

---

Retrospective Theses and Dissertations

---

1977

## Hardware and Software Considerations for Improving the Throughput of Scientific Computation Computers

Glenn Allen Sullivan  
*University of Central Florida*

 Part of the [Engineering Commons](#)

Find similar works at: <https://stars.library.ucf.edu/rtd>

University of Central Florida Libraries <http://library.ucf.edu>

This Masters Thesis (Open Access) is brought to you for free and open access by STARS. It has been accepted for inclusion in Retrospective Theses and Dissertations by an authorized administrator of STARS. For more information, please contact [STARS@ucf.edu](mailto:STARS@ucf.edu).

---

### STARS Citation

Sullivan, Glenn Allen, "Hardware and Software Considerations for Improving the Throughput of Scientific Computation Computers" (1977). *Retrospective Theses and Dissertations*. 382.

<https://stars.library.ucf.edu/rtd/382>

HARDWARE AND SOFTWARE CONSIDERATIONS  
FOR IMPROVING THE THROUGHPUT OF  
SCIENTIFIC COMPUTATION  
COMPUTERS

BY

GLENN ALLEN SULLIVAN  
B.S.E. Florida Technological University, 1971

THESIS

Submitted in partial fulfillment of the requirements  
for the degree of Master of Science in Engineering  
in the Graduate Studies Program of the College of Engineering  
of Florida Technological University

Orlando, Florida  
1977



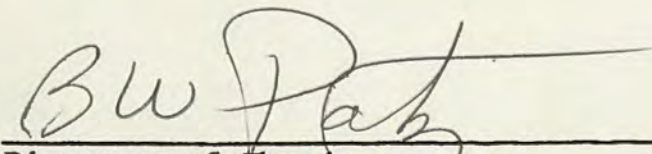
HARDWARE AND SOFTWARE CONSIDERATIONS FOR IMPROVING  
THE THROUGHPUT OF SCIENTIFIC COMPUTATION COMPUTERS

by

Glenn Allen Sullivan

ABSTRACT

In this paper, hardware and software techniques are presented for improving the Throughput (defined as Computations per dollar) of computing systems which are oriented towards high-precision floating point computations. The various improvements are referenced to a baseline of the PDP 11/20, the NOVA 1200, and the TI 960A, all 16 bit minicomputers. The most beneficial hardware improvement is the inclusion of a Floating Point Processor, which yields up to 200X Throughput increase over a software floating point package. The inclusion of a cache high speed local memory and the availability of Polish Notation format instructions are shown to provide less than a 5X increase each. The use of 48 bit data paths, numerous registers devoted to various processor functions, instruction lookahead, a system I/O controller which frees the processor from I/O work, and partitioned main memory, result in a combined Throughput increase of 5.9X.

  
\_\_\_\_\_  
Director of Thesis

## ACKNOWLEDGEMENT

I would like to recognize the guidance, urging and patience of Dr. Benjamin Patz, who I consider to be not only a teacher and advisor, but also a friend.



## TABLE OF CONTENTS

ACKNOWLEDGEMENT . . . . .	iii
I. INTRODUCTION . . . . .	1
II. HARDWARE CONSIDERATIONS . . . . .	6
III. SOFTWARE CONSIDERATIONS . . . . .	31
IV. CONCLUSIONS . . . . .	49
APPENDIX . . . . .	51
LIST OF REFERENCES . . . . .	53

## I. INTRODUCTION

This research paper is concerned with several hardware and software approaches to improving the Throughput of number-crunching minicomputers, i.e., the primary task of the minicomputer is the execution of high-precision arithmetic operations, typically with 32 to 48 bits resolution.

The intent is to provide guidelines for an examination of available computers, and not to exactly specify the characteristics of the computer. Thus while a 48 bit word (configured as in Figure 1) is frequently used in the examples, some other word size may be available and best suited for the projected applications.

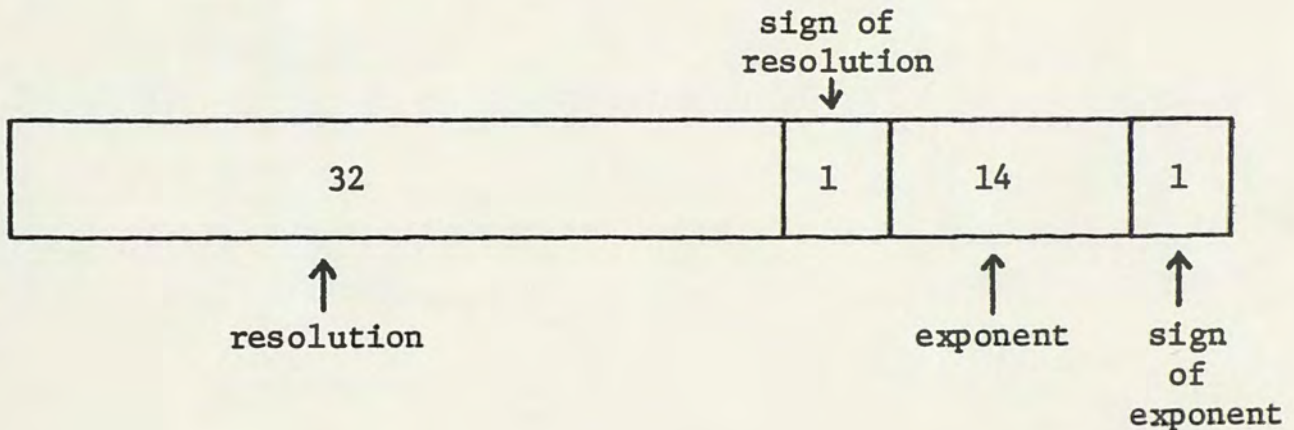


Fig. 1. Configuration of 48 bit Floating Point Word

Despite the frequent discussion of details such as cycle time, memory size, I/O channels, etc., the basis for comparing computer



systems can best be a matter of economics--how much computation per dollar, herein referred to as Throughput. (Foster (1) suggests that Throughput per unit of time be used, while neglecting cost.) The phrase "computation per dollar" is preferred to "instructions executed per dollar" since a fast but poorly-considered computer could easily appear superior to a somewhat slower computer with a well-considered instruction set, although the slower computer may equal or surpass the faster on a "computation per dollar" basis.

