This paper analyzes the addition operation of floating-point systems.

The analysis of a million executed floating-point additions is presented as an aid in optimizing design and measuring performance.

The frequency of the various shifts for floating-point additions with different radices was derived from the basic data so that designs with various radices may be evaluated.

An analysis of floating-point addition

by D. W. Sweeney

The major problem in designing a floating-point system is the addition operation. Knowledge of the relative importance of each of the suboperations can assist the designer in materially improving performance. Of special import are the provisions for shifting in pre-addition *alignment* of the operand radix points and in post-addition *normalization* of the result. Since these shifts vary over a wide range, provision must be made for the worst cases, without requiring the time of these extremes for all cases.

This paper provides data on the amount and type of shifting related to the floating-point operation. The data were collected by tracing the execution of a representative set of problems. An analysis of the results and some comments on design choices are included.

floating-point notation

Instead of placing the radix point in a number, scientific notation often employs an exponent which indicates the displacement of the radix point from a fixed position, e.g., the number -278.437 is represented as -2.78437×10^{2}, and the number $+.000743$ is represented as $+7.43 \times 10^{-4}$. In a computer, these numbers are conveniently represented as $+03-27843700$, and $-03+74300000$, the first two digits being the *exponent*, and the last eight digits the *fraction*. The radix (in this example, 10) and the position of the radix point (to the left of the most significant fraction digit) are normally fixed for a given arithmetic unit. By restricting the exponent to be less than some modulus and adding it to that modulus, the exponent can be treated as a non-negative integer.

For example, if the modulus is 50, +03−27843700 becomes −5327843700 and −03+74300000 becomes +4774300000. Thus, a representation is achieved that looks like a fixed-point number to all parts of the computing system (I/O equipment, storage, data flow paths, registers, etc.), except for the specialized controls for floating-point operations.

The above examples are of normalized numbers, i.e., the fraction contains a non-zero digit immediately to the right of the radix point. When fractions are normalized, they obviously can represent a number with the greatest precision. Therefore, most computer programs are written in such a manner that all data are handled in a normalized mode. However, there are times when it is convenient to use representations with leading zero digits (unnormalized fractions) and to perform unnormalized operations, i.e., leading zeros are not removed from the result. Unnormalized representations of +.000743, for example, are: +4807430000 with one leading zero and 4900743000 with two leading zeros, etc.[1]

floating-point addition

The rules for floating-point addition can be simply stated: align the actual radix points and add the operands; then, if necessary, move the radix point and adjust the exponent so that the result is in "conventional" form (usually the normalized form). Because of the limit on precision imposed by the computer word length, additional rules must also be stated to handle the cases where precision is exceeded. On alignment, for example, the shifting and addition would serve no purpose if the exponent difference were greater than the number of bits or digits in the fraction. Therefore, time can be saved by recognizing and bypassing this event. Result exceptions can also pose problems. If the final exponent exceeds its largest allowable value (exponent overflow) or its smallest allowable value (exponent underflow), or if the result fraction is zero, the definition of the result is arbitrary. These events should be indicated in a manner similar to fixed-point overflow or improper division. If an interrupt system is part of the design, these events should be monitored. Even if these results are indicated, the design should, if possible, provide a definition that propagates these results through succeeding operations. If the exponent overflows, it should be treated as "infinity." If the exponent underflows or the result fraction is zero, the result should be treated as "zero."

Some computer designers do not redefine the result if the fraction is zero, preferring to leave the result as an order-of-magnitude zero. In some instances, this is valuable for subsequent analysis, although it may cause considerable difficulty. Consider the least-squares problem $\sum (x - x_i)^2$, where the x's are large (say about 10^3), and the deviations are small (say about 10^{-3}). The value of the sum is destroyed if one value has a result fraction of zero, since the relatively large remaining exponent is doubled (say 10^6) and the valid part of the sum (which is on the order of 10^{-6}) is shifted out of range. Similar difficulties can be cited for other applications. In the opinion of the author, a number should be provided that behaves like zero in the floating-point operations,

and zero fractional results and exponent underflows should be set to this "zero."

