^{m}_{N}\cdot 2^{j} + {}^{j}_{p}N}{{}^{m}_{D}} = {}^{j}_{p}Q + \frac{{}^{m}_{p}R}{{}^{m}_{D}} \quad \text{----------} \quad (2)$$

where $^{j}_{p}Q$ is an j-bit quotient, $^{m}_{p}R$ is an m-bit remainder and of course $^{m}_{p}R < {}^{m}_{D}$

then

$$\frac{N}{{}^{m}_{D}} = {}^{j}_{p}Q\cdot 2^{(p-1)j} + \frac{{}^{m}_{p}R\cdot 2^{j} + {}^{j}_{p-2}N}{} + \frac{{}^{j}_{p-2}N\cdot 2^{(p-3)j} + \text{---} + {}^{j}_{1}N}{} \quad \text{--} \quad (3)$$

again let

$$\frac{{}^{m}_{p}R\cdot 2^{j} + {}^{j}_{p-1}N}{{}^{m}_{D}} = {}^{j}_{p-1}Q + \frac{{}^{m}_{p-1}R}{{}^{m}_{D}} \quad \text{----------} \quad (4)$$

we have

$$\frac{N}{m_D} = {}^{j}_{p}Q \cdot 2^{(p-1)j} + {}^{j}_{p-1}Q \cdot 2^{(p-2)j} + \frac{{}^{m}_{p-1}R \cdot 2^{j} + {}^{j}_{p-2}N}{m_D} \cdot 2^{(p-3)j}$$

$$+ \text{---} + \frac{{}^{j}_{1}N}{m_D} \text{ ---------- (5)}$$

By repeating the same procedure, we get

$$\frac{N}{m_D} = {}^{j}_{p}Q \cdot 2^{(p-1)j} + {}^{j}_{p-1}Q \cdot 2^{(p-2)j} + \text{---} + {}^{j}_{k}Q \cdot 2^{(k-1)j}$$

$$+ \text{---} + {}^{j}_{1}Q + \frac{{}^{m}_{1}R}{m_D} \text{ ---------- (6)}$$

Recognizing that for each term ${}^{j}_{k}Q$ on the right hand side of the equal sign of equation (6), the maximum bit number possible is j, and the order difference between two consecutive terms of (6) is 2^{j}, we get

$$\frac{N}{m_D} = {}^{j}_{p}Q \; {}^{j}_{p-1}Q \; {}^{j}_{p-2}Q \text{ ---- } {}^{j}_{k}Q \text{ --- } {}^{j}_{1}Q + \frac{{}^{m}_{1}R}{m_D} \text{ ---------- (7)}$$

In other words, the final quotient Q of N/D is

$$Q = {}^{j}_{p}Q \; {}^{j}_{p-1}Q \; {}^{j}_{p-2}Q \text{ --- } {}^{j}_{1}Q$$

and the final remainder R of N/D is

$$R = {}^{m}_{1}R$$

Parallel Look-Up Implementation

Step (4) of the division algorithm described above can be implemented with a read-only memory of 2^{m+j} words of m+j bits each. For each m+j bit subdividend, the j-bit partial quotient and the m-bit partial remainder are stored in the ROM addressed m+j as shown in Figure 1.

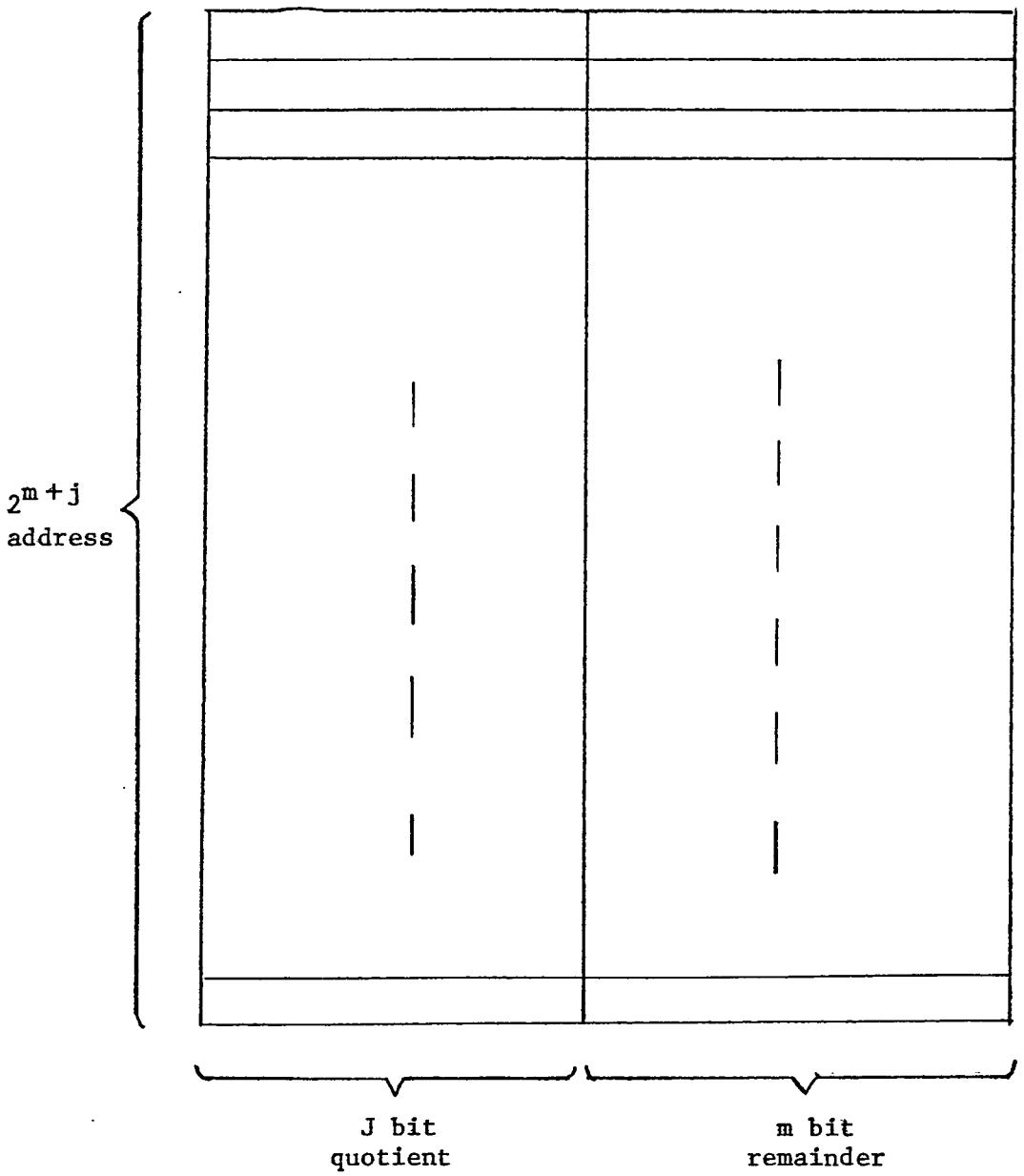

Figure 1: A basic building block
2^{m+j} x (m+j) ROM

Using this ROM as a basic building block, the division of N/D can be implemented as shown in Figure 2.

Discussion of the Parallel Look-Up Implementation

1. For an n-bit dividend, an m-bit divisor and a selected j value, the total number of ROMs needed is $\left[\frac{n+1-m}{j} \right]$ where [ ] denotes the smallest integer which is greater than or equal to the value of the expression enclosed.

2. Since each ROM unit has 2^{m+j} words of $m+j$ bits per word, the total number of bits is

$$\left[\frac{n+1-m}{j} \right] \times 2^{m+j} \times (m+j)$$

3. Figure 5 lists the number of ROM units needed and the ROM size for various values of j when m = 5, and N = 16, 24 and 32 respectively. Figure 7 lists the ROM requirements for m = 8.

4. Comparing Figure 5 and Figure 7, we see that for a selected value j, the total number of ROMs needed increases almost linearly with n, rather than exponentially as the case that a single big ROM is used as a direct look-up table.

5. The second column from the right in Figure 5 and Figure 6 shows the total number of bits in all ROMs as a function of the chosen value of j when the dividend has 16 bits. The last column gives the ratio of the total number of bits required by this algorithm to the total number of bits if a single big ROM of 2^{16} x 17 (5 bit remainder and 12 bit quotient) were to be used for the same 16 bit dividend.