#### A. APPROACHES TO PROBLEM

Throughput may be enhanced by improving the efficiencies of the two basic computer operations: (1) moving data, and (2) operating on data. One solution is to move data as little as possible, and to use generous amounts of hardware to achieve largely parallel data operations.

Accordingly, Section II examines the number and type of registers available to the programmer, the number of buses internal to the processor, and the necessity for a separate I/O controller and a Floating Point Processor (FPP) as well as other hardware features.

Section III, software considerations, examines the need for variable length instructions, compound operation instructions, and the I/O controller.



## B. RESULTS OF THE VARIOUS PROPOSALS

For typical scientific computations such as trigonometric function generation, matrix inversion or numerical integration, with a big percentage of the actual computations being high-precision operations, the usage of a hardware FPP is easily justified; there may be as much as a 100X improvement in Throughput as the time to execute floating point multiplications is reduced from 500 usec with software execution, to the range of 3 to 15 usec with various hardware execution techniques.

By using a high-speed local-store memory with 75 nanosec effective access time, compared to typical main memory times (core or MOS) of 400 to 700 nsec effective access times, and with both instructions and operands contained in local-store memory, the time to execute the shorter arithmetic and logic instructions can be reduced by as much as 80%. By using compound instructions, such as the Data General Nova computer family instructions which combine arithmetic or logic operations with condition testing and branching, the time to execute the shorter instructions can be further reduced by 50%.

Thus, depending on the instruction mixture, with a baseline of the PDP-11 or Nova series computers, we can expect from 4X to 200X improvement in Throughput as a result of implementing the various proposals of this report. Figure 2 illustrates the various system elements.



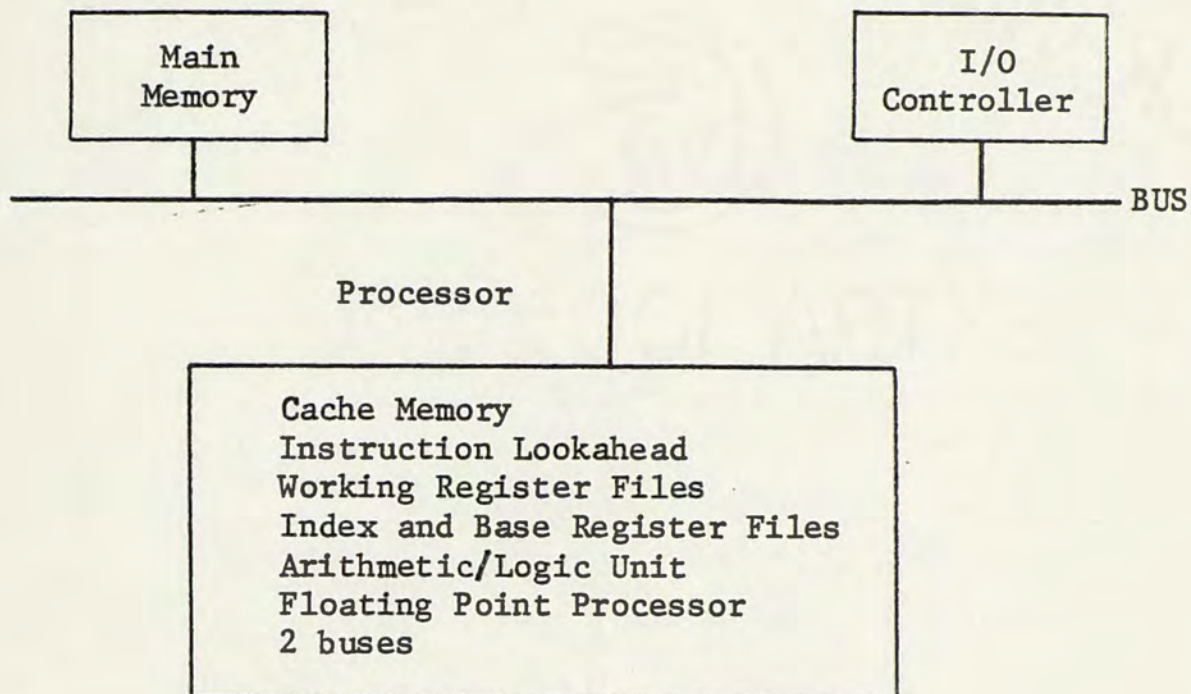


Fig. 2. Computer System Elements

### C. BACKGROUND MATERIAL

The investigative phase of the report development was concerned with becoming familiar with the instruction set characteristics of the Data General Nova 1200 (2), the Digital Equipment PDP-11/20 (3), and the Texas Instruments TI-960A (4), all 16 bit minicomputers.

Assembly language codings were generated for fixed point 16 bit divide and Floating Point 32 bit addition and multiplication. It was immediately obvious that the PDP-11 and TI-980 offered an advantage with their 6 and 8 registers, respectively, which are undedicated and therefore available for 32 bit computations.

The Nova 1200 required much more register-memory-register activity which completely negated the benefits of its compound

instructions.

As the report development continued, it was realized that it was folly to not have a Floating Point Processor. Further reflection inspired the inclusion of a number of working registers, so as to minimize the need for register-memory swapping, also improving I/O interrupt handling or switching from Worker to Supervisor mode.

The result is the realization that simply defining a better "set of instructions" for a scientific minicomputer will not yield the desired result, which is a significant improvement in Throughput as compared to the three minicomputers examined.

The proper approach is a combination of hardware and software (or instruction set) improvements. These improvements are presented in Sections II and III.



## II. HARDWARE CONSIDERATIONS

The key to good hardware performance is determined by the results of the two dominant computer operations, moving and operating on data. The overall goal is to keep data and instructions coming to and going from the data operation modules, where as much parallel processing is done as is affordable.

