A 16-Bit LSI Digital Multiplier

Rodney Tak Masumoto

Computer Science Department
California Institute of Technology

4204:TR:78

A 16 BIT LSI DIGITAL MULTIPLIER

by

Rodney Tak Masumoto

Technical Report #4204

Department of Computer Science

California Institute of Technology

In partial fulfillment of the requirements for the Degree of Electrical Engineer to my wife

ACKNOWLEDGMENTS

The author expresses his grateful appreciation to Dr. Carver A. Mead for his patient guidance and many contributions to this work. Special acknowledgments also go to Dr. Ivan Sutherland, Dr. William Heller, Dr. Amr Mohsen, Dr. Edmund Cheng, Mr. Vance Tyree, and Mr. Vincent Wong for their contributions to the development of this material. Many thanks are due to Barbara Lum for her aid in the typing of this manuscript; and a continuing gratitude is expressed to my parents for their constant patience and encouragement.

The work reported here was suported in part by the Office of Naval Research. Financial assistance from the California Institute of Technology in the form of several research and teaching assistantships is also gratefully acknowledged. Fabrication of the integrated circuit was performed by Maruman Integrated Circuits, Inc.

ABSTRACT

Multiplication in digital machines is often done sequentially by the processor's arithmetic logic unit. However, this method is very time consuming due to the many sequential shifts and additions required. Implementing this multiplication directly with hardware increases speed, but at added cost. By implementing an interesting multiplication algorithm on an LSI chip, it is possible to achieve high performance with little added cost.

This paper describes a single chip LSI implementation of such a hardware multiplier. This multiplier/accumulator chip performs a fast multiplication of two 16 bit 2's complement words. It was designed and implemented in silicon gate NMOS with depletion loads. By using a multiple bit examination algorithm, the circuitry requirements were significantly less than that of a standard hardware multiplier. Also, by employing carry-save adders and carry lookahead logic, multiplication delay times are competitive with bipolar implementations, but require one-fifth the power.

An on-chip accumulator allows successive products to be summed without tying up the external data bus. Special completion sensing logic allows the chip to be used in asynchronous timing applications. The 16 bit by 16 bit multiplier chip measures 180 by 180 mils. All circuits are modular, and chips of arbitrary word size can be generated by changing only two parameter values during the computer aided mask layout generation process.

TABLE OF CONTENTS

I. INTRODUCTION

The multiplication of two numbers is often one of the more time consuming operations performed by modern computer CPU's. Recent advances in integrated circuit technology have made possible the opportunity to implement this function on a single LSI chip. A 16 bit by 16 bit LSI multiplier using bipolar technology has been fabricated by TRW [1]. To date no such chip exists that uses MOS technology. Thus, an implementation using the MOS process was investigated.

Like any of the other LSI processes, MOS implementations have both strong and weak points. As demonstrated by MOS memories and other MOS LSI chips, it is possible to achieve high density and relatively low power dissipation with this process. This is particularly true when dynamic circuits are used. Internal speeds of 3 to 5 ns per gate delay are also common. It is only when signals go off-chip that the time delays become significantly slower than with other processes such as bipolar. Also, circuit implementations of a given logic design tend to be less complex in MOS than with other processes. In many cases, active devices occupy a surprisingly small percentage of the total chip area. Interconnections usually end up occupying much of the space. Finally, the MOS process itself has not fully matured. An order of magnitude improvement in performance is still expected.

On the less desirable side, MOS circuits have a relatively high source drive impedance. This is most noticeable by the long delay times (40-50 ns) required to drive the relatively high capacitance external loads. Even internal to the chip, long interconnect wiring can cause appreciable signal delays. A 100 micron signal line presents a load similar to that of a standard gate input. However, this problem can be kept minimal through the use of designs that recognize this shortcoming. Due to MOS's simple circuit structure, regularity in circuit design and layout can be achieved. Thus long signal paths can usually be kept to a minimum. An additional benefit from eliminating long wires is that chip area can be used more efficiently.

At the gate level, MOS pull-up signal response tends to be much slower than pull-down delays. Through proper use of precharged dynamic circuits, this biased speed characteristic can be used to advantage. It is usually possible to pre-charge the heavily loaded signal lines to a high state. Thus if any level transition must occur, the delay will be that of the faster pull-down case. Nodes where pull-up transitions must occur can be designed to have minimal loading.

II. THE COMBINATORIAL MULTIPLIER

It will be assumed that the multiplication will involve a combinatorial process. Therefore no discussion of any pipeline methods will be made. Our concern is with the general area of parallel multiplication methods. The basis from which this discussion will develop is the straightforward combinatorial array. This array is simply the hardware equivalent of the standard software multiply. This particular multiply algorithm generates the product through a series of adds and shifts. For the hardware case, all shifts are hardwired in. Thus the implementation reduces to an array of adders. For an n-bit multiplier and an m-bit multiplicand, n m-bit adders are required. Each bit of the multiplier controls an adder. If the bit=1, the multiplicand is added to the partial product at that level. If the bit=0, no addition takes place at that level. It is also assumed that the two numbers being multiplied in this simple case are positive. If 2's complement numbers are used, additional logic is usually added to handle negative values [2].

THE ADDER ARRAY

There are basically four array configurations that can be implemented in MOS LSI [3]. They are shown in figure 2.1. For an adder array, it does not matter in what order the sums and carries for a weighted position are assimilated. Thus the direction in which they propagate can be chosen. To avoid the wire routing problems of tree

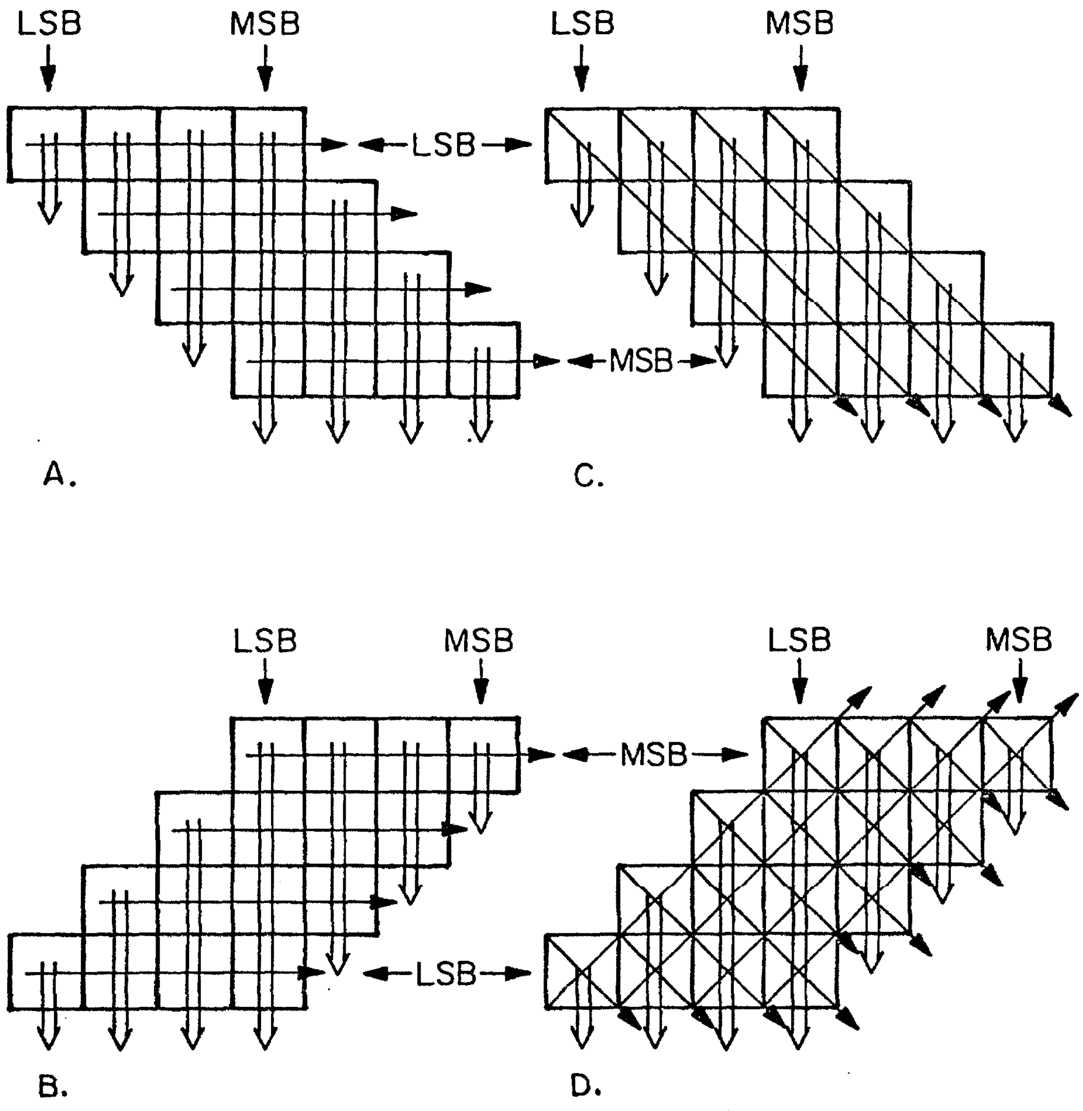


Figure 2.1 Adder array configurations.

structures, we are limited to passing carries and sums to adjacent cells. The sums are thus restricted to run either from LSB levels to MSB levels (figures A and C) or from MSB levels to LSB levels (figures B and D). The carries must flow from LSB's towards MSB's. However, they can be transmitted horizontally to cells in the same level (figures A and B), or diagonally to cells in an adjacent level (figures C and D). Horizontal carry transmission is used by full adders and diagonal carry transmission is used by carry-save adders.

On the basis of timing considerations, configuration B is preferred over configuration A. In A, a given cell must wait for both carry and sum delays before it can function. In B, the carries and sums propagate in a parallel fashion. Also final assimilation of the carries and sums in configuration A may require a cumbersome wiring topology. For similar reasons, configuration C is preferred over configuration D.

If only magnitude additions are involved, both configurations B and C are equivalent in terms of performance. However, when full 2's complement additions are required, configuration B exhibits difficulties. Straightforward additions of negative 2's complement numbers is not possible with this arrangement. It would be necessary to perform magnitude additions, then calculate the sign separately [2]. Configuration C does not exhibit this problem. Since carries propagate diagonally, the MSB sign determination is deferred to later levels. When the final carries are assimilated, an accurate MSB sign is then determined. This configuration was used for the adder array.

TWO'S COMPLEMENT MULTIPLICATION

When multiplication is implemented with a combinatorial process, Wallace [4] has indicated three areas where the processing performance can be improved:

1) Accelerate the formation of the summands.
2) Accelerate the addition of the summands.
3) Reduce the number of summands.

When an adder array is used, the summand formation time is quite minimal relative to the other delay factors. If other techniques such as ROMs were used, this item might be of more concern. The summand addition time is determined by the actual adder circuit design and layout. This area will be discussed in a later section. Reducing the number of summands, however, warrants further discussion.

Standard binary multiplication involves repeated additions and one bit shifts. This operation is simplified if multiple bit shifts are allowed. If the ability to subtract as well as add is combined with multiple shifts, a very powerful method of multiplication results. This method is known as ternary multiplication [2]. It is so-named because examination of the multiplier bits may require any one of three decisions:

1) Add a multiplicand multiple and shift.
2) Subtract a multiplicand multiple and shift.
3) Shift without arithmetic.

In conventional software multiplication, the multiplier bits are examined one at a time. If the digit is a '0', nothing is done. If the digit is a '1', the multiplicand is added to the partial product. The multiplicand is shifted one bit relative to the partial product, then the next multiplier bit is examined. This process is continued until all the multiplier bits have been examined. If, however, multiple bit shifting is allowed and several multiplier bits can be examined at one time, this process can be speeded up.

An obvious advantage can be seen if the multiplier contains a string of zeros. With this technique, one merely shifts the multiplicand the appropriate number of bits and examines the next non-zero multiplier digit. This is clearly faster than deciding whether or not to add after each single bit shift. A less obvious advantage can be extended to a string of ones as well. For example, the decimal number 127 = 001111111 (binary 2's complement, LSB on the right). Straightforward multiplication would require seven additions and six shifts of the multiplicand, 13 separate operations. But note that:

$$\begin{aligned} 127 &= 128 - 1 \text{ (decimal notation), or} \\ 001111111 &= 2^7 - 2^0 \text{ (binary notation)} \\ &= 010000000 - 000000001 \\ &= 0\overset{+}{1}000000\overset{-}{1} \end{aligned}$$

Thus the process reduces to two addition/subtractions and one seven bit shift. The worst case occurs when the multiplier consists of a string of alternating ones and zeros. An example would be:

$$01010101 = 0\overset{+}{1}0\overset{+}{1}0\overset{+}{1}0\overset{+}{1}$$

For this case, the process is the same as straightforward multiplication. In general, however, a multiplier word will consist of alternating strings of ones or zeros. It would be interpreted as the following example shows:

$$0100111010000111 = 0\overset{+}{1}0\overset{+}{1}00\overset{-}{1}0\overset{+}{1}000\overset{+}{1}00\overset{-}{1}$$

A direct implementation of this algorithm into hardware would be extremely complex. Variable shifts are not easily handled. If, however, the number of bits to be shifted is made constant, the shifting logic can easily be fixed in the hardware. For the present implementation, a constant shift of two bits occurs between examinations of multiplier bit sets. Each examination requires looking at the present two multiplier bits (Y_i & Y_{i+1}) and the previous bit (Y_{i-1}). Using the algorithm just discussed, it is possible to derive the decoding logic required. If $Y = Y_{i+1}Y_iY_{i-1}$ = 000 or 111, then nothing is done. In this case it is assumed that this set is part of a string of zeros or string of ones. The other cases are:

$Y = 001 = 0\overset{+}{1}0$: at MSB end of a string; add multiplicand.

$Y = 010 = 0\overset{+}{1}0$: single digit string; add multiplicand.

$Y = 011 = \overset{+}{1}00$: at MSB end of a string; add 2x multiplicand.

$Y = 100 = \overset{-}{1}00$: at LSB end of a string; subtract 2 x multiplicand.

$Y = 101 = \overset{-+}{11}0 = 0\bar{1}0$: at LSB end of one string and at MSB end of another string; subtract multiplicand.

$Y = 110 = 0\bar{1}0$: at LSB end of a string; subtract multiplicand.

These results are summarized in table 2.1. This special case of ternary multiplication is often referred to as the modified Booth's Algorithm. A formal proof is given by Rubenfield [5].

This technique can, of course, be applied to constant shifts of three bits or more. However, in such cases, one must have available

Y_{i+1}	Y_i	Y_{i-1}	Operation
0	0	0	Add Zero
0	0	1	Add 1x multiplicand
0	1	0	Add 1x multiplicand
0	1	1	Add 2x multiplicand
1	0	0	Subtract 2x multiplicand
1	0	1	Subtract 1x multiplicand
1	1	0	Subtract 1x multiplicand
1	1	1	Subtract zero

Table 2.1 The Modified Booth's Algorithm

pre-calculated numbers representing 3x the multiplicand, etc. In the two bit shift case, 1x and 2x the multiplicand can be obtained simply by shifting or not shifting the multiplicand. In a three bit shift case, three bits of the multiplier plus the previous bit are examined. A case such as $Y = 0110 = \overset{+}{1}0\overset{-}{1}0 = 0\overset{+}{1}\overset{+}{1}0$ would require 3x the multiplicand to be added to the partial product. At best this value would be pre-calculated and stored external to the array. This would require extra hardware. If the multiplier word has a large number of bits, this initial overhead might result in a net saving in hardware. However, each adder in the array still would require input lines representing 3x the multiplicand in addition to the multiplicand and partial product. These extra interconnect lines, when coupled with a more complex decoding logic, adds significantly to the area occupied by each cell in the array. Constant shifts of more bits increases this complexity. In the case of a 16 bit multiplier, a fixed shift of two bits is a reasonable compromise for an MOS implementation. Discrete hardware implementations appear to support this conclusion as evidenced by the use of the modified Booth's algorithm in IBM's floating point processor and in the Am 2505 2 by 4 multiplier chip. [5], [6].

THE 16 BIT BY 16 BIT MULTIPLIER

The chip described in this thesis was designed to multiply two 16 bit words and optionally to accumulate successive products. The result is a 32 bit word plus an overflow flag. As shown in figure 3.1, the multiplier is composed of four functional sections. The timing diagrams are illustrated in figure 3.2. Figure 3.2a shows a synchronous timing case. Figure 3.2b shows the asynchronous timing case. The timing for the synchronous case will be discussed first.

A multiplication cycle begins with the latching of the data input registers. These registers are shown in figure 3.3. The 16 bit word representing the multiplicand is stored in the X-register as shown in figure 3.3a. Data are loaded into this register while the X-IN control is high. This register holds the data and amplifies them for driving internal data lines. The Y-register holds the 16 bit multiplier word. This register is shown in figure 3.3b. The Y-register section also performs a preliminary decoding of the multiplier bits. It is here that the modified Booth's Algorithm is applied to reduce the number of adders required in the combinatorial array. Special drivers place these signals onto the internal control lines.

