S. F. Anderson
J. G. Earle
R. E. Goldschmidt
D. M. Powers

The IBM System/360 Model 91: Floating-Point Execution Unit

Abstract: The principal requirement for the Model 91 floating-point execution unit was that it be designed to support the instruction-issuing rate of the processor. The chosen solution was to develop separate, instruction-oriented algorithms for the add, multiply, and divide functions. Linked together by the floating-point instruction unit, the multiple execution units provide concurrent instruction execution at the burst rate of one instruction per cycle.

Introduction

The instruction unit of the IBM System/360 Model 91 is designed to issue instructions at a burst rate of one instruction per cycle, and the performance of floating-point execution must support this rate. However, conventional execution unit designs cannot support this level of performance. The Model 91 Floating-Point Execution Unit departs from convention and is instruction-oriented to provide fast, concurrent instruction execution.

The objectives of this paper are to describe the floating-point execution unit. Particular attention is given to the design of the instruction-oriented units to reveal the techniques which were employed to match the burst instruction rate of one instruction per cycle. These objectives can best be accomplished by dividing the paper into four sections—*General design considerations*, *Floating-point terminology*, *Floating-point add unit*, and *Floating-point multiply/divide unit*.

The first section explains how the desire for concurrent execution of instructions has led to the design of multiple execution units linked together by the floating-point instruction unit. Then the concept of instruction-oriented units is discussed, and its impact on the multiplicity of units is pointed out. It is shown that, with the instruction-oriented units as building blocks and the floating-point instruction unit as the "cement," an execution unit evolves which rises to the desired performance level.

The section on floating-point terminology briefly reviews the System/360 data formats and floating-point definitions.

The next two sections describe the design of the instruction-oriented units. The first of these is the floating-point add unit description which is divided into two sub-sections, *Algorithm* and *Implementation*. In the algorithm sub-section, the complete algorithm for execution of a floating add/subtract is considered with emphasis on the difficulties inherent in the implementation. Since the add unit is instruction-oriented, (i.e., only add-type instructions must be considered), it is possible to overcome the inherent difficulties by merging the several steps of the algorithm into three hardware areas. The implementation section describes these three areas, namely, characteristic comparison and pre-shifting, fraction adder, and post-normalization.

The last section describes the floating-point multiply/divide unit. This section describes the multiply algorithm and its implementation first, and then the divide algorithm and its implementation. The emphasis of the multiply algorithm sub-section is on recoding the multiplier and the usefulness of carry-save adders. In the implementation sub-section the emphasis is on the iterative hardware which is the heart of the multiply operation. An arrangement of carry-save adders is shown which, when pipelined by adding temporary storage platforms, has an iteration repetition rate of fifty Mc/sec. The divide algorithm is described next with emphasis on using multiplication, instead of subtraction, as the iterative operator. The discussion of divide implementation shows how the existing multiply hardware, plus a small amount of additional circuitry, is used to perform the divide operation.

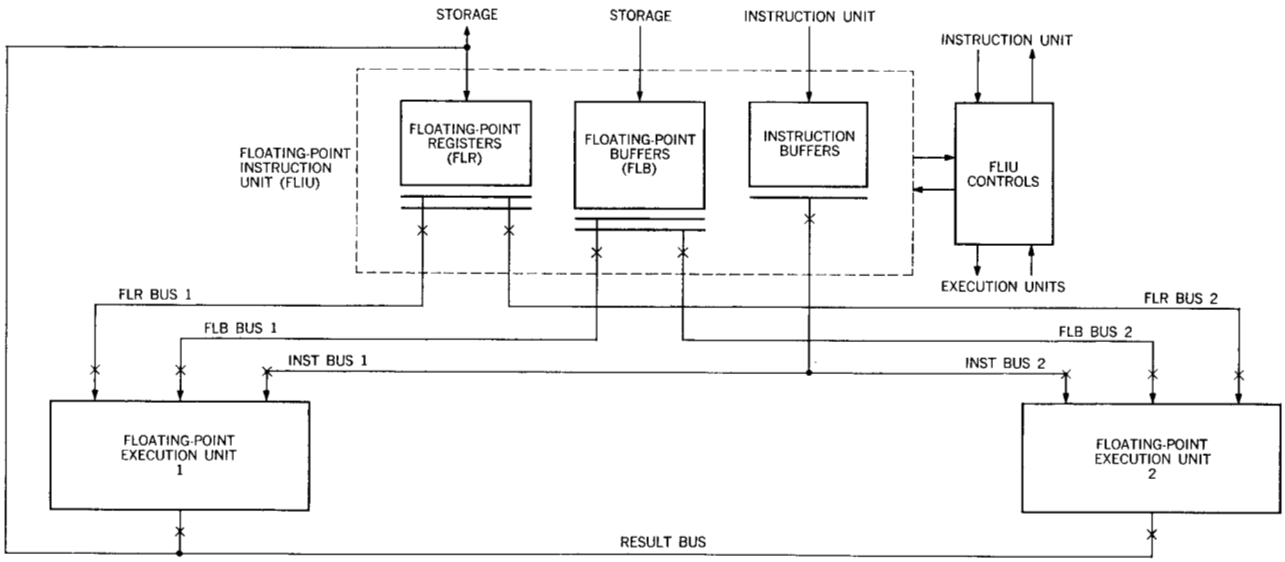


Figure 1 Floating-point execution unit capable of concurrent execution.

General design considerations

The programs considered "typical" by the user of high-performance computers are floating-point oriented. Therefore, the prime concern in designing the floating-point execution unit is to develop an overall organization which will match the performance of the instruction unit. However, the execution time of floating-point instructions is long compared with the issuing rate of these instructions by the instruction unit. The most obvious approach is to apply a faster technology and with special design techniques reduce the execution time for floating-point. But a study of many "typical" floating point programs revealed that the execution time per instruction would have to be 1 to 2 cycles in order to match the performance[3] of the instruction unit.* Conventional execution unit design, even with state-of-the-art algorithms, will not provide these execution times.

Another approach considered was to provide execution concurrency among instructions; this obviously would require two complete floating-point execution units.† An attendant requirement would be a floating-point instruction unit. This unit is necessary to sequence the operands from storage to the proper execution unit; it must buffer the instructions and assign each instruction to a non-busy execution unit. Also, since the execution time is not the same for all instructions the possibility now exists for out-of-sequence execution, and the floating-point instruction must insure that executing out of sequence does not produce incorrect results.* The organization for an execution unit capable of concurrent execution is shown in Fig. 1. Buffering and sequence control of all instructions, storage operands, and floating-point accumulators are the responsibility of the floating-point execution unit. Each of the execution units is capable of executing all floating-point instructions.

One might be led to believe that this organization is a suitable solution in itself. If multiply can be executed in seven cycles and two multiplies are executed simultaneously, then the effective execution time is 3.5 cycles. Similarly, for add the execution time would go from three cycles to 1.5 cycles. However, the operating delay of the floating-point instruction unit must be considered, and it is not always possible to execute concurrently because of the dependence among instructions. When these problems are considered the effective execution time is close to three cycles per instruction, which is not sufficient. A third execution unit would not help because the complexity of the floating-point instruction unit increases, and the amount of hardware becomes prohibitive.

The next solution to be considered was to improve the execution time of each instruction by employing faster algorithms in the design of each execution unit. Obviously this would increase the hardware, but since the circuit

* Even though the burst rate of the instruction unit is one instruction per cycle, it is not necessary to execute at the same rate.

† Since two complete execution units are necessary for concurrent execution, the cost-performance factor is important. Analysis showed that execution times of three cycles for add and seven cycles for multiply were reasonable expectations.

* Dependence among instructions must be controlled. If instruction $n + 1$ is dependent on the result of instruction n, instruction $n + 1$ must not be allowed to start until instruction n is completed.

Table 1 Floating-point instructions executed by floating-point execution unit.

Type	*Instruction*	*Condition code*	*Floating-point exceptions**	*Arithmetic unit*
RR-RX	Load (S/L)	NO		FLIU
RR	Load and Test (S/L)	YES		FLIU
RX	Store (S/L)	NO		FLIU
RR	Load Complement (S/L)	YES		ADD
RR	Load Positive (S/L)	YES		ADD
RR	Load Negative (S/L)	YES		ADD
RR-RX	Add Normalized (S/L)	YES	U, E, LS	ADD
RR-RX	Add Unnormalized (S/L)	YES	E, LS	ADD
RR-RX	Subtract Normalized (S/L)	YES	U, E, LS	ADD
RR-RX	Subtract Unnormalized (S/L)	YES	E, LS	ADD
RR-RX	Compare (S/L)	YES		ADD
RR	Halve (S/L)	NO		ADD
RR-RX	Multiply	NO	U, E	M/D
RR-RX	Divide	NO	U, E, FK	M/D

* *Exceptions:*
U—Exponent-underflow exception
E—Exponent-overflow exception
LS—Significance exception
FK—Floating Point Divide Exception

delay is a function not only of the circuit speed but also of the number of loads on the input net and the length of the interconnection wiring, more hardware may not make the unit faster.[5] These two factors—the desire for faster execution of each instruction and the size sensitivity of the circuit delay, have produced a concept which is unique to the organization of floating-point execution units, and which was adopted for the Model 91: the concept of using separate execution units for different instruction types. Faster execution of each instruction can be achieved if the conventional execution unit is separated into arithmetic units designed to execute a subset of the floating-point instructions instead of the entire set. This conclusion may not be obvious, but a unit designed exclusively for a class of similar instructions can execute those instructions faster than a unit designed to accommodate all floating-point instructions. The control sequences are shorter and less complex; the data flow path has fewer logic levels and requires less hardware because the designer has more freedom in combining serial operations to eliminate circuit levels; the circuit delay per level is faster because less hardware is required in the smaller, autonomous units. To implement the concept in the Model 91, the floating-point instruction set was separated into two subsets: add and multiply/divide. Table 1 shows a list of the instructions and identifies the unit in which each instruction is executed. With this separation, an add unit which executed all add class instructions in two cycles, and a multiply/divide unit which executed multiply in six cycles and divide in eighteen cycles, were designed.

The use of this concept somewhat changes the character of concurrent execution. It is possible to have concurrent execution with one execution unit—i.e., two arithmetic units, add and multiply/divide. The performance is not quite as good as that attainable using two execution units, but less hardware is required for the implementation. Therefore, more arithmetic units can be added to improve the performance. First, two add units and two multiply/divide units were considered. But the floating-point instruction unit can assign only one instruction per cycle. Therefore, since an add operation is two cycles long, two add units could be replaced by one add unit if a new add class instruction could be started every cycle. This would introduce still another example of concurrent execution: concurrent execution within an arithmetic unit.

Such concurrency within a unit is facilitated by the technique of pipelining. If a section of combinatorial logic, such as the logic to execute an add, could be designed with equal delay in all parallel paths through the logic, the rate at which new inputs could enter this section of logic would be independent of the total delay through the logic. However, delay is never equal; skew is always present and the interval between input signals must be greater than the total skew of the logic section. But temporary storage platforms can be inserted which will separate the section of combinatorial logic into smaller synchronous stages. Now the total skew has been divided into smaller pieces; only the skew between stages has to be considered. The interval between inputs has decreased and now depends on the skew between temporary storage platforms. Essentially the temporary storage platform is used to separate one complete job, such as an add, into several pieces; then

several jobs can be executed simultaneously. Thus, inputs can be applied at a predetermined rate and once the pipeline is full the outputs will match this rate.