Guidelines for achieving this are presented in Section II A (Moving Data) where the dominant theme is to move data and/or instructions as little as possible but move them quickly when needed, and in II B (Data Operations) where parallel processing is interpreted to mean not just one-step clockless multiplication, but the elimination of certain instruction execution approaches which particularly penalize the less complex instructions.

### A. DATA MOVEMENT

Improving Throughput requires that the processor be able to move data when needed, not when the I/O peripherals so permit. Accordingly, two types of data paths are defined: (1) a BUS, which major system elements use to transfer among themselves, and (2) a bus, which is a data path within the processor.

#### 1. BUSES and buses

The number of data Buses greatly influences the system Throughput. Systems which need simultaneous I/O and processor executions must be configured so as to minimize conflicts between the



two; if there is only one wideband data path within the system, conflicts will be unavoidable.

For a system with multiple processors, multiple I/O controllers and numerous I/O devices, it certainly makes sense to have several BUSES; the BUS priority hardware may be simplified, Throughput should be enhanced, etc. But for a one-processor system, where processor execution might be inhibited while main memory is used for I/O communication, only one Bus can be readily justified.

Thus most computing systems can only justify one BUS; partitioned main memory and an I/O controller can require and justify more than one BUS.

The buses within the processor itself are a different matter, although subject to the same reasoning. A processor has numerous data sources and sinks, such as the main memory port(s), cache memory, registers, and data operators.

One obvious choice is to have no special processor bus, but to extend the BUS inside the processor. This choice is economical because no BUS switch is needed to link a processor bus to the BUS; however, one common BUS will reduce Throughput because of being able to move only one word at a time and because memory-to-I/O operations inhibit transfers involving any processor units (note that processor units such as the Floating Point Processor should be working while the BUS is busy elsewhere).

A second choice is to have one BUS and one bus, which allows independent I/O and memory-reference-free processor functioning but



does require a Bus switch between the BUS and the bus.

However, two buses will permit providing two operands to those units which can operate on two operands, without having to load one operand in a register and then provide the second. Two buses do require twice as much driving and receiving logic to interface to processor units. But the time saved and the ability to access two different operands simultaneously are strong favorable arguments. In addition, the bus interface circuitry is often designed into contemporary TRI-STATE output Integrated Circuits, therefore, only bus control logic need be designed, not bus driving circuitry. (A typical unit is SN74S200 (5), a 256 bit TRI-STATE memory.)

Three buses are even better, because of being able to provide two operands to a unit and then move the answer to its storage location. But unless the processor register files are able to supply two operands and receive the result, which implies three data ports for the files, the three buses will not be simultaneously busy and thus two processor buses are enough.

Figure 3 summarizes the points of each choice.

Thus, for a scientific machine, a good choice is one BUS and two buses, for these reasons:

1. Minimum of conflict between I/O and the processor
2. A scientific machine which is not highly parallel may be slow enough that two processor buses can provide sufficient bandwidth



3. Two buses can move in parallel, two operands from registers or memory and allow the execution of one-step operations from the buses instead of a temporary holding register.

We must include a dedicated bus from the instruction look-ahead circuitry to main memory, as shown in Figure 6 on page 21 of this report. Hellerman (6) further discusses the need for various buses.

# buses	0	1	2	3
I/O -- processor conflict	maximum	only over memory usage		
speed of moving operands	1	2	3	4
financial cost	low	low	some more	even more
micro-programming cost	low	low	little more	and more
temporary register cost	high	high	medium	lowest

Fig. 3. Processor Buses strongly influence processor Throughput.

How many bits wide should the BUS and the buses be? Since the human-interface devices typically use 7 bit ASCII codes and the industry standard mass-storage data word is an 8 bit byte, 8 or 16 bits might be adequate. But if the processor and main memory size is 48 bits, then a 48 bit wide system BUS sounds good.



Considerable logic circuitry will be wasted in multiplexing 48 bit words onto a 16 bit BUS and then demuxing into 48 bit registers, and transferring 48 bit words will take 3 times as long as one 16 bit word, probably 300 nsec versus 100 nsec.

An alternative approach is to realize that once a block of data has been transferred to the processor, and operations have begun, then there will be only infrequent demand for other data words until a whole new block of data is needed, and a 3-step transfer is acceptable, for occasional demands. Unfortunately, if this occasional demand for memory access occurs in the middle of an iterative execution, then Throughput suffers. Again a 48 bit BUS is needed.

The final point is the continually increasing speeds of main memory technology. A 1 usec access time core memory is only slightly worsened by a 300 nsec transfer time, while modern dynamic MOS RAM memories, with 400 nsec effective access times, certainly justify a 48 bit wide BUS.

The processor buses can be examined with the same criteria in mind, but transferring data from FPP to registers to cache memory or Arithmetic-Logic-Unit or Main Memory. Again, 48 bit buses are needed.

In summary, partial word transfers seriously degrade system Throughput, and as will be seen in the rest of Section II, the recommended hardware is best utilized with full-width data paths.

Figure 4 illustrates system configuration at this point.