A CLK/Precharge period then initializes the combinatorial array. As shown in figure 3.4, the combinatorial array consists of an 8 cell by 17 cell matrix. Each cell in the array is composed of a shifter/

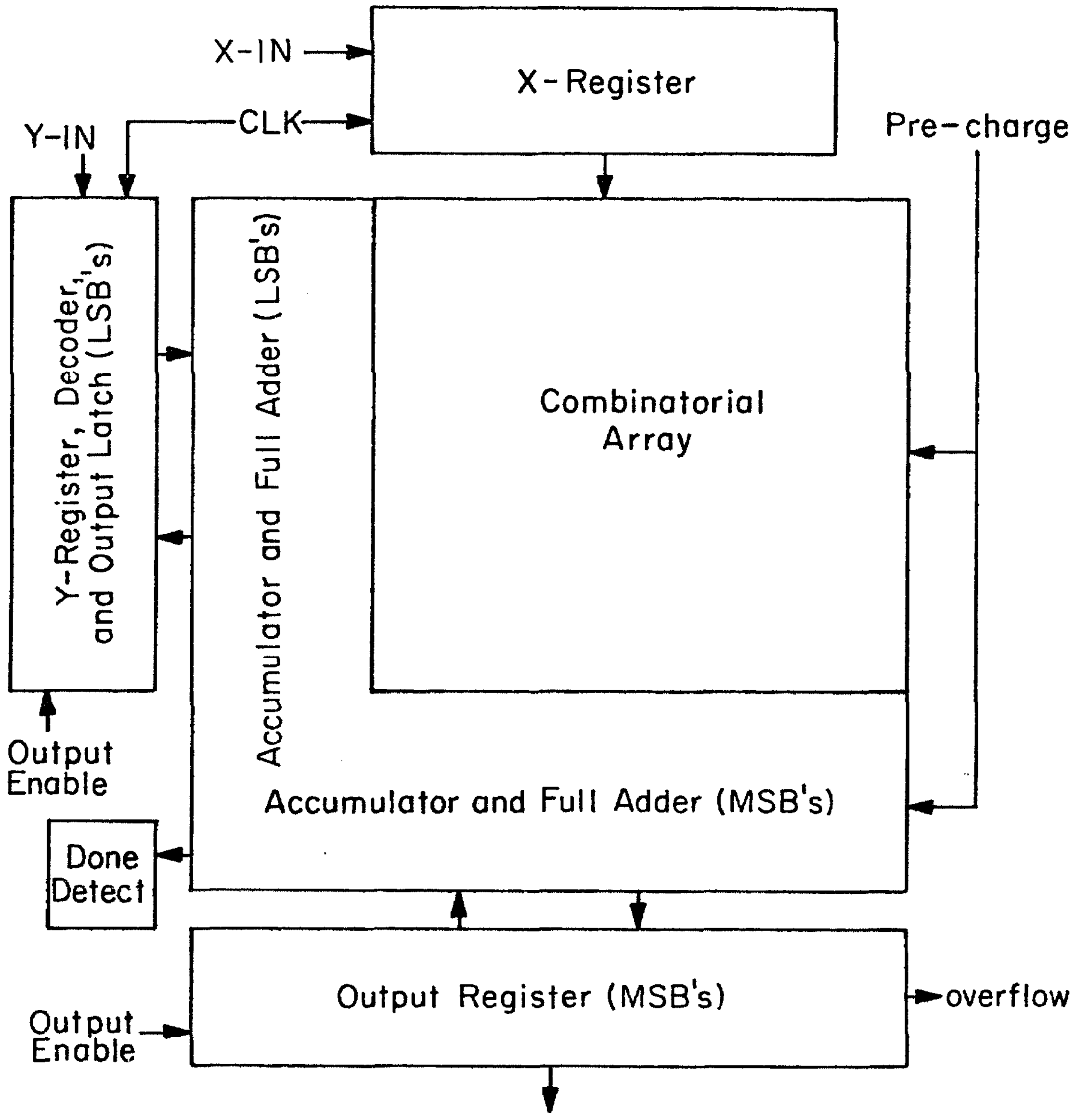
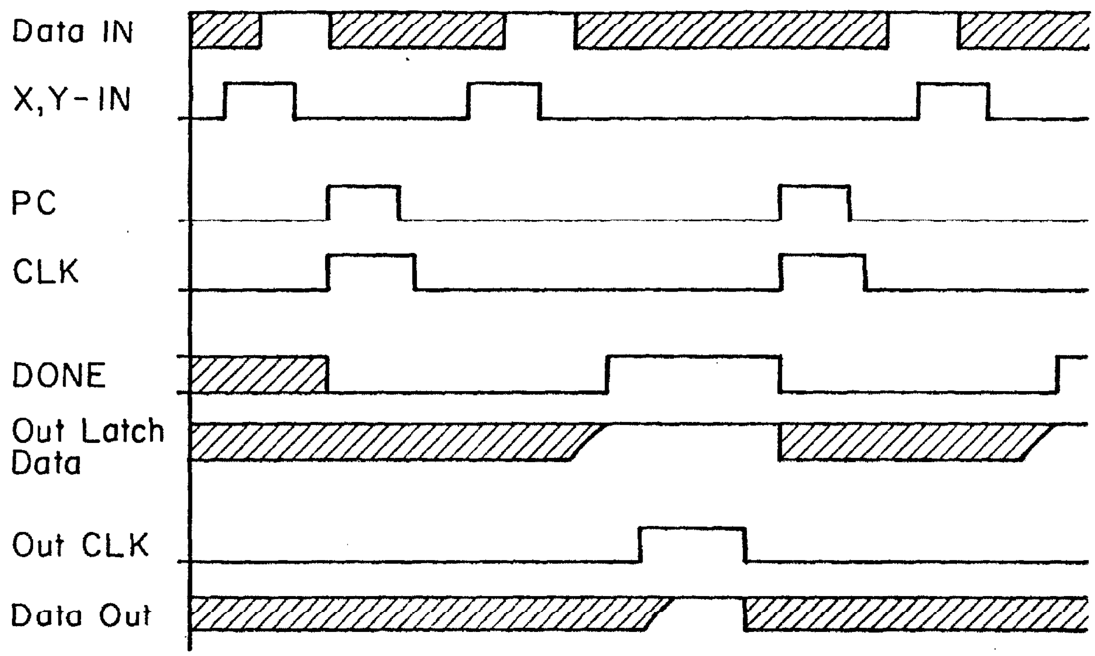
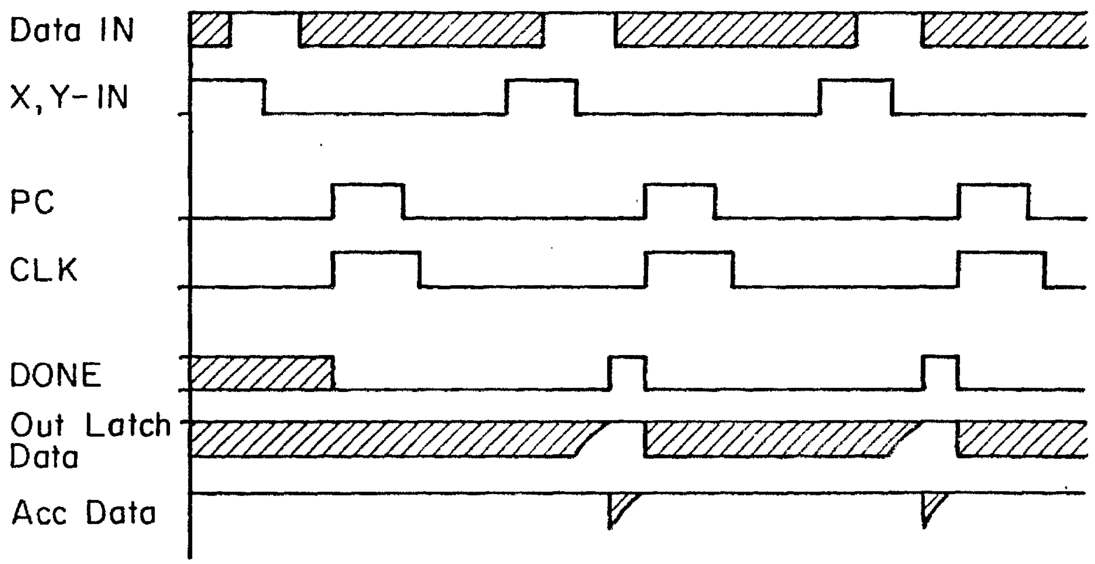


Figure 3.1 Block diagram of the 16 x 16 multiplier.

inverter and a carry-save adder. The shifter/inverter completes the Booth's algorithm operation that was begun in the Y-register decoder. When CLK and Precharge go high, they activate the decoder controls that enter the shifter/inverter. These controls operate on the incoming

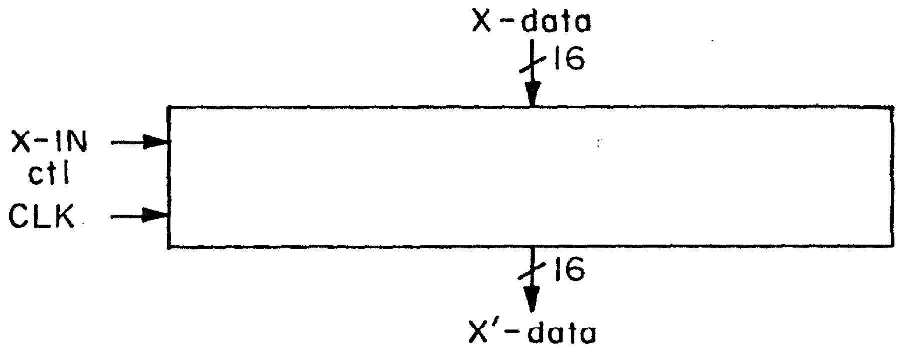


a. Synchronous timing

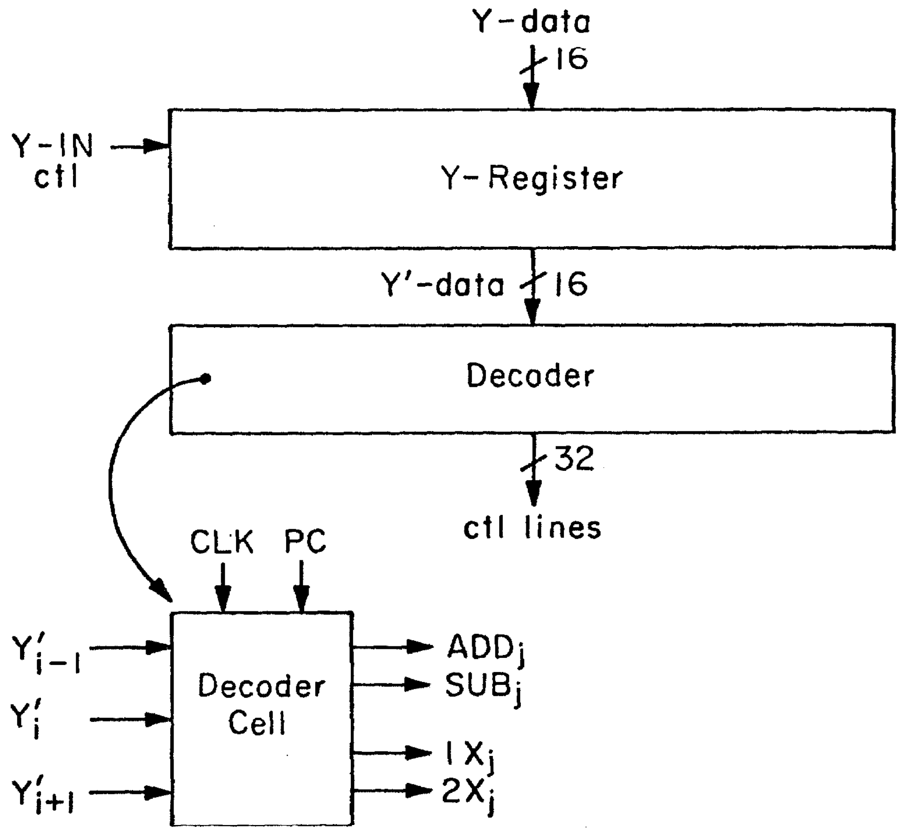


b. Asynchronous timing

Figure 3.2 Multiplier timing.



a. X-register and driver



b. Y-register and decoder

Figure 3.3 Input registers.

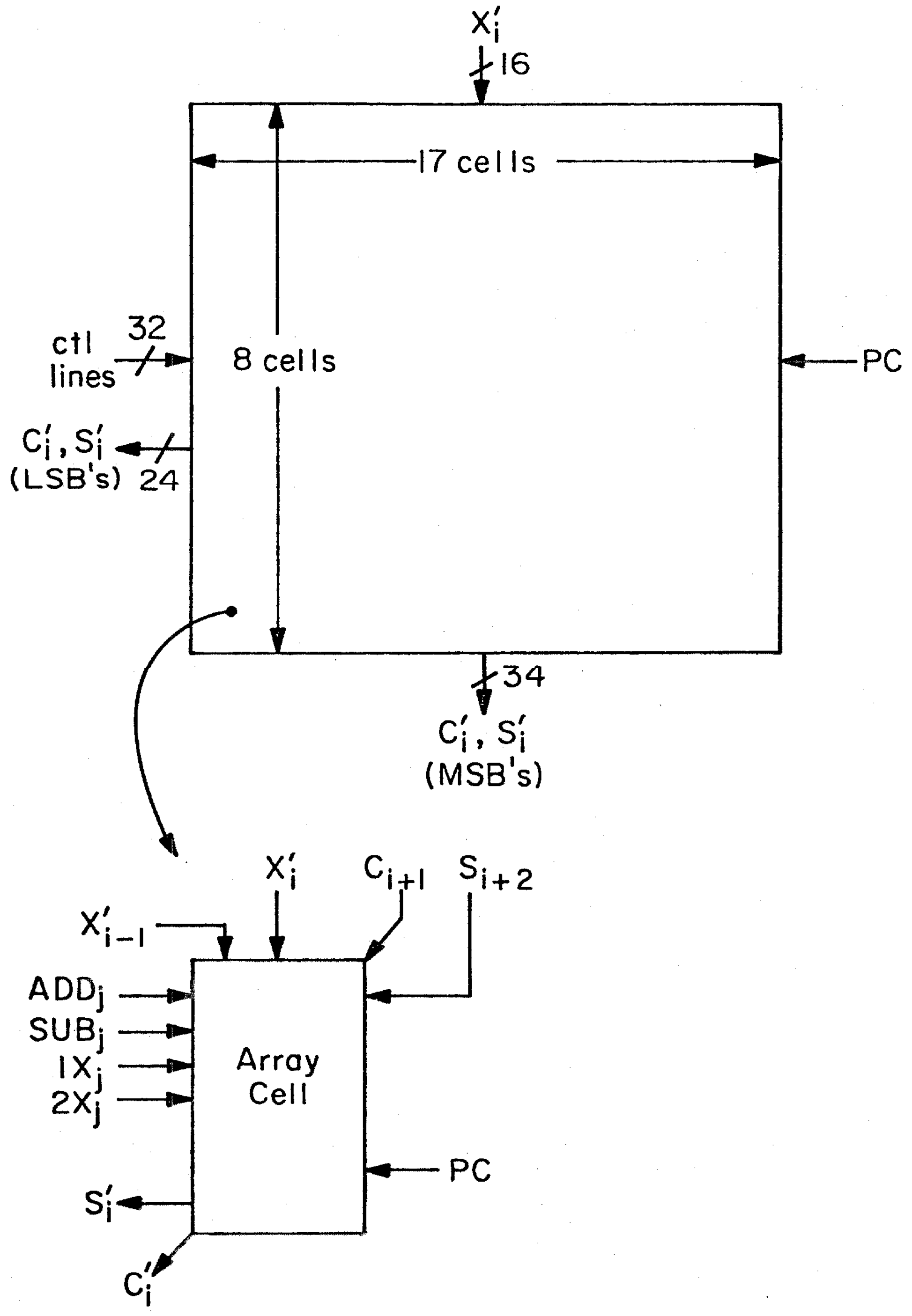


Figure 3.4 The combinatorial array.

multiplicand and send a modified value to a fast carry-save adder section. While Precharge is high, the carry-save adder is reset to an initial state. Precharge is then set low. The CLK control soon follows suit. This begins an asynchronous multiply period. At each adder level, the modified multiplicand is added to an incoming partial product. A new partial product is generated and output to the next adder level. It should be noted that once Precharge is set low, the internal array is isolated from the input registers. This implies that new input data can be loaded into the input registers while the multiply process i~~n~~s continuing.

The final product, still in carry-save form, is then passed to the accumulator and full adder section which is shown in figure 3.5. Both the accumulator and the full adder are initialized while Precharge is high. When Precharge is set low, this section waits for the final product from the combinatorial array. When the final product information arrives, the accumulator adds it to the previous result. The full adder then converts the accumulated sum into a 2's complement form for output. Due to the unique design of this array, it is possible to detect the completion of the multiply/accumulate operation. This completion condition is detected at the bit level and is used to latch the associated output bit into its output register cell. By NORing the completion indicators together for all the output bits, a 'DONE' signal for external use is also generated. As shown in the timing diagrams of figure 3.2, 'DONE' is set high when the data in the output register are valid.