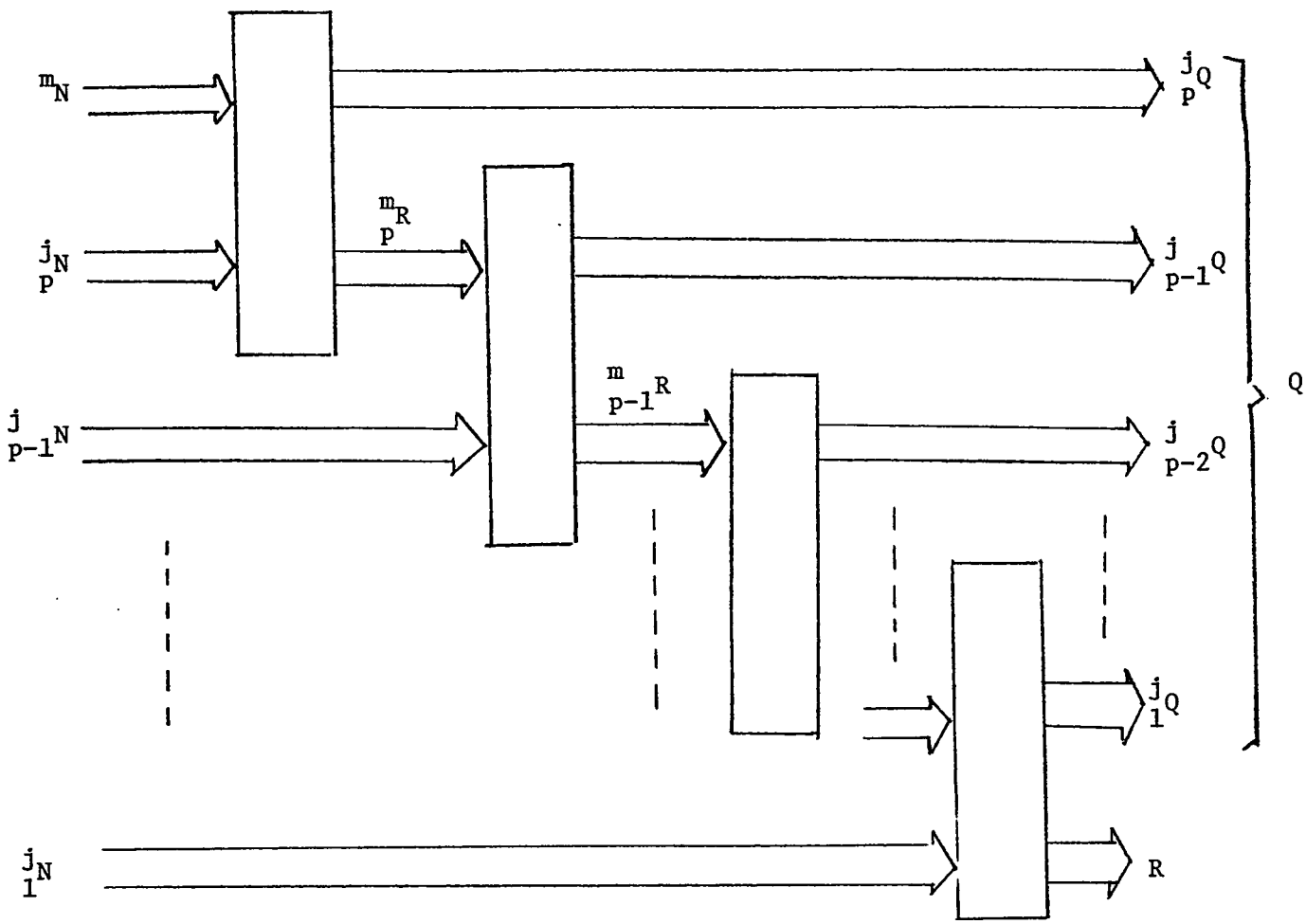

Figure 2: Implementation of a divider for fixed-integer divisors by cascading the basic 2^{m+j} x $(m+j)$ ROMs.

AN EXAMPLE: a 12-bit dividend divided by 13 (decimal).

In this case, a 12-bit binary integer is to be divided by the fixed-divisor 13. Since 13 is a 4-bit binary number, m = 4. If one chooses j = 2, the basic building block ROM will have 64 x 6 bits with the contents illustrated in Figure 3.

Using this ROM as a basic unit, a hardware divider for a 12-bit dividend can be implemented by cascading four basic units as shown in Figure 4. Also shown in Figure 4 is the immediate partial-dividends and partial-remainders for an arbitrarily-selected dividend 1127 (decimal) which is equivalent to the binary number 010001100111.

ADDRESS							OUTPUT							ADDRESS							OUTPUT					
DECI-	BINARY												DECI-	BINARY												
MAL	A_5	A_4	A_3	A_2	A_1	A_0	q_1	q_0	r_3	r_2	r_1	r_0	MAL	A_5	A_4	A_3	A_2	A_1	A_0	q_1	q_0	r_3	r_2	r_1	r_0
0	0	0	0	0	0	0	0	0	0	0	0	0	32	1	0	0	0	0	0	1	0	0	1	1	0
1	0	0	0	0	0	1	0	0	0	0	0	1	33	1	0	0	0	0	1	1	0	0	1	1	1
2	0	0	0	0	1	0	0	0	0	0	1	0	34	1	0	0	0	1	0	1	0	1	0	0	0
3	0	0	0	0	1	1	0	0	0	0	1	1	35	1	0	0	0	1	1	1	0	1	0	0	1
4	0	0	0	1	0	0	0	0	0	1	0	0	36	1	0	0	1	0	0	1	0	1	0	1	0
5	0	0	0	1	0	1	0	0	0	1	0	1	37	1	0	0	1	0	1	1	0	1	0	1	1
6	0	0	0	1	1	0	0	0	0	1	1	0	38	1	0	0	1	1	0	1	0	1	1	0	0
7	0	0	0	1	1	1	0	0	0	1	1	1	39	1	0	0	1	1	1	1	1	0	0	0	0
8	0	0	1	0	0	0	0	0	1	0	0	0	40	1	0	1	0	0	0	1	1	0	0	0	1
9	0	0	1	0	0	1	0	0	1	0	0	1	41	1	0	1	0	0	1	1	1	0	0	1	0
10	0	0	1	0	1	0	0	0	1	0	1	0	42	1	0	1	0	1	0	1	1	0	0	1	1
11	0	0	1	0	1	1	0	0	1	0	1	1	43	1	0	1	0	1	1	1	1	0	1	0	0
12	0	0	1	1	0	0	0	0	1	1	0	0	44	1	0	1	1	0	0	1	1	0	1	0	1
13	0	0	1	1	0	1	0	1	0	0	0	0	45	1	0	1	1	0	1	1	1	0	1	1	0
14	0	0	1	1	1	0	0	1	0	0	0	1	46	1	0	1	1	1	0	1	1	0	1	1	1
15	0	0	1	1	1	1	0	1	0	0	1	0	47	1	0	1	1	1	1	1	1	1	0	0	0
16	0	1	0	0	0	0	0	1	0	0	1	1	48	1	1	0	0	0	0	1	1	1	0	0	1
17	0	1	0	0	0	1	0	1	0	1	0	0	49	1	1	0	0	0	1	1	1	1	0	1	0
18	0	1	0	0	1	0	0	1	0	1	0	1	50	1	1	0	0	1	0	1	1	1	0	1	1
19	0	1	0	0	1	1	0	1	0	1	1	0	51	1	1	0	0	1	1	1	1	1	1	0	0
20	0	1	0	1	0	0	0	1	0	1	1	1	52	1	1	0	1	0	0	X	X	X	X	X	X
21	0	1	0	1	0	1	0	1	1	0	0	0	53	1	1	0	1	0	1	X	X	X	X	X	X
22	0	1	0	1	1	0	0	1	1	0	0	1	54	1	1	0	1	1	0	X	X	X	X	X	X
23	0	1	0	1	1	1	0	1	1	0	1	0	55	1	1	0	1	1	1	X	X	X	X	X	X
24	0	1	1	0	0	0	0	1	1	0	1	1	56	1	1	1	0	0	0	X	X	X	X	X	X
25	0	1	1	0	0	1	0	1	1	1	0	0	57	1	1	1	0	0	1	X	X	X	X	X	X
26	0	1	1	0	1	0	1	0	1	1	1	1	58	1	1	1	0	1	0	X	X	X	X	X	X
27	0	1	1	0	1	1	1	0	0	0	0	1	59	1	1	1	0	1	1	X	X	X	X	X	X
28	0	1	1	1	0	0	1	0	0	0	1	0	60	1	1	1	1	0	0	X	X	X	X	X	X
29	0	1	1	1	0	1	1	0	0	0	1	1	61	1	1	1	1	0	1	X	X	X	X	X	X
30	0	1	1	1	1	0	1	0	0	1	0	0	62	1	1	1	1	1	0	X	X	X	X	X	X
31	0	1	1	1	1	1	1	0	0	1	0	1	63	1	1	1	1	1	1	X	X	X	X	X	X

X: Don't Care

Figure 3: A basic Building Block for Division by 13, Choosing j = 2.