Other limitations inherent in the data flow paths and the register-adder complex, as defined for other operations, can also pose implementation problems. The greatest difficulty for the designer lies in determining the appropriate compromises between cost and speed when he is designing actual hardware, or between storage space and speed when he is designing a floating-point subroutine.[2] To assist the designer in his choice, relevant data are needed with regard to the frequency of floating-point additions, the various amounts of shifting, the number of times the shift amounts exceed the precision representable, and the frequency of recomplementation, etc.

data gathering

The frequency of the various shifts depends on the radix of the computer, the type of problem, and the data associated with the problem. This frequency is relatively independent of the method by which a particular computer executes the floating-point addition, except for certain small differences in handling extreme cases. It is important to obtain representative problems with realistic data and examine the types of shifting encountered during execution of these problems.

The study involved two phases. The first phase contributed to the design of a particular binary computer.[3] This design involved a shifter that could shift a few places (say 0 to 6) very quickly. Long shifts could then be done by taking multiple jumps (of 6), with the excess (1 to 5) being done in the same manner as a short shift. The problem was to determine if this design was sufficiently fast or if a larger shifter had to be considered.

Six representative problems were selected from those processed by an IBM 704 at a computational laboratory serving a wide variety of engineers and scientists. A tracing program already available for the 704 was modified to gather the data. One of these problems was traced twice with different input data. If an entire problem could not be traced, the process was terminated after tracing about 250,000 floating additions. These data sufficed to show that the chosen design was adequate.

The second phase of the study was undertaken to confirm the results of the first phase by gathering more data and examining different problems. The study was expanded to include the analysis of floating-point additions with radices greater than two. Again, six problems were analyzed, this time selected from several laboratories. One of the problems was traced twice with different input data. In this phase of the study, the analysis was terminated after tracing about 500,000 instructions (unless the problem had been completed earlier). The number of instructions traced, and the number and percentage of floating-point additions included are shown in Table 1.

A frequency count of the amount of alignment and normalization shifting was kept for each problem traced. In summarizing these data, it was decided to express each shift as a percentage of

operand signs differ. These results show that almost all operands are normalized, that most exponent differences are small (55 percent are less than 4), and that there is very little normalization (86 percent are less than 2).

Cases with unnormalized operands can be constructed in which the total shifting is as great or greater than twice the length of the fraction. With normalized operands, the total shift need not be greater than the length of the fraction plus one position, since no more than one bit of the shifted operand is shifted back into the result if normalization is required. This is shown by the following analysis. The fractional part of a normalized operand is in the range $1/2 \leq |f| < 1$, and since each alignment shift halves this range, the possible results must be as shown in Table 4.

If signs are alike, there can only be two possible results, regardless of the amount of alignment shifting. Either an overflow occurs or no normalization is necessary. If the signs are unlike and there is no alignment shift, the result sign may change. Also, any amount

Table 2 Weighted data on alignment and normalization in percentages

Shift	*Equal weight by shift*		*Equal weight by shift and problem*	
	Alignment	*Normalization*	*Alignment*	*Normalization*
result 0	—	0.39	—	0.82
overflow	—	23.30	—	17.43
0	24.29	58.04	26.28	60.25
1	17.14	8.42	13.29	8.01
2	10.59	2.69	8.77	3.17
3	7.64	1.42	6.53	1.54
4	6.55	0.89	7.77	1.33
5	6.44	0.70	8.07	1.03
6	5.81	0.56	5.13	0.63
7	4.50	0.56	3.78	1.02
8	0.90	0.40	1.10	0.93
9	1.05	0.31	1.23	0.56
10	0.96	0.23	1.84	0.66
11	0.78	0.13	1.35	0.16
12	0.99	0.25	1.54	0.22
13	0.95	0.35	0.81	0.31
14	0.54	0.13	0.48	0.16
15	0.60	0.04	0.58	0.03
16	0.38	0.03	0.29	0.06
17	0.38	0.07	0.31	0.09
18	0.43	0.09	0.50	0.17
19	0.34	0.09	0.32	0.15
20	0.27	0.10	0.26	0.08
21	0.37	0.09	0.40	0.07
22	0.28	0.07	0.30	0.07
23	0.22	0.10	0.24	0.19
24	0.23	0.15	0.25	0.49
25	0.24	0.10	0.26	0.09
26	0.19	0.30	0.16	0.28
27–53	0.73	—	0.86	—
54–255	6.21	—	7.30	—