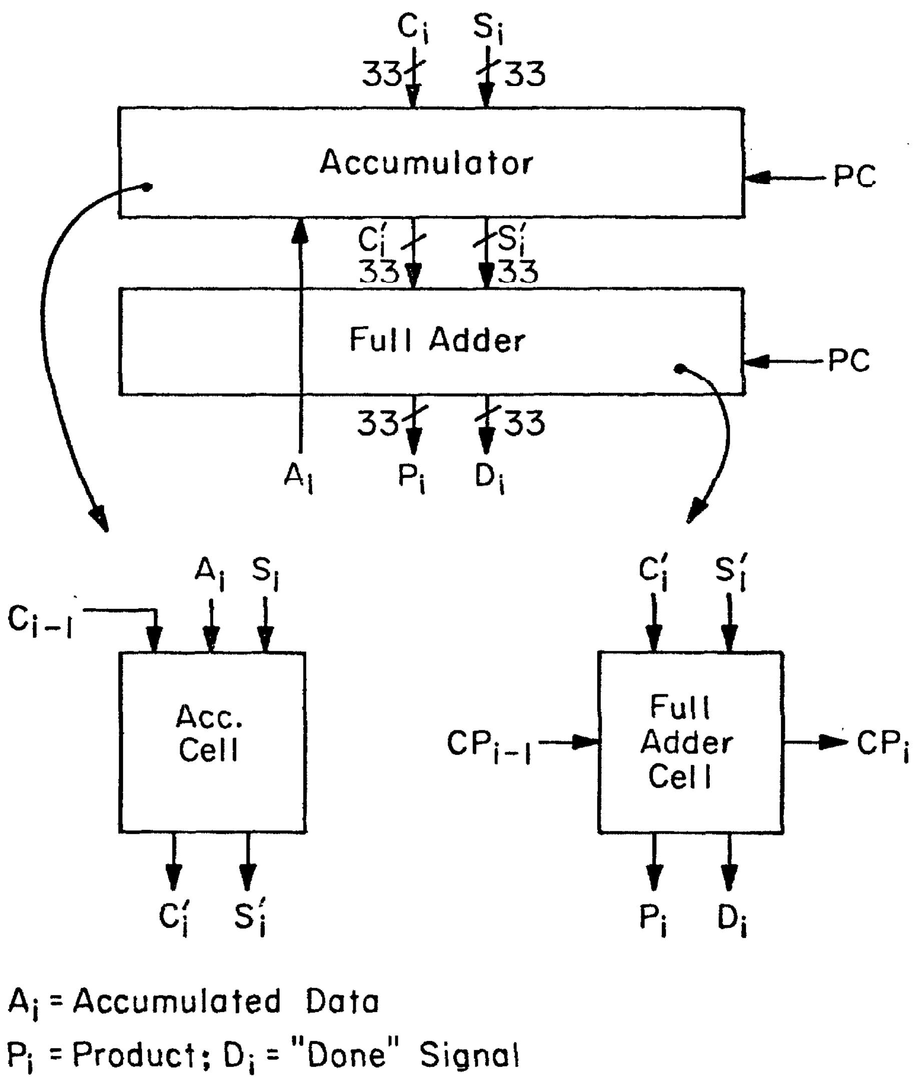


Figure 3.5 Accumulator and full adder.

The final section contains the output register and drivers, which are shown in figure 3.6. The output register stores the final result both for output at a later time and for use on the next multiply/accumulate cycle. When the final result is to be sent off-chip, the OUTCLK

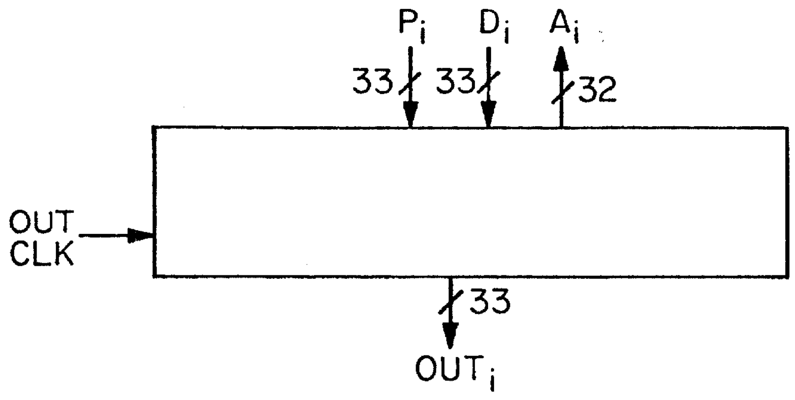


Figure 3.6 Output register and driver.

control is set high. This control causes the tri-state output drivers to deposit the output data onto external data lines. When OUTCLK is set low, the output drivers return to a high impedance, inactive state. An overflow flag is set whenever an overflow out of the accumulator occurs.

The timing for the asynchronous case of figure 3.2b is similar to that of the synchronous case except for the following condition. If new data has been loaded into the input registers by the time the 'DONE' signal is set high, a new multiply cycle can be initiated immediately. Thus the 'DONE' signal can be used to activate the CLK and Precharge signals, which would asynchronously start a new multiply cycle.

IV. DESIGN DESCRIPTION

X-REGISTER

The X-register holds the 16 bit multiplicand word. Information is input while the X-IN control is high. A NOR gate equivalent circuit is shown in figure 4.1. Also included is a buffer which drives the internal data lines. The MOS circuit for this register is shown in figure 4.2. During the X-IN high time, input data are loaded into the register. This information must be stable before X-IN goes low. While the X-In control is low, the register outputs are connected back to their inputs. Data may be held in this state indefinitely.

Since the multiplicand data lines running into the combinatorial array are quite long, a special push-pull driver is included in this register. It is driven by the CLK signal. Between multiplies, both CLK and X are low. When new data are input, a condition is set up that

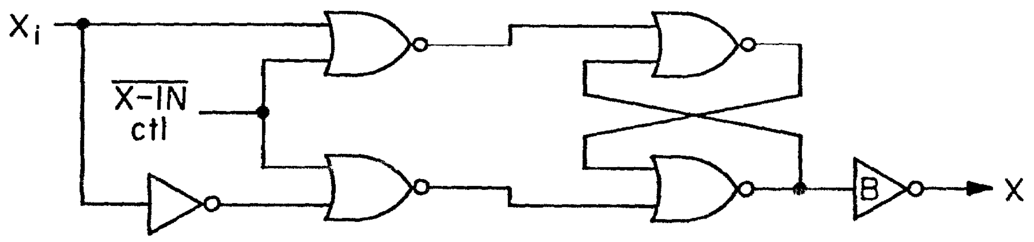


Figure 4.1 X-register NOR gate equivalent circuit.

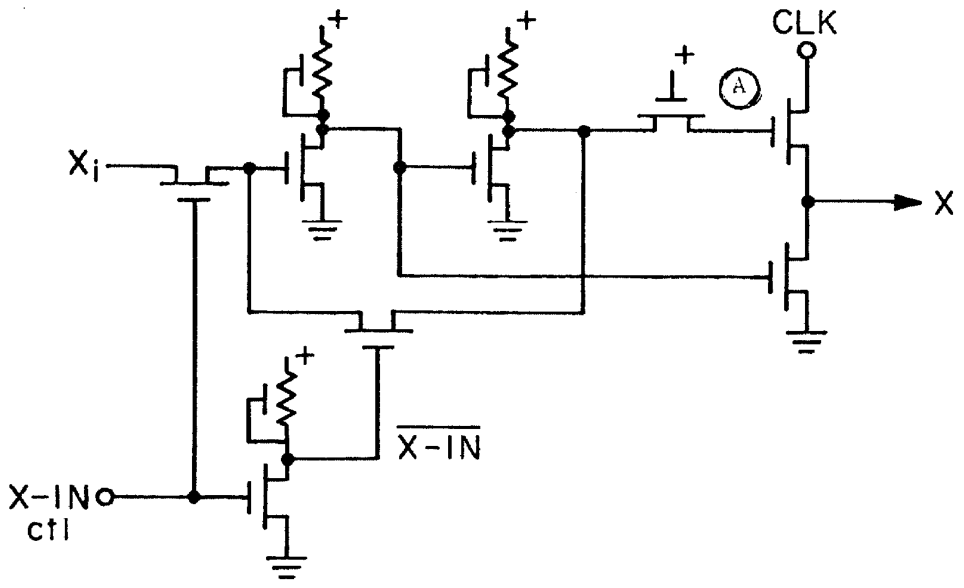


Figure 4.2 X-register MOS circuit.

either keeps X low or charges node 'A' to a high state. When the multiply cycle is initiated, CLK goes high. If node 'A' is low, the associated pull up transistor is off, and X remains low. If node 'A' is high, the pull up is 'on' and X rises as CLK goes high. A bootstrap effect causes node 'A' to become isolated so that sufficient drive can be provided to allow X to rise to the CLK high level. Since the driver is push-pull, the pull up can be a high current device. This allows the X lines to be set with minimal time delays.

THE Y-REGISTER and DECODER

The Y-register performs the input latching of the 16 bit multiplier word. In addition, through multiple bit examination, preliminary decoding of the multiplier word is performed. Special drivers place the decoded signals onto the internal control lines.

The Y-register section is similar to the X-register. Data are input during the Y-IN control high time. They are held in a static state during the Y-IN low time. The NOR gate equivalent circuit is shown in figure 4.3 and the MOS circuit is shown in figure 4.4.

The decoder implements part of the modified Booth's Algorithm that was discussed earlier. Each cell examines three bits of the multiplier (with one bit overlap between cells) and generates commands based on the truth table of table 4.1. The logic equations are as

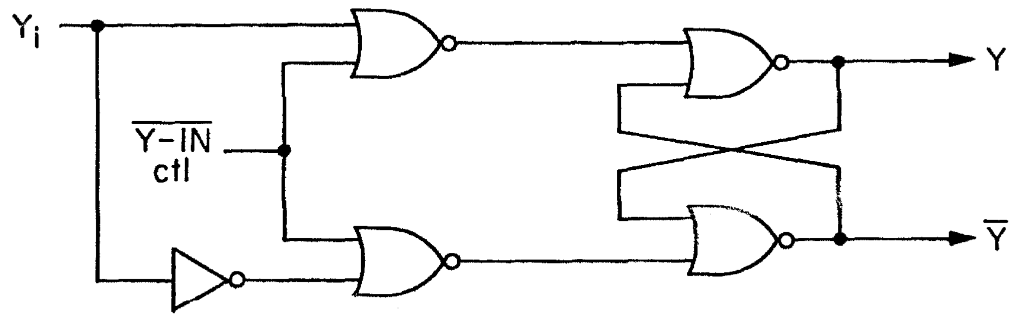


Figure 4.3 Y-register NOR gate equivalent circuit.

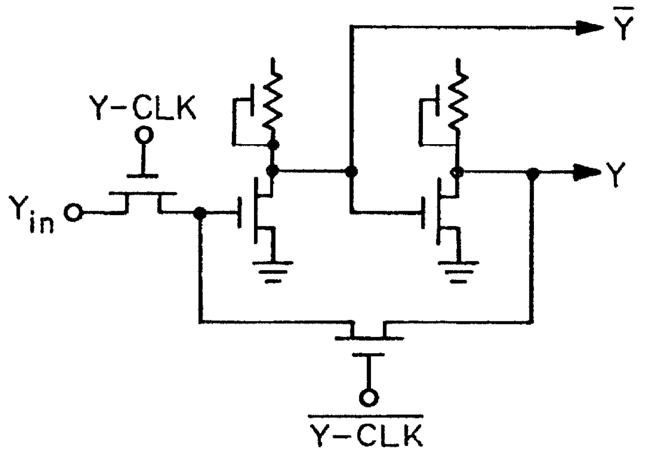


Figure 4.4 Y-register MOS circuit.

Y_{i+1}	Y_i	Y_{i-1}	ADD	1x	2x	
0	0	0	1	0	0	Add Zero
0	0	1	1	1	0	Add 1x
0	1	0	1	1	0	Add 1x
0	1	1	1	0	1	Add 2x
1	0	0	0	0	1	Subtract 2x
1	0	1	0	1	0	Subtract 1x
1	1	0	0	1	0	Subtract 1x
1	1	1	0	0	0	Subtract 0

Table 4.1 Decoder Truth Table

follows:

$\text{ADD} = \overline{Y}_{i+1} = \text{add}$

$\text{SUB} = Y_{i+1} = \text{subtract}$

$1x = Y_i \oplus Y_{i-1}$ = add or subtract the multiplicand

$2x = \overline{(Y_i \oplus Y_{i-1})}\ (Y_{i+1} \oplus Y_i)$ = add or subtract twice the multiplicand

The logic circuit for the decoder is shown in figure 4.5, and the NOR gate equivalent is in figure 4.6. The logic equations were carefully selected to allow implementation with a minimum number of devices. The inverters with a 'B' in them are special drivers that are similar to those in the X-register. They are necessary because of the heavy drive

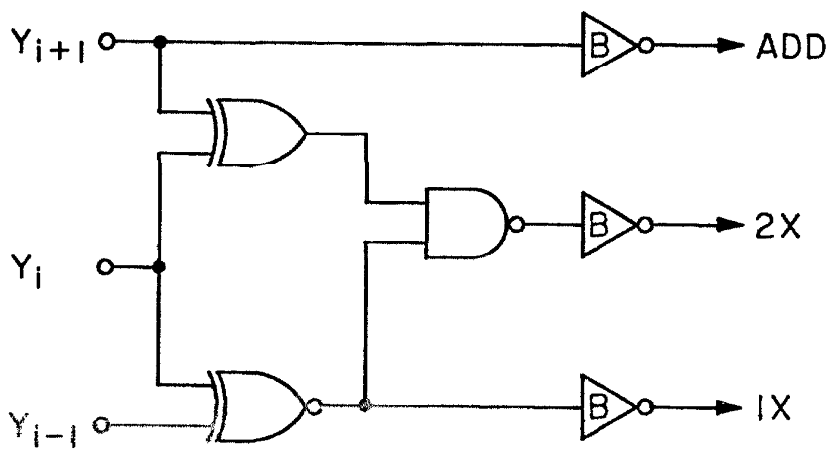


Figure 4.5 Decoder logic circuit.

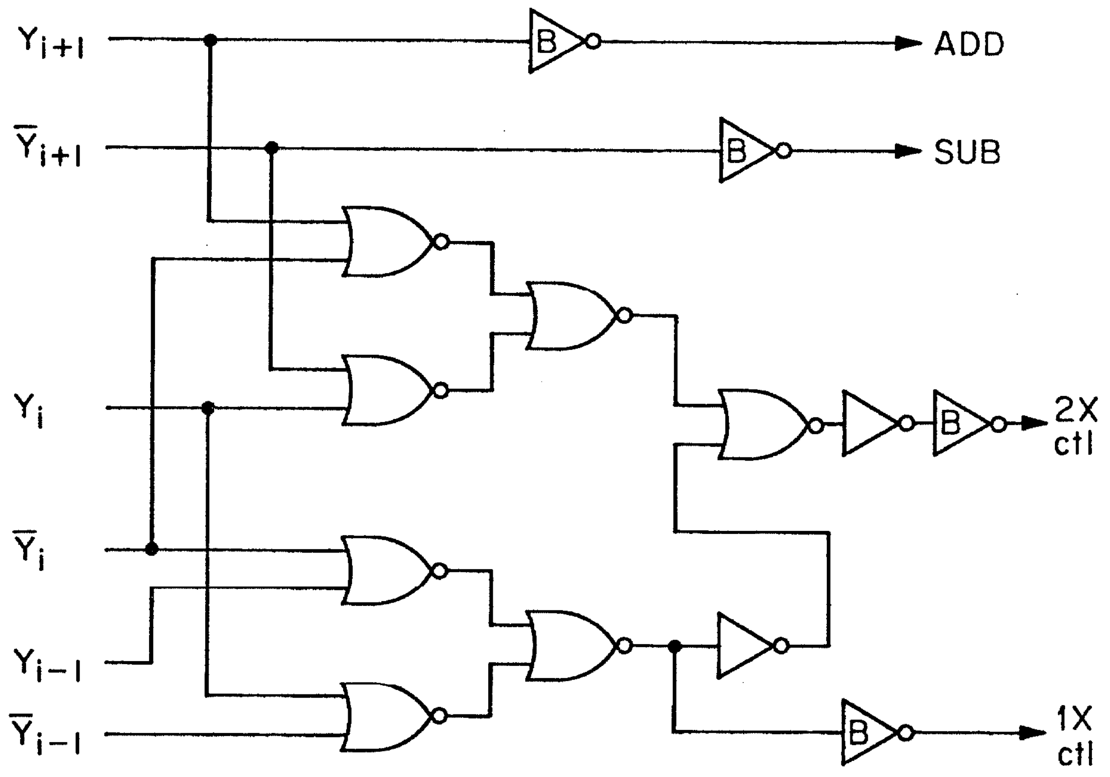
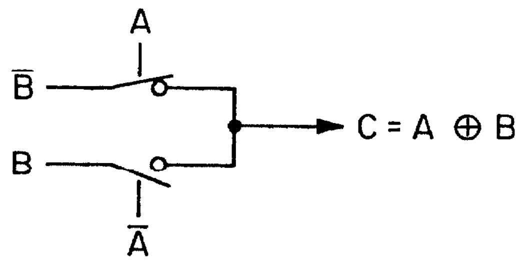


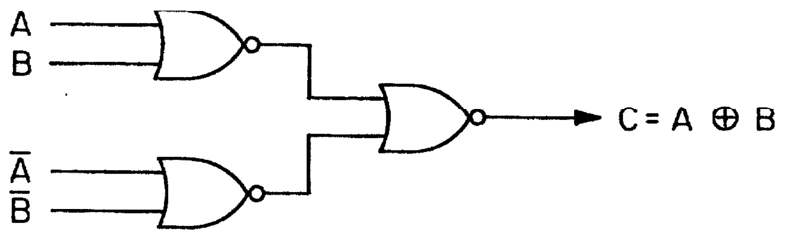
Figure 4.6 Decoder NOR gate equivalent circuit.

requirements of the control lines. The MOS circuit of the Y-register as was shown in figure 4.4 reveals that both data and data bar are available. This means that a very compact implementation of the decoder circuit can be achieved by making use of switching circuits.