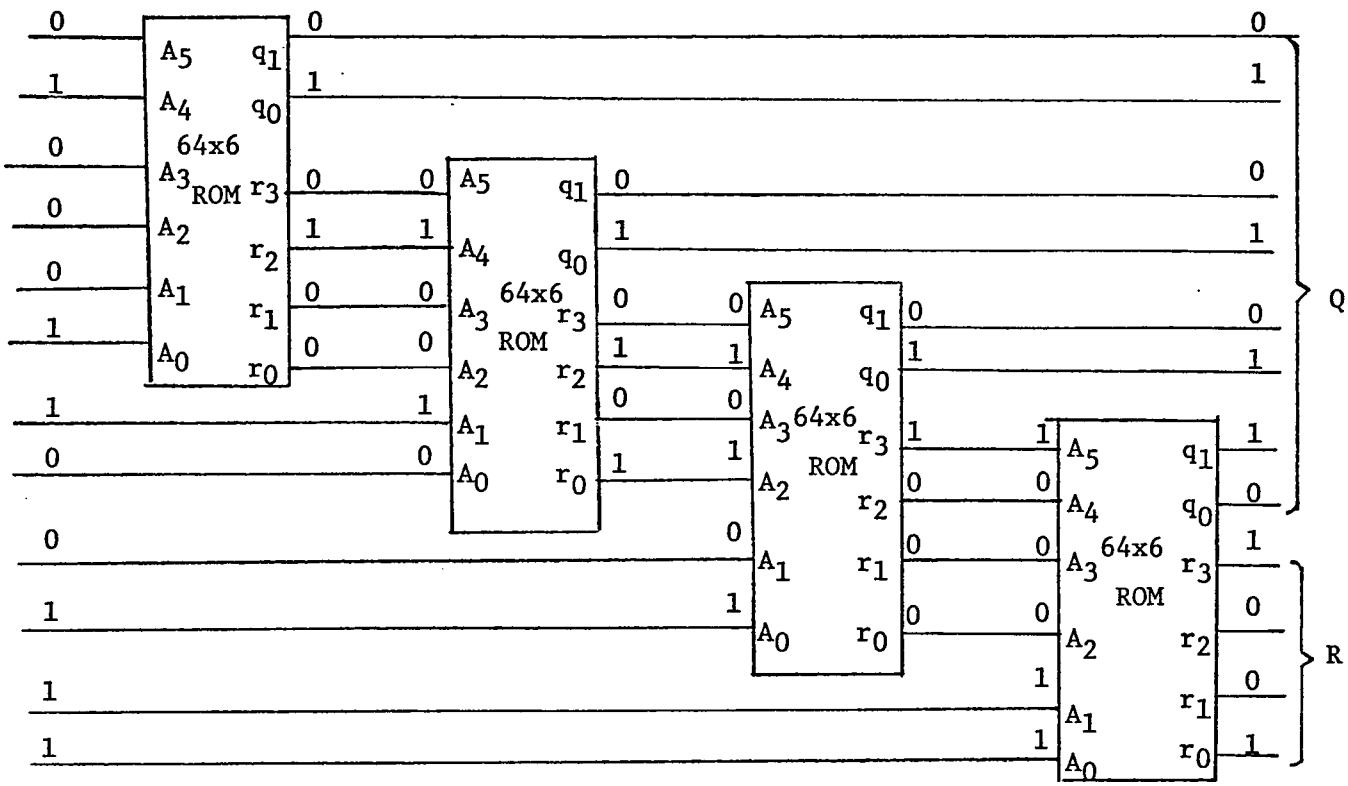

Figure 4: An example showing 1127/13 = 86 + 9/13.

Value of j selected	# of ROM needed			Words per ROM	Bits per word	Total bits for N = 16	Ratio
	N = 16	N = 24	N = 32				
j = 2	6	10	14	2^7	7	5376	.00482
j = 3	4	7	10	2^8	8	8192	.00735
j = 4	3	5	7	2^9	9	13824	.0124
j = 5	3	4	6	2^{10}	10	30720	.0275
j = 6	2	4	5	2^{11}	11	45056	.0404
j = 7	2	3	4	2^{12}	12	98304	.0882
j = 8	2	3	4	2^{13}	13	212992	.1911
j = 9	2	3	4	2^{14}	14	458752	.4117
j = 10	2	2	3	2^{15}	15	983040	.8823

Figure 5: The total number of ROM's needed and the ROM size tabulated for N = 16, 24, 32 and for selected j values from 2 to 10 for m = 5.

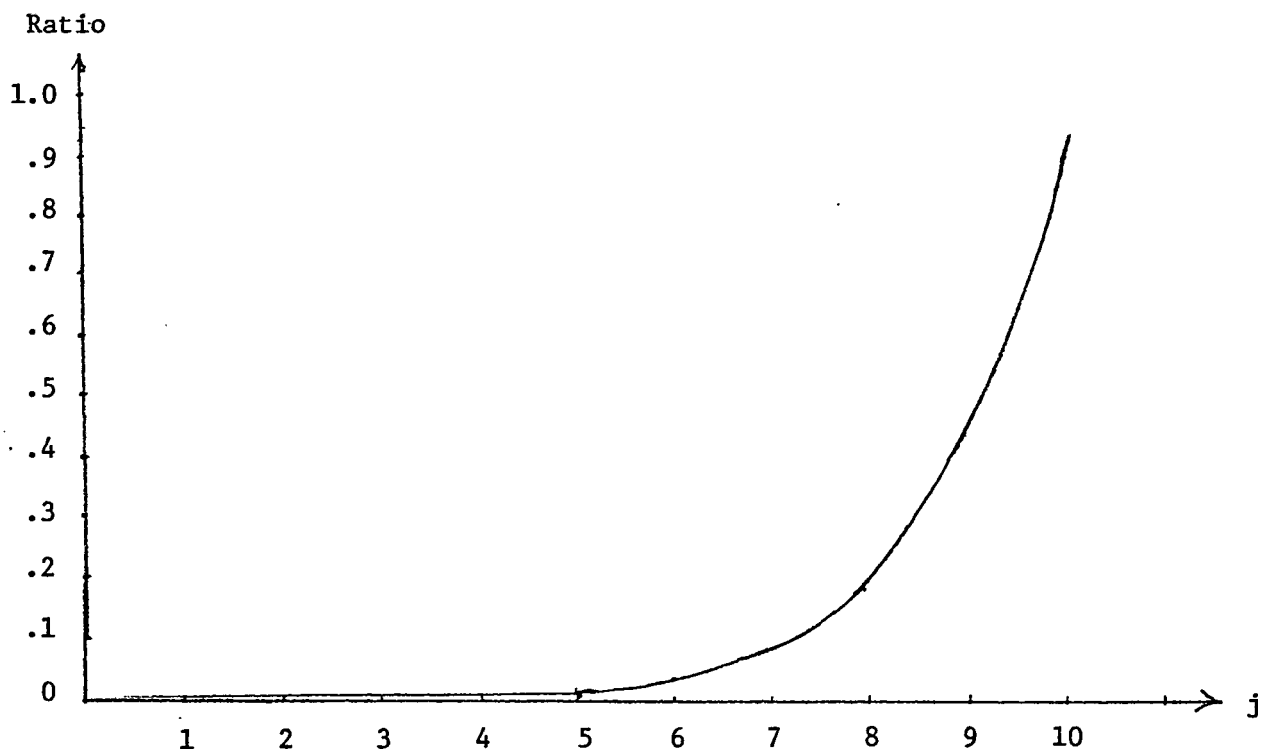

Figure 6: The ratio as a function of j.

divisor bit lengths = 8

Value of j selected	# of ROM needed			Words per ROM	Bits per word
	N = 16	N = 24	N = 32		
j = 2	5	9	12	2^{10}	10
j = 3	3	6	9	2^{11}	11
j = 4	3	5	7	2^{12}	12
j = 5	2	4	5	2^{13}	13
j = 6	2	3	5	2^{14}	14
j = 7	2	3	4	2^{15}	15

Figure 7: The total number of ROMs needed and the ROM size tabulated for N = 16, 24, 32 and for selected j values from 2 to 10 for m = 8.

6. The ratio shown in the last column of Figure 5 is drawn as a function of j in Figure 6. It is obvious from Figure 6 that the ratio and thus the total number of bits increase with the selected j value.

7. The implementation cost and propagation delay for a 5-bit divisor, choosing j = 3, is listed in Figure 8.
TI 74S471 PROM is used, as an example, for the 256 x 8 basic building unit.

8. The implementation cost and propagation delay for an 8-bit divisor, choosing j = 2, is listed in Figure 9. Since there is no single package PROM with 2^{10} x 10 bit organization, a 2^{10} x 4 PROM 82S137 are combined to serve as a basic building block. The cost per package, based on a 25-99 purchase, is taken from the Signetics price list published May, 1977. MOS Memory were to be used, this cost could be cut down by half. However, this reduction in cost would be paid for by a longer propagation delay.

9. Since the size of the ROM used as the basic building block still tends to increase exponentially with the divisor bit length m, the cost of parallel implementation will increase exponentially with the bit number of the divisor. If a large divisor is to be implemented, it is necessary to modify this parallel implementation. Two methods that can be used to improve the parallel implementation for larger divisors are presented in sections which follow.

	N=16	N=24	N=32
Basic ROM Size	256 x 8	256 x 8	256 x 8
# of Package Required	4	7	10
Cost per Package	4.95	4.95	4.95
Access Time (ns)/pkg.	50	50	50
Total Cost	19.80	34.65	49.5
Total Delay Time (ns)	200	350	500

Figure 8: The implementation cost and propagation delay for a 5-bit divisor when j is chosen to be 3. The basic building unit used in this example is the TI 74S471 PROM.

	N=16	N=24	N=32
Basic ROM Size	1024 x 10	1024 x 10	1024 x 10
IC's used as basic unit	82S181 & 82S137	same	same
# of basic units required	5	9	12
cost per 82S181 ($)	31.50	31.50	31.50
cost per 82S137 ($)	11.78	11.78	11.78
cost per basic unit ($)	43.28	43.28	43.28
access time (ns)/basic unit	50	50	50
Total Cost ($)	216.40	389.52	519.36
Total Delay Time (ns)	250	450	600

Figure 9: The implementation cost and propagation delay for an 8-bit divisor, choosing $j = 2$. A 2^{10} x 8 PROM 82S181 and a 2^{10} x 4 PROM 82S137 are combined to serve as a basic ROM unit.

Sequential Look-Up Implementation

When the divisor is long, the cost of the individual ROM can be great if a straight table-look-up scheme is implemented. The size of the individual ROM increases exponentially with the number of bits used in the divisor. When this fact is combined with the fact that the number of these ROMs needed in a parallel table-look-up implementation increases almost linearly with the size of the dividend, it becomes clear that the total cost of such an implementation can be quite large, indeed. A different implementation can provide some reduction in the size of the system. This implementation is designated as the look-up-add implementation instead of the straight-look-up scheme that has been described up to this time. This look-up-add implementation uses some of the memory in a sequential manner, thus resulting in considerable economy when the divisor and dividend are both large.

Since each of the basic ROM modules used in building the system has the same size and same bit pattern as the others, the same ROM can be used over and over in a sequential mode. The remainder of each operation along with the next j-bit group of the dividend is shifted left j-bits at a time, so that it will be in the proper place for the next part of the table-look-up. Furthermore, if the output quotient of the ROM is right-rotated back to the shift register, that register will contain both the quotient and the remainder when the divide process is completed.

A sequential look-up implementation using this approach appears in Figure 10, where the dividend has been partitioned into

j-bit groups and at least one leading zero bit has been added to form the most significant m-bit group.