Table 3 Shifting as a function of the effective signs

Shift	Like signs		Unlike signs		Total	
	Alignment	*Normalization*	*Alignment*	*Normalization*	*Alignment*	*Normalization*
result 0	—	—	—	0.82	—	0.82
overflow	—	17.43	—	—	—	17.43
0	12.54	29.66	13.74	30.59	26.28	60.25
1	7.07	0.01	6.22	8.00	13.29	8.01
2	3.93	0.00	4.84	3.17	8.77	3.17
3	2.89	0.02	3.64	1.52	6.53	1.54
4	2.98	0.01	4.79	1.32	7.77	1.33
5	4.51	0.01	3.56	1.02	8.07	1.03
6	2.85	0.01	2.28	0.62	5.13	0.63
7	1.86	0.02	1.92	1.00	3.78	1.02
8	0.44	0.00	0.66	0.93	1.10	0.93
9	0.60	0.00	0.63	0.56	1.23	0.56
10	0.94	0.00	0.90	0.66	1.84	0.66
11	0.63	0.01	0.72	0.15	1.35	0.16
12	0.83	0.01	0.71	0.21	1.54	0.22
13	0.39	0.00	0.42	0.31	0.81	0.31
14	0.12	0.00	0.36	0.16	0.48	0.16
15	0.36	0.01	0.22	0.02	0.58	0.03
16	0.10	0.00	0.19	0.06	0.29	0.06
17	0.10	0.00	0.21	0.09	0.31	0.09
18	0.28	0.01	0.22	0.16	0.50	0.17
19	0.05	0.01	0.27	0.14	0.32	0.15
20	0.06	0.03	0.20	0.05	0.26	0.08
21	0.14	0.01	0.26	0.06	0.40	0.07
22	0.12	0.01	0.18	0.06	0.30	0.07
23	0.07	0.00	0.17	0.19	0.24	0.19
24	0.06	0.00	0.19	0.49	0.25	0.49
25	0.05	0.00	0.21	0.09	0.26	0.09
26	0.06	0.00	0.10	0.28	0.16	0.28
27–53	0.43	—	0.43	—	0.86	—
over 54	2.81	—	4.49	—	7.30	—
Totals	42.27		52.73		100.00	

of normalization can occur, depending upon the number of the leading digits that are equal. In any case, the result must be normalized at least one position since it is less than one-half. If signs are unlike and the alignment shift is one position, any amount of normalization may occur. If the alignment shift is more than one position, the normalization shift can be, at most, one position. If the operands are unnormalized, there may be a normalization shift. Table 3 shows that a small percentage of operands caused such normalization.

If recomplementation is time consuming, Table 3 also contains a clue to the design of the sign and true-complement controls so that this step can be bypassed in most cases. For normalized operands, an ambiguous result sign appears only if there is no alignment shift and the operand signs are unlike. This situation occurs only 13.7 percent of the time, and no more than half of these

Table 4 Possible results for normalized operands, radix 2

Shift	*Operands after alignment*		*Result* *Like signs*	*Unlike signs*
0	$1/2 \leq f_1 < 1$	$1/2 \leq f_2 < 1$	$1 \leq r < 2$	$-1/2 < r < 1/2$
1	$1/2 \leq f_1 < 1$	$1/4 \leq f_2 < 1/2$	$3/4 \leq r < 3/2$	$0 < r < 3/4$
2	$1/2 \leq f_1 < 1$	$1/8 \leq f_2 < 1/4$	$5/8 \leq r < 5/4$	$1/4 < r < 7/8$
3	$1/2 \leq f_1 < 1$	$1/16 \leq f_2 < 1/8$	$9/16 \leq r < 9/8$	$3/8 < r < 15/16$

cases can be expected to cause recomplementation. Therefore, if the sign of the larger exponent is always assigned to the result, and if this sign is used to govern the true-complement controls, the adder output seldom needs recomplementation.

Another item of interest is the number of interchanges required. If the shift mechanism is associated with only one of the registers, it is necessary to put the operand with the smaller exponent into that register before shifting. The data gathered showed that 36.7 percent of the operands coming from storage had an exponent smaller than the operand in the accumulator register. If the interchange operation is time consuming, additional circuitry may be justified to minimize this time.

timing of floating addition

The results in Table 3 can be used to calculate the average floating addition time for the IBM 704, 709, 7090 and 7094 as an example of the use of these results in design evaluation. The 704 and 709 require seven cycles (six cycles for the 7090) to perform the floating addition if the alignment shift is less than 11 positions and the normalization shift is less than 5 positions. Each additional 12 positions (or less) of alignment, or 4 positions (or less) of normalization, require an additional cycle. The 704 allows alignment shifts up to 255 positions. The 709, 7090, and 7094 allow alignment shifts less than 64 positions only. Table 5 shows the data in Table 3 combined into these ranges.