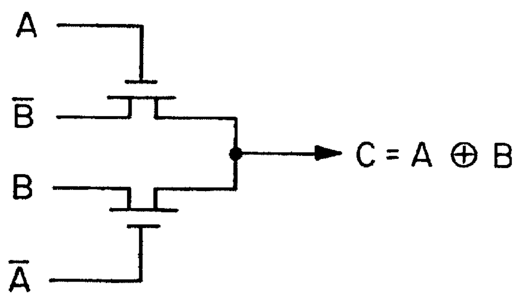
Switching circuits represent one of our oldest forms of logic. They originated in the days of relay logic. As shown in figure 4.7a, two wires and two switches can perform the equivalent of an exclusive OR or an exclusive NOR function. A NOR gate equivalent circuit is shown in figure 4.7b. If both A and B are active, or if both are not active,



a. Switching circuit



b. NOR gate equivalent circuit



c. MOS circuit

Figure 4.7 Exclusive-OR circuits.

then C is not active. Since switches are easy to implement with MOS, an equivalent circuit shown in figure 4.7c can perform the XOR or XNOR functions. When applied to the decoder circuit, the schematic of figure 4.8 is arrived at.

The MOS circuit for the control line drivers is shown in figure 4.9. They are the same as the X-register drivers. Their operation is the same as described in the X-register section.

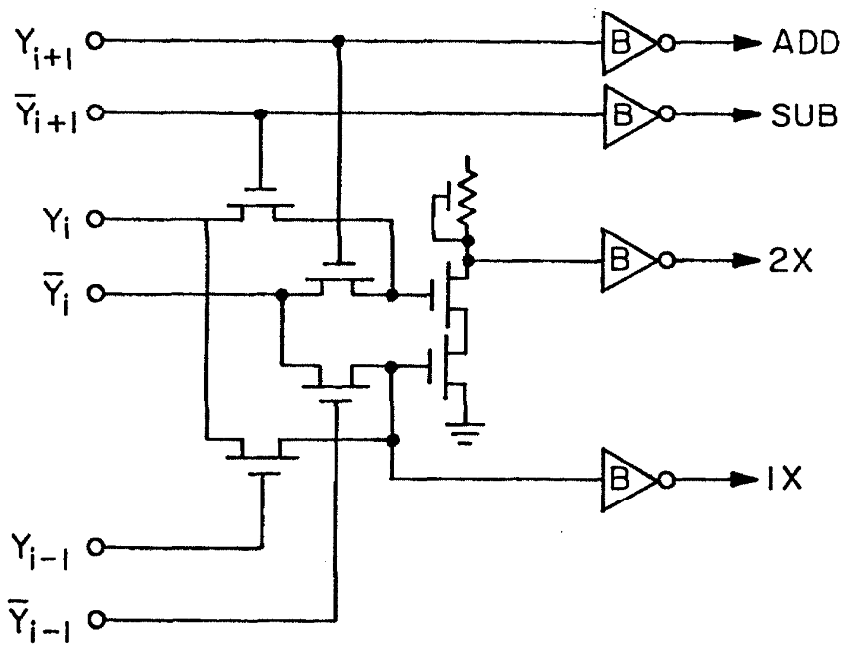


Figure 4.8 Decoder MOS circuit.

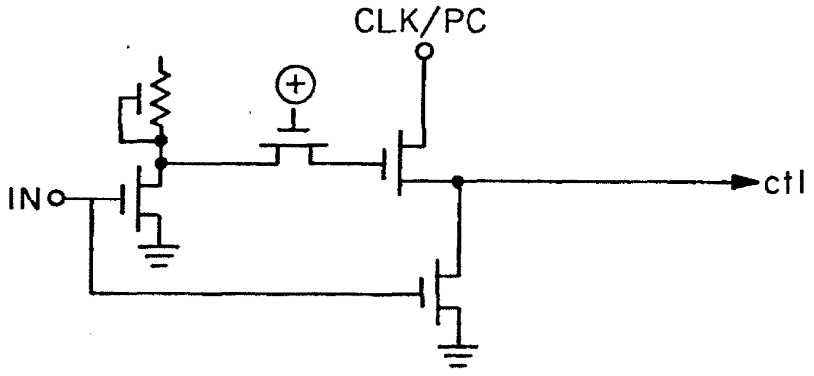


Figure 4.9 Decoder drivers.

THE COMBINATORIAL ARRAY

The combinatorial cell consists of two basic sections. Besides the carry-save adder section, this cell also includes a section to shift or invert the incoming multiplicand bit.

The Shifting Array and Complementer.

The shifter/inverter section must interpret the controls from the decoder as shown in table 4.2. The logic circuit is given in figure 4.10 and a NOR gate equivalent is shown in figure 4.11. For 2's complement operation, the LSB of each adder row is attached to Y_{i+1}. Thus $C_{in} = Y_{i+1}$ = INVERT. Whenever a subtraction is required, $C_{in} = Y_{i+1} = 1$ and ADD = 0. As a result, 1x or 2x the multiplicand is inverted and "1" is added to the LSB position. This operation is the equivalent of a

ADD	2x	1x	x
0	0	0	1 Subtract zero
0	0	1	$\overline{X}_n$
0	1	0	$\overline{X}_{n-1}$
0	1	1	not allowed
1	0	0	0 add zero
1	0	1	X_n
1	1	0	X_{n-1}
1	1	1	not allowed

Table 4.2 Shifter/inverter truth table

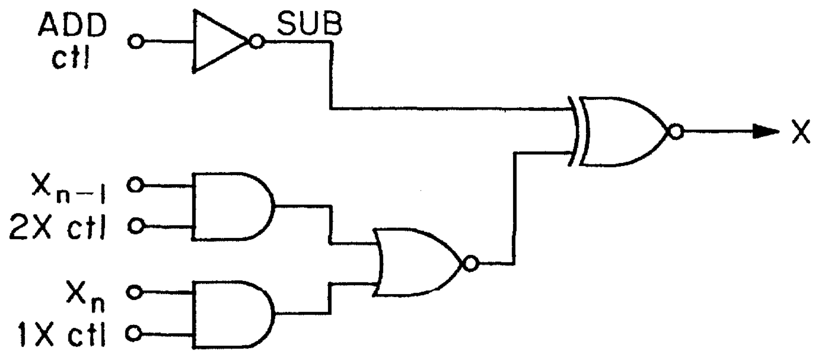


Figure 4.10 Shifter/inverter logic circuit.

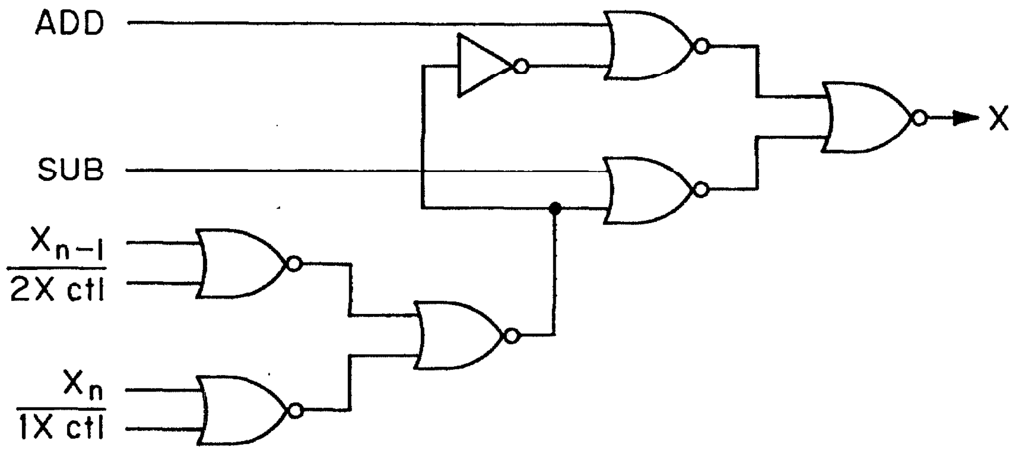


Figure 4.11 Shifter/inverter NOR gate equivalent circuit.

two's complement subtraction. When 1x = 2 x = 0, zero is to be added or subtracted. If ADD = 1, X = 0 as required. If SUB = 1, X = 1. But C_{in} will also equal "1". Thus the carry-in, when added to X will zero the effective input. As a side effect, it will also cause an overflow of the MSB. However, this implementation ignores such overflows.

Since this circuit also uses an XNOR, the switching circuit previously discussed can be used. Also, the decoder implementation assures that two of the eight states of table 4.2 will never occur. Making use of all this information results in the MOS circuit shown in figure 4.12. The two AND gates and the NOR gate shown in figure 4.10 are combined into a more compact NAND-NOR circuit. The extra buffering to generate X and X bar are required because they are needed in the sum and carry circuits. The 1x and 2x lines are activated by the CLK signal. The ADD and SUB lines are activated by the PC signal. It is assumed that the decoder states are set before CLK and PC go high. When PC goes low, ADD and SUB also go low to isolate the cells from the input latches. Thus new X and Y data can then be loaded while the multiply cycle is finishing. Between multiplies, the 1x, 2x, ADD, and SUB lines are low. Thus the circuit dissipates minimal power during non-use.

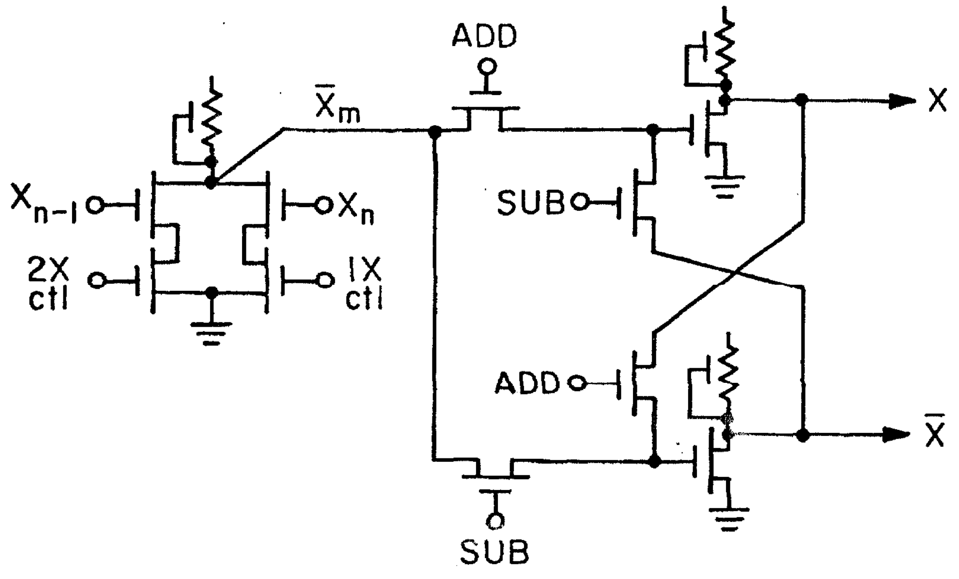


Figure 4.12 Shifter/inverter MOS circuit.

THE CARRY-SAVE ADDER

Carry-save adders are sometimes referred to as half-adders. The truth table for a carry-save adder, which is shown in Table 4.3, is identical to that for a full adder. The difference between the two is in the intercircuit connections. A full adder propagates carries through adjacent cells within a single adder level. A carry-save adder defers the carry propagation to the next adder level. Thus there is no carry assimilation delay time. Each cell of the adder operates independently of the others. Carries and sums are generated in parallel and are functions only of the three input bits. Of course a conservation of computation law still holds. These intermediate carry-save values must eventually be converted back to standard form with the use of a full adder.

Si	X	Ci	So	Co
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Table 4.3 Carry-Save Adder Truth Table

The logic circuit for a standard adder cell is presented in figure 4.13. The NOR gate equivalent circuit is given in figure 4.14. The equivalent MOS circuit is given in figure 4.15. Note that the carry-in to carry-out path involves two inverter delays. A similar situation exists for the sum-in to sum-out path. This type of circuit is representative of a ripple carry type of adder.

Ripple-carry adders are the simplest in terms of functional operation. An example is shown in figure 4.16a. Each cell or bit position pre-processes the two addend input bits, then waits for the carry-in from the adjacent LSB cell. A carry-in signal initiates final sum generation processing for that cell. It also causes a carry-out to be generated for use by the next MSB cell. If we let t_c equal the single cell carry delay and let n equal the number of adder bits (cells), then the total addition

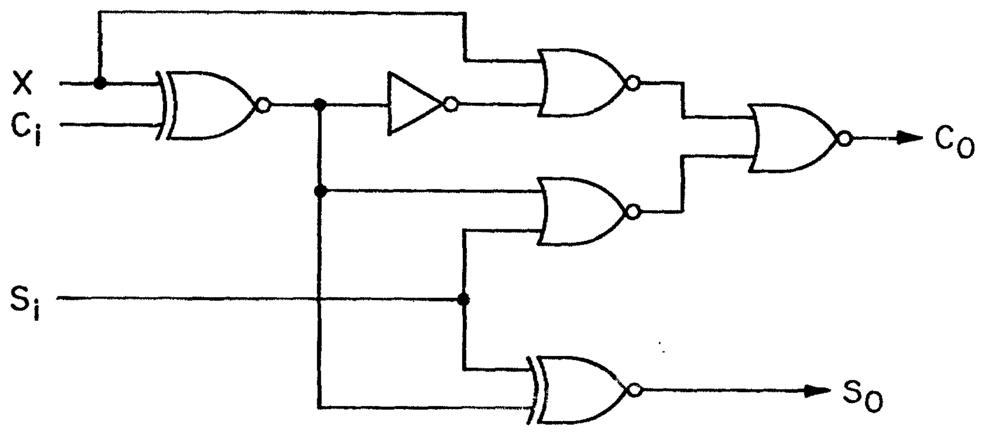


Figure 4.13 Carry-save adder logic circuit.

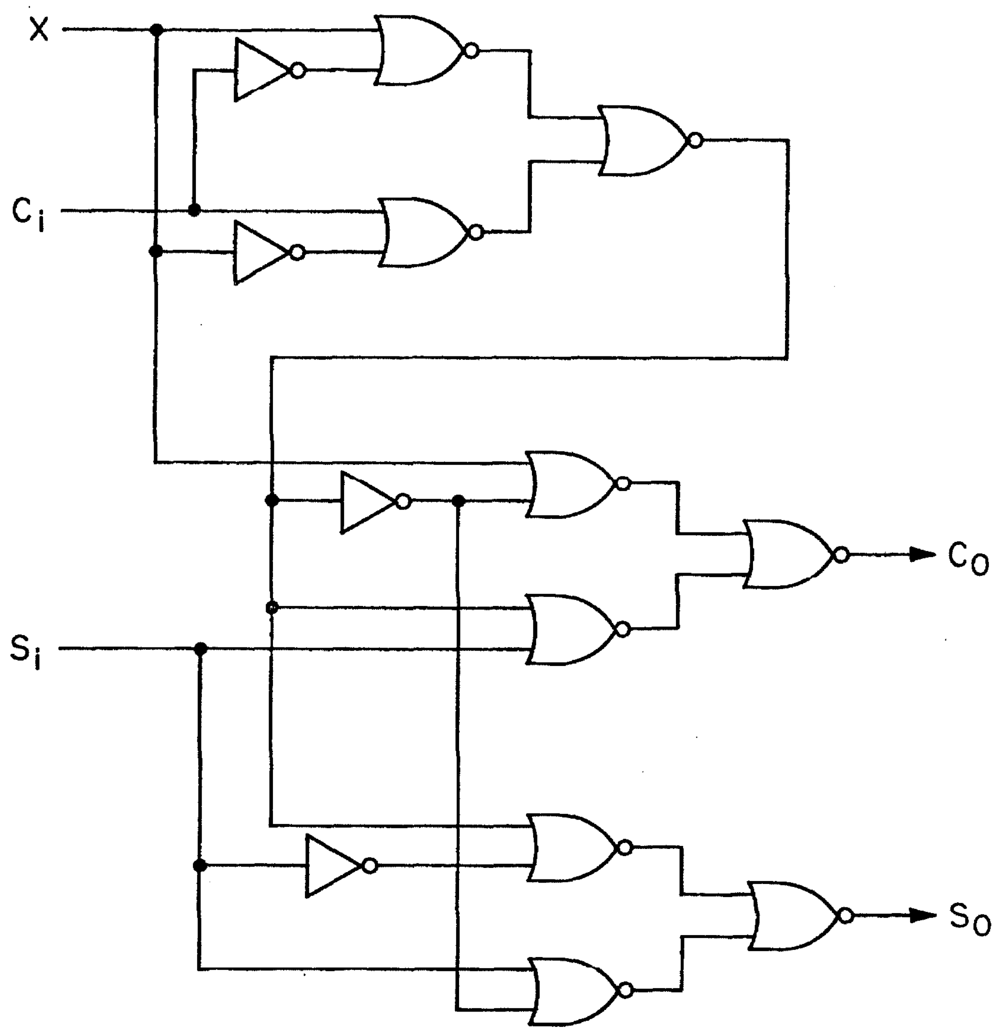


Figure 4.14 Carry-save adder NOR gate equivalent circuit.

time is: $t = n*t_C + t_s$, where t_s accounts for the pre and post processing delays. Thus for a ripple-carry adder, the time delay increases linearly with the number of bits.