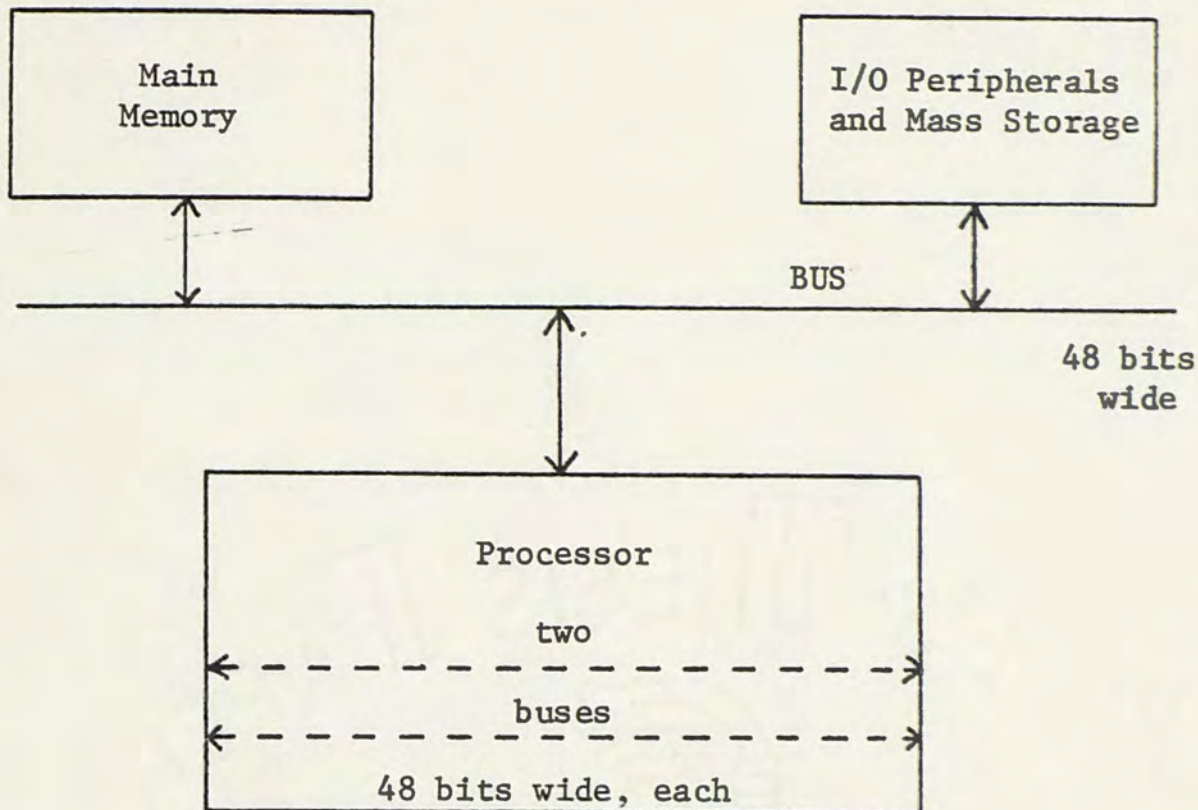


Fig. 4. System Data Paths

## 2. PROCESSOR REGISTERS

Now that the system can move data quickly when required, we need to minimize the movement of data (and instructions) by including, within the processor, the following accumulators/registers.

1. Registers used to hold data before and after operation; for generality of use, the width should be 48 bits, they should be available by either of two ports, so that two operands can come from the same register file and the file should hold at least 8 registers so that the register specification field in the instruction word is of non-trivial width; suggested source is SN74172, 3-port register file

2. To save time in computing memory addresses, there should be separate registers which act as index registers for list ac-



cessing and as base registers for relocatable instruction accessing; by having these registers separate from the 48 bit registers, they can be permanently wired to parallel adders and the Program Counter, thus allowing rapid address computation; suggested length is 32 bits, allowing a main memory of 4 Megawords, although the base registers will typically be referring to blocks of words of 512 word size or larger and something less than 32 bits would suffice; suggested source is SN74170 4WX 4 bit register file. The need for at least 8 working registers (including index) to allow the writing of position-independent code is discussed in a book from General Automation (7). Lorin (8) shows that index or base registers are needed for multiprogramming activity.

3. A third set of registers will be used whenever the processor is forced to switch from number-crunching to managing the system, as defined by the Operating System Program (OSP). These registers will be 48 bits wide, so as to be able to handle any size word. To minimize register-memory swapping while executing the OSP, 8 registers will be provided.

4. A fourth set of registers is in the FPP, so as to further minimize the movement of operands. These will be discussed in Section II B1.

Lest the reader be appalled by the numerous registers in the processor, remember that registers are relatively cheap, less than \$.2 per bit. Adding extra registers is one hardware technique which greatly improves Throughput because the data can be available



within 50 nsec instead 400 nsec, and the use of fewer bits to select a register than to specify a main memory word allows shorter instructions.

Speaking of instructions, why can't they be in registers as well?

### 3. HIGH SPEED LOCAL STORAGE (CACHE MEMORY)

In computers with only a few registers the instruction execution cycle most often requires two main memory accesses: one to fetch the instruction and a second to locate the desired operand. Having data in registers reduces the frequency of second accesses. Likewise, having the instructions in fast store/registers would reduce or eliminate first accesses. Having the instructions in fast store would reduce the instruction execution time by nearly 40% because of having a 50 nsec register access time replacing a .4 usecond memory access.

Storing the entire program in fast store would be considerably more expensive than using conventional memory, although the Throughput would increase considerably. Programmed loops, which will fit into the available fast store, can be executed at a very fast pace without requiring the main memory to be nearly as fast. Lorin (8) discusses this under "Moving a Single Processor System to Its Limit."

To permit the use of fast storage, two conditions must be satisfied. These are (1) the loop must fit within the available fast storage, and (2) instructions must exist for loading the loop



instructions into fast storage and for switching the processor to and from execution of the fast storage loop. The first condition is satisfied by purchasing a suitable block of fast storage (less than \$.2 per 16 bit word for 75 nsecond access) and by allotting sufficient bits in the instruction words to select any one word of fast storage, which can be avoided by using a Cache Program Counter. Thus 1024 bits of fast storage requires 10 bits to select any one word.

The second condition cannot be satisfied by purchasing components; instead two new instructions must be defined. Multiple Fast Transfer (MFT) is intended to load several words into sequential storage locations. Before executing MFT, an index register could be initialized as an autoincrementing pointer to the desired data block. MFT contains the two essential numbers of (1) the first word of the data block, and (2) the number of words to be transferred to fast store or to main memory.

The second new instruction is Conditional Control Transfer; program control is handed from the regular program counter to a Fast Store program counter, or vice versa, if a specified processor state exists.