For the 709 we have, therefore,

$$7 + (.0714 + .0361) + 2(.0130 + .0160) + 3(.0030 + .0056) + 4(.0013 + .0049) + 5(.0082) + 6(.0037) = 7.3$$

cycles and one cycle less for the 7090.

For the 704, extra time is necessary for alignment shifts of more than 63 positions. This time, determined from the original data, is about 0.7 cycles.

For the 7094, the timing rules are more complex. The operation always requires at least two cycles for instruction and data fetches. Exponent differencing and operand interchanging are overlapped with the last half of the data fetch cycle. Also, if the exponent difference is zero, one operand is normalized, and if the signs are alike, the addition can be executed. Overflow and exponent adjustment and trapping are overlapped with the first half of the next instruction cycle. Additional cycles are required only for cases

Table 5

Alignment *Shift*	*Percentage*
0–10	83.79
11–22	7.14
23–34	1.30
35–46	0.30
47–58	0.13
over 58	7.34
Normalization *Shift*	*Percentage*
0–4	92.55
5–8	3.61
9–12	1.60
13–16	0.56
17–20	0.49
21–24	0.82
25–26	0.37

Table 6 Floating addition times in cycles

Computer	*Average*	*Maximum*
704	8.0	34
709	7.3	15
7090	6.3	15
7094	3.0	12

of unlike signs and for alignment or normalization shifting. Each of these cycles consists of six subcycles. During a subcycle, an addition can take place, or either one or two alignment or normalization shifts can be executed. For like signs, shifts from 1 to 10 positions require one extra cycle (one subcycle is reserved for the addition), shifts from 11 to 22 positions require two extra cycles, etc. For unlike signs, shifts from 1 to 8 positions require one extra cycle (one subcycle has been reserved for the addition, and another for a possible normalization of 1), shifts from 9 to 20 positions require two extra cycles, etc. For unlike signs, no shifts or shifts of one position cause most of the normalization (19.96 out of 22.14 percent). Therefore, allowing two subcycles for alignment and addition, normalization of 1 to 8 positions requires one extra cycle, of 9 to 20 positions requires two extra cycles, etc. This counting is somewhat conservative because allowance for normalization is counted twice in some cases. On the other hand, no allowance has been made for (1) normalization of the small percentage of unnormalized operands, or (2) recomplementation of the multiplier quotient (MQ) register and adjustment of the MQ sign.

The results of these computations are given in Table 6. It will be noted that the design modifications based on knowledge of the suboperations improved the average speed significantly, but had relatively little effect on the maximum speed.

other radices

Many of the principles discussed apply equally well to computers employing other radices. A decimal computer is obviously of interest. Other possibilities are radices that are powers of 2 (i.e., 4, 8, 16, etc.), since a binary adder can still be used in such computers. For examples, in the 7094, the floating-point format has an 8-bit exponent based on radix 2, with a normalized 27-bit fraction f in the range $1/2 \leq |f| < 1$.

The same format could represent an exponent with implied radix 8 and a fraction in the range $1/8 \leq |f| < 1$.

This length of normalized fraction gives less precision than the 7094, since there is the possibility of one or two leading zeros. The exponent has a much larger range ($8^7 = 2^{21}$). In this configuration, each bit of exponent difference on alignment will force a shift of three positions. Normalization also takes place in shifts of three positions, with a corresponding adjustment of the exponent by 1 for each jump shift.