The technique of pipelining does have practical limits, and these limits differ for each application. In general the rate at which new inputs can be applied is limited by the logic preceding the pipeline (e.g., add is limited to one instruction per cycle by the floating-point instruction unit) or by the rate at which outputs can be accepted. Also, both the rate of new inputs and the length of the pipeline are limited by dependencies among stages of the pipeline or between the output and successive inputs (e.g., the output of one add can become an input for the next).

The add unit requires two cycles for execution and is limited to one new input per cycle. Thus pipelining allows two instructions to be in execution concurrently, thereby increasing the efficiency with a small increase in hardware.

Further study of pipelining techniques would indicate that a three-cycle multiply and a twelve-cycle divide are possible. Here the technique of pipelining is used to speed up the iterative section of the multiply which is critical to multiply/divide execution. (This is discussed in detail in the section on the multiply/divide unit.)

The execution unit would consist at this point of a floating-point instruction unit, an add unit which could start an instruction every cycle, and a multiply/divide unit which would execute multiply in three cycles and divide in twelve cycles. However the performance still would not match the instruction unit. The execution times would be adequate but the units would spend considerable time waiting for operands. Therefore, instead of duplicating the arithmetic unit (which is expensive) extra input buffer registers have been added to collect the operands and necessary instruction control information. When both operands are available, the control information is processed and a request made to use an arithmetic unit. These registers are referred to as "reservation stations." They can be and are treated as independent units.

The final organization is shown in Fig. 2. It consists of three parts: the floating-point instruction unit; the floating-point add unit; and the floating-point multiply/divide unit. Another paper in this series[3] explains the floating-point instruction unit in detail. The problems involved and both the considered solutions and the implemented solutions are discussed. The floating-point add unit has three reservation stations and, as stated above, is treated as three separate add units, A1, A2 and A3. The floating-point multiply/divide unit has two reservation stations, M/D1 and M/D2. The last two sections of this paper describe the design of these two units in detail.

Floating-point terminology

The reader is assumed to be familiar with System/360 architecture and terminology.[2] However, the floating-point

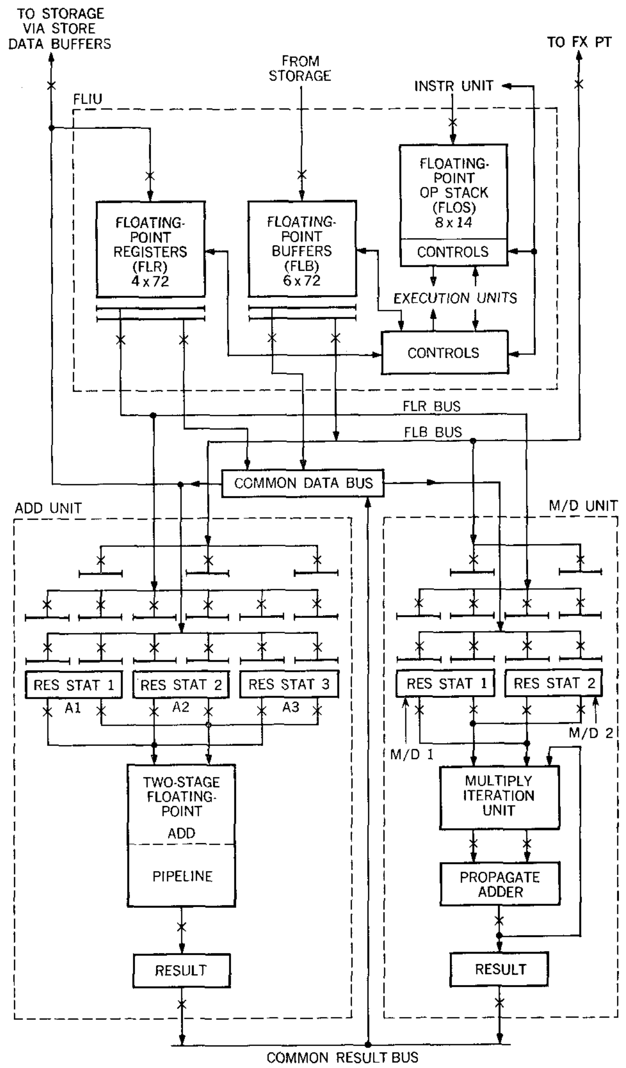


Figure 2 Overall organization of floating-point unit.

data format and terminology will be briefly reviewed here.

Floating-point data occupy a fixed-length format, which may be either a full-word short format or a double-word format:

Short Floating-Point Binary Format

Sign Characteristic Fraction

0 1 — — — — — 7 8 — — — — 31

Long Floating-Point Binary Format

Sign Characteristic Fraction

0 1 — — — — 7 8 — — — — — 63

The first bit(s) in either format is (are) the sign bit(s). The subsequent seven bit positions are occupied by the charac-

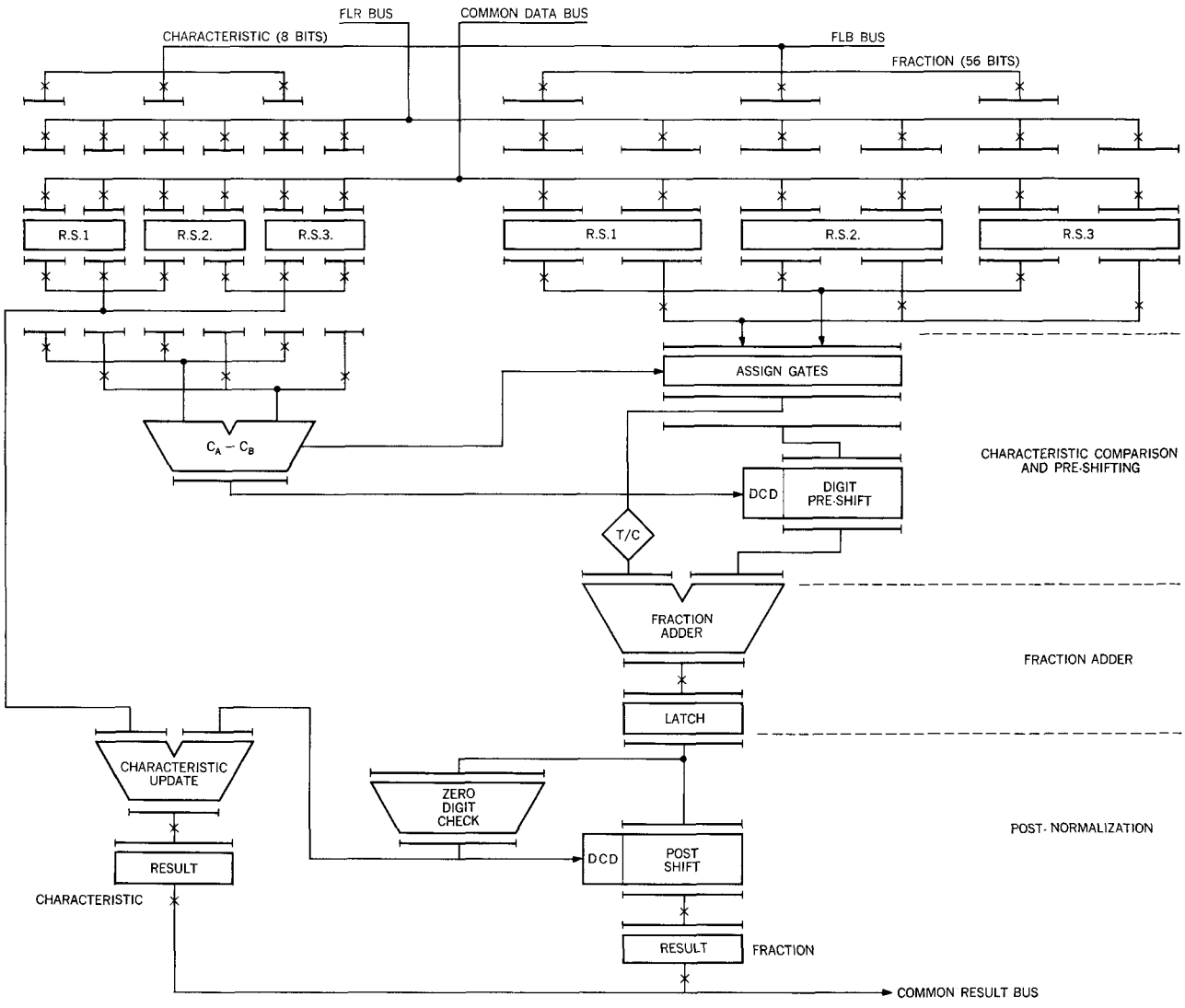


Figure 3 Floating-point add data flow.

teristic. The fraction consists of six hexadecimal digits for the short format or 14 hexadecimal digits for the long.

The radix point of the fraction is assumed to be immediately to the left of the high-order fraction digit. To provide the proper magnitude for the floating-point number, the fraction is considered to be multiplied by a power of 16. The characteristic portion, bits 1–7 of both floating-point formats, indicates this power. The characteristic is treated as an excess 64 number with a range from −64 through +63 corresponding to the binary expression of the values 0–127.

Both positive and negative quantities have a true fraction, the difference in sign being indicated by the sign bit. The number is positive or negative accordingly as the sign bit is zero or one.

A normalized floating-point number has a non-zero high-order hexadecimal fraction digit. To preserve maximum precision in subsequent operation, addition, subtraction, multiplication, and division are performed with normalized results. (Addition and subtraction may also be programmed to be performed with unnormalized results. The operands for any floating-point operation can be either normalized or unnormalized.)

Floating-point add unit

The challenge in the design of the add unit was to minimize the number of logical levels in the longest delay path. However, the sequence of operations necessary for the execution of a floating-point add impedes the design goal.

Consider the following operations:

(a) Since the radix point must be aligned before an add can proceed, the characteristics of the two operands must be compared and the difference between them established.
(b) This difference must be decoded into the shift amount, and the fraction with the smaller characteristic shifted right a sufficient number of positions to make the characteristics equal.
(c) Since subtraction is to be performed by forming the two's complement of one of the fractions and then adding the two fractions in the fraction adder, one of the fractions must pass through true/complement logic.
(d) The two operand fractions are added in a parallel adder. The carries must propagate from the low order end to the high order end.
(e) Because of subtraction, the output must provide for both the true sum and the complement sum, depending on the high-order carry.
(f) If the system architecture calls for left justification or normalized operation, the result out of the adder must be checked for high-order zeros and shifted left to remove these zeros.
(g) The characteristic must be reduced by the amount of left shift necessary to normalize the resultant fraction.
(h) The resultant operand must be stored in the proper accumulator.

The above sequence of operations implies a series of sequential execution stages, each of which is dependent on the output of the previous stage. The problem then, is to arrange, change and merge these operations to provide fast, efficient execution for a floating-point add.

None of the steps can be eliminated. Each step is required in order to execute add; but the steps can be merged so that the interface between them is eliminated,* and each step can be changed to provide only the necessary information to the next stage. For example, the long data format consists of 14 hexadecimal digits; therefore any difference between the two characteristics which is greater than 14 will result in an all zero fraction. This means that the characteristic difference adder need not generate a sum for the high-order three bits. Instead, if the difference is greater than 14, a shift of 15 is forced. As a result, the characteristic difference adder is faster and less expensive.

The add unit algorithm is separated into three parts: characteristic comparison and pre-shifting, fraction adder, and post-normalization (Fig. 3). The first section, the characteristic comparison and pre-shifting operation, merges the first three operations from the sequence given above; the second section—the fraction adder—merges the next two operations; the final section—post normalization—merges the final three operations. The hardware implementation of each of these three sections is discussed below.