Figure 11 gives an example of the timing relationship between the control signals. First, the tri-state output of the input register is enabled and the input dividend is loaded into the input register and then into the shift register. After the input data is loaded into the shift register, the tri-state buffer of the input register is disabled and that of the ROM is enabled. After the maximum access time of the basic ROM, the highest order m+j bits of the shift register are updated by loading the m+j bits output of the ROM into the shift register. This is then followed by j fast-shift-right clock pulses to rotate the shift register by j bits. This process repeats cyclically until the final remainder appears as the most significant m bits of the shift register and the final quotient appears as the other bits of the shift register.

The shift register in Figure 10 can be replaced by a general storage register by properly hardwiring the output lines of the ROM and of the storage register to the input lines of the register as shown in Figure 12. The m-bit remainder of the ROM are fed back to the most significant m-bit inputs of the storage register. The j-bits of the ROM are applied as the least significant j-bits of the input. The output lines of the register, besides the highest order m+j lines which are fed to the ROM, are applied to the input of the register in between (see Figure 12).

The advantage of the hardwiring is two-fold. First, the j-bit shift operation can be replaced by a single loading operation. This speeds up the shift operation by a factor of j. Second, the

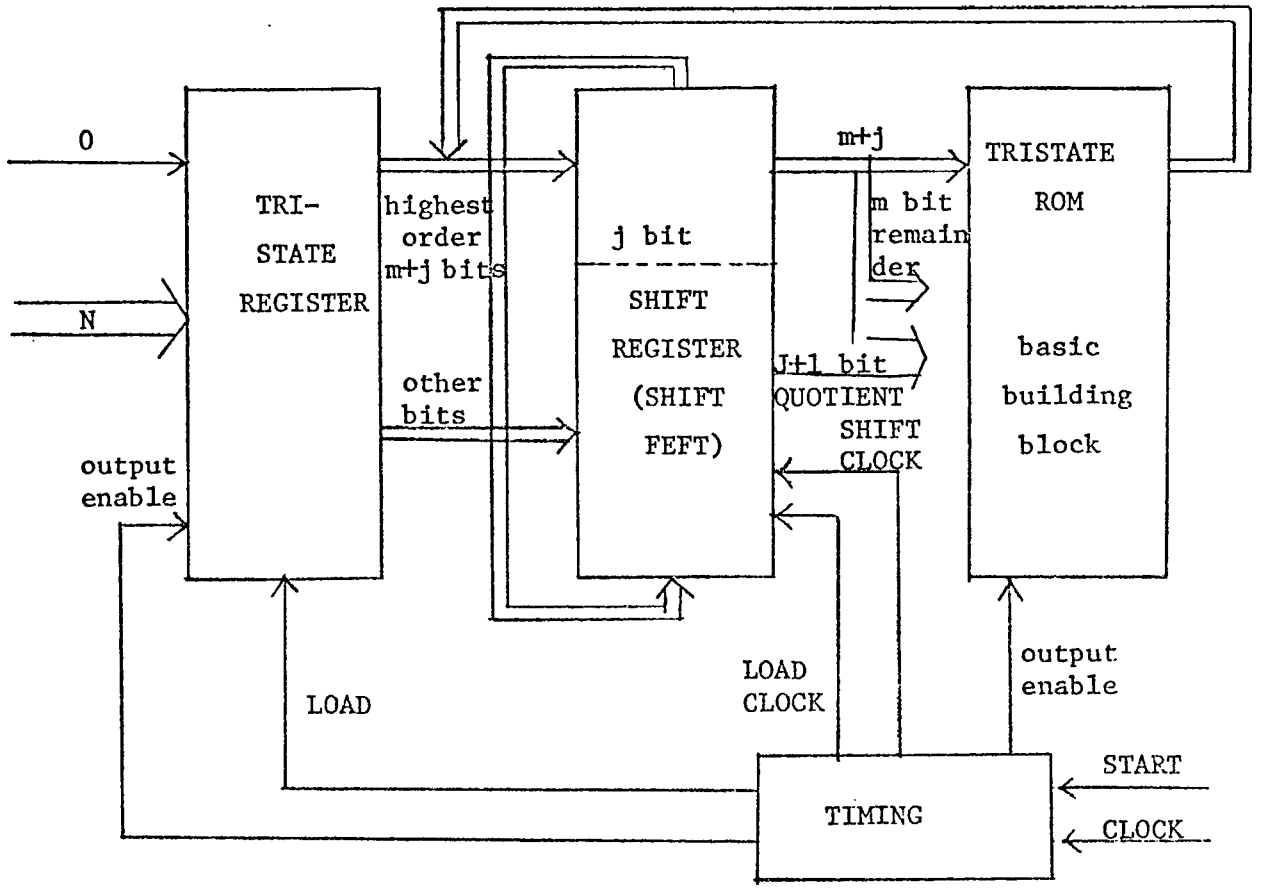

Figure 10: Sequential implementation of a hardware divider for fixed-integer divisors.

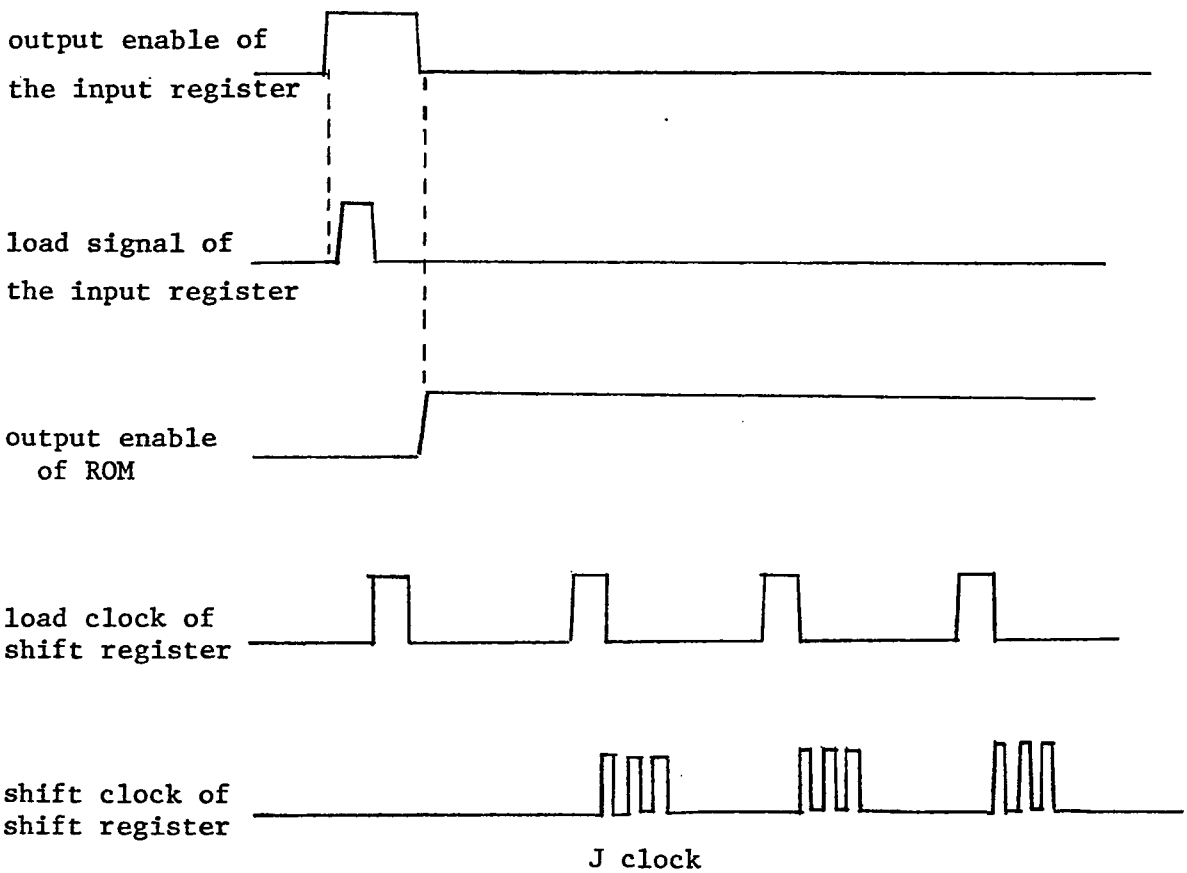

Figure 11: Timing example for Figure 10 assuming every signal is high-active or positive-trigger.