The Throughput improvement provided by Cache Memory is illustrated with a software implementation of the Booth algorithm for multiplication, which goes as follows:

1. Logical-shift the multiplier and the partial product

2. Add the multiplicand to the partial product if the multiplier LSB is a 1

3. Go to 1 unless finished

This operation is executed as follows:

1. Load the multiplier and multiplicand into the proper registers, clear the register wherein the product will appear, and load a down counter with  $17_{10}$

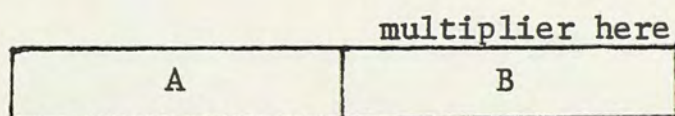
2. Load an autoincrementing index register with the address of the first instruction of the add-shift loop

3. Execute a MFT of the add-shift loop into a block of fast store

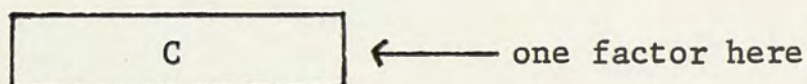
4. Execute a CCT--unconditionally transferring control from the program counter to a fast storage program counter

The loop is executed requiring 0.5 usecond per instruction, until a CCT is satisfied (after 16 loop iterations) and control is transferred back to the program counter.

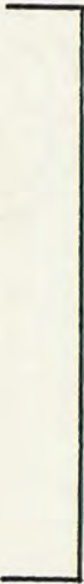
With the following add-shift loop



double length product appears here





<pre> LOOP:   Shift right A         Shift right B         CCT (if counter = 0)         Skip (if carry = 0)         Add (A+C into A)         Decrement counter         Jump (to LOOP) </pre>		<pre> loaded into fast store </pre>
<pre> NEXT:   next instruction in the program </pre>		

conventional execution (with instruction CCT changed to a conditional Jump TO NEXT) requires the following execution times:

1. Load multiplicand and multiplier into B and C registers, clear A register, and load  $17_{10}$  into a down counter (these initializations are identical for both cases and thus are neglected)

2. 16 iterations of the loop from Shift A through Jump (to LOOP) requiring 16 iterations times 7 instructions times 1.5 useconds (the 1.5 usec is composed of 1 usec instruction fetch time, and 0.5 usec execute time because all operands are in registers) per instruction, or 168 useconds

3. Execution of Shift A, Shift B, and then Jump to NEXT which ends the loop--4.5 useconds

for a total of 172.5 useconds.

A similar execution, using fast store, required the following times:

1. Initializations
2. Load an autoincrementing index register--2.5 usec, and



execute a MFT (of 7 words)--1 usec to fetch plus 7 transfers times 1.5 usec or 11.5 usec

3. Transfer program control to a fast storage program counter--CCT--1.5 usec

4. Execute loop--16 iterations times 7 instructions times 0.5 usec or 56 usec

5. Execution of Shift A, Shift B, and CCT--1.5 usec for a total  $11.5 + 1.5 + 56 + 1.5$  or 70.5 usec; this is  $\approx 40\%$  of conventional execution times.

Thus execution of loops requiring many iterations--where the critical number of iterations is inversely proportional to the loop length--will reduce program execution time. For combinations of long loops and many iterations, the execution time is bounded by limits of 60% and 20% of conventional execution times, where 60% results from instructions being in cache memory and the operands in main memory, and 20% results from instructions in cache and all operands in registers. This assumes that all data massaging occurs in 500 nsec, no matter what the operation.

The program requires three additional instructions:

1. To initialize an index register
2. MFT
3. CCT

It is felt that the additional instructions will prove useful, MFT for restoring register contents after a POWER FAIL INTERRUPT (indeed if the entire processor state were contained in registers one MFT



would suffice to restore the processor state) and both MFT and CCT for changing processor states and reassigning processor control in a multiuser/multiprogramming/multiprocessor/time-shared computational environment.

For consistency, if nothing else, it is necessary to make the cache memory word size 48 bits. To determine the necessary number of words in the memory requires more effort, but an examination of several program loops (see Appendix A) showed that a 1K word cache memory is adequate. Besides, Section III shows how to pack several instructions in one 48 bit word, so there is the capability of holding quite large loops in a 1K cache memory.

A possible source is the SN74S200, a 256 bit RAM. Probable cost is greater than \$500 for a 48 bit memory.

#### 4. INSTRUCTION LOOKAHEAD

It was previously mentioned that the processor needs to keep data and instructions coming to and going from the data operation modules. With the inclusion of several types of registers and the cache memory, the data and instructions are available faster than the processor can finish one instruction and move to the next.

For example, with data and instructions in cache memory, and assuming 25 nsec to compute the next instruction address, 75 nsec cache memory access time for the instruction, 100 nsec instruction decode time and 75 nsec to access the new operands from either register or cache memory, then a 100 nsec execution time ( a rea-



sonable value for fully parallel operations such as ADD, COMPARE) is totally swamped by the 275 nsec instruction setup time. This flow of operations follows:

1. Compute next instruction address--0→150 nsec; 0 typically, 150 nsec if different index register is used; allow 25 nsec
2. Access next instruction in cache memory--75 nsec
3. Clock instruction into holding register and decode--100 nsec
4. Locate new operands and prepare to gate them onto processor buses--75 nsec
5. Gate operands onto buses and execute instruction--100 nsec
6. Return to 1

By adding extra logic to implement an Instruction and Data Lookahead module, then these 5 operations can be split into 2 parallel activities as illustrated in Figure 5.