* Levels are used to encode the output of one step, which is subsequently decoded in the next step. Merging the two steps will eliminate these levels.

$C_A > C_B$	$C_B = 1\ 1\ 1\ 1\ 1\ 0\ 0$		$C_B = 1\ 1\ 0\ 1\ 0\ 0\ 0$
	$1\ 1\ 1\ 1\ 1\ 0\ 0$	C_A	
	$0\ 0\ 1\ 0\ 1\ 1\ 1$	$\overline{C}_B$	
	$\quad\quad\quad\quad\quad\quad 1$	HOT ONE	
(RESULT IS TRUE) 1	$1\ 1\ 0\ 1\ 1\ 1\ 1$	$C_A - C_B$	
$C_A < C_B$	$C_B = 1\ 1\ 0\ 1\ 0\ 0\ 0$		$C_B = 1\ 1\ 1\ 1\ 1\ 0\ 0$
	$1\ 1\ 0\ 1\ 0\ 0\ 0$	C_A	
	$0\ 0\ 0\ 0\ 0\ 1\ 1$	$\overline{C}_B$	
(RESULT IS COMPLEMENT) 0	$\quad\quad\quad\quad\quad\quad 1$	HOT ONE	
	$1\ 1\ 0\ 1\ 1\ 0\ 0$		
COMP. RESULT	$0\ 0\ 1\ 0\ 0\ 1\ 1$		
MUST ADD HOT ONE	$\quad\quad\quad\quad\quad\quad 1$		
	$0\ 0\ 1\ 0\ 1\ 0\ 0$	$C_A - C_B$	
$C_A < C_B$ (END-AROUND CARRY)			
	$1\ 1\ 0\ 1\ 0\ 0\ 0$	C_A	
	$0\ 0\ 0\ 0\ 0\ 1\ 1$	$\overline{C}_B$	
(NO CARRY) 0	$1\ 1\ 0\ 1\ 0\ 1\ 1$		
COMPLEMENT	$0\ 0\ 1\ 0\ 1\ 0\ 0$	CORRECT RESULT	

Figure 4 Examples of exponent arithmetic.

• *Implementation*

Characteristic comparison and pre-shifting

The first stage of execution for all two-operand instructions (floating-point add, subtract, and compare) is to compare the characteristics and establish the magnitude of the difference between them. The characteristic (C_B) of one operand is always subtracted from the characteristic (C_A) of the other operand ($C_A - C_B$). Characteristic B is always complemented as it is gated in at the reservation station.

If the output of the characteristic difference adder is the true sum or the complement of the true sum, the output can be decoded directly at the pre-shifter. But the adder always subtracts C_B from C_A and if $C_B > C_A$ the sum would be negative. Therefore, to eliminate the possibility of having to add a 1 in the low order position and complement when C_B is greater than C_A, an "end-around-carry" adder is used. This is shown by the example in Fig. 4.

The characteristic comparison can result in two states—$C_A \geq C_B$ or $C_B > C_A$. If $C_A \geq C_B$, there is a carry out of the high order position of the characteristic difference adder, and the carry is used to gate the fraction of operand B to the pre-shifter. The true sum output of the characteristic difference adder is the amount that the fraction must be shifted right to make the characteristics

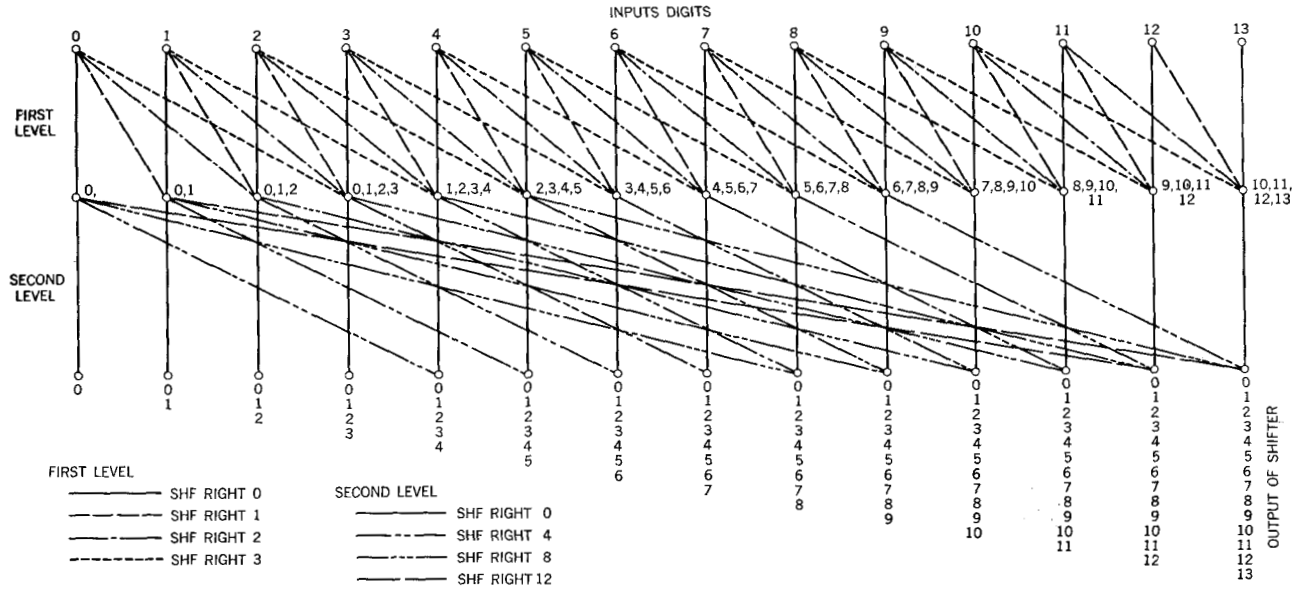


Figure 5 Digit pre-shifter.

equal. If $C_B > C_A$, there is no carry out of the high order position of the characteristic difference adder, and the absence of a carry is used to gate the fraction of operand A to the pre-shifter. In this case the complement of the sum output of the characteristic difference adder is the amount that the fraction must be shifted right to make the characteristics equal. In both cases the second operand fraction (the one with the larger characteristic) is gated to the true-complement input of the fraction adder.

The characteristic of the unshifted fraction becomes the resultant characteristic. It is gated to the characteristic-update adder, and after updating, if necessary, it is gated to the accumulator specified by the instruction.

The output of the characteristic difference adder is decoded by the pre-shifter and the proper fraction shifted right the necessary number of positions. The pre-shifter is a parallel digit-shifter which shifts each of the 14 digits right any amount from zero to fifteen. The decode of the shift amount is designed into each level, thereby eliminating serial logic levels for decoding.

The pre-shifter consists of two circuit levels. The first level shifts a digit right by 0, 1, 2 or 3 digit positions. The second level shifts a digit right by 0, 4, 8, or 12 digit positions. Thus, by the proper combination of these amounts any right digit shift between and including 0 and 15 can be executed. Figure 5 shows an example of the pre-shifter.

The un-shifted fraction is gated to the true/complement gates of the adder. Here the fraction is gated unchanged if the effective operation is ADD and complemented if the effective operation is SUBTRACT. The true/complement gating is overlapped with the pre-shifter on a time basis. The output of both the true/complement logic and the pre-shifter are the inputs to the fraction adder.

Fraction adder

Most of the time required for binary adders is carry propagation time. Two operands must be combined and the carries allowed to ripple from right (low order) to left (high order). The usual method of finding the sum is to combine the half sum* of bit n (higher order) with the carry from bit $n - 1$ ($S_n = A_n \forall B_n \forall C_n$).† The carry ($C_n$) into bit position n is also a three term expression which includes the carry into bit position $n - 1$

$$(C_n = A_{n-1} \cdot B_{n-1} \vee A_{n-1} \cdot C_{n-1} \vee B_{n-1} \cdot C_{n-1}).$$

If the carry term is rearranged to read

$$C_n = A_{n-1} \cdot B_{n-1} \vee (A_{n-1} \vee B_{n-1}) C_{n-1},$$

two new terms can be defined which separate the carry into two parts—generated carry, and propagated carry. The generated carry (G_{n-1}) is defined as $A_{n-1} \cdot B_{n-1}$, and the carry propagate function (often abbreviated to simply propagate or P_{n-1}) is defined as $A_{n-1} \vee B_{n-1}$. Now the

* The half sum is the exclusive OR of the two input bits, ($A_n \forall B_n$).

† The two operand fractions are designated as A, B and the bits as A_n, B_n, A_{n-1}, B_{n-1}, etc. C_n is the carry into bit position n, which is the carry out from bit $n - 1$.

carry expression can be rewritten as:[1,6]

$$C_n = G_{n-1} \vee P_{n-1}C_{n-1}$$

$$C_n = G_{n-1} \vee P_{n-1}G_{n-1} \vee P_{n-1}P_{n-2}C_{n-2}$$

$$C_n = G_{n-1} \vee P_{n-1}G_{n-1} \vee P_{n-1}P_{n-2}G_{n-2} \vee P_{n-1}P_{n-2}P_{n-3}C_{n-3}$$

$$\vdots$$

The expansion can continue as far as one desires and one could conceive of C_n being generated by one large OR block preceded by several AND blocks (in fact n AND blocks—one for each stage). But it is obvious that the limiting factor would be the circuit fan-in. Only a limited number of circuit stages can be connected together in this manner. This technique is defined as carry look-ahead, and by cascading different levels of look-ahead the technique can be made to fit the circuit fan-in, fan-out limitations.

For example, assume that four bits can be arranged in this manner, and that each four bits form a "group." The adder is now divided into groups and the carries and propagates can be arranged for carry look-ahead between groups just as they were for look-ahead between bits. It is possible to carry the concept even further and define a section as consisting of one or more groups. Now the adder has three levels of carry look-ahead: the bit level of look-ahead, the group level, and the section level.

The fraction adder of the floating-point add unit is a carry look-ahead adder. A group is made up of four bits (one digit) and two groups form a section. Since it must be capable of adding 56 bits, the fraction adder consists of seven sections and 14 groups. Each pair of input bits generate the three bit functions: half-sum ($A \veebar B$), bit carry generate ($A \cdot B$) and bit propagate ($A \vee B$). These functions are combined to form the group generate and propagate which in turn are combined to form the section generate and propagate. A typical group is shown in Fig. 6 and the group and section look-ahead are shown in Fig. 7.

The high-order sum consists of nine bits to include the end-around carry for subtraction and the overflow bit for addition. The end-around carry is needed for subtraction because the fraction which is complemented may not be the subtrahend. This is illustrated by the example given in the description of the characteristic comparison. If the effective sign of the instruction is minus (the exclusive OR of the sign of the two fractions and the instruction is the effective sign) the effective operation is subtract. Also, the high-order bit (ninth bit of the high order section) is set to a one, thus conditioning it for an end-around-carry. If there is no end-around-carry when the effective sign is minus the adder output is complemented.

Post-normalization

Normalization or post-shifting takes place when the intermediate arithmetic result out of the adder is changed to the final result. The output of the fraction adder is checked for high-order zero digits and the fraction is left-shifted until the high-order digit is non-zero.

The output of the fraction adder is gated to the zero-digit checker. The zero-digit checker is simply a large decoder, which detects the number of leading zero digits, and provides the shift amount to the post-shifter. Since this same amount must be subtracted from the characteristic, the zero-digit checker also must encode the shift amount for the characteristic update adder.

