Memorandum to: Dr. S. G. Campbell September 12, 1961

Subject: Design of Floating Point Arithmetic

The following consists of a number of suggested improvements on 7030 Floating Point Arithmetic. If implemented, the machine will have a unique system of three interconnected accumulators, an unusually powerful instruction set and extremely fast arithmetic. Most of the hardware needed is already in the 7030, and the additional hardware investment will be small.

The present memo will cover the following points:

1. Format
2. The Multiplier Register
3. The Triple-Accumulator System
4. Postgressive Indexing
5. Immediate Operands
6. Data Flagging and Overflow
7. Sorting and Merging

Other topics will be discussed by a different memo.

This memo is more or less a continuation of an earlier one ("Floating Point Arithmetic", August 23, 1961), and relevant material may be found in two other memos ("Index Word Format and Index Arithmetic", July 26, 1961 and "Progressive Indexing", July 28, 1961).

The triple accumulator concept evolved in several discussions with Don Gibson (Kingston). I have benefited greatly from discussions with Don Gibson and Robert Rockefeller (Kingston), and stimulating discussions with C. T. Apple, N. Hardy, G. Hira and Gordon Zook (Kingston).

1. FORMAT

The 7030 floating point fraction is preceded by the exponent and followed by the 4-bit sign byte. It is desirable in double precision work to have the second order fraction adjacent to the first order fraction in the double-length accumulator to accommodate VFL data handling. In the 7030 this is achieved by detaching the sign byte from the floating point number and placing it in a special sign byte register ($SB).

This arrangement has a number of drawbacks. First, the result of a floating point "load" or ("load with flag"), even if unnormalized, will not reproduce the bit configuration of the word in the accumulator area.

Secondly, any floating point instruction involving the left accumulator address (8.0) will be an unusual instruction, for which 60 bits from address 8.0 plus 4 bits from address 10.03 (fourth bit of sign byte register) must be mobilized by hardware.

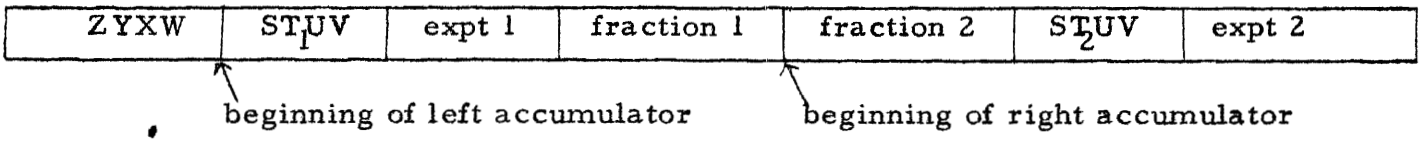
A third difficulty is that, again for VFL accommodation, the second order fraction begins immediately after the end of the first order fraction, namely at bit 60 of the left accumulator. This artificial placement has the effect that, so far as floating point arithmetic is concerned, the lower accumulator ceases to be an independent entity, but mainly a slave of the upper accumulator.

It is more desirable to place the sign byte at the front of the floating point word. The exponent sign bit should also be the leading bit of the exponent field for the sake of uniformity. During a floating point unnormalized "load with flag", the entire word can occupy the left accumulator without change of bit configuration. Floating point instructions involving the left accumulator address will then be a nonexceptional operation.

The need for a separate sign byte register will be removed; there need only be a 4-bit "zone byte register" preceding the left accumulator.

An immediate consequence of this new arrangement is that the fraction ends precisely with the left accumulator. The rather unnatural partition at the 60th bit is thus avoided. Further, this enhances the semi-independent position of the right accumulator.

To ensure the fraction continuity of a double-precision number in the double precision accumulator, the ideal low order fraction should again be adjacent to the high order fraction, with the low order sign byte and exponent displaced to the extreme right, as seen in Figure 1.



It is important to allow the lower accumulator the freedom of having its own sign byte and exponent. It may not be easy to ensure the low order sign byte and exponent to be compatible with the high order counterparts, nor is it really necessary. There should, however, be the instruction CSLO (compatibilize and store low order) which treats the low order fraction as the low order part of a double precision word, creates and inserts the correct sign byte and exponent before storing.

2. THE MULTIPLIER REGISTER

In the K-2 improvement program a multiplier register ($MR) is provided to facilitate the "multiply and add" instruction. This reduces the vector multiply arithmetic time by a factor of two.

The multiplier register in K-2 is accessible to only three instructions: LMR (load multiplier register), *+ (multiply and add) and STM (store multiplier). It has no address.

It is important that this register be given an address, for there is a genuine need for addressable transistor registers in the accumulator area serving as temporary storages to avoid unnecessary interaction with memory.

In the "double divide" instruction instead of sending the remainder to location 13.0 in the memory, it would be much more expedient to put it in $MR for rapid reaccess.

3. THE TRIPLE-ACCUMULATOR SYSTEM

The 7030 employs the universal accumulator concept: the most important results of arithmetic are always placed in the accumulator. This, plus the addressability of the accumulator, removes much need for using temporary storages.