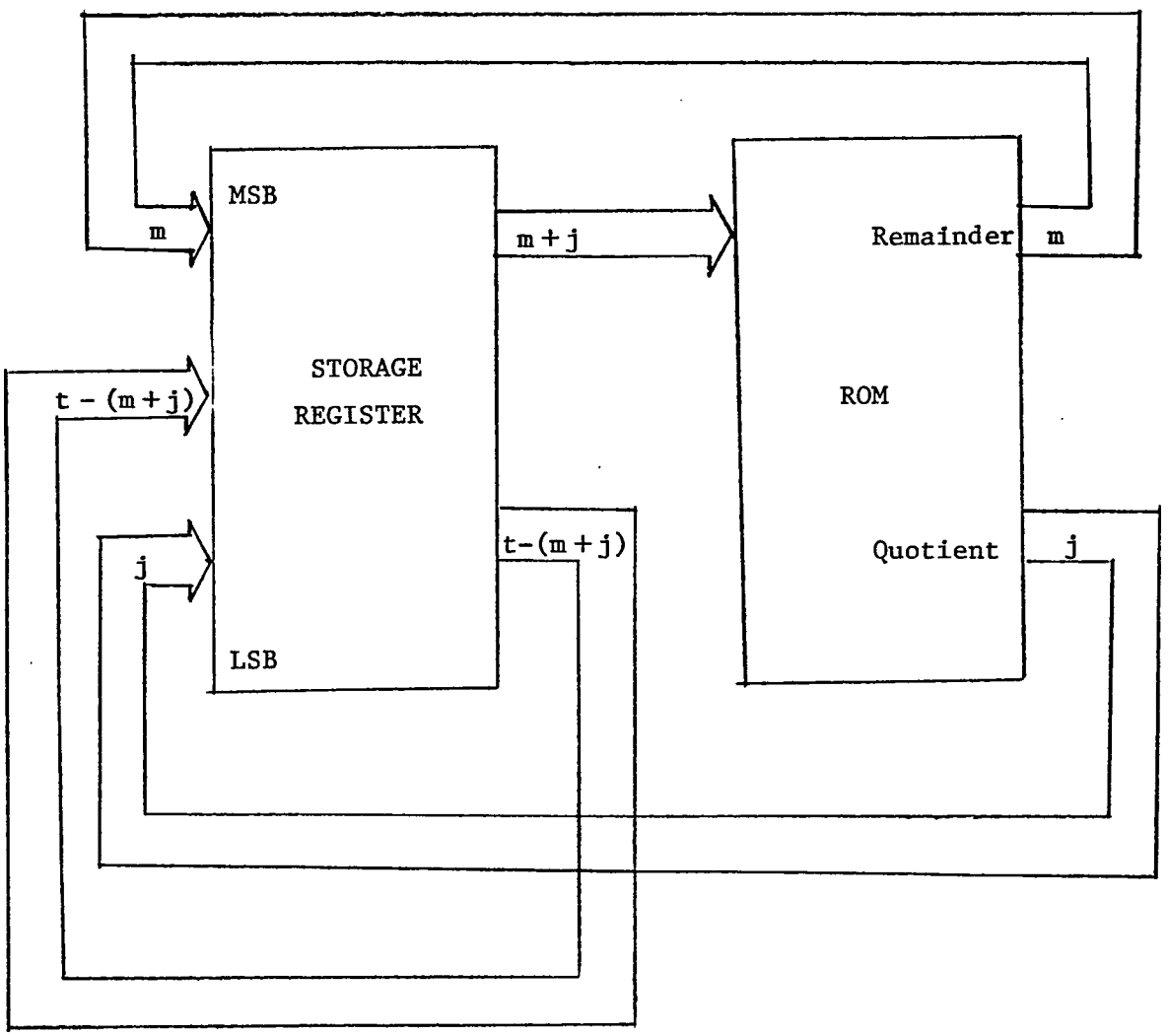

Figure 12: Hardwired rotate left j bits
t is the total number of bits of the storage register.

shift register can be replaced by a more inexpensive storage register and the timing circuitry is relatively simple. The implementation can be more economical.

Figure 13 illustrates the sequential look-up implementation in step-by-step manner. This example shows the intermediate output of the storage register and basic ROM for division of decimal numbers 1127 by 13.

Discussion of the Sequential Look-Up Implementation

(1) The sequential look-up implementation process changes the hardware requirement from several identical basic modular ROMs to a single modular ROM with a simple timing circuit and a storage register. (The input register is not considered as an additional hardware requirement, since it is required in both cases, though it does not have to be tri-state in the parallel implementation.) This will reduce the implementation cost to a different extent according to the bit length of the dividend and divisor.

(2) Figure 14 gives a comparison of the total cost of implementation and the total delay time of the sequential implementation versus the parallel implementation when the divisor is eight bits long. The corresponding information for the parallel implementation is copied from Figure 9. A $40.00 implementation cost is estimated for the timer. This, plus the cost per basic ROM ($43.28) and a few dollars for the storage registers, will make the estimated total cost of $90.00. Although any larger dividend takes more storage

Loading Input	Shift Register	0 1 0 0 0 1	1 0 0 1 1 1
	ROM	0 1 0 1 0 1	
First Rotate	Shift Register	0 1 0 0 1 0	0 1 1 1 0 1
	ROM	0 1 0 1 0 1	
Second Rotate	Shift Register	0 1 0 1 0 1	1 1 0 1 0 1
	ROM	0 1 1 0 0 0	
Third Rotate	Shift Register	1 0 0 0 1 1	0 1 0 1 0 1
	ROM	1 0 1 0 0 1	
Fourth Rotate	Shift Register	1 0 0 1 0 0	0 1 0 1 1 0
		Remainder	quotient

Figure 13: The intermediate output of the shift register and the basic ROM by the sequential implementation of 1127/13. Refer to Figure 4 for comparison.

		N = 16	N = 24	N = 32
Total Cost ($)	Parallel	216.4	389.52	519.36
	Sequential	90.0	85.0	85.0
Propagation Delay (ns)	Parallel	250	450	600
	Sequential	375	675	950

Figure 14: Comparison of sequential implementation versus parallel implementation for an eight bit divisor.

space, the cost of this storage is relatively small compared to other costs ($0.81 per 6-bit 74174 register). Hence, the same cost is assumed for all dividends.

(3) A propagation delay time of 25 ns for type 74174 registers is added to the 50 ns delay time of the basic ROM to give the 75 ns delay time per rotating operation. The total delay time given in Figure 14 is calculated by multiplying the number of rotating operations by 75 ns.

(4) It is obvious from Figure 14 that the sequential implementation cost stays constant as N (the total number of bits of the dividend) increases.

(5) Sequential implementation has longer delay time.

Look-Up-Add Implementation of the Basic Building Block

The minimum size of the basic building block is $2^{m+1} \times (m+1)$ for an m-bit divisor in a straight look-up implementation. A 16-bit divisor, would need a ROM of at least $2^{17} \times 17$ to implement a basic building block. Since a ROM of this size is impractical by today's technology, another method must be sought in order to make the implementation practical if the divisor is large.

Each basic building block has $(m+j)$ address lines. These $(m+j)$ lines can be partitioned into two groups, the most significant $(j+1)$ bit group and the remaining least significant $(m-1)$ bit group. Thus the $(m+j)$ bit dividend can be regarded as the sum of two components. The first component, component A, is an $(m+j)$ bit number and is formed by adding $(m-1)$ bits of 0's to the right of the $(j+1)$

bit group. The second component, B, is an (m - 1) bit number and is the same as the (m - 1) bit group.

By the division property of (A + B)/c = A/c + B/c, we know that the final quotient and remainder can be computed from the quotient and remainder of each component. Since component B is an (m - 1) bit number, there is no quotient directly generated from component B. Component A, although it is an (m + j) bit number, has only (j + 1) significant bits. The quotient and remainder generated from component A can thus be directly looked-up from a 2^{j+1} x (m + j) ROM. The remainder from component A are then added to component B. The sum, (sum S), is always less than twice the divisor, since both component B and the remainder from component A are less than the divisor.

The final quotient and remainder can be derived from the sum S and the quotient from component A as follows:

If sum S is less than the divisor, sum S itself is the final remainder, and the quotient from component A is the final quotient. If sum S is not less than the divisor, the final quotient is one plus the quotient from component A, and the remainder can be extracted by subtracting sum S by the divisor.

From the above introduction, it is clear that a basic building block for large divisors can be implemented as shown in Figure 15. The single line output from the comparator is fed to the carry input of an j bit adder. The sum from the m bit adder is applied to the minuend part of the subtractor. The subtrahend to the subtractor is either zero or the divisor depending on the output of the comparator.

The look-up-add implementation for an 8-bit divisor 1101 0101 (decimal 213) is shown in Figure 16. A 12-bit number 1011 1101 0011