The implementation of the digit post-shifter is the same as the digit pre-shifter except for the fact that the post-shift is a left-shift. The first level of the post-shifter shifts each of the 14 digits left 0, 1, 2 or 3 and the second level shifts each digit 0, 4, 8, or 12. The output of the second level is gated into the add unit fraction result register, from which the resultant fraction is routed to the proper floating-point accumulator.

The characteristic update is executed in parallel with the fraction shift. The zero-digit checker provides the characteristic update adder with the two's complement of the amount by which the characteristic must be reduced. Since it is not possible to have a post-shift greater than 13, the high-order three bits of the characteristic can only be changed by carries which ripple from the low order four bits. The update adder makes use of this fact to reduce the necessary hardware and speed up the operation.

Floating-point multiply/divide unit

Multiply and divide are complicated operations. However, two of the original design goals were to select an algorithm for each operation such that (1) both operations could use common hardware, and (2) improvement in execution time could be achieved which would be comparable to that achieved in the floating-point add unit. Several algorithms exist for each instruction which make the first design goal attainable. Unfortunately, the best of the algorithms generally used for divide are not capable of providing an improvement in execution comparable to the improvement achievable by those used for multiply. The algorithm developed for divide in the Model 91 uses multiplication as the basic operator. Thus, common hardware is used, and comparable improvement in the execution time is achieved.

In order to give a clear, consistent treatment to both instructions, this section discusses the multiply algorithm and hardware implementation first. Then the divide algorithm is discussed separately. Finally, it is shown how divide utilizes the multiply execution hardware and the hardware which is unique to the execution of divide is described.

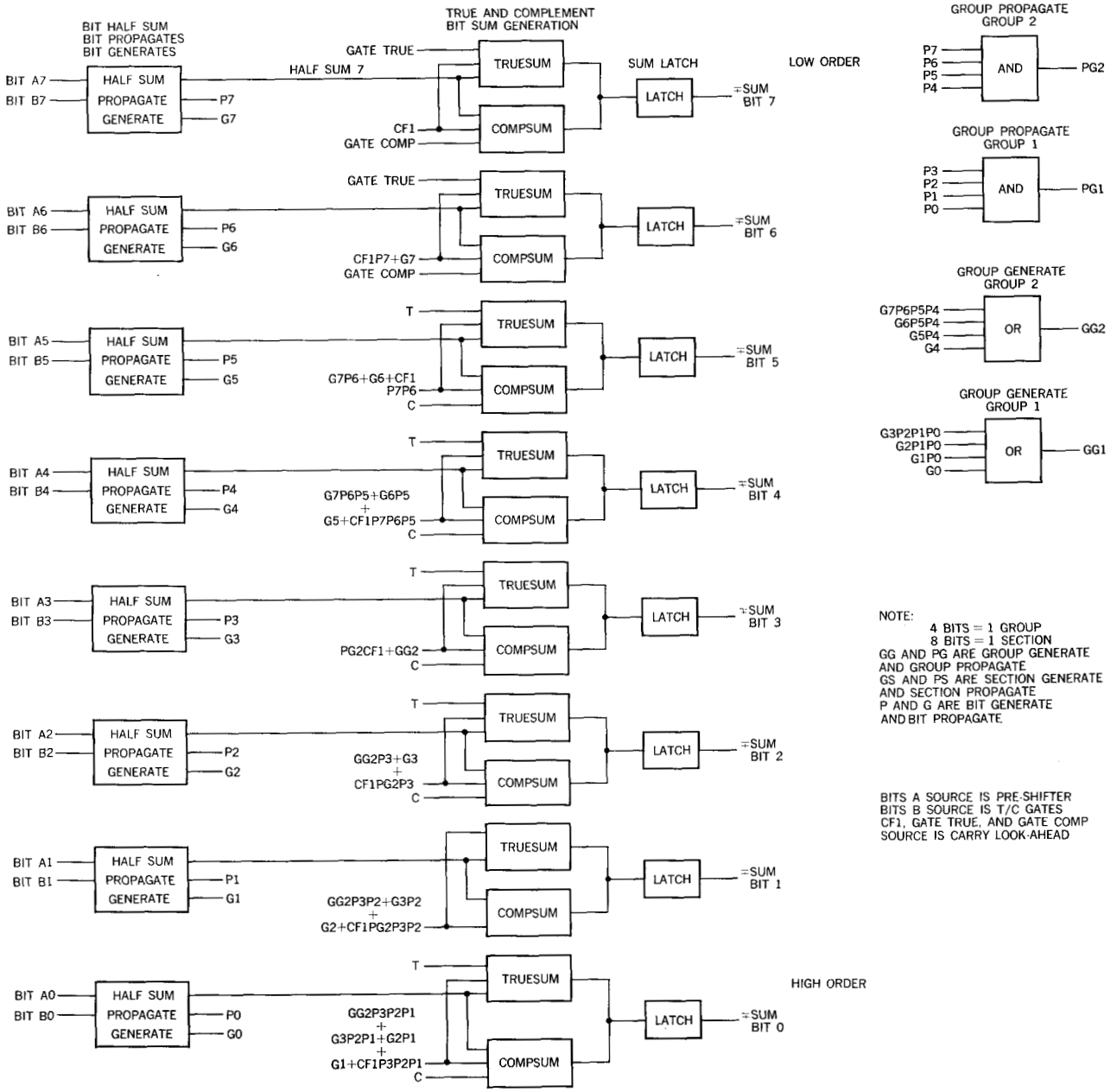


Figure 6 Fraction adder, section 1 (high-order).

• *Multiply algorithm*

Computers usually execute multiply by repetitive addition, and the time required is dependent on the number of additions required.[1,6] A zero bit in the multiplier results in adding a zero word to the partial product. Therefore, because shifting is a faster operation than add, the execution time can be decreased by shifting over a zero or a string of zeros. Any improvement in the multiply execution beyond this point is not obvious. However, certain properties of the binary number system combined with complementing to allow subtraction as well as addition can be used to reduce the number of necessary additions.

An integer in any number system may be written in the following form:

$$a_n b^n + a_{n-1} b^{n-1} + \cdots + a_1 b^1 + a_0 b^0,$$

where

$0 \leq a \leq b - 1$, and b = base of the number system

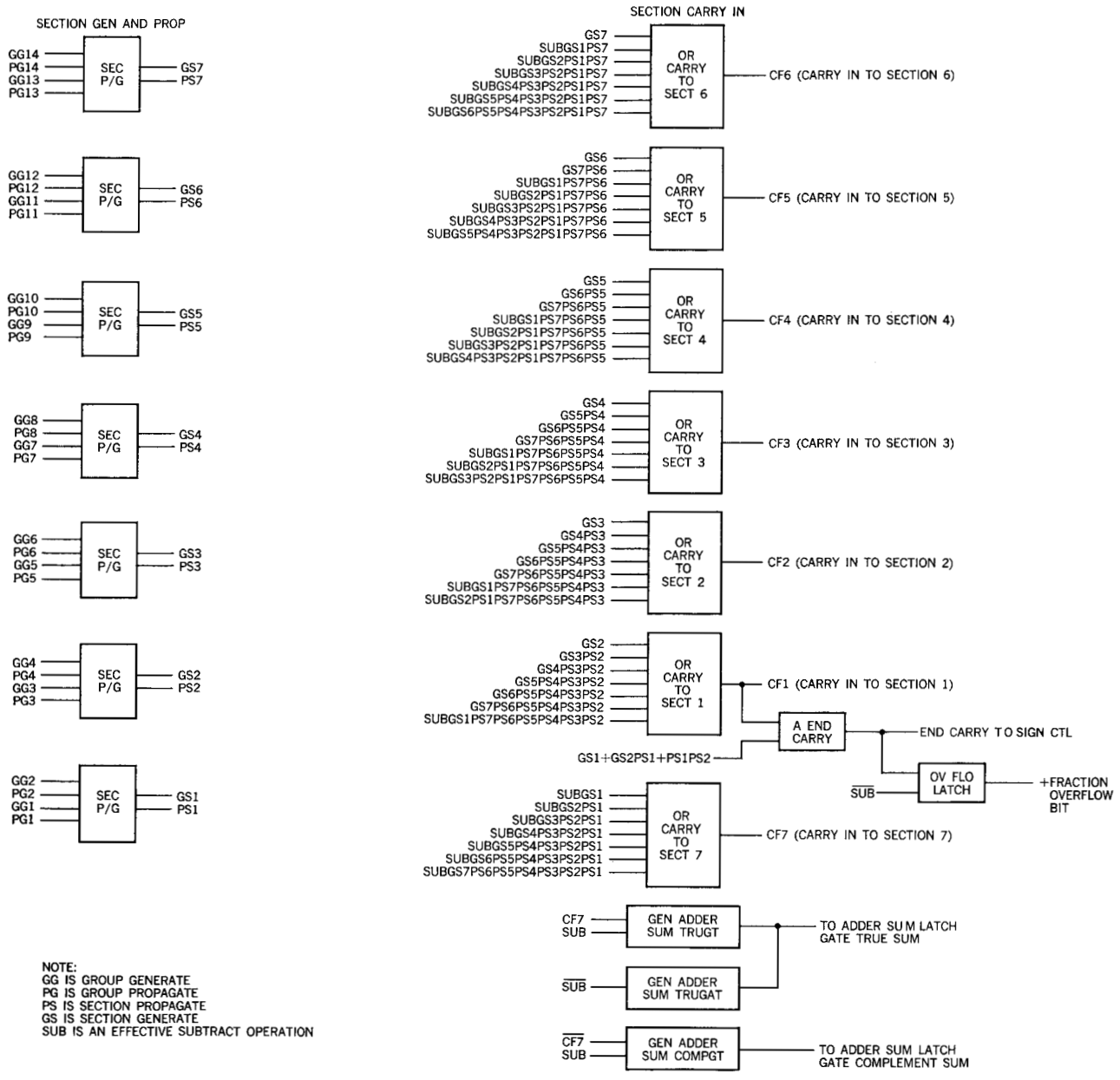


Figure 7 Fraction adder, carry look-ahead.

One of the properties of numbering systems which is particularly interesting in multiply is that an integer can be rewritten as shown below.

$$a_n b^n + a_{n-1} b^{n-1} + \cdots + a_k b^k + \cdots + a_{n-x} b^{n-x},$$

where

$$a_k = b - 1 \text{ for any } k.$$

In the binary number system a_k can take only the values 0 and 1. Thus, using the above property, a string of 1's can be skipped by subtracting at the start of the string and adding at the end of the string:

$$112_{10} = 2^6 + 2^5 + 2^4 = 2^7 - 2^4,$$

$$112_{10} = 111000_2 = 10000000_2 - 10000_2.$$

Therefore, a string of 1's in the multiplier can be reduced from an addition for each 1 in the string to a subtraction for the first 1 in the string, shift the partial product one position for each 1 in the string, and an addition for the last 1 in the string.

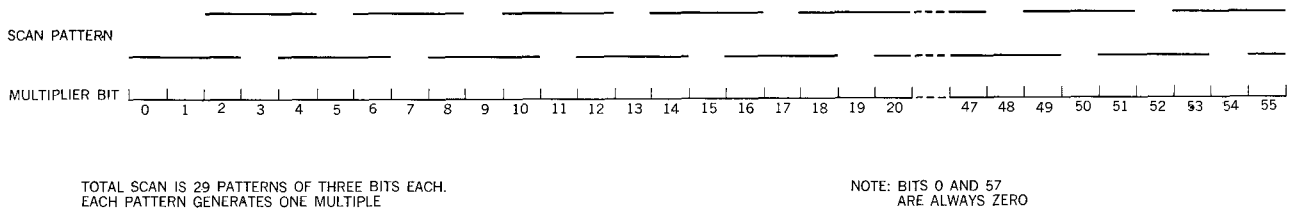


Figure 8 Scanning pattern for multiplier.