It is still necessary to protect a result from being destroyed by the next few instructions. Since the forwarding implementation is uneconomical, it is necessary to have high speed registers for temporary storage. The lower accumulator and the multiplier register, when given high speed linkage to the upper accumulator, clearly will serve the purpose.

One can proceed a step farther to save the reaccessing the temporary storage, by endowing these two registers the property of universal accumulators.

The universal accumulator in the 7030 is achieved by performing arithmetic with special hardware outside the accumulator. It is thus extremely simple for the hardware to pick up operands from, and submit results to, any of the three "accumulators".

It turns out that the linkages, through the need for "double" operations and "multiply and add" are already present, although a little strengthening may lead to higher speed.

A single accumulator is not really enough to demonstrate the power of 7030 arithmetic, since the storing and refetching from temporary storages are redundant operations. More than four accumulators would lead to another type of redundancy, namely redundancy of hardware. The optimum balance of performance with hardware calls for about three or four accumulators. The natural expansion of 7030 facilities with little increase of hardware leads to the near optimum number of three.

There should be a more powerful instruction set to help bring out the potential of the triple accumulator scheme. There should be elementary arithmetic instructions, (add, multiply, etc.) available to all three accumulators separately. Then there should be cooperative instructions (such as "add double" "multiply and add") to allow the accumulators to pool their resources together. Finally there could be automatic "filing" and "retrieving" schemes to reduce the number of redundant operations to a minimum.

The triple-accumulator possibility changes the entire complexion of floating point arithmetic. Not only will programming be vastly simpler and easier to debug, but the scheme allows great improvement in arithmetic speed. For instance, if two consecutive instructions refer to different accumulators, the second instruction can be started long before the first one finishes.

4. POSTGRESSIVE INDEXING

Index arithmetic on the 7030 is relatively slow. A sequence of "load index" instructions take 5.7 μs each, a sequence of "add to value" instructions take 5.2 μs each. For floating point inner loops the most important instruction is "add to value immediate" a sequence of which takes 3.3 μs each.

When interlaced by floating point instructions, much of the time can be overlapped. The amount dead time which cannot be overlapped is 1.5 to 1.8 μs.

There are various ways to improve the index arithmetic speed. For instance by refraining from loading index recovery levels into the lookahead, the dead time can be reduced to zero, if ther is enough arithmetic action to cover up the I-box index arithmetic time.

However, it is not easy to cover up several microseconds of I-box time by concurrent floating arithmetic, particularly if the latter is made faster.

The solution is to make useful index arithmetic an automatic secondary operation on floating point instructions. This saves instruction space and can allow overlap (within the I-box) of decoding time. Further if the index arithmetic is similar to routine effective address creation, the I-box time will be cut drastically, to the extent that overlapping with E-box time is practically assured.

In 7030 VFL arithmetic, progressive indexing is a powerful secondary index arithmetic operation. Somewhat unfortunately, the feature is quite different from usual effective address creation, and is therefore not too fast and is hard for new programmers to learn.

In an earlier memo (Progressive Indexing, July 28, 1961) a scheme to replace the present progressive indexing has been suggested. In the new "postgressive indexing" scheme the effective address is generated in a standard fashion (address field plus index value field), but the effective address may or may not replace the index value field, dependent on the specification by the programmer.

The postgressive indexing scheme saves a good bit of hardware, is just as powerful as progressive indexing, is potentially faster, and is easier to learn and use by programmers.

A method to avail the postgressive indexing feature to practically all instructions was also described in the same memo. The precise specifications of the four options (normal, immediate, V+I and V+ICR), are relegated to the two unused index register bits, and only one bit per instruction is needed to specify whether the specifications are to be ignored or not. This greatly generalizes immediate index arithmetic (by taking the immediate option in direct index arithmetic. Such instructions previously cannot be indexed, but will now be fully indexable), and makes bug-free floating point arithmetic particularly easy to program.

5. IMMEDIATE OPERANDS

A side-effect of the postgressive indexing scheme is that all floating point instructions now can have optional immediate operands. Namely, the address field can be used to contain the operand, rather than an address which refers to the operand.

The use of immediate operands reduces memory access, hence trends to cut down the number and durations of memory conflicts in the machine.

Of much greater importance is the forthright nature of coding achievable through immediate operand addressing. It will now be possible to say "multiply the quantity in the accumulator by 13" directly in one instruction, rather than "multiply the accumulator contents by the contents of 25724.0". By the time the instruction is being executed the contents of 25724.0 may no longer contain the number 13.

The instruction "add to exponent immediate" will now be a special case of the instruction "add to exponent", and can be removed.

6. DATA FLAGGING AND OVERFLOW

The data flagging feature in 7030 floating point arithmetic is extremely convenient for boundary value problems and matrix manipulations.

The presence of three flag bits (TUV) per floating point word, however, may be a luxury. It turns out that the presence of one flag is very helpful, but there are few instances calling for two data flags and virtually no need for all three flags at all. It seems desirable to turn the lower flag bits to better use.