The possible results are slightly different for radices larger than 2. For example, in the case of radix 8, the fractional result

Table 7 Possible results for normalized operands, radix 8

Shift	*Operands after alignment*		*Result* *Like signs*	*Unlike signs*
0	$1/8 \leq f_1 < 1$	$1/8 \leq f_2 < 1$	$1/4 \leq r < 2$	$-7/8 < r < 7/8$
1	$1/8 \leq f_1 < 1$	$1/64 \leq f_2 < 1/8$	$9/64 \leq r < 9/8$	$0 < r < 63/64$
2	$1/8 \leq f_1 < 1$	$1/512 \leq f_2 < 1/64$	$65/512 \leq r < 65/64$	$7/64 < r < 511/512$

Table 8 Alignment shift frequencies for various radices

Shift	*Radix* 2	4	8	10	16	32	64
0	32.64	38.24	45.77	47.15	47.32	52.52	55.84
1	12.11	18.54	19.77	23.22	26.02	26.37	26.64
2	8.61	12.83	11.92	11.27	10.47	5.92	3.77
3	6.72	9.87	6.26	3.26	2.24	1.82	2.35
4	7.17	3.04	1.73	1.39	1.31	2.08	1.98
5	3.88	2.05	1.10	0.93	1.70	1.87	
6	4.39	1.01	0.89	1.54	1.24		
7	4.82	0.72	1.52	1.28			
8	1.29	0.63	1.00				
9	1.28	0.94					
10	1.31	0.72					
11	0.48	0.97					
12	0.58	0.74					
13	0.38	0.27					
14	0.38						
15	0.32						
16	0.33						
17	0.32						
18	0.40						
19	0.48						
20	0.36						
21	0.53						
22	0.48						
23	0.33						
24	0.36						
25	0.36						
26	0.19						
over	9.50	9.43	10.04	9.96	9.70	9.42	9.42

r as a function of fractional operands f_1 and f_2 is shown in Table 7. The range of r narrows very rapidly in this instance. As the exponent difference increases, the possibility of overflow or figure loss diminishes. If the exponent difference is zero, an overflow or figure loss (depending on the signs) always occurs for radix 2, whereas the results may not require normalization for larger radices.

Tables 8 and 9 show the frequency of shifting for various designs with radices 2, 4, 8, 10, 16, 32, and 64 compiled from data gathered in the second phase. The numbers in the "over" row of Table 8 show the percentage of shifts greater than the presumed single-length accumulator. The assumed accumulator lengths are based on information content that is approximately equivalent to the 27-bit fraction.

The trend displayed in Tables 8 and 9 is obvious: as the radix increases, there is substantially less alignment and normalization shifting. In a particular design, the limiting case would be a radix sufficiently large that a shift of one would exceed the fraction length. A practical limit would be much less than this; given a large

Table 9 Normalization shift frequencies for various radices

Shift	*Radix* 2	4	8	10	16	32	64
result 0	1.42	1.42	1.42	1.42	1.42	1.42	1.42
overflow	19.65	10.67	6.52	7.19	5.50	5.69	2.60
0	59.38	72.11	79.40	79.80	82.35	83.86	87.36
1	6.78	7.96	8.75	8.04	7.29	5.99	6.04
2	3.47	3.35	1.64	1.55	1.38	0.87	1.23
3	2.35	1.49	0.38	0.28	1.01	0.88	0.47
4	1.91	0.34	0.43	1.03	0.30	0.41	0.88
5	1.06	0.14	0.71	0.16	0.32	0.88	
6	0.56	0.92	0.25	0.25	0.43		
7	0.48	0.18	0.22	0.28			
8	0.16	0.13	0.28				
9	0.14	0.15					
10	0.08	0.18					
11	0.09	0.17					
12	0.32	0.27					
13	0.55	0.52					
14	0.16						
15	0.02						
16	0.04						
17	0.09						
18	0.08						
19	0.07						
20	0.12						
21	0.07						
22	0.07						
23	0.09						
24	0.11						
25	0.16						
26	0.52						

radix, many leading zeros are possible in a normalized fraction, and any shift deletes a larger part of the precision available.

For a serial arithmetic unit, the speed advantages of fewer shifts are apparent. For a parallel unit, design studies comparing multi-path radix 2 shifters against a jump shift with a larger radix suggest that the latter requires less circuitry for an equivalent speed.

These were among the considerations that led to the choice of radix 16 for SYSTEM/360. A further discussion can be found in Reference 4.

The trend in post-shifting for overflow and other shifts is somewhat erratic. At first, it was thought that the conversion and analyzing routine was incorrectly programmed. However, examination of the original data shows that there are certain conditions with respect to overflow on like-sign addition, and with respect to figure loss on unlike-sign addition, that can cause abrupt discontinuities. This is illustrated by Table 10 which shows how the fractional part of a number N could be represented as the radix varies.