However, the method described above requires a variable shift and thus does not permit one to predict the exact number of cycles required to execute multiply. Furthermore, it does not permit the use of carry-save adders in the implementation. (Carry-save adders will be discussed later.)

A multiplier recoding-algorithm, which is based on the property described above, but which uses uniform shifts is used in the Model 91. The multiplier is divided into uniform groups of k bits each. These k bits are recoded to generate a multiple of the multiplicand, which is added to or subtracted from the partial product. The multiples are generated by shifting the position of the multiplicand in relation to the normal position at which it would enter the adder for a k equal to one. After adding the generated multiple to the partial product, the partial product is shifted k positions and the next group of k bits is considered.

The correct choice for k is important since an average of $1/2^k$ of the generated multiples will have a value of zero, and increasing k (over k equal to one) reduces the amount of operand reduction capability that is used inefficiently. However, if k is greater than two, carry propagate addition is necessary to generate the needed multiplicand multiples (shifting can only be used to generate multiples which are a power of two). In the context of a fast multiply, the carry-propagate adder increases the start-up time, which is undesirable. The Model 91 uses a k equal to two.

The technique used to scan the multiplier is shown in Fig. 8. Overlapping the high-order bit of one group and the low-order bit of the next group insures that the beginning and end of a string of 1's is detected once and only once. Table 2 shows which multiples are selected for all possible combinations of the two new bits and the overlapped bit.

Since the objective is fast multiply execution, six groups of multiplier bits are recoded at one time, and the resultant six multiples are added to the partial product. Five iterations are sufficient to assimilate the full 56 bits of the multiplier fraction. Figure 9 shows how the multiplier fraction is separated for each iteration and how each iteration is separated for the six generated multiples.

A tree of carry-save adders is used to reduce the generated multiples from six to two. A carry-save adder, which can be used whenever successive addition of several operands is necessary, requires less hardware, has less data skew and has less delay than a carry-propagate adder.[6] The individual carry-save adder takes three input operands and generates the resulting sum and carry. However, instead of connecting the carries to the next higher-order bits and allowing them to ripple, they are treated as independent outputs. In accordance with the customary rules for addition, the carries will be added to the next higher-order bits as separate inputs to the next carry-save adder down the tree.

Figure 10 illustrates a tree of carry-save adders which will reduce six input operands to two, thereby retiring 12 bits of the multiplier on each iteration. Note that the final output of the carry-save adder tree is two operands—sum and carry—which are shifted right 12 positions and loop back to become input operands. Thus, the partial product is accumulated as a partial sum and a partial carry. After the multiplier has been assimilated, these two operands, sum and carry, are added in a carry propagate adder to form the final product.

- *Implementation*

A block diagram of the data flow for the execution of a multiply is shown in Fig. 11. This data flow can be separated into two parts, the iterative hardware and the peripheral hardware (that hardware which is peripheral to the iterative hardware). The latter includes the input reservation stations, the pre-normalizer, the post-normalizer, the propagate adder, the result register, and the characteristic arithmetic. The peripheral hardware is described first, but since the iterative hardware is the heart of multiply execution, the major part of this section is devoted to a discussion of this hardware.

Input peripheral hardware

The input hardware includes the reservation stations, pre-normalizer, and the characteristic arithmetic. As was stated earlier, the multiply unit has two reservation stations and appears to the floating-point instruction unit for assign-

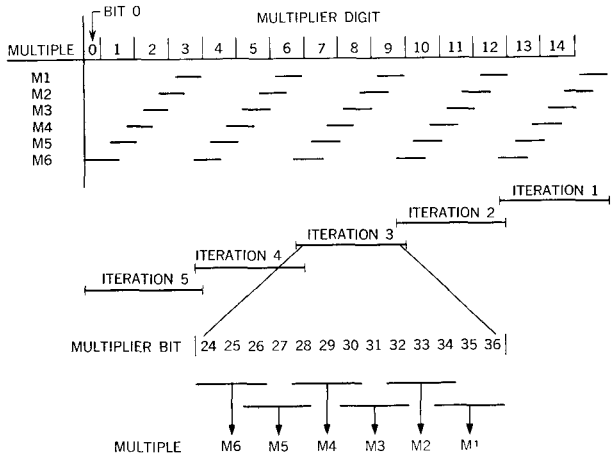


Figure 9 Iterations and multiple generation for multiply.

ment purposes as two distinct multiply units. If both units have been selected for a multiply operation, the first unit to receive both operands is given priority to begin execution. In the case where both units receive their second operand simultaneously, the unit which was selected by the floating-point execution unit first is given priority for execution.

The system architecture specifies that multiply is a normalized operation. Thus, if the input operands are unnormalized, they must be gated to the pre-normalizer, normalized, and then returned to the originating reservation station. In some cases, one additional machine cycle is added to the execution time for each unnormalized operand. However, normalization takes place as soon as the first operand enters the reservation station, provided there is not an operation in execution. Thus, normalizing can take place while the unit is waiting for the second operand.

The design of the zero digit detector and the left-shifter are similar to those described earlier for the add unit. If the zero digit detector, detects an all-zero fraction, the multiply is executed normally, but the outgate of the result to the floating-point accumulator is inhibited. Thus the required result, and all-zero-fraction, is stored.

The amount of left shifting necessary to normalize an operand is gated to the characteristic arithmetic logic, where the characteristic is updated for this shift. Characteristic arithmetic for multiply simply requires the two characteristics to be added but this operation can be overlapped with the execution of the multiply. Thus, the implementation is simple and straightforward.

It remains only to update the characteristic because of post-normalization. The post-shift can never be more than one digit because the input operands are normalized. Therefore, in order to eliminate logic levels at the end of multiply execution, two characteristics are generated: the normal resultant characteristic and the normal characteristic minus one. Subsequent to post-normalization the correct characteristic is outgated.

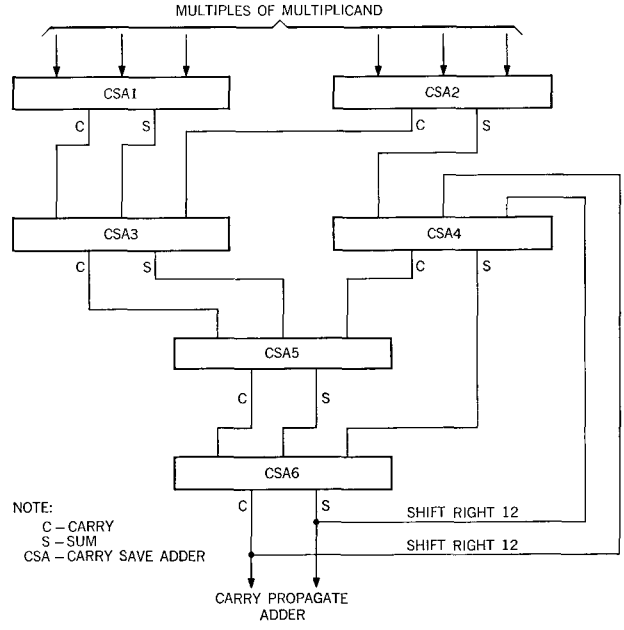
Output peripheral hardware

The output peripheral hardware includes the carry-propagate adder, the result register and the post-normalizer. Since the product is accumulated as two operands (sum and carry) the output of the iterative hardware is gated to a carry-propagate adder to form the final product. The design of the carry propagate adder is similar to the one used in the add unit with the exception that multiply does not require an end-around carry adder. A result register is created by latching the last level of the carry propagate adder. The output of the result register is gated to the common data bus via the post-normalizer. Detection of the need for post-normalization is done in parallel with the carry propagate adder and the result is gated to the common data bus, either shifted left one digit or unshifted.

Iterative hardware

The multiply execution area has conflicting design goals. The execution time must be short but the amount of hardware necessary for implementation has a practical upper limit. One could design a multiply unit which would take two cycles for execution. A large tree of twenty-eight carry-save adders could be interconnected so that the multiplicand and the multiplier would be the input to the tree and the output would be the product.[8] The performance of this multiply unit would be acceptable but

Figure 10 Carry-save adder tree.



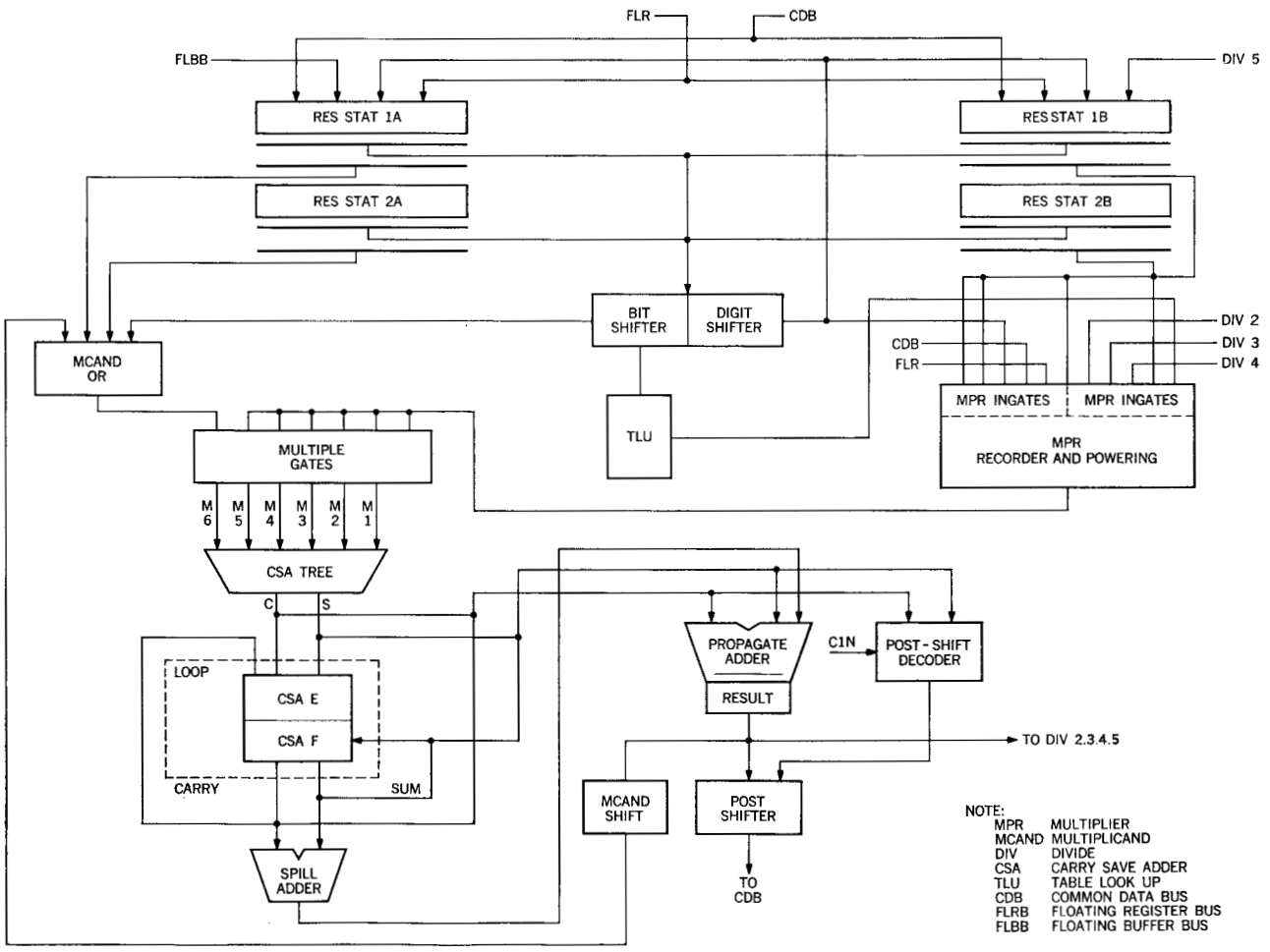


Figure 11 Floating-point multiply/divide data flow.

the amount of hardware necessary for implementation is much too high.