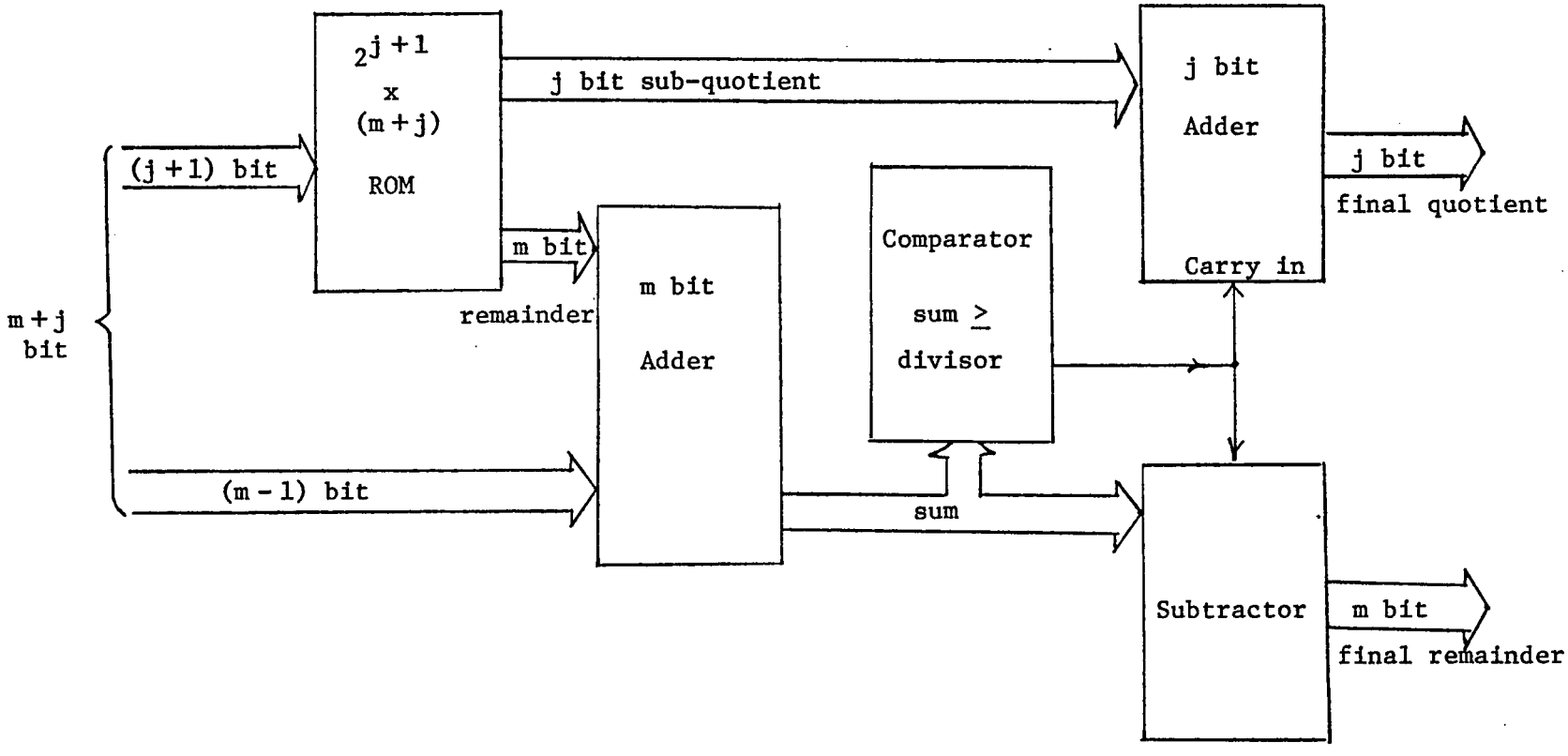

Figure 15 Look-up-add implementation of the basic building block.

(decimal 3027) is arbitrarily selected as the dividend. Since this is an example of m = 8 and j = 4, the most significant 5 bits (10111) of the dividend are applied to the address input of the 32 words by 12 bits ROM. The 4 bit quotient 1101 and the 8 bit remainder 1010 1111, from the 12 bit component 1011 1000 0000, appear at the output part of the ROM.

The subtractor in Figure 15 is replaced by an adder with the addend lines hardwired as shown in Figure 16. The lines corresponding to the "1" bit in the 2's complement of the divisor are tied together and connected to the comparator output. The lines corresponding to the "0" bit are tied to ground. Hence, if the comparator output is 1, the 2's complement of the divisor is added to the sum of the first m bit adder. If the comparator output is 0, nothing is added and the sum of the first adder appears as the sum of the second adder and as the final remainder.

Discussion of the Look-Up-Add Implementation

(1) For an m-bit divisor and a chosen value of j, the look-up-add method changes the hardware requirement from a 2^{m+j} x (m+j) ROM to a smaller 2^{j+1} x (m+j) ROM, two m-bit adders, a j-bit incrementer and a (m+1) bit comparator. This reduces the ROM requirement by a factor of 2^{m-1}.

(2) The word size of the new ROM is 2^{j+1}. This is not a function of the bit length of the divisor. Hence no matter how large the divisor is, the ROM will have the same word size.

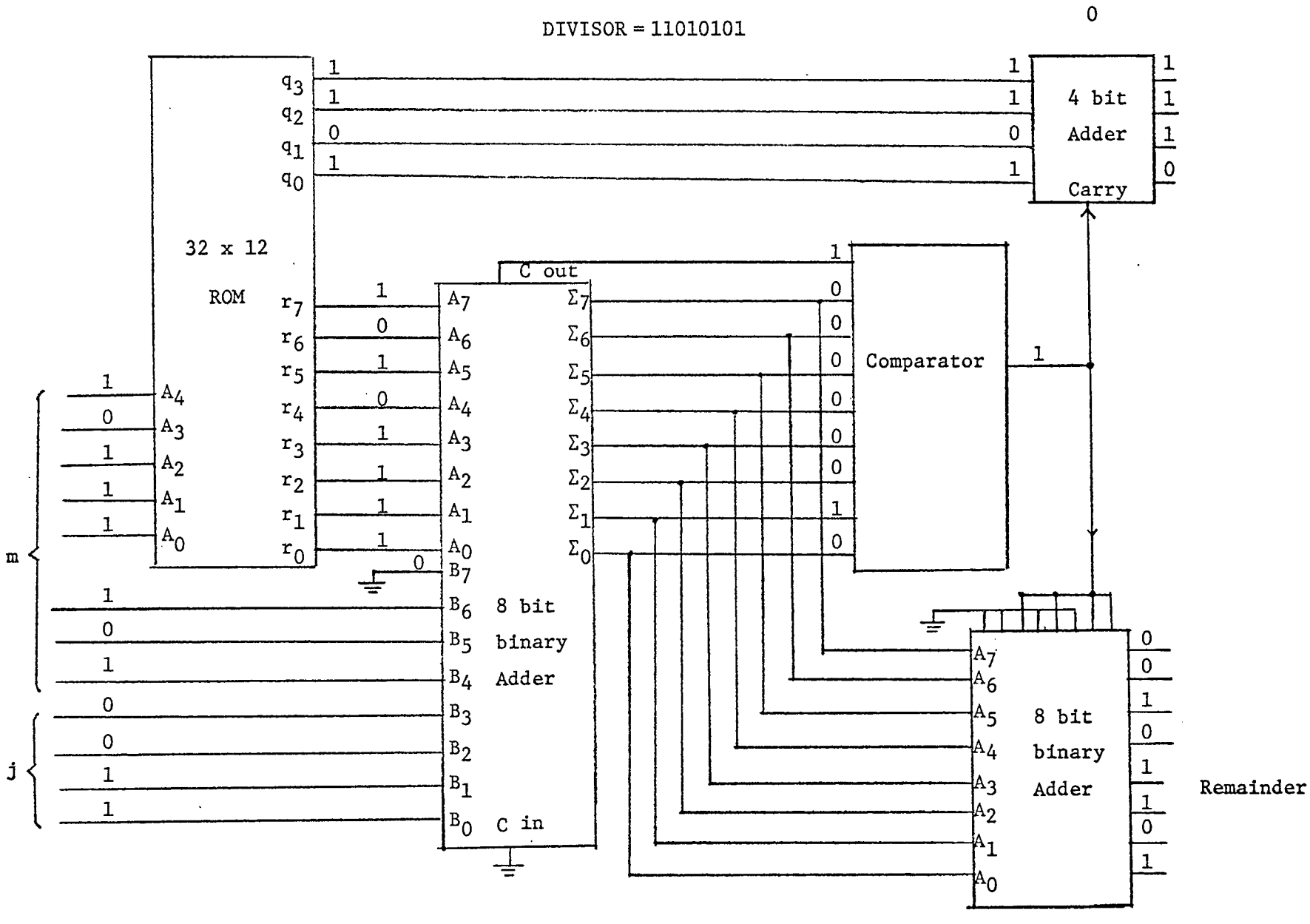

Figure 16: Example showing look-up-add implementation for 3027 ÷ 213.

(3) Besides the j-bit incrementer (or adder), the hardware cost of the comparator and the adders is a linear function of m. Since the cost of the 2^{j+1} x (m+j) ROM is also a linear function of m. We can conclude that the look-up-add implementation cost will increase nearly linearly rather than exponentially with increases in the bit length of the divisor.

(4) It was suggested that in the parallel look-up implementation, a small j be selected to reduce the total number of bits. (See Figure 5 and Figure 6). In the look-up-add implementation, present technology permits the choice of a slightly larger value, around 8. The ROM is not the major cost in the basic building block and a larger value of j reduces the total number of basic building blocks needed. For example, a 24-bit dividend divided by a 16-bit fixed-integer divisor can be processed by a single basic building block if j is chosen to be 8. The total hardware requirement will be a 512 x 24 ROM, two 16-bit binary adders, an 8-bit incrementer and a 17-bit comparator. The division time is in the range of 200 ns, if one uses an 82S141 for the 512 x 24 ROM, a type 7483 for the adder and type 7485 for the comparator.

(5) One shortcoming with the look-up-add implementation is that it increases the total chip count. It is not suited for parallel implementation since the circuitry has to be duplicated in each basic building block. But when used in a sequential implementation, it is a powerful method to process very large dividends and divisors. For example, a 64-bit dividend divided by a 32 bit divisor can be processed in

four sequences with j = 8. The division time is in the range of 1.2 μsec with an estimated 300 nsec delay time per sequence.

Hardware Divider for Small Variable-Integer Divisors

Several methods of implementing a hardware divider for fixed-integer divisors have been presented in the previous sections. An interesting question arises at this point: "Can any of these methods be modified to implement a general purpose hardware divider for any variable divisors, or at least for small variable-integer divisors?"

An interesting tentative approach is to try adding the m bits of the divisor to the address of each ROM in a parallel implementation. The size of each ROM is hence enlarged by a factor of 2^m from 2^{m+j} words to 2^{2m+j} words. The 2^{2m+j} words can be partitioned into 2^m regions of 2^{m+j} words each. The region number is the most significant m bits of the address. The contents in each region will be the quotient and the remainder of a division. The divisor of this division is the region number, the dividend is the less significant m+j bits of the address.

To make the parallel implementation compatible with this approach, a minor modification, (other than adding m bits to the address of each ROM), must be made. This modification is associated with the first stage ROM. We have assumed that a (m+j)-bit dividend divided by an m-bit divisor will have an j-bit quotient. This is true only under the condition that the most significant bit of the m-bit divisor is a "1", and the (m+j)-bit dividend is less than the divisor. If the divisor can be any m bit number including 000---01, the quotient will have the same number of bits as the dividend in

the worst case. Hence a $(m+j)$-bit space must be reserved for the quotient of the first stage ROM. This increases the size of the first stage ROM to $2^{2m+j} \times (2m+j)$

The m-bit remainder from the first state ROM will be less than the divisor, hence a j bit space will be enough for the quotient of the ROM's from the second stage and on.