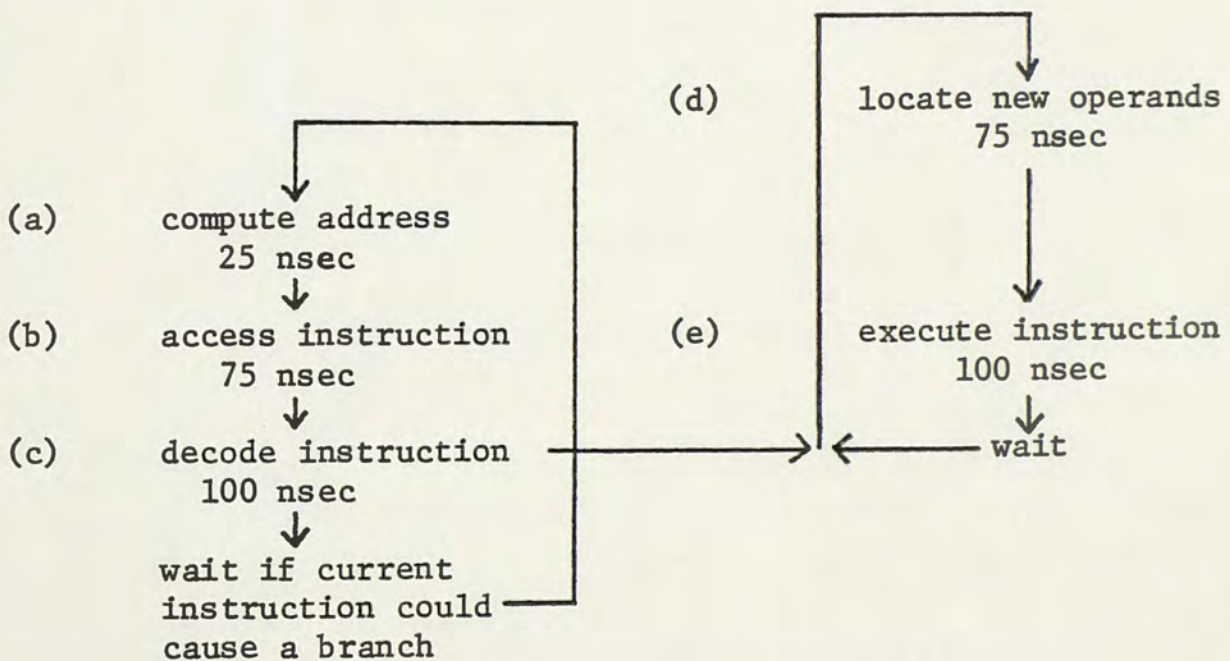


Fig. 5. Flow of Instruction Lookahead



This parallel flow reduces the typical execution time to 250 nsec from 375 nsec, and is well worth the extra circuitry, which will mainly consist of logic to allow the locating of operands to have priority over instructions, especially desirable if two operands are sequentially pulled from the cache memory, and logic to halt the instruction sequence (steps a, b & c) if the present instruction could result in a program flow branch and thus invalidate the address that would have been computed. It should be noted that a branch within the boundaries of cache memory results in much less time delay (before returning to pipelined execution) than does a branch to main memory.

To expedite instruction transfer from the cache, a dedicated path exists between the cache and the lookahead unit, as shown in Figure 6.

## 5. I/O CONTROLLER

After improving the Throughput by adding the hardware suggested in Section II A1 to A4, it is necessary to ensure that the processor will not be bothered by the need to handle the I/O devices. We particularly do not want the processor to have to handle data transfers to and from mass storage.

By using an I/O Controller to handle all interrupt servicing and block data transfers, and to buffer I/O device data transfers to/from memory, the processor can be isolated from most of the problems that I/O devices inflict upon a computing system, particularly



where the cache memory is reading in a block of data and an I/O service routine memory access would delay the beginning of a computation loop.

Figure 6 presents the hardware suggestions of Section II A.1-5.