The adopted alternative approach was to select a subset of the carry-save adder tree such that one iteration through the tree retires 12 bits of the multiplier. This iteration is repeated until the full 56 bits of the multiplier have been exhausted. If each iteration is fast enough, the multiply execution time for this method approaches that for the large tree of carry-save adders. In fact, if each iteration can be 20 nanoseconds the second method can execute a multiply in three cycles, and the iterative hardware can be reduced to 20% of that required for the first method. Thus, with an iterative loop, the primary design problem is to design the carry-save adder tree so that the iteration period is minimized. The faster the repetition rate of the iterative hardware, the better the cost-performance ratio of the multiply area.

There are several ways to arrange the carry-save adders, and each method affects the iteration period differently. For example, if they are arranged as shown in Fig. 12, the feedback loop (the partial product) is from the output back to the input. In this case, the iteration period becomes the time required to make one complete pass through the tree. However, the adopted arrangement, shown in Fig. 13, allows the iteration period to approach the delay through the last carry-save adders (these two carry-save adders are accumulating the partial product). But the delay through the path leading to the last two carry-save adders (the multiplier recoding, multiple generation and the first four carry-save adders) is much longer than the delay through the adders. If, however, temporary storage platforms are inserted in the iterative loop the concept of pipelining, explained earlier, can be put to use here. Temporary storage platforms are inserted in the iterative hardware for deskewing so that the rate of inserting new inputs (twelve bits of the multiplier) and the

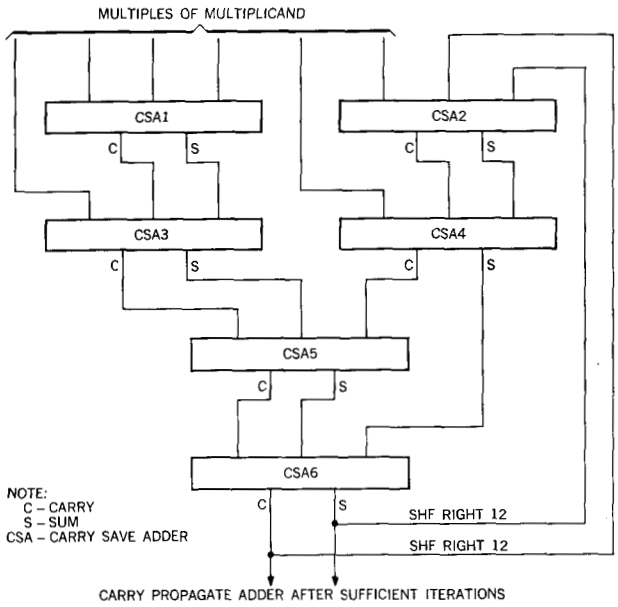


Figure 12 CSA tree with feedback loop from output.

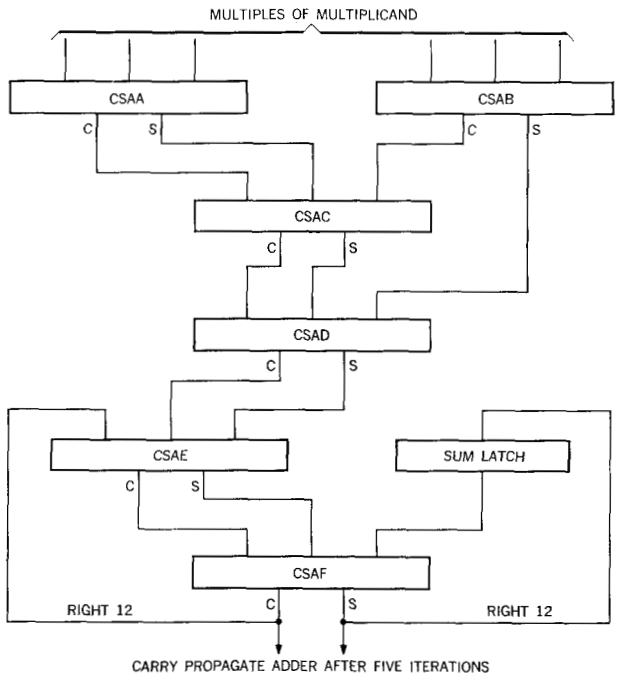


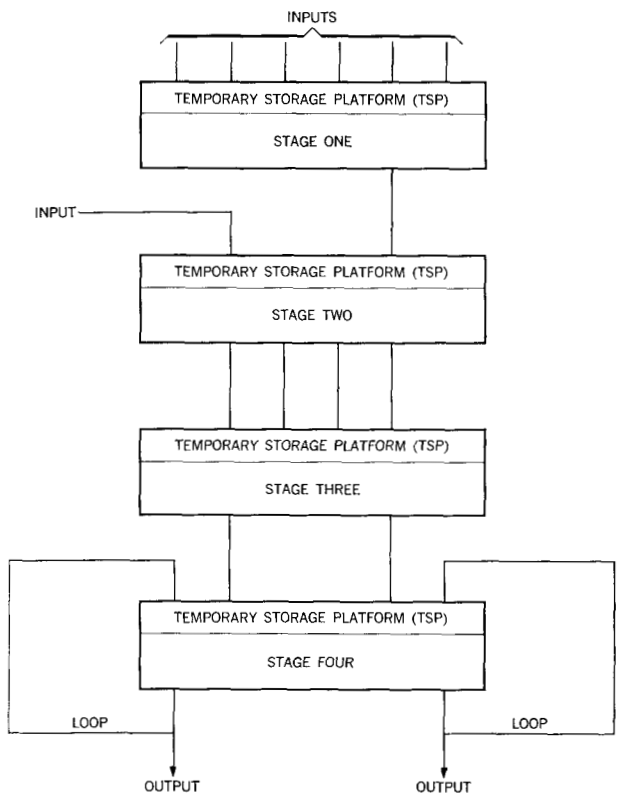
Figure 13 CSA tree with accumulating loop at output.

rate of accumulating the partial product may safely be made equal. Therefore, by pipelining the carry-save adder tree, the second arrangement can be used and the iteration period is equal to the delay through the last two carry-save adders.

In order to explain the pipelined tree, the path is abstracted in Fig. 14. Each block represents the logic associated with the stages of the pipeline and the first level of each block represents the temporary storage platform. The period of the clock is set by the logic delay of the accumulating loop. In the abstract design the logic delay of all paths between stages of the pipeline is assumed to be the same as the clock period.

Figure 15 is a timing diagram for the abstracted iterative hardware. At clock time zero, the first input, I_1, is gated into the temporary storage in stage one. At clock time one, I_1, after being operated on by the logic in stage one, is gated in at stage two and I_2 is gated in at stage one. This process continues until at clock time three, the original input, I_1, is entering stage four. During this clock time, the pipeline is filled, i.e., each stage of the pipeline now contains data in various forms of completion. At clock time four, the last input, I_5, enters stage one, and the partial product starts to accumulate at stage four. The next three clock times are used to drain the pipeline and accumulate the full partial product. Thus the total iterative loop time is that necessary to fill up the carry-save adder

Figure 14 Abstract drawing of "pipelined" iteration.



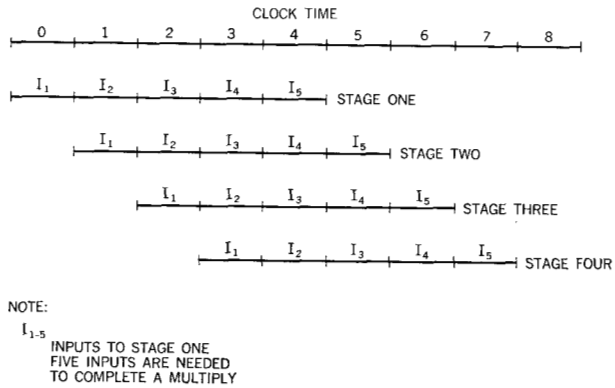


Figure 15 Timing diagram for abstracted iterative hardware.

tree plus five passes around the accumulating loop, or eight clock periods. If the feedback loop were from output to input, as shown in Fig. 12, the total iterative loop time would be twenty clock periods. Therefore the iterative loop time has been reduced by a factor of 2.5, with only a small increase in hardware. (This is described later.)

The actual implementation of the pipeline is not simple. First, the temporary storage platforms require extra hardware and add delay to the path. Second, the placement of the temporary storage platforms is important for two reasons: (1) The purpose of the temporary storage platform is to deskew the logic (difference between fast and slow logic paths) and the logic delay is not ideally distributed, and (2) the placement can affect the amount of hardware necessary for implementation.

The solution to the first problem led to a design in which the logic function was designed into the temporary platform; e.g., a latched carry-save adder or a latched multiple gate. The extra hardware is only that required for the feedback loop which latches the logic function; the added delay is eliminated because the logic function is designed into the temporary storage. The solution to the second problem was more complex. First, the clock used to control the temporary storage platform ingate was designed as a series clock. All of the pulses of an iteration are initiated by a single oscillator pulse and then delayed to drive the ingates of the successive pipeline stages. The clock delay between successive temporary storage ingates is equal to the long path circuit and wiring delay of the logic between these ingates. The time between iterations (the oscillator period) is still the delay of the accumulating loop, but the time between pipeline stages is not equal to the clock period. This allows the placement of temporary storage to vary without being dependent on the clock.

The relationship between the logic skew and clock period can be expressed as

Short path > [long path − clock period] + gate width,

where short path is the shortest logic delay between two temporary storage platforms; long path is the longest logic delay between two temporary storage platforms; and gate width is the time necessary to set and latch the temporary storage platform.

The temporary storage platforms were placed to minimize the hardware; then a careful data path analysis was made to determine the logic skew. The above relationship was next applied and the short paths "padded" with additional delay to satisfy the relationship. The result is shown in Fig. 16. The temporary storage platforms are at the multiplier recorder, the multiple gates, carry-save adder C and the accumulating loop, and carry-save adders E and F.

Since the design goal was to make the iteration period as short as possible, the design of the last two carry-save adders required a minimum number of levels and was constrained to account for the "short path around the loop." Carry-save adders E and F are each designed as a temporary storage platform and are orthogonal—i.e., are not ingated simultaneously. The first, carry-save adder E, is ingated on the first-half of the clock period and the second, carry-save adder F, is ingated on the second half of the clock period.

The low order thirteen bits of the multiplier are gated into the latched multiplier recoder at clock time zero and recoded to six control lines. Every clock period—20 nanoseconds—a new set of bits is gated into the multiplier recoder until the full word (56 bits) is exhausted. The next step in the pipeline is the latched multiple gates. Six multiples are generated by shifting the multiplicand, under control of the output from the multiplier recoder. These six multiples are reduced to four (two sums and two carries) by carry-save adders (CSA) A and B. Carry-save adder C takes three of these outputs and reduces them to two latched outputs. The sum from CSA-B is latched in parallel with CSA-C and combines with the two outputs from CSA-C to provide CSA-D with three inputs. At the output of CSA-D, the sum and carry are the result of multiplying twelve bits of the multiplier and the full multiplicand. The next two latched carry-save adders are used to accumulate the partial product. Each iteration adds the latest sum and carry from CSA-D to the previous results. After five iterations of the accumulating loop the output of CSA-F is the bit product in carry-save form. Now the sum and carry operands are gated to the carry propagate adder and the carries allowed to ripple to form the final product.