A parallel implementation of a general purpose hardware divider is shown in Figure 17. The first stage ROM has $2^{2m+j} \times (2m+j)$ bits. The other stage ROM's have $2^{2m+j} \times (m+j)$ bits.

Figure 18 shows the sequential implementation of a general purpose hardware divider. The circuitry illustrated in Figure 10, with the m bits of the divisor added to the address of the ROM, is represented by the sequential divider block in Figure 18. The first stage ROM is not included in this block, since it is different from all other ROMs.

The proposed approach represents a simple means of implementing a general purpose hardware divisor. It is very efficient for small divisors and can be implemented either in parallel or sequential configuration. For a divisor with medium length, sequential implementation with the look-up-add method can reduce the size of memory required for the basic building block from 2^{2m+j} words to 2^{m+j+1} words with the aid of two 8 bit adder, a 9 bit comparator and a j bit incrementer. For a large divisor, say 16 bit, a basic building block will require a ROM of 2^{18} words at least. ($j = 1$, $m = 16$). This is not practical by today's technology.

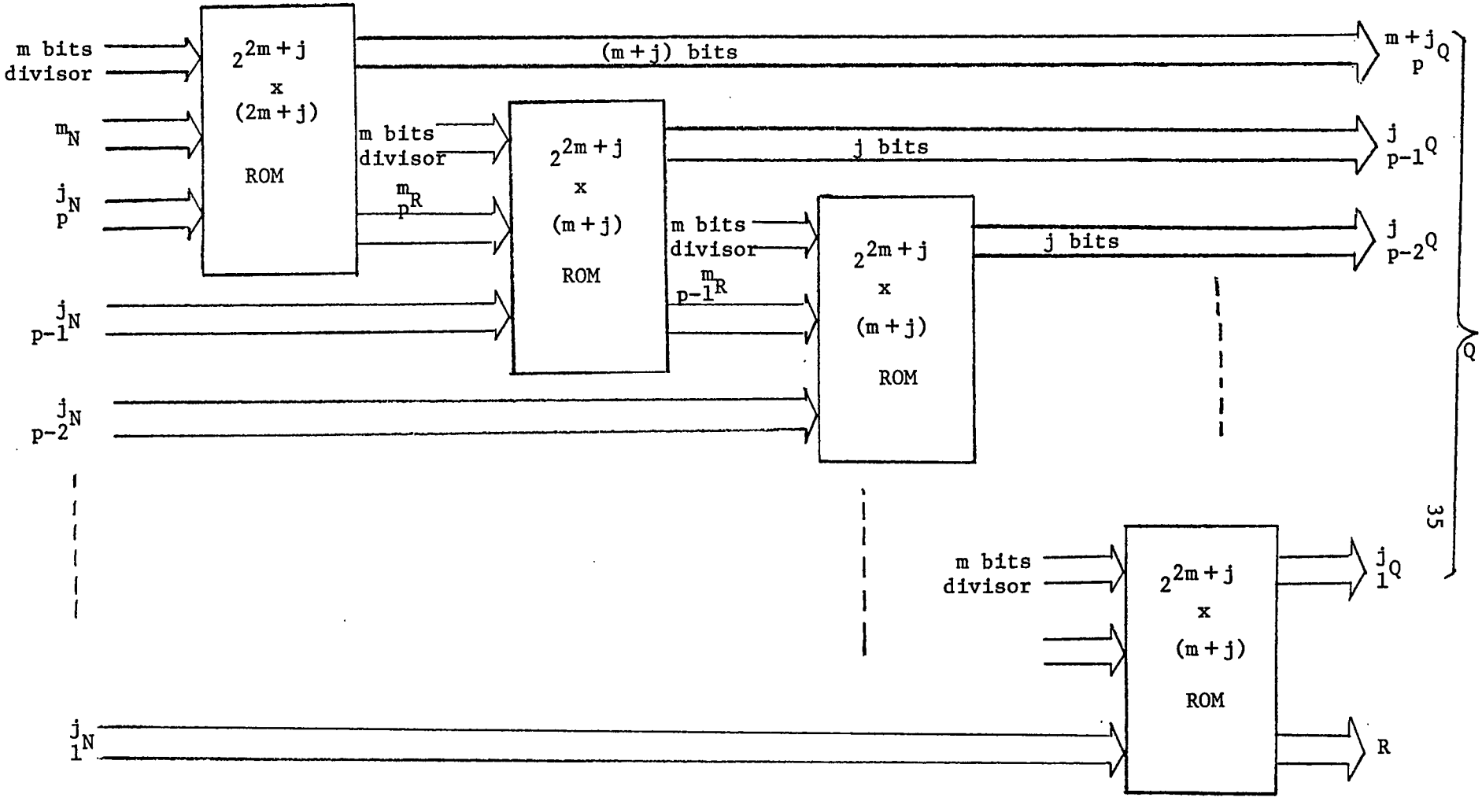

Figure 17: Parallel implementation of a general purpose hardware divider for small variable-divisors. Refer to Figure 2 for symbol definition.

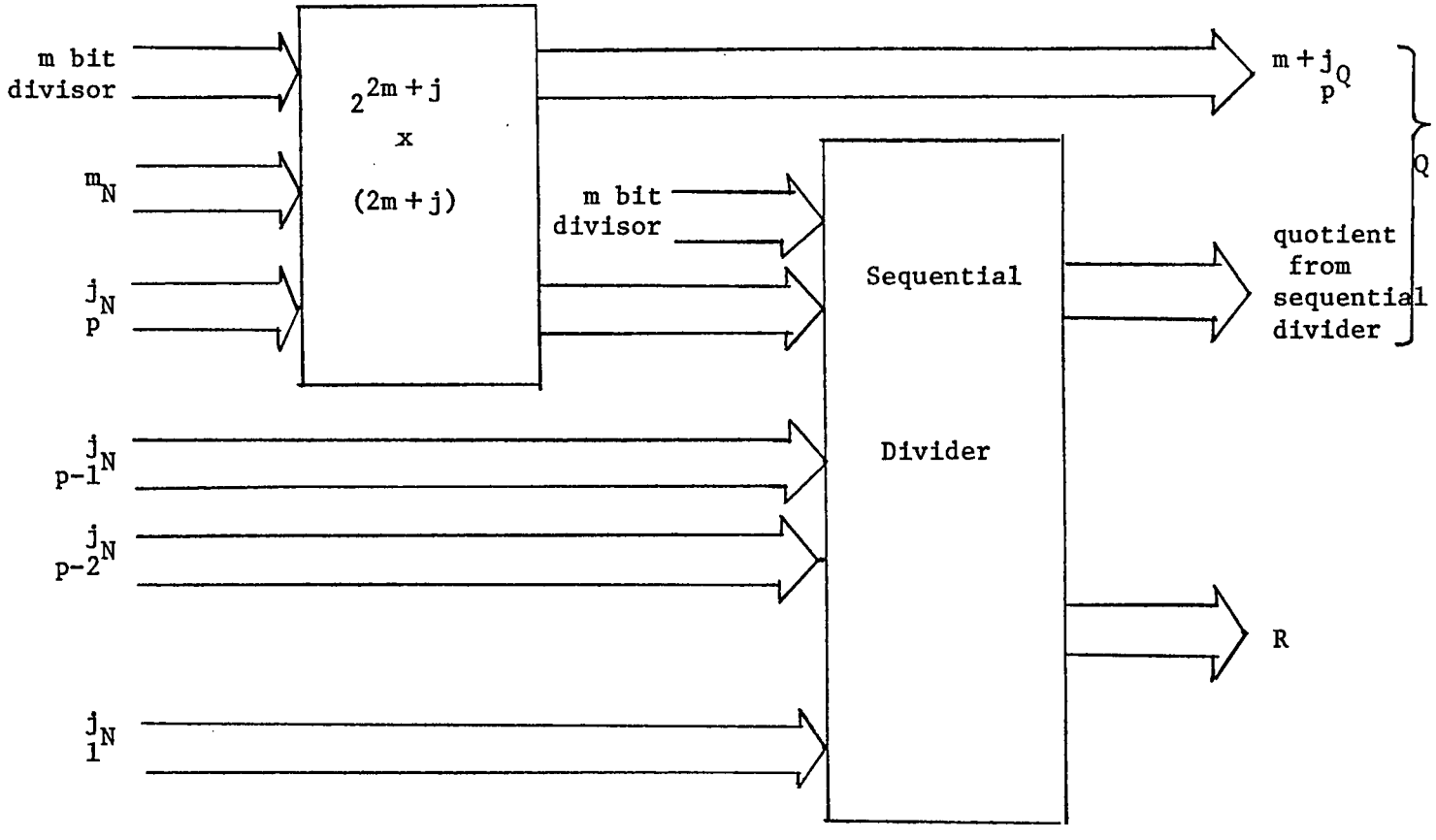

Figure 18: Sequential implementation of a general purpose hardware divider for small variable divisors.

Applications

EXAMPLE 1: Sector Location in a Track-Organized Disk

In this example, it is desired to locate a physical sector in a 19 sectors/track disk. The sectors are to be addressed contiguously independent of track boundary. In total, the disk has 256 x 19 physical sectors. In order to move the head to a new physical sector, the disk controller would have to divide this new address by a fixed divisor 19 to translate to its corresponding track and sector number.

Since 256 x 19 has a 13-bit dividend and a 5-bit divisor, the divider can be implemented using four 256 x 8 ROMs (choosing j = 3; see Figure 5).

EXAMPLE 2: BCD-to-Binary Converter

The look-up-add method can be applied also for BCD-to-Binary conversion. A 6 decade BCD number is selected here as an example to illustrate the general algorithm. The result will be compared to the 74184 converter.