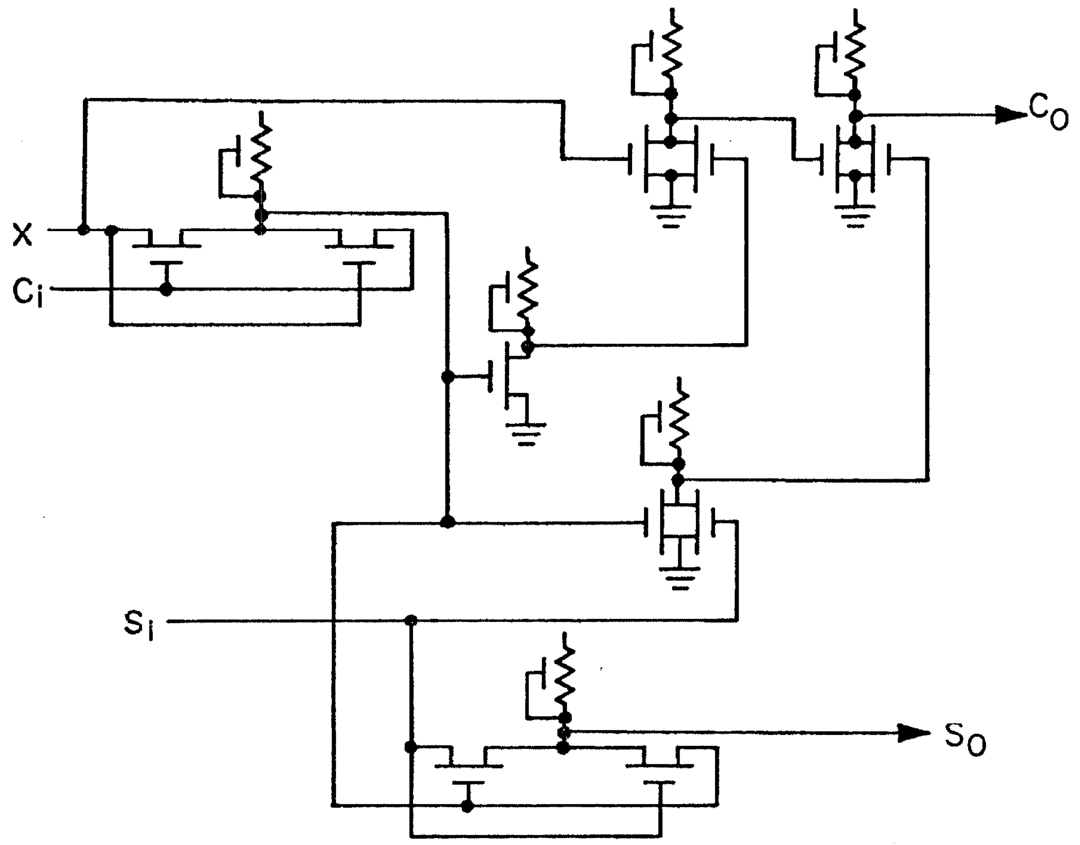
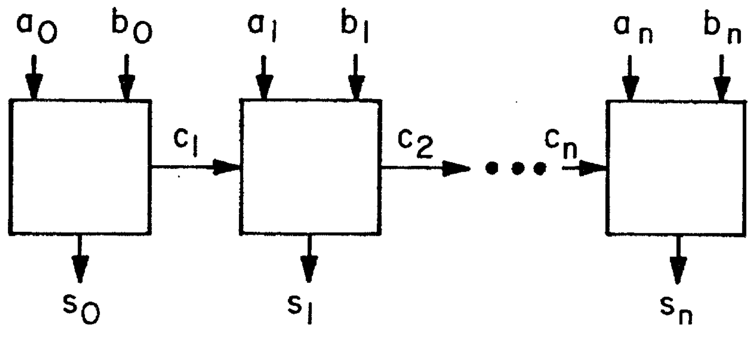
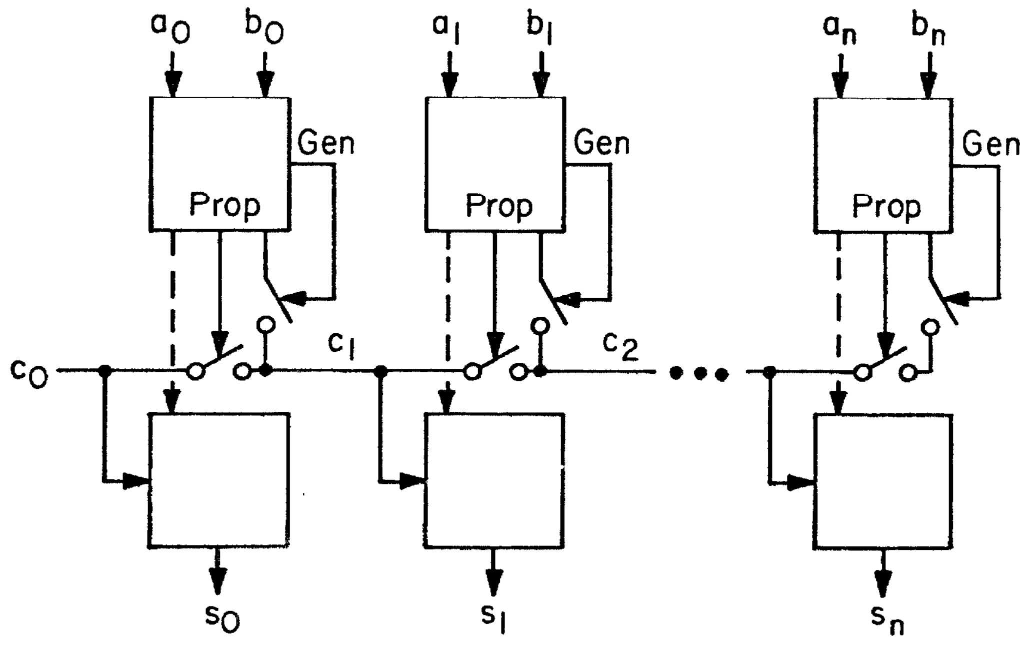


Figure 4.15 Carry-save adder direct MOS circuit implementation.

Manchester type adders encompass those that include carry generate logic along with carry propagation. An example is shown in figure 4.16b. In some literature they are refered to as switched ripple-carry adders. This description goes back to their original implementation in the old relay logic technology. In that technology, switches were opened or closed depending on whether the carry signals were being generated, killed, or propagated. In the propagation mode, carries essentially propagated through a series of closed switches. A very fast carry propagation time resulted.



a. Ripple carry adders



b. Manchester type adders

Figure 4.16 Adder types.

Thus, use of MOS devices as switches opens up other approaches to the adder circuit design. If switching logic is applied to adders, a modified truth table can be used (Table 4.4). For sum generation, the parity between X and C_{in} can be determined. If it is assumed that both S_{in} and S_{in} bar are available, this parity condition can be used to control a group of switches. Figure 4.17 shows the NOR gate equivalent circuit for such an implementation. Figure 4.18 shows the MOS circuit equivalent. Although the logic circuit appears rather complex, the actual implementation is quite simple. Note that the S_{in} to S_{out} path now involves only a one transmission gate delay.

The carry generation logic for a Manchester type adder is shown in Table 4.5. The parity between X and S_{in} can be used to switch between either a propagate state or a kill/generate state. If it is assumed that both C_{in} and C_{in} bar are available, a switching scheme similar to that for sum generation can be used. Figure 4.19 shows the NOR gate equivalent circuit and figure 4.20 presents the MOS circuit equivalent. As with sum generation, the C_{in} to C_{out} path involves only a one transmission gate delay.

X	C_i	S_o	$\overline{S}_o$
0	0	S_i	$\overline{S}_i$
0	1	$\overline{S}_i$	S_i
1	0	$\overline{S}_i$	S_i
1	1	S_i	$\overline{S}_i$

Table 4.4 Modified Sum Truth Table

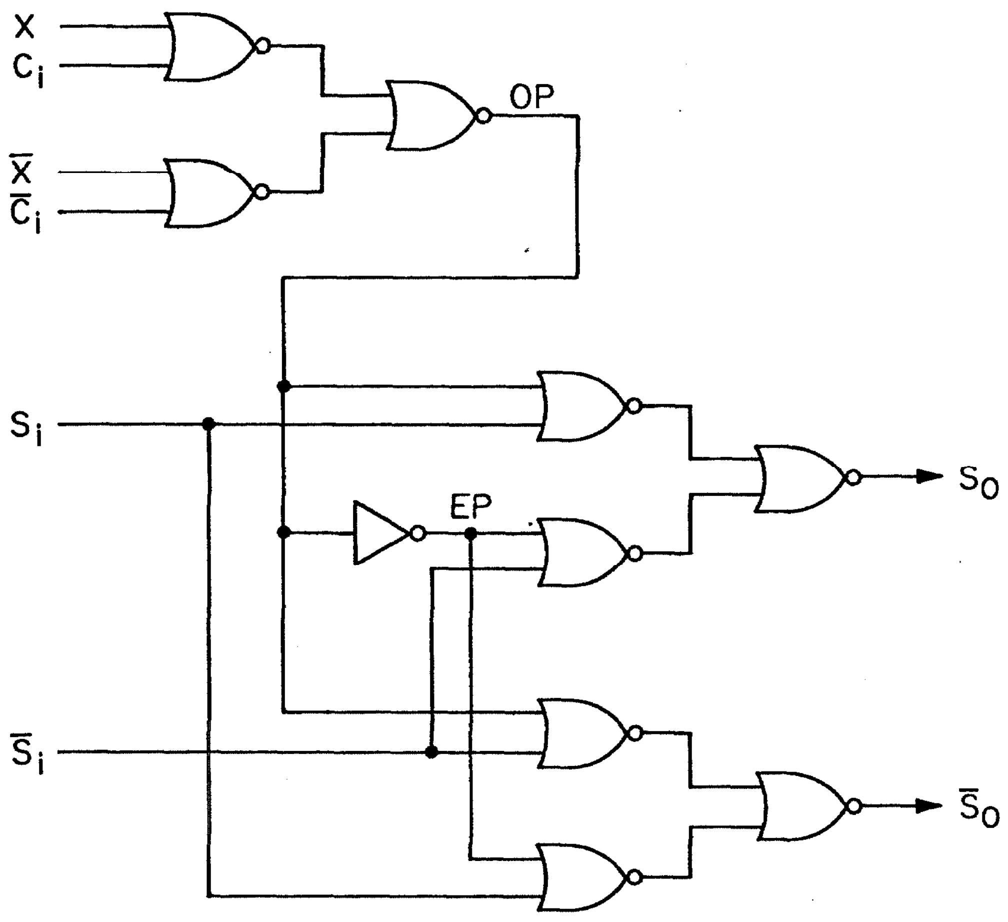


Figure 4.17 Modified SUM, NOR gate equivalent circuit.

For the specific case of a carry-save adder, both the carry generation delay and the sum generation delay have equal importance. Since both sum and carry propagate in parallel through an adder array, the time response for both paths should be equally optimized. There is nothing to be gained from optimizing either path at the expense of the other. Keeping this in mind, the combined logic circuit of figure 4.21

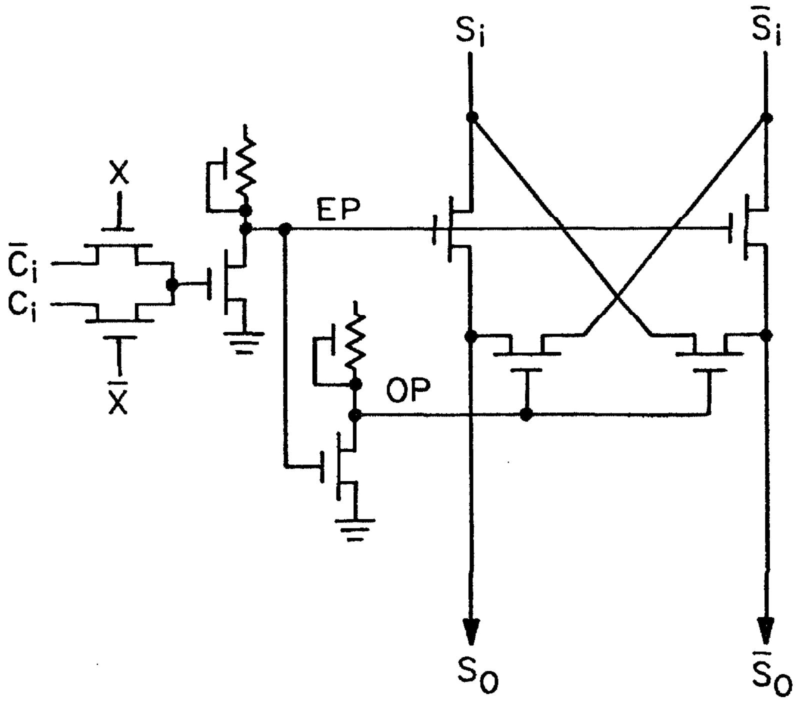


Figure 4.18 Modified SUM, MOS circuit.

S_i	X	P	K	G	C_o	$\bar{C}_o$
0	0	0	1	0	0	1
0	1	1	0	0	C_i	$\bar{C}_i$
1	0	1	0	0	C_i	$\bar{C}_i$
1	1	0	0	1	1	0

Table 4.5 Modified carry truth table

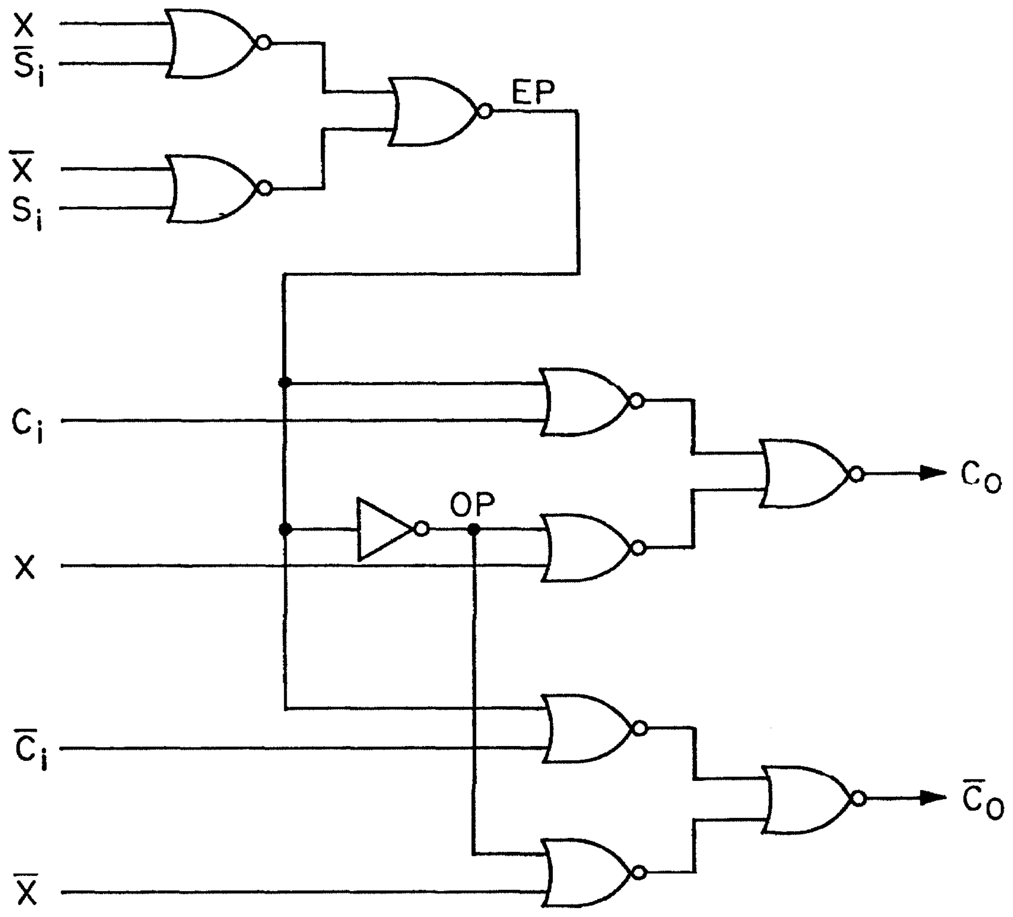


Figure 4.19 Modified CARRY, NOR gate equivalent circuit.

results. This circuit deviates somewhat from the NOR gate equivalent circuit of figure 4.22. At the front end, both an exclusive-OR and an exclusive-NOR of the X and C_{in} lines are taken in parallel

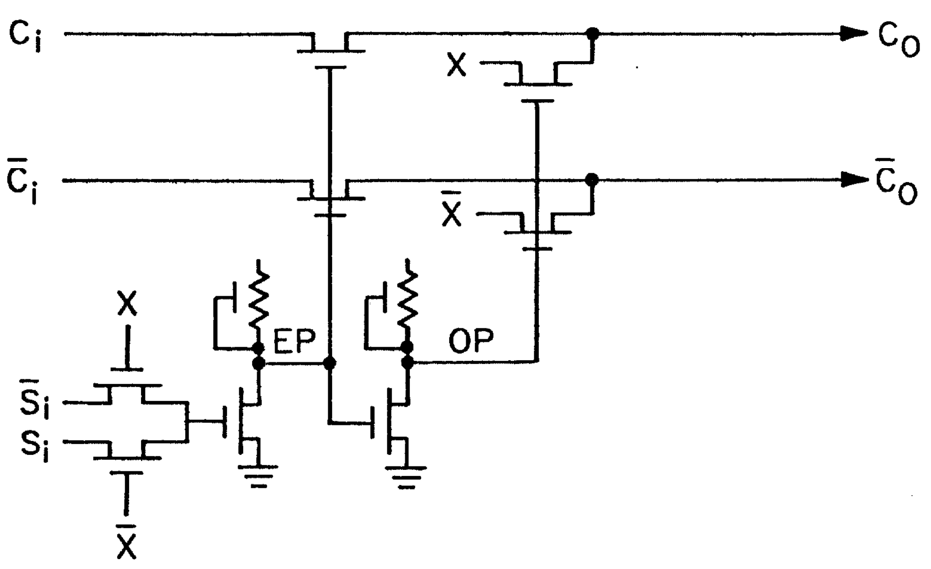


Figure 4.20 Modified CARRY, MOS circuit.

to save time. Note that both data and data bar are carried on pre-charged lines. The final S_o and C_o can thus be generated with only a group of pass gates. Because of the exclusive-OR/NOR circuits used, the carries between cells are always level restored. The sums that propagate between stages are also level restored. Since the lines are pre-charged, voltage swings to ground only occur. Thus no active pull-up is required.