- *Divide algorithm*

Several division algorithms exist,[1,6] of varying complexity, cost and performance, which could be used to execute the divide instruction in the Model 91. But because of the relatively complex and iterative nature of divide

algorithms, the execution time is out of balance with other processor functions. Even the higher-performing conventional algorithms contain a shortcoming which requires that successive subtractions be separated by a performance-degrading decode interval.* The Model 91, however, utilizes a unique divide algorithm which is based on quadratic convergence.[7,8,9,10] A major advantage is that the number of required iterations is reduced (proportional to $\log_2$ of the fraction length), which reduces the number of data-control interactions. Another important advantage is that MULTIPLY is the basic iterative operator. This both reduces the cost, by exploiting existing hardware, and enhances the execution time, because in the Model 91 MULTIPLY is extremely fast.

The divisor and dividend are considered to be the denominator and numerator of a fraction. On each interation a factor, R_k, multiplies both numerator and denominator so that the resultant denominator converges quadratically toward one (1) and the resultant numerator converges quadratically toward the desired quotient.

$$\frac{N}{D} \times \frac{R}{R} \times \frac{R_1}{R_1} \times \frac{R_2}{R_2} \times \cdots \times \frac{R_n}{R_n}$$

$$\Rightarrow NRR_1 \cdots R_n = \text{Quotient},$$

where N = numerator = dividend,
D = denominator = divisor, and
$D\,R\,R_1\,R_2 \cdots R_n \Rightarrow 1$.

The selection of the factor R_k is the essential part of the procedure and is based on the following: The divisor can be expressed as

$$D = 1 - x,$$

where $x \leq 1/2$ since D is a bit-normalized, binary floating-point fraction of the form

$$0.1\ \text{xxx} \cdots.$$

Now, if the factor R is set equal to $1 + x$ and the denominator is multiplied by R

$$D_1 = DR = (1 - x)(1 + x) = 1 - x^2,$$

where $x^2 \leq 1/4$, since $x \leq 1/2$.

The new denominator is guaranteed to have the form $0.11\ \text{xxxx} \cdots$. Likewise, selecting $R_1 = 1 + x^2$ will double the leading 1 on the next iteration to yield

$$D_2 = D_1R_1 = (1 - x^2)(1 + x^2) = 1 - x^4$$

$$= 0.1111\text{xxxx} \cdots,$$

where $x^4 \leq 1/16$ since $x \leq 1/2$.

* Conventional refers to previous division algorithms which use subtraction as the iterative operator. The faster algorithms generate more than one quotient bit in parallel through the use of pre-wired multiplies. However, the selection of the multiplies for the next iteration is dependent upon a decode of the partial remainder of the previous iteration.

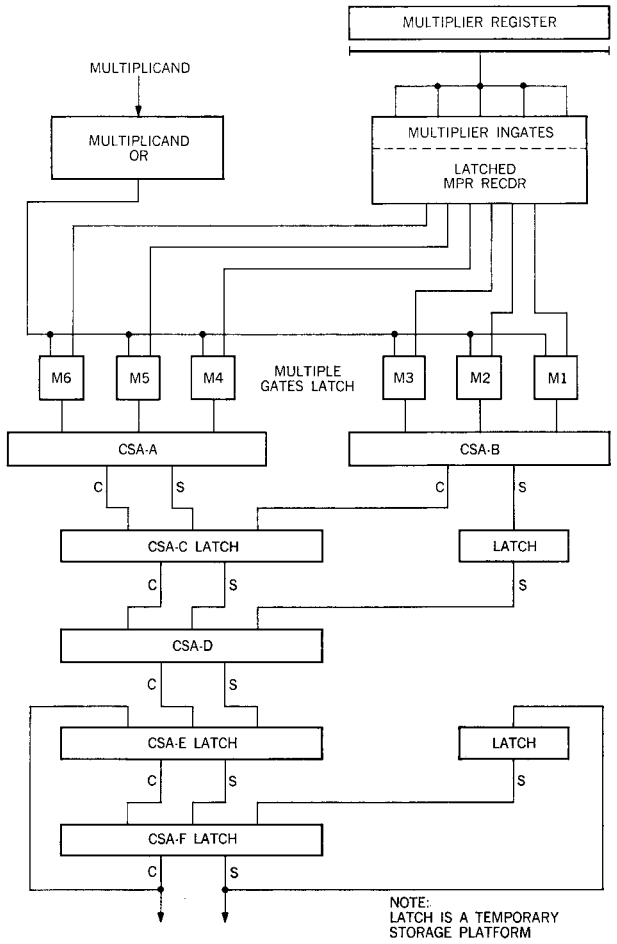


Figure 16 Multiply iterative loop showing temporary storage.

In general, if $x_k < 1/2^n$ then $x_{k+1} < 1/2^{2n}$. Thus, by continuing the multiplication until x_k is less than the least significant bit of the denominator (divisor fraction), the desired result, namely a denominator equivalent to one $(0.1111111 \cdots 111)$, is obtained.

It is important to note that the multiplier for each iteration is the two's complement of the denominator,

$$R_{k+1} = 2 - D_k = 2 - (1 - x_n) = 1 + x_n.$$

Thus the multiplier for iteration k is formed by taking the two's complement of the result of iteration $(k - 1)$. However, in this form the algorithm is still not fast enough. For a 56-bit fraction, eleven multiples are required with a two's complement inserted between six of the multiples:

$$Q = NRR_1R_2R_3R_4R_5$$

$$R_5 = 2 - D_4 \quad \text{and} \quad D_4 = DRR_1R_2R_3R_4.$$

Table 2 Multiplier recoder rules.

Input			Output multiple	Reason
n*	$(n+1)$	$(n+2)$		
0	0	0	0	No string
0	0	1	+2	End of string
0	1	0	+2	Beginning and end
0	1	1	+4	End of string
1	0	0	−4	Beginning of string
1	0	1	−2	Beginning and end
1	1	0	−2	Beginning of string
1	1	1	0	Center of string

* Bit n is the high-order position

But if the number of bits in the multiplier could be reduced, the time for each multiply would be decreased. If in order to obtain n bits of convergence the multiplier is truncated to n bits [$1 + x_T$ where $(x_T - x) < 2^{-n}$] it can be shown that the resultant denominator is equivalent to

$$(1 + x_T)(1 - x) = 1 - x^2 + |T|,$$

where $0 < T$ (which is due to truncation) $< 2^{-n}$.

Because the additional term T is always positive, the resultant denominator can now have two forms:

$$D_k = \begin{cases} 0.11111 \cdots \text{xxxxx} \cdots \\ 1.00000 \cdots \text{xxxxx} \cdots \end{cases}.$$

The denominator can converge toward unity from above or below, but it will converge, so no additional problems are encountered.

Therefore, the number of bits in the multiplier can be reduced to the string bits (all 0 or all 1) and the number of bits of convergence desired. The string bits, since they are all 0 or all 1, can be skipped in the multiply. Thus the multiply time has been improved considerably and so, consequently, has the divide time. To improve the initial minimum string length, thus reducing the number of iterations, the first multiplier, R, is generated by a table-lookup which inspects the first seven bits of the divisor. The first multiply guarantees a result which has seven similar bits to the right of the binary point ($1 \pm x$ has the form $\bar{a}.aaaaaaa \cdots$ etc.).

The following sequence outlines the operations which result in the execution of a divide.

1. Bit normalize the divisor and shift the dividend accordingly.
2. Determine the first multiplier, R, by a table-lookup.
3. Multiply D by R forming D_1.
4. Multiply N by R forming N_1.
5. Truncate D_k and complement to form R_k.
6. Multiply D_k by R_k forming D_{k+1}.
7. Multiply N_k by R_k forming N_{k+1}.
8. Iterate on 5, 6 and 7 until $D_{k+n} \Rightarrow 1$ and then N_{k+n} = Quotient.

• *Divide implementation*

Each iteration of divide execution consists of three operations as shown above. The problem in implementation is to accomplish these three operations utilizing the multiply hardware described previously and accomplish them in the minimum amount of time. But there are three points which create difficulty. First, the multiplier is a variable length operand, the length being different on each iteration. The first multiplier, determined by table-lookup, is ten bits and yields a minimum string length of seven; the second multiplier is fourteen bits; the third multiplier is twenty-eight bits, etc. In other words, the minimum string length can be doubled on each iteration after the first. Second, the result of one iteration is the multiplicand for the next iteration. Since the output of the multiply iterative hardware is two operands—carry and sum—the carry propagate adder must be included in the divide loop. Third, two multiplies are required in each iteration—one determines what to do on the next iteration (multiplier × denominator) and one converges the numerator towards the quotient (multiplier × numerator).

When all three of these points are considered simultaneously they present a dilemma. Since two multiplies are necessary it is desirable to overlap the two and save time, but any multiply for which the multiplier is greater than twelve bits requires that the carry-save adder loop be used. Also, the fact that the carry propagate adder must be included in the loop lengthens the time for each iteration. Several design iterations were required before arriving at the correct solution.

First consider the entries in Table 2 and note that the leading string of 1's or 0's in the multiplier can be skipped since they result in a zero multiple out of the multiplier recoder. Also, if the input of the multiplier recoder is complemented the sign of the output changes but the magnitude remains the same. Thus, this property can be used to produce $\mp x_n$ at the output of the recoder.

Next consider a multiplier (complement of truncated denominator) such as the following:

$$\begin{array}{lllll} 1. & 0000 & 0000 & 000\left[0\right. & 00xx \quad xxxx \quad xxx1\left.\right] \\ 0. & 1111 & 1111 & 111\left[1\right. & 11xx \quad xxxx \quad xxx1\left.\right] \end{array}$$

If *all* positions were recoded, a bit of value 1 would be recoded from the high-order end and a set of bits of value $\mp x_k$ from the right end ($1 \pm x_k$). However, if only the

Table 3 Formats of the denominators and their multipliers.