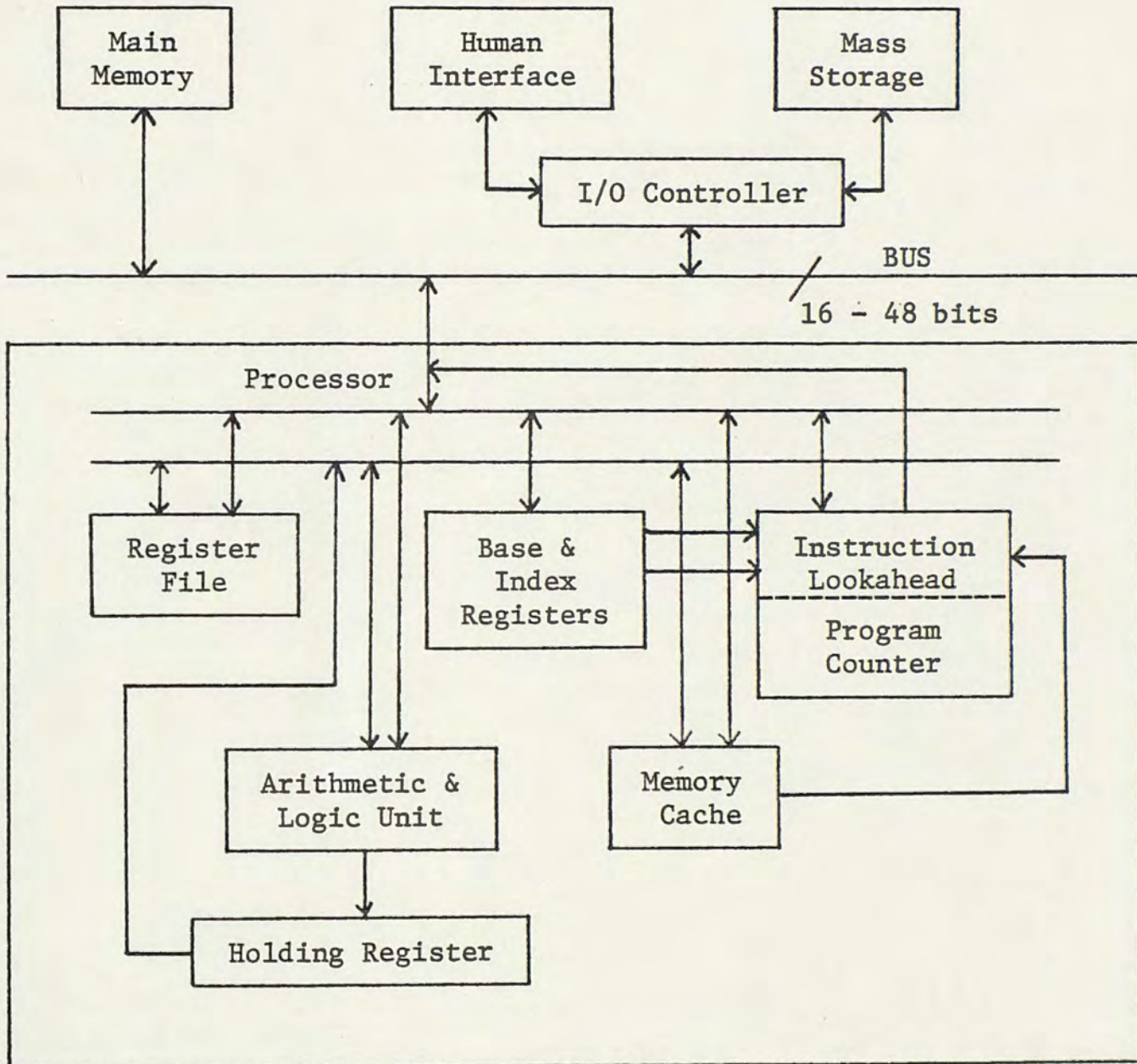


Fig. 6. System Configuration



## 6. MAIN MEMORY PARTITIONING

One way to prevent processor and I/O conflict over main memory is to partition main memory into sections, each with its own memory address and data registers and bus controller interface. Data awaiting I/O action would be available in one section while the other section(s) could simultaneously provide memory service for the processor.

There is a peculiarly interesting benefit if the number of memory sections available to the processor is a binary integer  $2^n$ ,  $n > 1$ . This benefit appears as a  $1/2^n$  reduction in effective memory access time when referencing sequential memory locations, as when transferring blocks of memory words to the processor cache memory.

For example, if there are 4 memory sections for the processor, and if words are written into these sections in a 4 word parallel fashion (e.g., word N in section 1 - location M, word N+1 in section 2 - location M, N+3 in section 4 - location M, word N+4 in section 1 - location M+1, etc.) as illustrated in Figure 7, then by accessing 4 words in parallel, the effective memory access time becomes 100 nsec instead of 400 nsec.

Keep in mind that to access any word takes 400 nsec but that once the Memory Buffer registers are filled, the effective word rate is 10 MHz instead 2.5 MHz. Once data is in the cache memory, however, the word rate rises to 13 MHz.

As was discussed in section II A, part 1, the inclusion of partitioned memory may justify two BUSES, with the I/O controller



moving I/O data to and from the processor portion of memory, and with the processor dumping I/O commands into an I/O controller parallel port, without directly talking with any I/O devices.