When the multiplier is idle, all the precharged lines discharge towards zero, causing all inverter outputs to rise towards a high state. Thus this circuit dissipates minimal power during non-use.

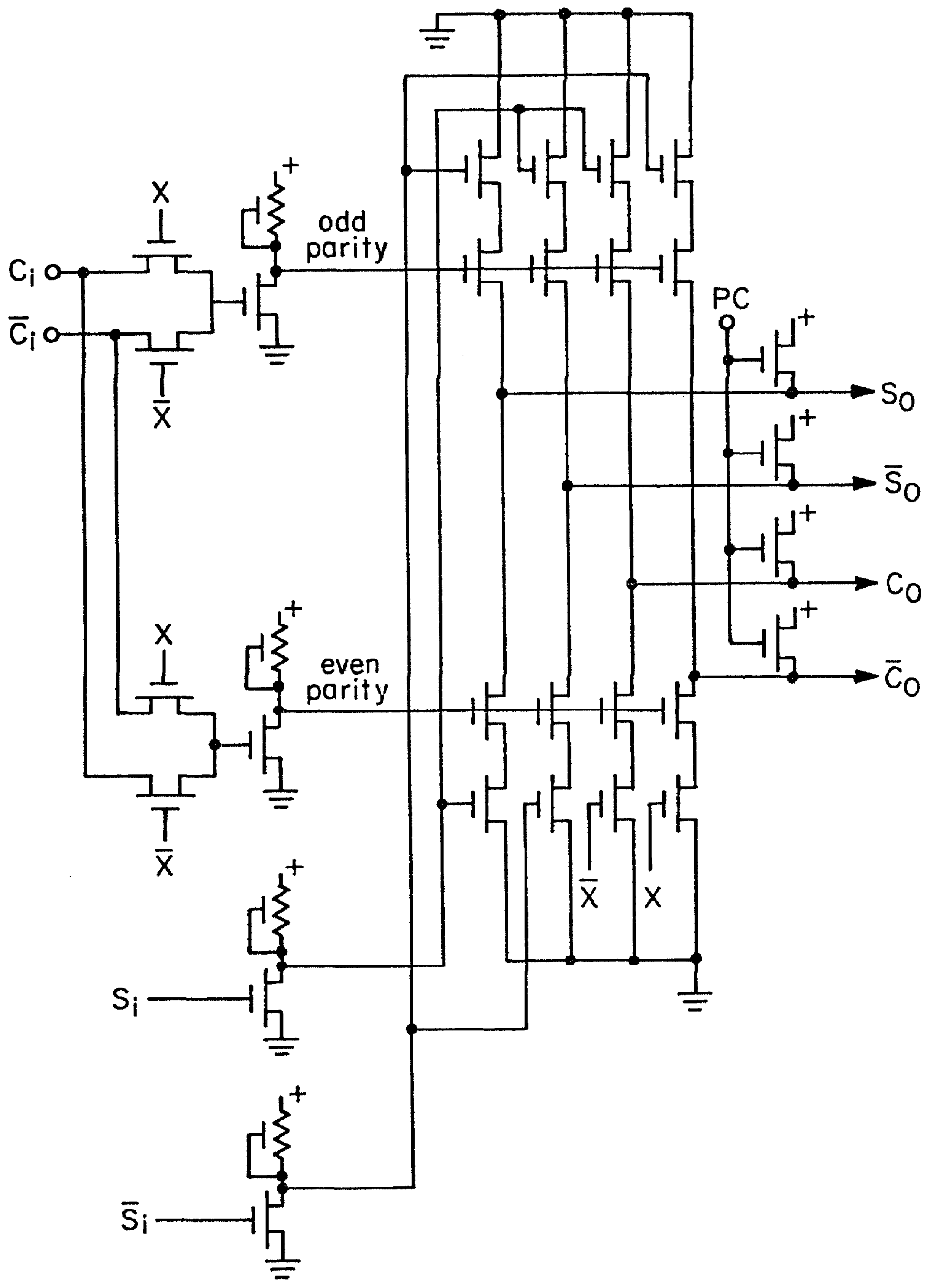


Figure 4.21 Carry-save adder, MOS circuit.

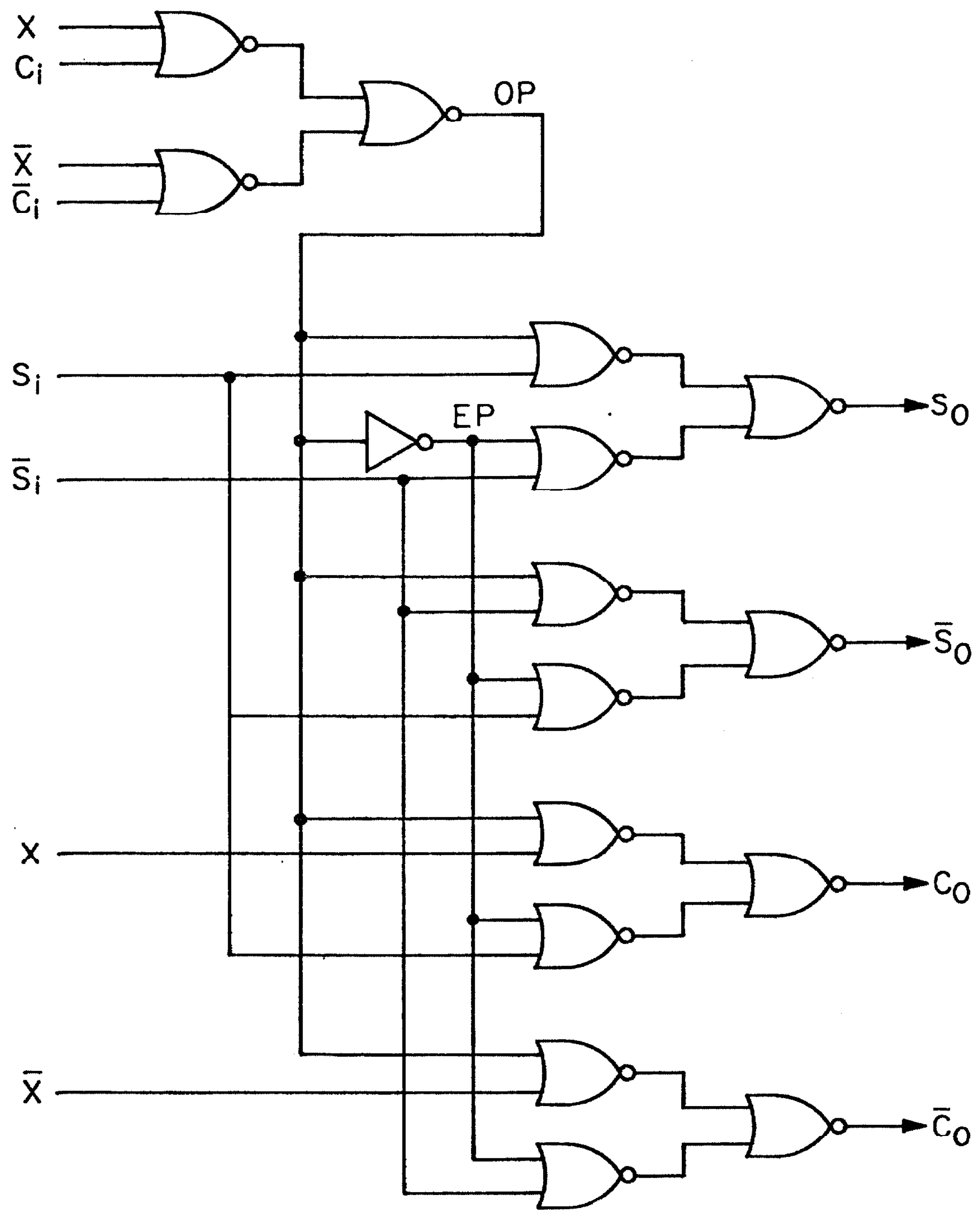


Figure 4.22 Carry-save adder, NOR gate equivalent circuit.

THE ACCUMULATOR AND FULL ADDER

The accumulator uses the same cell as the combinatorial array. The only difference is that the multiplicand word is not used as the X input. Instead, the previous output data are used as the X input to the carry-save adder. Sums and carries out of the accumulator then go to the full adder for final processing.

The full adder is logically similar to the carry-save adder. The only difference lies in the circuit layout implementation. As previously indicated, full adders are defined to be adders which input two addend words and generate a complete or full sum as an output. Any internally generated carries are accounted for in the sum. The delay time for sum generation is dominated by the carry assimilation delay time. A NOR gate equivalent circuit of a full adder is shown in figure 4.23.

The discussion involving carry-save adders also applies to MOS implementations of full adders. Figure 4.24 shows the MOS circuit implementation of the full adder that results. When compared with the carry-save adder of figure 4.21, it can be noted that the carry-in lines are set up for fast propagation. However, the carry lines do not have any level restoration between cells. Although represented as switches, MOS pass gates have a non-negligible series resistance when turned on. This causes cascaded carry stages to look like an RC ladder network. If n equals the number of stages, the time delay through this network is roughly proportional to n^2. Delays through a series of conventional

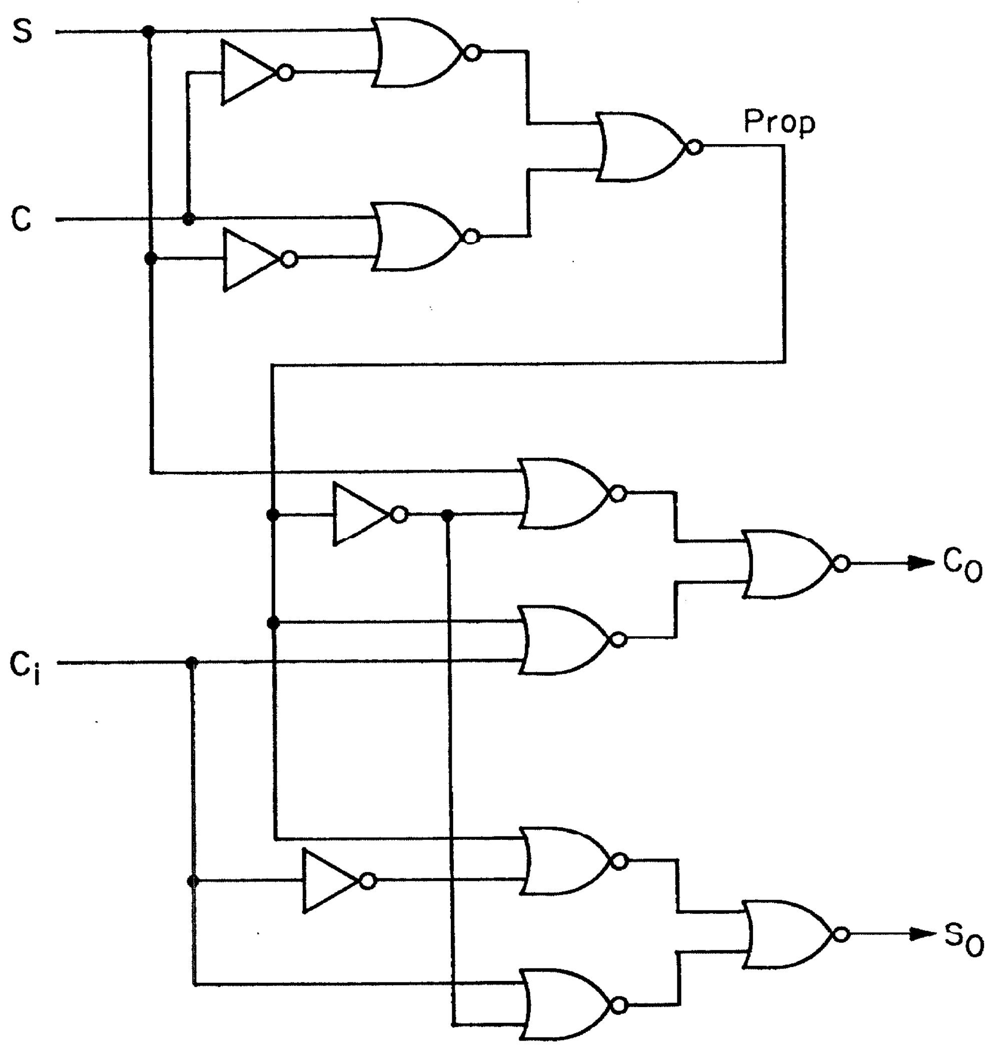


Figure 4.23 Full adder NOR gate equivalent circuit.

inverters is proportional to n. Thus there is an upper limit to the number of stages that can be cascaded before the delay time can be improved by inserting a restoring logic stage. For this MOS implementation, the magic number is approximately four [7].

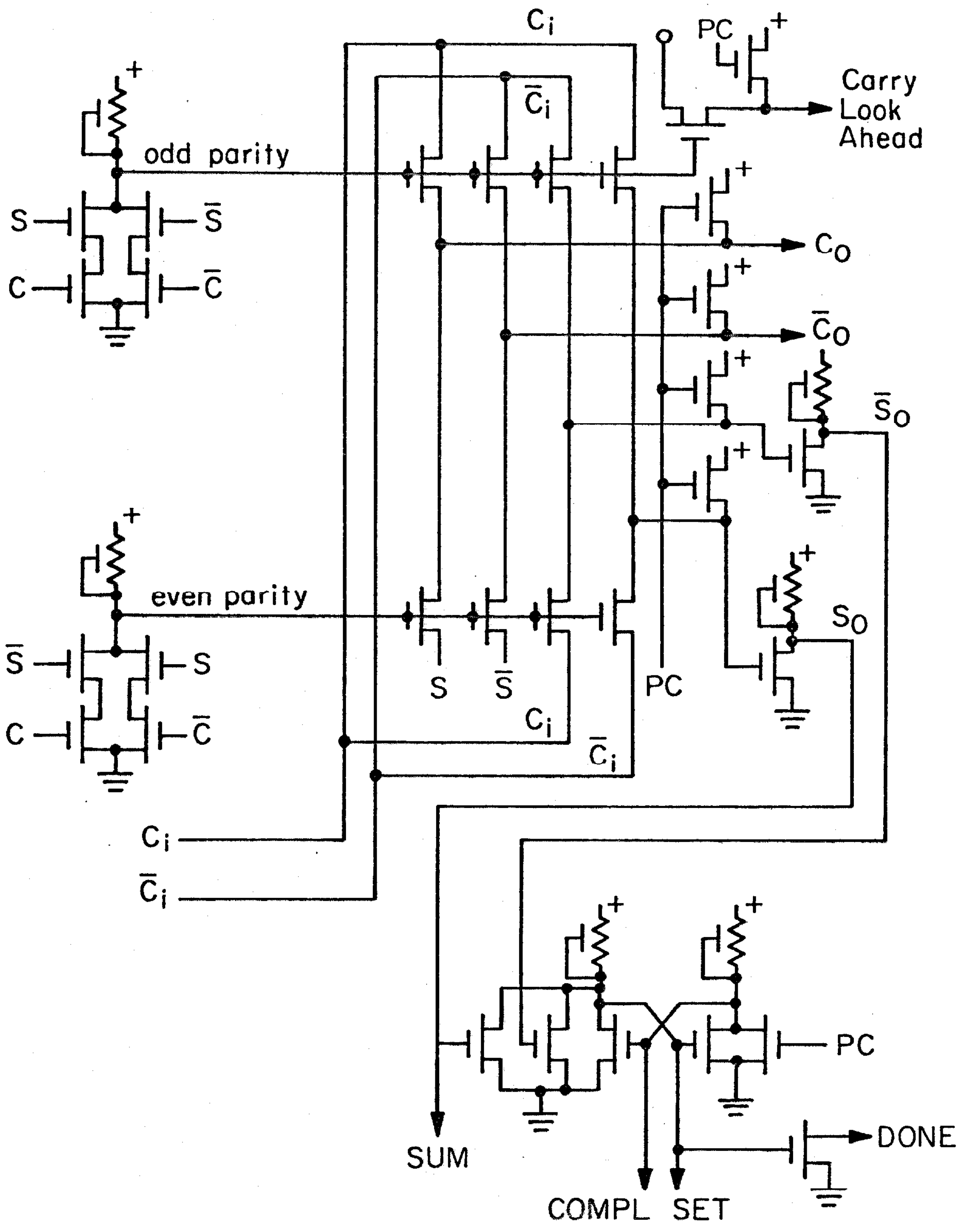


Figure 4.24 Full adder MOS circuit.