Unnormalized floating point arithmetic on the 7030 may lead to overflows in the fraction field. This overflow may only be intended by the programmer to be a temporary phenomenon to be retrieved in subsequent operations. A case in point is (0.5 + 0.5) - 0.5, where the result should be +0.5. On the 7030 this leads to a "lost carry" indicator being turned on, and the result is minus 0.5, because the machine does not allow retrievable overflow of the fraction field for unnormalized floating point arithmetic.

Because floating point arithmetic is much faster on the 7030 than VFL, it is advantageous to do unnormalized floating point arithmetic on VFL quantities to gain speed. The lack of proper overflow retrieval is a handicap.

It is therefore suggested that in the floating point word the V flag bit be used as an extension of the high order bits of the fraction, such that fractions twice as large as normal can be allowed to exist. Also the U flag should be made into a double overflow flag which is turned on whenever a fration double overflow occurred. Such double overflow should not be retrievable without explicit bit-setting instructions, and a U-flag interrupt can be made. as an indication for double fraction overflow. The interruptibility of the V flag,on the other hand, is not really crucial.

The new floating point word format would place the U, V flags ahead of the fraction, and the overflow interpretation is quite natural. The ability for a floating point word to carry its own overflow indications should be an added asset.

It is extremely important that the fraction overflow condition be retrievable on unnormalized add operations. For other instructions the precise action on V-flagged" quantities is not of grave concern.

7. SORTING AND MERGING

The 7030, because of the highly overlapped nature, is relatively slow on branches on the basis of E-box conditions. The indicator bits AL, AE, AH can be tested only by proceeding on an assumption with provision to back-track.

The assumption on X-1 and K-1 is that the branch is probably unsuccessful. In the K-2 and subsequent machines the assumption is that the condition of these indicators will probably remain unchanged between the decoding and the execution of the branch instruction.

The K-2 scheme can be quite effective if the "compare" instruction which sets the indicators AL, AE and AH is several instructions ahead of the conditional branch. For many situations this "instruction distance" can be maintained, and the conditional branch is very fast.

There are, however, situations which involve extremely short conditional branch inner loops, and the "instruction distance" can be maintained only through time consuming artifices. Outstanding cases are sorting, in which a given collection of numbers are rearranged in numeric sequence, and merging, in which two or more presorted sequences combine to form a longer sorted sequence.

The sort-merge problem is important particularly in commerical data processing. Usually the data may be in VFL format, but the key items to be sorted may well have been recorded in floating point format to gain speed.

The solution to the sort-merge problem lies in giving the E-box high speed facilities to do the right thing automatically without testing by conditional branches.

For the 7030 a new instruction, tentatively call KL (compare load) allows high speed automatic "anchor" sorting, and a new KC (compare status change) indicator bit with interruption facilities allows high speed merging.

In the compare load instruction, the memory operand is compared with the quantity in the upper accumulator. Whichever is larger goes to the upper accumulator, and the smaller quantity is loaded into the lower accumulator. The instruction produces automatically a sorted pair of numbers without any conditional branch instructions.

For a collection of N numbers, one number is placed first in the upper accumulator. Then, (N-1) pairs of KL, SLO instructions later the largest entry will be found in the upper accumulator. The (N-1) smaller numbers, now partially sorted, are stored in the memory ready to repeat the process with one fewer member. This "anchor" sorting process, in which the heftiest quantity is always found at the bottom after one sweep, is here done completely without any conditional branch instructions.

As a result of a compare instruction the AL, AE, AH bits are set. If the resultant status is different from the previous status, the KC bit is set to 1, otherwise it is set to zero. When the KC bit is further endowed with interrupt abilities, it can allow the merging process to proceed in a speedy manner.

Suppose there are two presorted sequences (p_k) and (q_ℓ). To merge the two into the sequence (r_m) one may assume (p_o) is larger than many leading members of (q_ℓ). (q_ℓ) can therefore be sent to the accumulator, be compared against p_o and stored into (r_m). As long as these quantities are less than p_o, nothing will happen to disrupt this process. When an exceptional case is found by the compare instruction, KC will be turned on, causing an interrupt. Then the exceptional q_ℓ is assumed to be larger than members of (p_k), and the process continues. When both (p_k) and (q_ℓ) are exhausted, (r_m) will be a sorted merged sequence.

The KC indicator bit has other useful properties. It can allow the removal of the "compare for range" type of instructions, can can give indication when the "anchor sort" scheme is completed ahead of expectations.

Suppose one wants to find out if C lies within the interval (A, B). It is only necessary to compare C against A, then C against B and interrogate the KC bit. If the bit is a 1, the comparison status with A must be different from the comparison status with B, and C must either lie in the middle, or is equal to one of the comparands. Note that it is not necessary to know whether B is greater than A or not. It suffices that B is different from A.

During the use of the KL instruction for anchor sorting not only the largest quantity is found at every sweep, but the sequence is partially sorted as well. If after a certain number of sweeps the probability of completion is high, it is possible to switch to KC interrupts to complete the sorting. If no interruption occurred during the sweep, the sequence must already be well ordered.

Tien Chi Chen

TCC:tm