Digit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
D	0	1xxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	0000	0000
R	[01	xxxx	xxxx	xx0]	Determined by table lookup of denominator												
$D \times R = D_1$	0	1111	111x	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx
	1	0000	000x	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx
R_1	1	0000	[000$\bar{x}$	$\bar{x}\bar{x}\bar{x}\bar{x}$	$\bar{x}\bar{x}11$	1]	Determined by complementing denominator										
	0	1111	[111$\bar{x}$	$\bar{x}\bar{x}\bar{x}\bar{x}$	$\bar{x}\bar{x}11$	1]											
$D_1 \times R_1 = D_2$	0	1111	1111	1111	11xx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx
	1	0000	0000	0000	00xx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx
R_2	1	0000	0000	0000	00$\bar{x}\bar{x}$	$\bar{x}\bar{x}\bar{x}\bar{x}$	$\bar{x}\bar{x}\bar{x}$1]										
	0	1111	1111	1111	11$\bar{x}\bar{x}$	$\bar{x}\bar{x}\bar{x}\bar{x}$	$\bar{x}\bar{x}\bar{x}$1]										
$D_2 \times R_2 = D_3$	0	1111	1111	1111	1111	1111	111x	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx
	1	0000	0000	0000	0000	0000	000x	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx
R_3	1	0000	0000	0000	0000	0000	[000$\bar{x}$	$\bar{x}\bar{x}\bar{x}\bar{x}$	$\bar{x}\bar{x}\bar{x}\bar{x}$	1]							
	0	1111	1111	1111	1111	1111	[111$\bar{x}$	$\bar{x}\bar{x}\bar{x}\bar{x}$	$\bar{x}\bar{x}\bar{x}\bar{x}$	1]							
$D_3 \times R_3 = D_4$	0	1111	1111	1111	1111	1111	1111	1111	1111	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx
	1	0000	0000	0000	0000	0000	0000	0000	0000	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx
R_4	1	0000	0000	0000	0000	0000	0000	0000	0[000	$\overline{xxxx}$	$\overline{xxxx}$	$\overline{xxxx}$	$\overline{xxxx}$	$\overline{xxxx}$	$\overline{xxxx}$	$\overline{xxxx}$	$\overline{xxxx}$	11]
	0	1111	1111	1111	1111	1111	1111	1111	1[111	$\overline{xxxx}$	$\overline{xxxx}$	$\overline{xxxx}$	$\overline{xxxx}$	$\overline{xxxx}$	$\overline{xxxx}$	$\overline{xxxx}$	$\overline{xxxx}$	11]
D_5 (not formed)	0	1111	1111	1111	1111	1111	1111	1111	1111	1111	1111	1111	1111	1111	1111	1111	1111	
	1	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	

Short precision divide result is $N_4 = NRR_1R_2R_3$
Long precision divide result is $N_5 = N_4R_4$

portion in brackets is gated to the recoder the output will have value $\mp 2^{12}x_k$.* The bits in the bracket are chosen such that the left-most three bits are identical. Thus, multiple six (refer to Fig. 9) is not used because a zero multiple is always recoded, and the product $\mp D_k x_k$ or $\mp N_k x_k$ is accomplished by the five operands gated to multiple gates one through five. If the unshifted multiplicand is gated simultaneously into the sixth multiple gate the sum of all six operands is $D_k + (\mp D_k x_k)$ or $D_k(1 \mp x_k)$, which is the desired result. The result which is generated by adding the carry and sum out of the carry-save adder tree (refer to Fig. 11) is the following:

$$D_{k+1} = \begin{cases} 0.\ 1111\ 1111\ 1111\ 1111\ 1111\ 111x\ xxxx \rightarrow \\ 1.\ 0000\ 0000\ 0000\ 0000\ 0000\ 000x\ xxxx \rightarrow \end{cases}$$

Thus, without using the carry-save adder loop the leading string has been increased by nine bits.

Table 3 presents the format of the multipliers and their denominators. Notice that the first multiplier is ten bits and the second is seven. These are fixed and cannot be changed without making the table-lookup decoder larger. Thus the third multiplier is the first one capable of using more than nine bits. But if a multiplier of more than nine bits is used, the carry-save adder loop must be included in the divide loop. Since this is undesirable (concurrency among multiplies is discussed below) the multiplier for the third and fourth iterations is chosen to be nine bits, thereby increasing the string length by nine each time. Thus, D_4 has 32 leading 1's or 0's. Now if D_4 is multiplied by multiplier four, R_4, the result will be 64 leading 1's or 0's, which is equivalent to unity within the desired accuracy. Therefore, since it is not necessary to calculate multiplier five, R_5, this multiply is not done and since only the numerator is going to be multiplied by multiplier four, the carry-save adder loop is used to speed up this last operation. (This is discussed more fully below.)

The second difficulty, which was that the carry propagate adder must be included in the path, was used to solve the third difficulty. Consider Fig. 17, which is the divide loop. To begin the execution of a divide the divisor is multiplied by the first multiplier (R), and the first denominator (D_1) is generated at the output of the CSA tree. These two outputs are added in the carry propagate adder; the output loops back to the input and becomes the new multiplicand; the truncated and complemented output forms the new multiplier. Note that the complete loop contains two temporary storage platforms—one at CSA-C and one at

* The multiplicand is shifted right twelve positions to compensate for the 2^{12} factor.

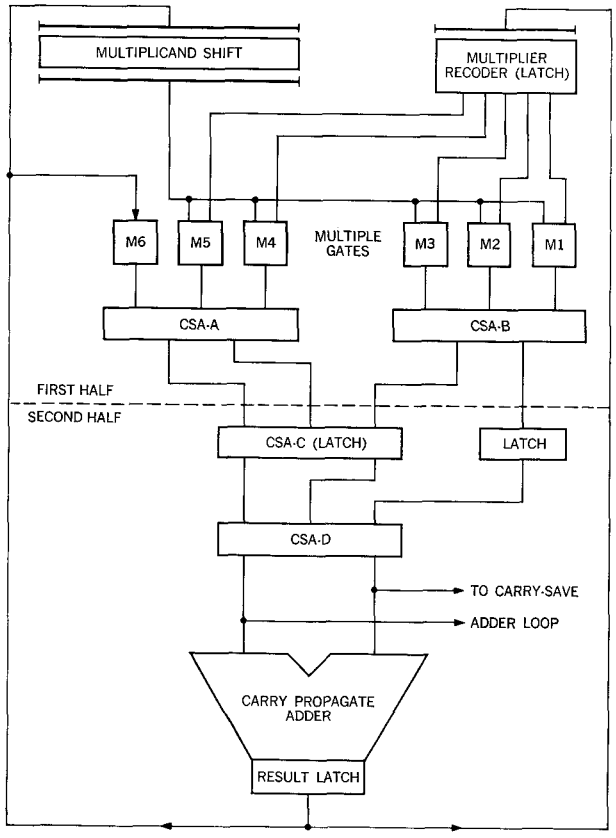


Figure 17 Divide loop.

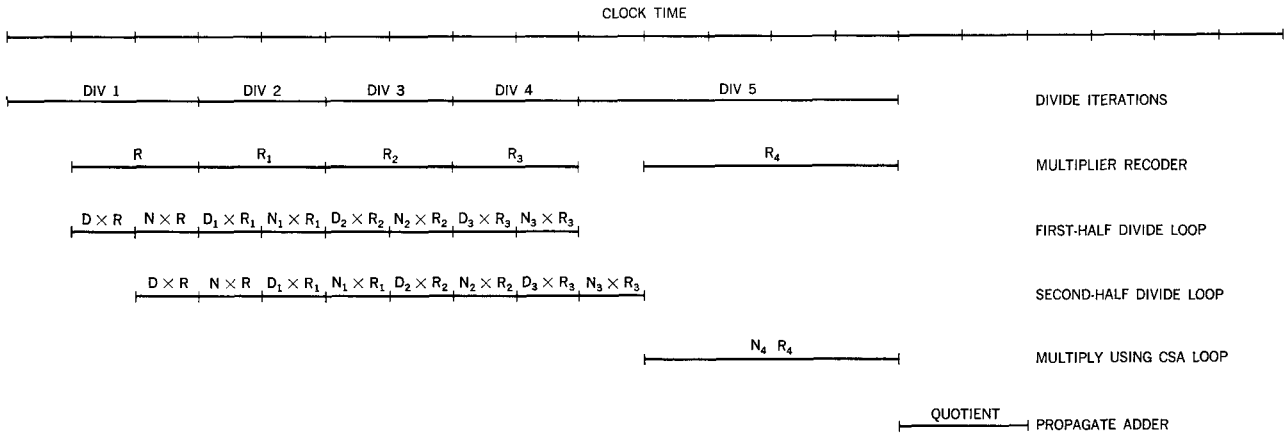
the output of the propagate adder, the result latch. Thus as soon as $R \times D$ is gated into CSA-C, the next multiply, $R \times N$, can be started. Now $R \times D$ advances to the result latch and loops back to start the next multiply $R_1 \times D_1$. At this time $R \times N$, which is latched in CSA-C, advances through the adder to the result latch. So the two multiplies follow each other around the divide loop. The first determines what the second should be multiplied by to converge eventually to the quotient.

This chain continues until multiplier four has been calculated. Since denominator five is equivalent to one, the multiply is not done. The 32-bit multiplier is gated into the reservation station and then gated to the multiplier twelve bits at a time as shown in Table 3. The result of this multiply, N_4R_4, is the final quotient. The diagram in in Fig. 18 shows the concurrency in the divide loop. The multiplier recoder latch is changed each time a denominator multiply is completed. Notice that two multiplies are always in execution, one in the first half of the divide loop (from input to CSA-C) and one in the second half of the divide loop (from CSA-C to the result latch).

Conclusions

The prime effort during the design of the floating-point execution unit was to develop an organization which would achieve a balance between instruction execution and preparation. Early in the design phase it appeared that an organization which would achieve this result would have a poor cost-performance ratio.

Figure 18 Timing diagram showing concurrency in divide loop.



Concurrency, obviously, had to be the key to high performance, but the connotation of concurrency in computers is parallel execution of different instructions. Thus the early organizations exhibited more than one execution unit and a high cost. In the final organization, concurrency is the key to the high performance, but this organization exhibits several levels of concurrency:

1. Concurrent execution among instruction classes.
2. Concurrent execution among instructions in the same class (add unit).
3. Concurrent execution within an instruction (multiply iterative hardware and divide loop).

The concepts of instruction-oriented units and reservation stations were used to keep the performance level sufficiently high but reduce the cost. These two concepts yield the same performance as several units without the cost of several units. The instruction-oriented units allow the design to be hand-tailored for faster execution and permit the use of a unique algorithm to execute divide.

Acknowledgments

The design of a computer unit such as this—containing nearly as many logical decisions as IBM's previous largest central processor—requires a great deal of decision making. The authors gratefully acknowledge the logical and engineering design contributions made by the following individuals: Mr. W. D. Silkman for the floating-point instruction unit; Messrs. J. J. DeMacedo, J. G. Gasparini, L. Grosman, R. C. Letteney and R. M. Wade for the multiply/divide unit; Messrs. M. Litwak, K. J. Pockett and K. G. Tan for the add unit; and Mr. E. C. Layden for the processor clock.

Acknowledgment is also made for the early planning efforts of Mr. R. J. Litwiller.

References

1. W. Buchholtz et al., *Planning a Computer System*, McGraw-Hill Publishing Co., New York, 1962.
2. G. M. Amdahl, G. A. Blaauw and F. P. Brooks, Jr., "Architecture of the IBM System/360," *IBM Journal* **8**, 87 (1964).
3. D. W. Anderson, et al., "Model 91 Machine Philosophy and Instruction Handling," *IBM Journal* **11**, 8 (1967) (this issue).
4. R. M. Tomasulo, "An Efficient Algorithm for Exploiting Multiple Arithmetic Units," *IBM Journal* **11**, 25 (1967) (this issue).
5. R. F. Sechler, A. K. Strube and J. R. Turnbull, "ASLT Circuit Design," *IBM Journal* 11, 74 (1967) (this issue).
6. O. L. MacSorley, "High Speed Arithmetic in Binary Computers," *Proc. IRE* 49, 67, (1961).
7. R. E. Goldschmidt, "Applications of Division by Convergence," Master's Thesis, MIT, June 1964.
8. C. S. Wallace, "A Suggestion for a Fast Multiplier," *Trans. IEEE*, **EC-13**, 14–17 (1964).
9. M. V. Wilkes et al., *Preparation of Programs for an Electronic Digital Computer*, Addison-Wesley Publishing Co., Cambridge, Mass., 1951.
10. T. C. Chen, "Fast Division Scheme," private communication, November 4, 1963.

Received November 1, 1965.