A further examination of figure 4.24 shows that the exclusive OR logic is implemented differently. The S and C lines in from the carry-save adders are both precharged. This means that they will never reach a high enough '1' state voltage level that can drive a pass gate. Thus a more conventional level restoring XOR circuit is used.

Initially, when PC goes high, S, S bar, C, and C bar are set high. Thus both "odd parity" and "even parity" are set low. This allows the C_i, C_i bar, C_o, C_o bar, S_o, and S_o bar nodes also to be precharged high. So initially the propagate, generate, and kill controls are off. At the appropriate time, only one set of controls will become active. Any timing race conditions are thus prevented.

SELF-TIMED LOGIC

The time response of many systems tends to be limited by the worst case time delay. When operated in a synchronous environment, adder speeds are limited by the slowest sum generation delay time. However, in many asynchronous timing applications, it may be desirable not to be limited in this manner. It would be useful to know when the addition operation is finished so that the next operating step can immediately be performed.

Completion sensing can be implemented for the sum-out or the carry line to indicate completion of the operation. For an asynchronous application, this self-timing feature allows a new operation to be started as soon as the old one is finished. On the average, this type of operation is much faster than the synchronous system, which must operate at a speed limited by the worst case delay time. Self-timed logic is not without its drawbacks, however. Both data and its complement are usually required throughout the circuit to allow for completion detection. Also, a means must be provided to allow resetting the logic to a known state. Completion is then detected by noting when either data or its complement change. In the present implementation this capability is already present.

The additional logic attached to the sum-out lines of figure 4.24 provides the sum completion sensing. The timing process is shown in figure 4.25. During the precharge time, both S_o and S_o bar are initially set to "0". Since precharge is high, COMPL is forced low and SET is forced high. The DONE line is also released. This DONE line is ORed with all the other adder sum output sensors. After precharge goes low, either S_o or S_o bar will eventually go high. This will cause COMPL to go high and SET to go low, indicating that the operation for that bit position has been completed. When this state is reached by all the cells, DONE is allowed to rise, thus indicating that the multiply operation is completed.

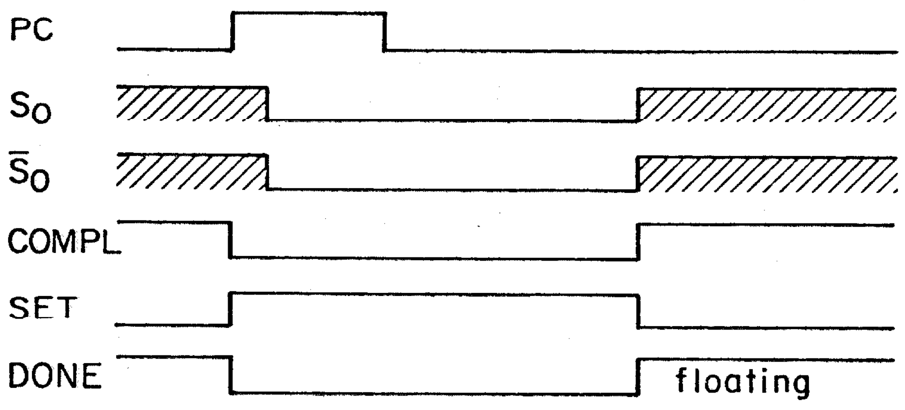


Figure 4.25 Multiplication completion timing.

CARRY-SKIP/CARRY LOOK AHEAD LOGIC

A wide variety of techniques for speeding up adders are based on the carry-skip idea [8]. These schemes date back to Babbage's anticipated carry device. At the cost of additional circuits, it is possible to detect a state when a contiguous group of adder stages will merely be propagating the carries within that group. An alternate path can then be set up to allow the carry-in to skip over that section and to go directly to the next group. A trade off between circuit complexity added and speed improvement gained determines the optimal group size. It is of course assumed that an alternate path exists that is significantly faster than the existing one.

Figure 4.26 shows a NOR gate equivalent circuit for carry lookahead. When a full propagate state (four stages in this case) is detected, a signal is generated to activate a bypass route. In figure 4.27a this is accomplished by turning on a bypass MOS switch. The approach of figure 4.27b uses an AND-OR circuit to access either the lookahead signal or the normal carry line. In the present implementation, there exists a need both to buffer the carry signal and to propagate both signal polarities. Since the lookahead lines are relatively high

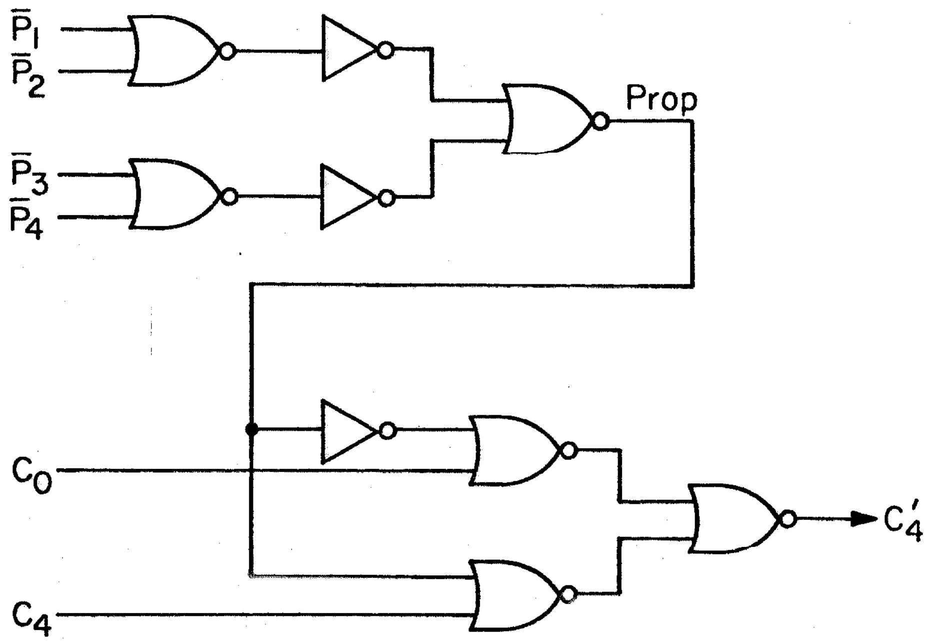
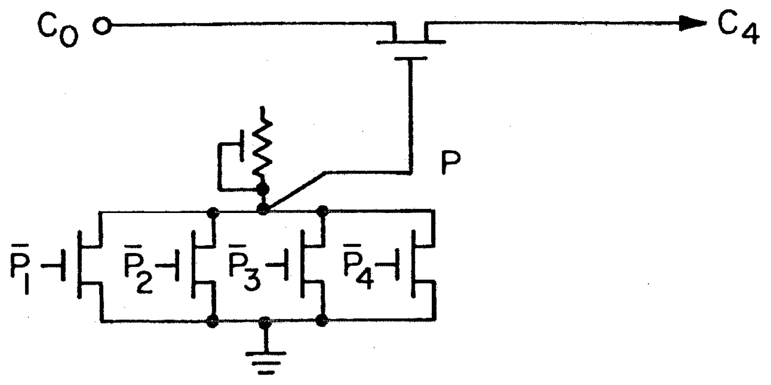
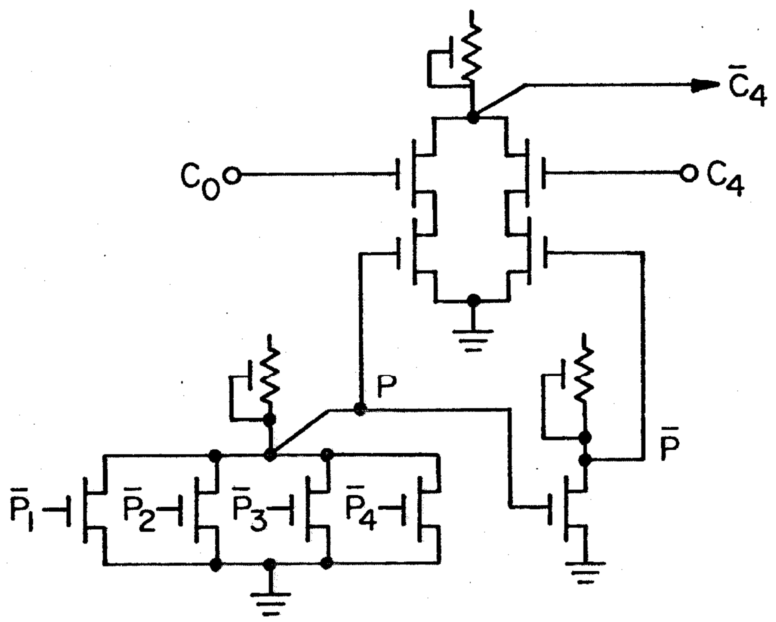


Figure 4.26 Carry lookahead NOR gate equivalent circuit.



a. Bypass MOS switch



b. AND-OR switching

Figure 4.27 Carry lookahead MOS circuits.

capacitance, they would cause the MOS bypass switch method to exhibit a rather slow signal response. Thus the ORing approach is preferred.

Since only one full adder is used, a precharged carry lookahead arrangement can also be employed. The requirement that any node change state at most once is not violated. The extra space used can be justified by the speed gains. Setting up a dual rail scheme helps even more. Figure 4.28 shows the full lookahead scheme. C_o is the carry entering the chain. C_4 is the chain output. After precharge, PROP is high, keeping the lookahead line inactive. If a lookahead condition develops, the lookahead line is given control of the C_4' lines.

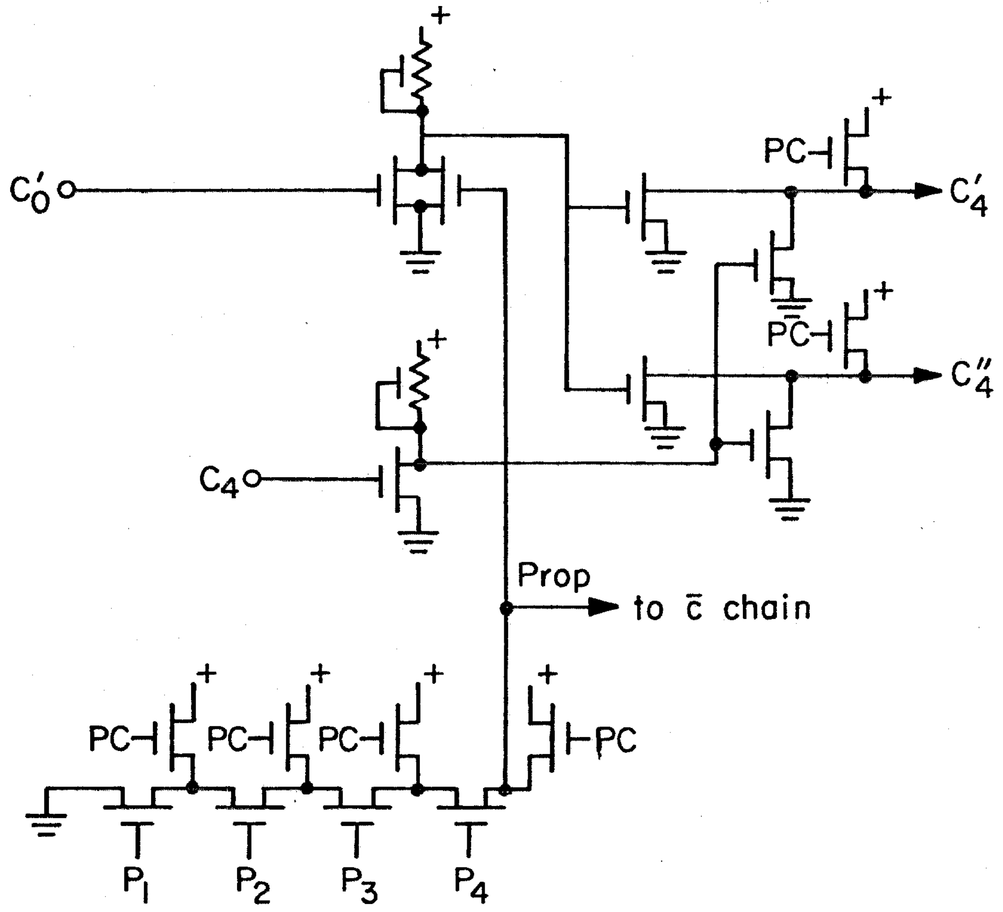
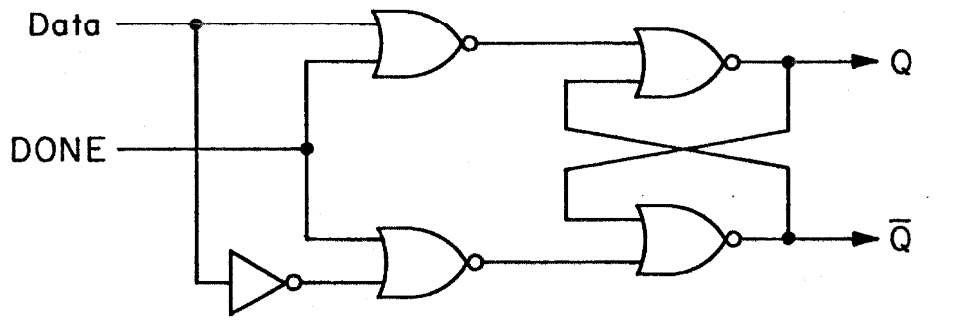


Figure 4.28 Carry_lookahead MOS circuit for C_n. (the $\overline{C}_n$ chain is identical)

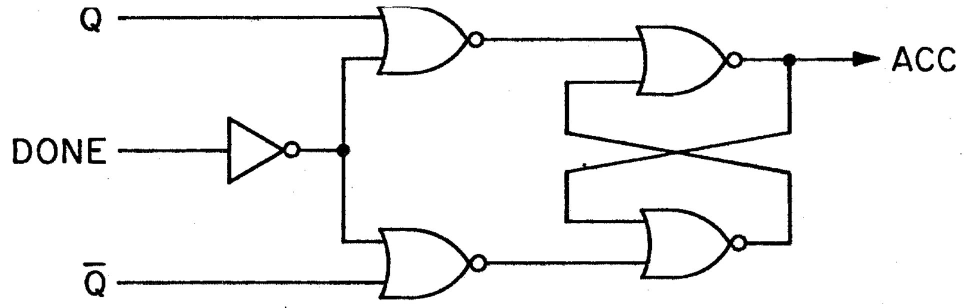
OUTPUT REGISTER AND DRIVER

The NOR gate equivalent circuit for the output register and driver is shown in figure 4.29. The accumulated product is gated to the output register during the 'DONE' low period. At the same time, the accumulator register saves the previous output value for use during the present cycle. When the'DONE' line goes high, the data become locked into the output register. They remain there until the beginning of the next multiply cycle when PC goes high again. Meanwhile, this new output value is gated into the accumulator register for use during the next cycle. The output driver remains in a tri-state, high impedance mode until the OUTCLK line goes high. During the OUTCLK high period, the output data are driven onto the external data lines.

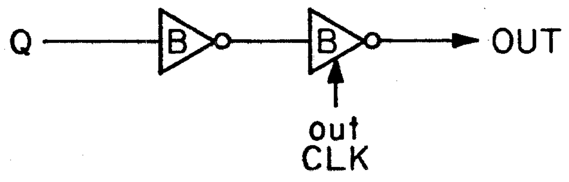
The MOS circuit implementation is shown in figure 4.30. It should be noted that the use of node 'A' as a dynamic storage node allows the accumulator register to be implemented with only one inverter and a transmission device. Also, in order to save power and still provide strong output drive, the OUTCLK data enabling uses a bootstrap principle to drive the output. An implication of this technique is that the output data are of dynamic quality. They will eventually lose their drive capability if OUTCLK is left in a static high state. However, in most applications the output is enabled for only one cycle period to read the data, then disabled again. This dynamic cycling of OUTCLK would make this driver appear no different from other types, while dissipating much less power.



a. Output register



b. Accumulator register



c. Output driver

Figure 4.29 NOR gate equivalent circuits for the output register and driver.