Table 10 Number representation as a function of the radix

N	Radix 2	4	8	16	32	64
1/8	.1	.1	.001	.001	.001	.001
1/2	.1	.01	.01	.01	.01	.01
1/2	.1	.1	.1	.1	.1	.1
1	.1	.01	.001	.0001	.00001	.000001
2	.1	.1	.01	.001	.0001	.00001
4	.1	.01	.1	.01	.001	.0001
8	.1	.1	.001	.1	.01	.001

Adding 1/2 + 1/2 causes an overflow for all radices. However, when adding 4 + 4, an overflow occurs for radices 2 and 8 only. Other examples of this discontinuity are apparent in Table 10 and can easily be constructed for larger or smaller values. A similar condition existing for zero alignment shifts (if the signs are unlike) can cause a figure loss and a post-shift of one position as the radix increases.

As pointed out earlier, the necessity for recomplementation of results can almost be eliminated. This same technique can be applied for computers with larger radices. Table 11, based on data from the second phase of the study, shows the percentages of floating additions with zero shift and unlike signs for various radices. Only these cases (except for the small fraction of unnormalized operands) can give an ambiguous result which may require recomplementation. In fixed point work, approximately 50 percent of the additions would be expected to involve unlike signs with one-half requiring recomplementation. Therefore, about 25 percent of all additions would involve recomplementation. Table 11 shows that, more than 13 percent of the time, recomplementation is unlikely for the largest radix examined. Thus, the same technique discussed for radix 2 would be effective for any design with a larger radix.

Table 11 Percentages of add operations with zero shift and unlike signs for various radices

Radix	*Percentage*
2	15.3
4	18.0
8	21.5
10	22.0
16	22.2
32	23.9
64	25.7

double precision

The data presented can also be used to evaluate double-precision designs. Some of the data would be slightly different if greater precision were used in the analysis. For example, with more bits to match, probably fewer results would have been zero, thus increasing the number of operations with relatively lengthy normalizations. However, lengthy alignments or normalizations do not have a substantial effect. For example, in the case of the data gathered, the average time for the 7094 would only increase from 3.0 to 3.08 cycles if all alignments greater than 27 positions (.0086 from Table 3) and all zero results (.0082) take five cycles.

If double-precision operations are not provided, the designer should facilitate double-precision programming. Double-precision addition, the most difficult floating-point operation, occurs in every double-precision multiply or divide subroutine. Consequently, an instruction for adding a single-precision operand to a

double-precision operand with a double-precision result has merit. This instruction can be used in a simple double-precision addition routine which first adds the two low-order portions of the operands and then successively adds the high-order portions to provide a double-precision result. Provision should also be made for a double-length product and for a remainder on division.

If double-precision operations are offered or facilitated, some additional design alternatives are available for improving single-precision performance. These designs concern the retention of the bits or digits shifted beyond the right end of a single-precision register during alignment. If the operands are normalized and the signs are alike, there is no need to retain any information shifted out of the register. If the operands are normalized and the signs are unlike, there is, at most, one non-zero digit that can be shifted back into the single-precision register. Normalization of more than one digit can only be the result of no alignment shift or an alignment shift of one position, regardless of the radix base. Also, in a serial arithmetic unit, performance is greatly improved if single-precision shifting suffices for single-precision operations. In a parallel arithmetic unit, it is possible to save the time and circuitry necessary to provide complementation in the low-order register if separate signs are retained as in the 7094.

ACKNOWLEDGMENT

The assistance of J. C. Gibson, L. O. Nippe, and others in analysis and computation is appreciated.

CITED REFERENCES AND FOOTNOTES

1. See S. G. Campbell, "Floating point operations," *Planning a Computer System*, W. Buchholz ed., McGraw-Hill Book Company, 1962, pp 92–106, for a discussion of floating-point representation, precision, and significance.
2. See F. P. Brooks, Jr., and K. E. Iverson, *Automatic Data Processing*, John Wiley and Sons, 1963, pp. 184–199, for further discussion of floating-point arithmetic and an example of an algorithm for floating-point addition.
3. This study was undertaken during the design of the IBM 7030.
4. G. M. Amdahl, G. A. Blaauw, and F. P. Brooks, Jr., "Architecture of the IBM System/360," *IBM Journal of Research and Development* **8**, No. 2, 87–101, April 1964.