By partitioning the 24-bit BCD number into 3 bytes, the 6 decade number can be viewed as a sum of three components. The first component is a 24 bit number and is formed by adding 2 bytes of 0's to the most significant byte. The second component is a 16 bit number. It is formed by adding 1 byte of 0's to the center byte. The third component is the same as the least significant byte.

Each of the three components has only 8 bit significance. A 256 x 20 ROM can be programmed for the first component. Similarly a 256 x 14 and a 256 x 7 ROM can be built for the second and the

third component respectively. By summing the output of the three ROM's, we will have the final converted binary number.

Each ROM is a direct look-up table with the converted BCD number stored. For example, if the converting BCD number is 129538, the output from the 256 x 20 ROM will be 0001 1101 0100 1100 0000 which is the binary representation of decimal 120000. The output from the 256 x 14 ROM is 100101 0001 1100 which is equal to decimal 9500. The output from the 256 x 7 ROM is 010 0110 which is decimal 38. The sum of the three output is 0001 1110 1010 0000 0010, which is exactly equal to decimal 129538.

From the above example, one might have observed that the least significant 4 bits output of the 256 x 20 ROM and the least significant 2 bits output of the 256 x 14 ROM are 0's. This is not a particular outcome of this example but instead is a universal feature. The proof is as follows:

The unit of the address of the 256 x 20 ROM is decimal 10000. This is equivalent to binary 0010 0111 0001 0000. Since the least significant four bits of the unit is 0000, any integer multiple of the unit should keep this attribute. This argument applies for the 256 x 14 ROM.

Taking advantage of this feature, the 256 x 20 ROM can be reduced to 256 x 16, and the 256 x 14 ROM can be reduced to 256 x 12.

Figure 19 shows a 6 decade BCD to binary converter. The total hardware requirement is four 256 x 8 PROM's, one 256 x 4 PROM and seven 4-bit binary adders. The total delay time is the sum of the access time of the ROM's, the 12 bit binary addition time and the 16 bit binary addition time. If TI 74S471, TI 74S387 and 7483A are

used for the 256 x 8 ROM, the 256 x 4 ROM and the 4-bit binary adder, respectively. The total delay time will be 136 nsec typically. (50 nsec for 74S471, 43 nsec for adding two 12 bit or 16 bit words as listed in the TI TTL and MEMORY DATABOOK).

Figure 20 gives a performance comparison between this look-up-add method and the TTL 74184 method. It is obvious that the look-up-add method is much better than the TTL 74184. Its use reduces the cost from $79.8 to $28.7, the delay time from 364 ns to 136 ns, and the chip count from 28 to 12. Hence, it is strongly recommended that the look-up-add method be used to replace the conventional TTL 74184 method.

EXAMPLE 3: Binary-to-BCD Converter

The look-up-add method can be applied to implement a binary to BCD converter. A 16 bit binary number is selected as an example to illustrate this method.

Again, the 16 bit number is split into two bytes. The 20 bit BCD representation of the first component and the 10 bit BCD representation of the second component are stored in a 256 x 20 and a 256 x 10 ROM, respectively. The first component is formed by adding one byte of 0's to the right of the high-order byte of the 16 bit number. The second component is the same as the low-order byte.

The least significant of the 20 bit output from the first component is always a zero. Since this one bit saving does not result in any significant cost reduction, it is ignored in the following descriptions.

The 20 bit output and the 10 bit output can be added decimally by a 20 bit BCD adder. The sum will be the final answer.

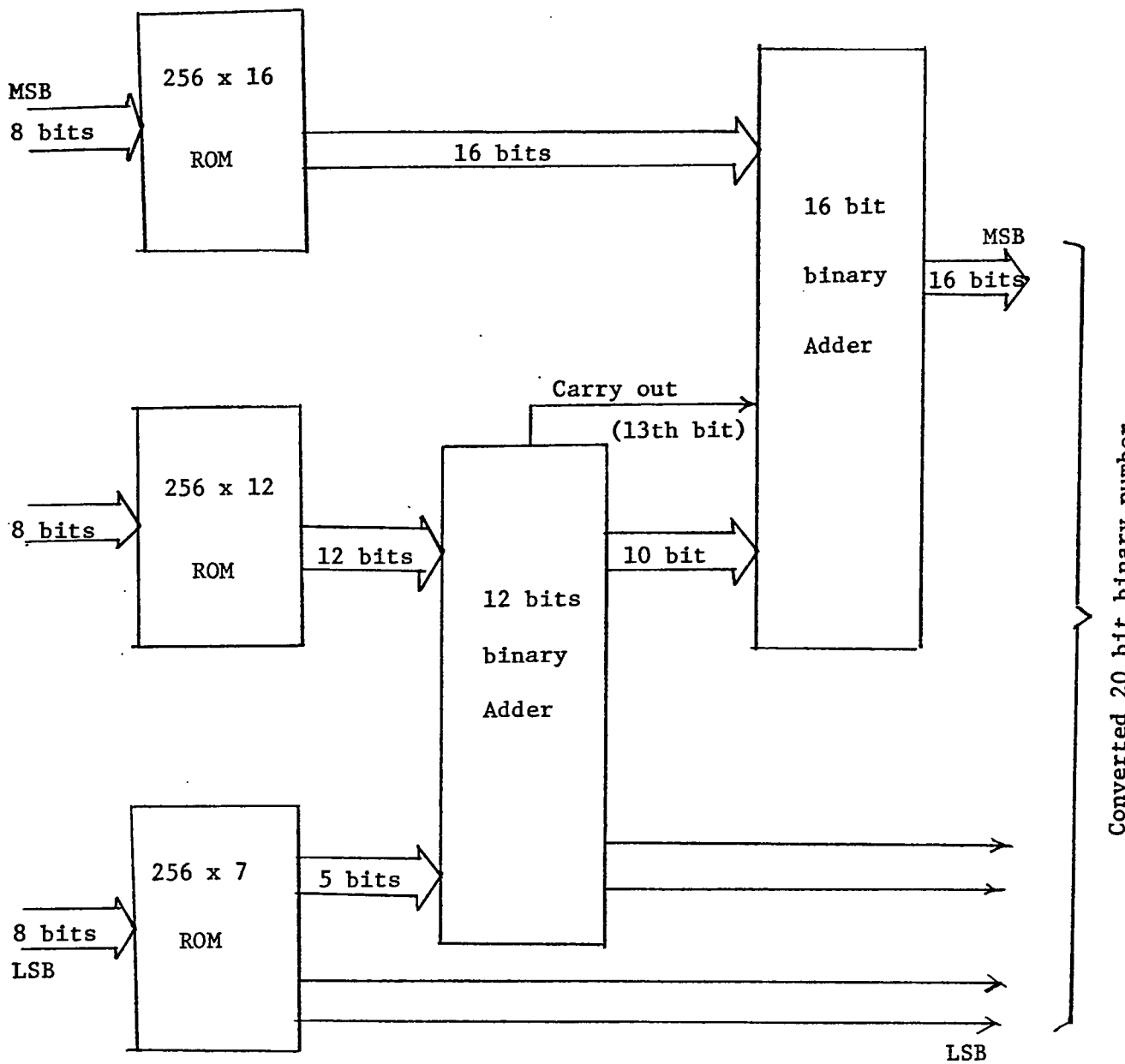

Figure 19: A 6 decade BCD to binary converter.

FEATURES COMPARED	74184 method	look-up-add method	factor improved
# of chips required			
74184 ($2.85 each)	28	—	—
74S471 ($4.95 each)	—	4	—
74S387 ($3.65 each)	—	1	—
7483A ($0.75 each)	—	7	—
total cost ($)	79.8	28.7	2.78
total delay time (ns)	364	136	2.68
total chip counts	28	12	2.33

Figure 20: Comparison between the 74184 converter and the look-up-add converter for a 6 decade BCD to binary conversion.

Figure 21 gives a comparison between this look-up-add method and the conventional 74185 method. It shows that the look-up-add method improves the typical delay time from 200 ns to 100 se and reduces the chip count from 16 to 10. Although the figure shows that the implementation cost is slightly higher, this is the result of the high cost of the Signetics 82S83 4-bit BCD adder. The BCD adder is more expensive than the binary adder mainly because of limited demand.

Conclusion

Several methods of implementing a hardware divider for fixed-integer divisors have been demonstrated. For small divisors, these methods can be expanded to implement a general purpose hardware divider. The applications of these methods include BCD-to-binary conversion and binary-to BCD conversion. The resulting converters are simpler and faster than other parallel-conversion schemes and the cost is much less for BCD-to-binary converter and slightly higher for binary-to-BCD converter.

FEATURES COMPARED	74185 method	look-up-add method	factor improved
# of chips required			
74185 ($2.85 each)	16	—	—
74S471 ($4.95 each)	—	3	—
74S387 ($3.65 each)	—	2	—
82S83 ($6.75 each)	—	5	—
total cost ($)	45.6	54.4	.84
total delay time (ns)	200	96	2.08
total chip counts	16	10	1.60

Figure 21: Comparison between the 74185 method and the look-up-add method for implementing a 16 bit binary-to-BCD converter.

REFERENCES

(1) "The IBM System/360 Model 91: Floating Point Execution Unit," IBM Journal (January 1967).

(2) Jacobsohn, David H. "A Combinatoric Division Algorithm for Fixed-Integer Divisors," IEEE Transactions on Computers (June 1972).

(3) "Making Small ROM's Do Math Quickly, Cheaply and Easily," Electronics (May 11, 1970).

(4) "Computing the Square Root of Binary Numbers," Computer Design (August 1972).

(5) "Binary-to-BCD - Conversion with Complex IC," Computer Design (Sept. 1970).

(6) "Type SN 54/74185 A Binary to BCD Converter," Databook. Texas Instruments.

(7) Computer Design Development. (1976).