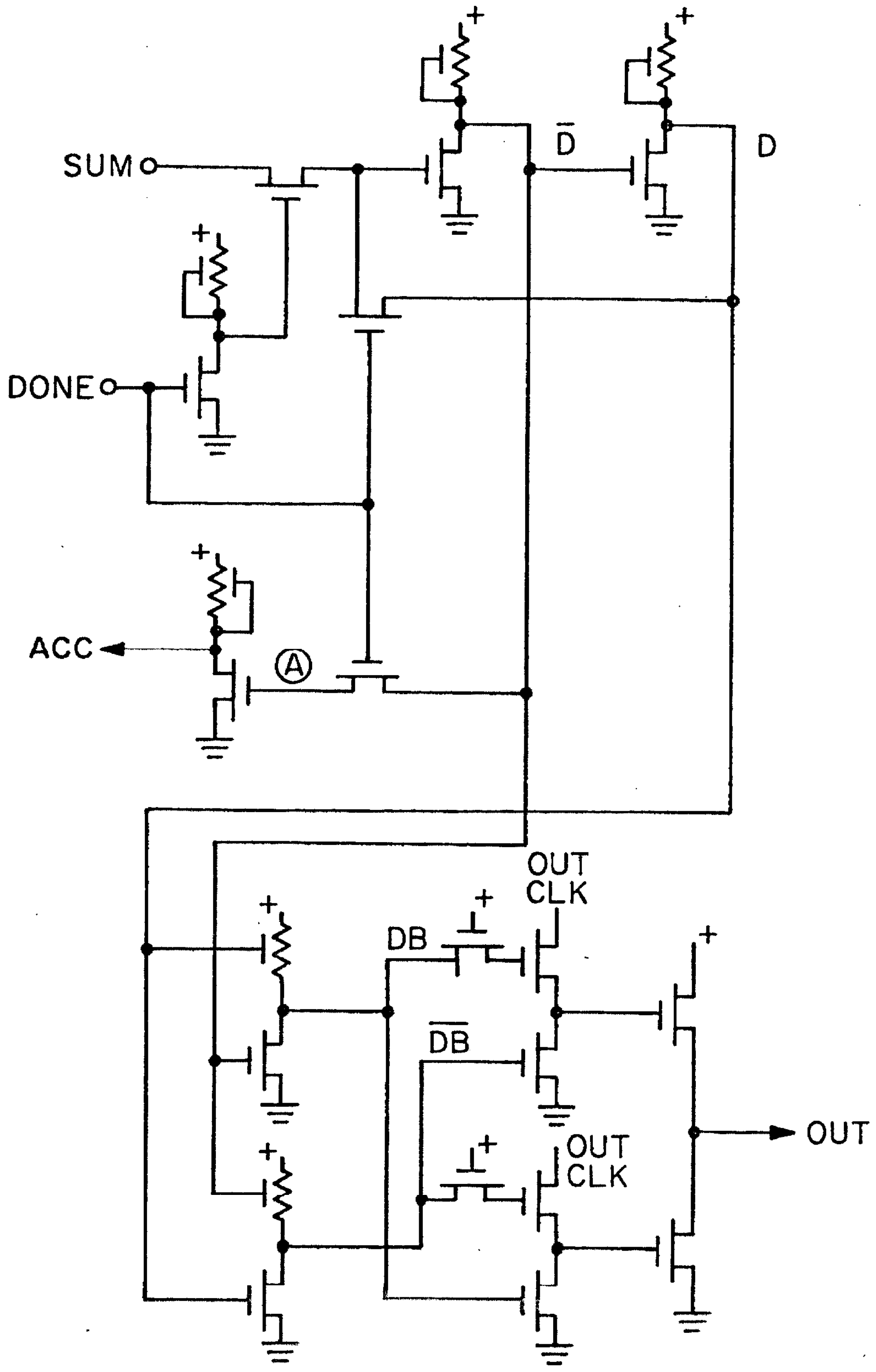


Figure 4.30   Output register and driver, MOS circuit.

SIGN AND OVERFLOW CONTROL

For a 2's complement multiplicand of n-bits, n+1 carry-save adder cells are used for each level in the array. The n+1 cell, which is called the ENDCELL, performs the sign calculation. Whereas standard sign generation for multiplication is performed through an examination of the signs of the two input words, using the modified Booth's Algorithm and carry-save adders allows this to be generated as part of the normal adder array process.

As discussed earlier, the algorithm assigns a negative weighting to the decoded multiplier word whenever the beginning of a string of 1's is encountered. A positive correction is applied only if the end of a 1's string is encountered. For the case of a negative two's complement number, this positive correction never occurs once the sign information is encountered. Thus the multiplier word sign is accounted for. The partial products formed from the multiplicand word can be maintained in correct form, provided an overflow never occurs into the sign position. Since intermediate results are maintained in carry-save form, an overflow into the sign position never occurs. The carry-save adders keep defering the sign bit calculations to later levels. Thus, by allowing for the 2x shift option at each level, merely adding a 17th cell to keep track of the sign position is all that is needed. The sum and carry out of the final ENDCELL in the array can then be combined in the full adder to generate the sign bit for the product.

The ENDCELL is functionally similar to the carry-save adder cell. The primary difference lies in the I/O interface. In the shifter section, only X_{15} is input. No shifting is necessary since we are dealing with the MSB which is the same in either case. The invert option is retained, however. In the carry-save adder section, the S_i and C_i from the previous ENDCELL serve as the inputs. The S_o generated is sent to the last three cells of the next adder level. The C_o is sent to the last two cells of the next level. This arrangement is illustrated in figure 4.31.

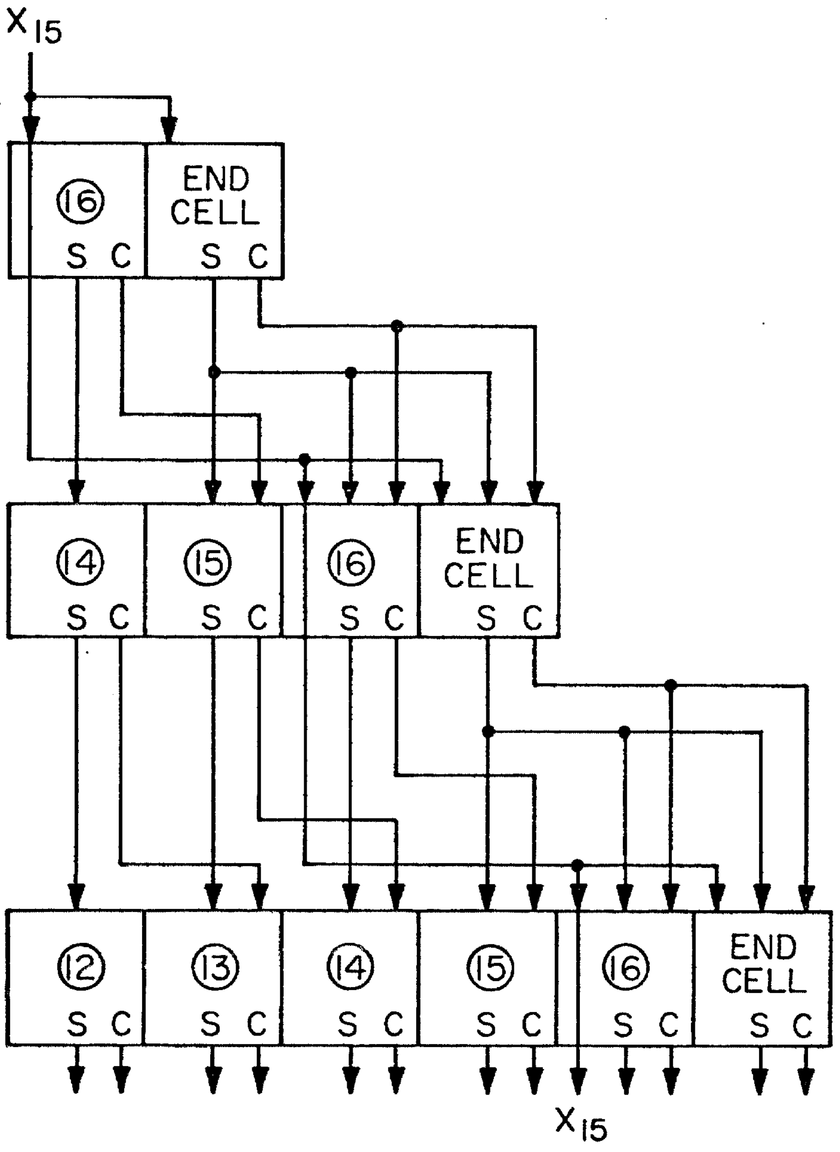


Figure 4.31 Array termination.

CONCLUSION

Recent advances in integrated circuit technology have made it possible to implement increasingly complex functions on a single chip. Improvements have reached a point where a fully parallel 16 bit by 16 bit multiplication function can be placed onto a single LSI chip. The design of such a multiplier has just been discussed. Unlike existing LSI multipliers, this design was implemented in n-channel silicon gate MOS with depletion loads. It used the modified Booth's algorithm to improve performance.

The functional design was implemented with circuits that were carefully selected to match the MOS process. Circuit designs that are unique to MOS were used to allow multiplication speeds that were comparable to bipolar equivalents while keeping chip power dissipation well below one watt. Fast asynchronous operation is also possible through the use of special completion sensing logic. All the cells in the circuit design are modular. When MOS scaling and process improvements make it possible, a 32 bit by 32 bit multiplier can easily be generated by merely changing only two variables in the chip design language. The 16 x 16 multiplier chip is shown in figure 5.1. Preliminary test data on the chip are provided in Appendix A.

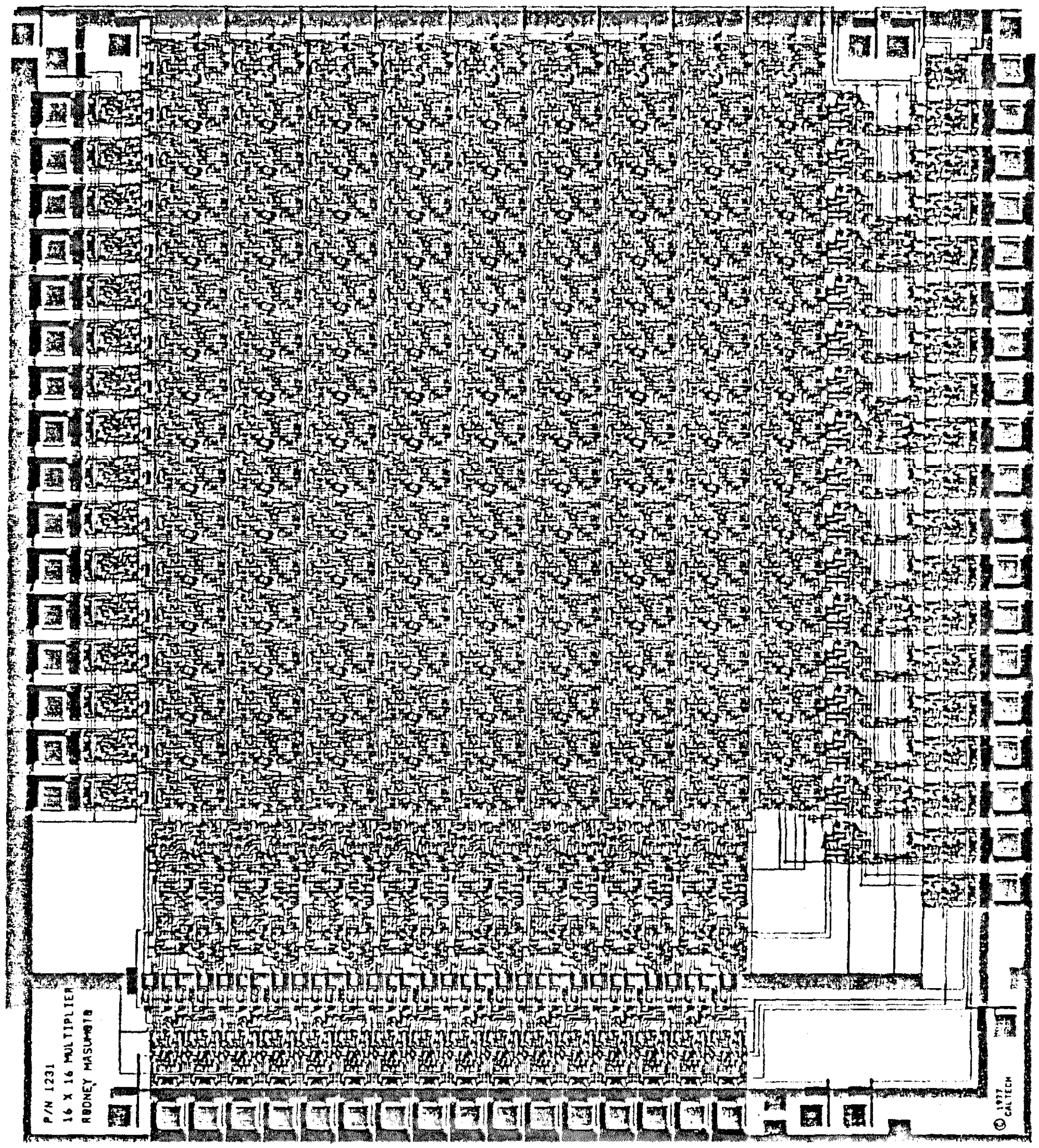


Figure 5.1 The 16 bit by 16 bit LSI multiplier/accumulator chip.

APPENDIX A

A preliminary performance test of the 16 bit multiplier/accumulator chip has been performed. Figure 1 shows a timing diagram of selected test points. The test conditions included a nominal supply voltage (V_D) of 5 volts. The minimum multiply clock pulse width (t_{pw}) was 75 nsec. Typical switching characteristics were as follows:

multiply delay to accumulator output, t_{ac} = 200 nsec.

multiply delay to full adder output, t_{fc} = 410 nsec.

multiply delay to output register, t_{oc} = 460 nsec.

'DONE' detect time, t_{dc} = 470 nsec.

OUTCLK to data out, t_o = 40 nsec.

multiply and accumulate cycle, t_c = 535 nsec.

Chip power dissipation with OUTCLK = 0 volts was as follows:

P_D = 210 mW @ t_c = 0.5 μsec.

= 200 mW @ t_c = 1 μsec.

= 190 mW @ t_c = 10 μsec.

= 100 mW (standby, PC = CLK = 0 volts)

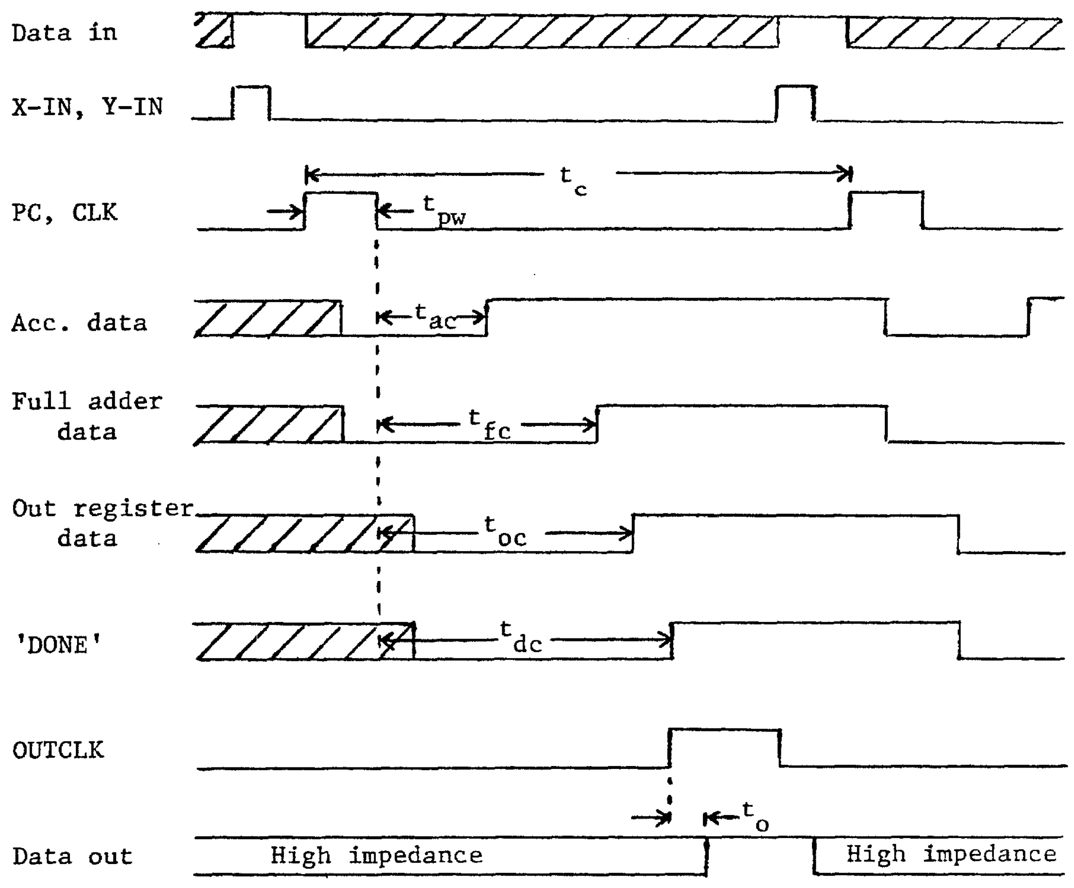


Figure 1. Timing diagram.

REFERENCES

1. G. W. McIver, R. W. Miller, and T. G. O'Shaughnessy, "A monolithic 16 x 16 digital multiplier," in _1974 Int. Solid-State Circuits Conf., Dig. Tech. Papers_, pp. 54-55.

2. I. Flores, _The Logic of Computer Arithmetic_. Englewood Cliffs, N. J.: Prentice-Hall, Inc., 1963. 493p.

3. I. E. Sutherland, internal document.

4. C. S. Wallace, "A suggestion for a fast multiplier," _IRE Trans. Elect. Comput._, vol. EC-13, pp. 14-17, Feb. 1964.

5. L. P. Rubinfield, "A proof of the modified Booth's algorithm for multiplication," _IEEE Trans. Comput._, vol. C-24, pp. 1014-15, Oct. 1975.

6. R. C. Ghest, _Application Note: A 2's Complement Digital Multiplier - the AM2505_. Sunnyvale, California: Advanced Micro Devices, Inc., 1971, 11p.

7. C. A. Mead and L. A. Conway, _Introduction to LSI Systems_. (unpublished).

8. M. Lehman and N. Burla, "Skip techniques for high-speed carry propagation in binary arithmetic units," _IRE Trans. Elect. Comput._, vol. EC-10, pp. 691-98, Dec. 1961.