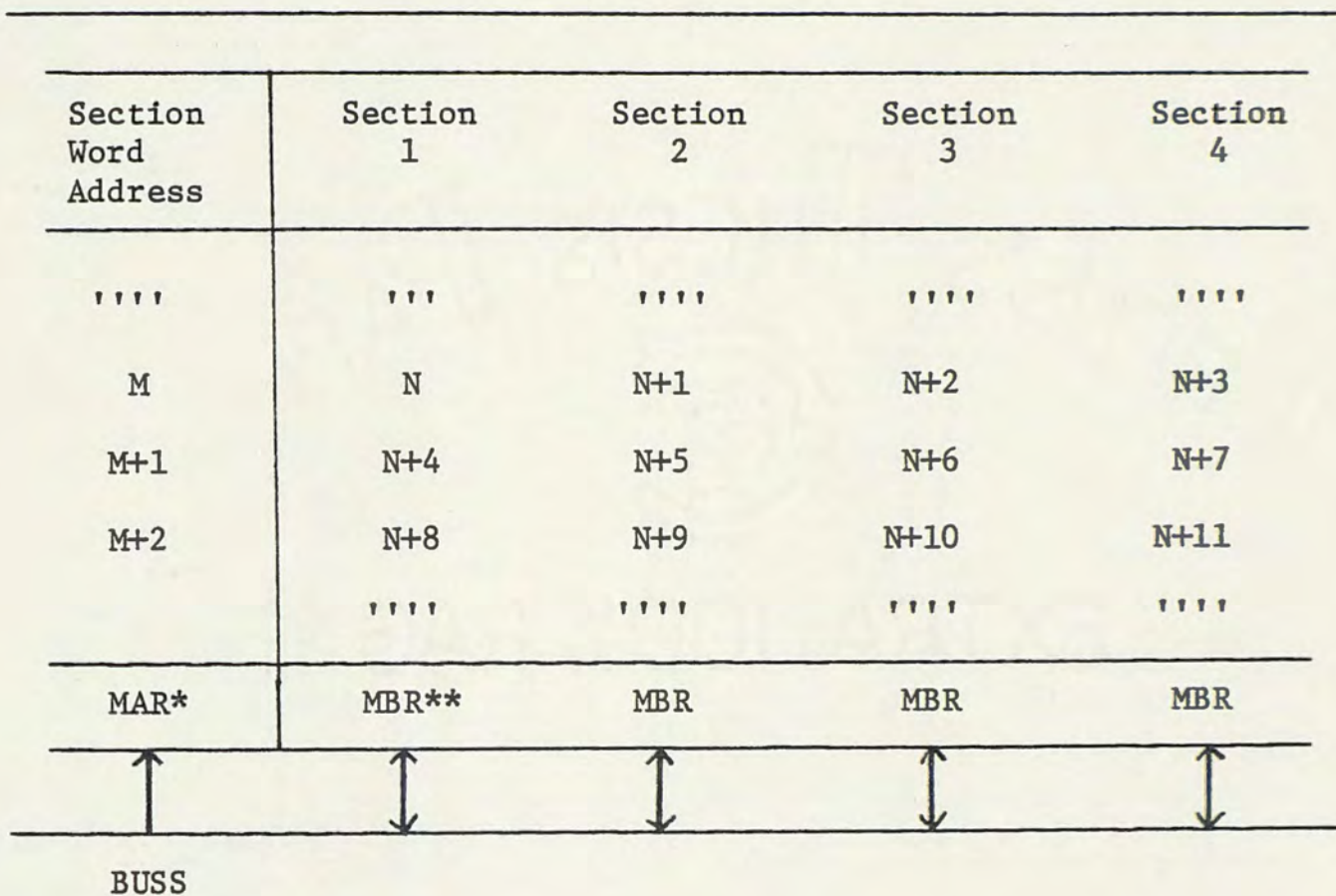


Fig. 7. Parallel Storage Increases Memory

\* MAR = Memory Address Register

\*\* MBR = Memory Buffer Registers

## B. DATA OPERATIONS

For best processor Throughput, the data operations need to be as parallel or one-step a procedure as is feasible.

### 1. FLOATING POINT PROCESSOR (FPP)

One of the key points of this report is that a scientific-computation-oriented minicomputer needs to have a hardware Floating



Point Processor to handle the high-precision arithmetic operations. Since the FPP can provide a 100X Throughput improvement, either the computer system should have one from the start, or one should be designed and built by some graduate students as a research project. But if there is no FPP, there is no reason to implement the other proposals of this report, since the FPP gives such a big benefit.

The following parameters need to be considered when specifying the FPP:

1. Is it an integral part of the processor or is it treated as an I/O device with the attendant data movement delays
2. How many full width registers are included in the FPP, which provide needed storage to minimize the moving of data at inopportune moments
3. Is the FPP expandable to wider words and greater precision by a control instruction, or must triple-word (48 bit resolution) operations be executed by software or software-hardware combinations at a serious Throughput penalty
4. What degree of parallelism should the FPP provide for the multiply operation

For best Throughput, the FPP should be an integral part of the processor, with immediate access to the processor buses, registers, and cache memory. Particularly for Floating Point Addition and Subtraction, where the majority of the instruction execution time will be spent in aligning the decimal points before parallel add or subtract and the additional time needed to move two operands



to an I/O device and move the result back to the processor register files compares with the actual execution time, keeping the FPP in the processor is justified. In addition, the BUS is then less needed by the processor, and I/O data movement is enhanced.

The second FPP parameter is the number and size of registers it retains for its own use. Since maximizing Throughput requires keeping the FPP as busy as possible without delaying operations because the operands are not available, at least 6 registers, 48 bits wide, are needed to hold the operands and results of two successive, completely separate arithmetic operations which were executed while the processor buses or cache memory were busy with other activities. Therefore, the SN74172 dual-port register file is suggested, supplying 8 words X 2 bits in each integrated circuit, and being able to drive two buses with different operands.

The third FPP parameter, expandability, is determined by the size of the adders and shift registers of the FPP. One-step addition and subtraction requires a 32 bit adder (which assumes 32 bit resolution) as does the iterated steps multiply and divide, so including the capability for 48 or 64 bit resolution computation merely requires 4 or 8 more 4 bit adders and 1 or 2 lookahead logic functions (which is used to keep the time to add 64 bit operands down to 2 or 3 times the delay of a single 4 bit adder). The multiply and divide functions also will require a 96 or 128 bit shift register, which is 16 SN74198 ICs. By including at least 13 more ICs, the FPP can be expanded to 64 bit arithmetic operations, thus



avoiding obliging the programmer who needs more than 32 bit operations to fall back to software implementation or a  $(M + N) (a + b)$  partial product approach.

The fourth FPP parameter, degree of parallelism of the actual act of multiplication, is determined mainly by affordability. Secondary considerations are space and power, which at least for earth-bound computer-systems, still reduce to a matter of cost. The cheapest implementation, the add-the-multiplicand-to-the-partial-product-if-the-next-multiplier-LSB-is-1, can easily yield step times of  $\sim 150$  nsec/bit, or 4.8 usec for the basic operation plus 0.5 usec instruction setup time (with the sign and exponent of the product being computed during the 4.8 usec) which yields 5.3 usec for 32 bit multiplication.

The use of clockless multiply ICs such as the Fairchild 9344 (9) will give a 32 bit product, truncated from 54 bits, in 750 nanoseconds. An expansion of 64 bit operands requires 4 times as many ICs and power, or --- of the 9344 ICs.

A third approach uses the Advanced Micro Devices AM25LS14, (10) a one-clock-pulse per bit of product serial multiplier function, which enables the use of 4 ICs for a 32 bit multiplier. One operand is presented in parallel to the 8 inputs of each of the ICs, and the other operand is clocked serially into the end of each of the multipliers. The allowable clock rate for 32 bits is 6 MHz, or 10.2 usec for a 32 bit multiplication. By using 16 of



the ICs to generate 4 partial products, with only 32 clock cycles, and then adding the partial products with 3 adders, the time for a complete 32 bit multiply is 0.5 usec setup + 5.6 usec partial multiply + 0.2 usec addition, a total of 6.3 usec, no speed improvement over the first approach, the Booth algorithm, mainly because 32 clock pulses are required.

The non-parallel version of this approach is readily expanded to 48 or 64 bit operands by simply using 6 or 8 multiplier chips and thus is recommended if more than 32 bit operations are likely.

The most reasonable pseudo-parallel approach is a partial product approach using Medium Scale Integration logic which yields partial products in 8 clock pulses instead of 32, requiring about 40 ICs. If used with a 10 MHz clock rate, it would result in partial products in 0.8 usec and complete results in 0.5 usec setup + 0.8 usec multiply + 0.2 usec addition, totaling 1.5 usec. This approach is diagrammed in Figure 8. By reconfiguring the shift registers and adders, 64 bit multiplications can be performed in 0.5 usec setup + 3.2 usec multiply + 0.2 usec addition, totaling 3.9 usec.

This last technique, because of its inherent parallelism, speed, and expandability, is recommended for use in a scientific computing system.



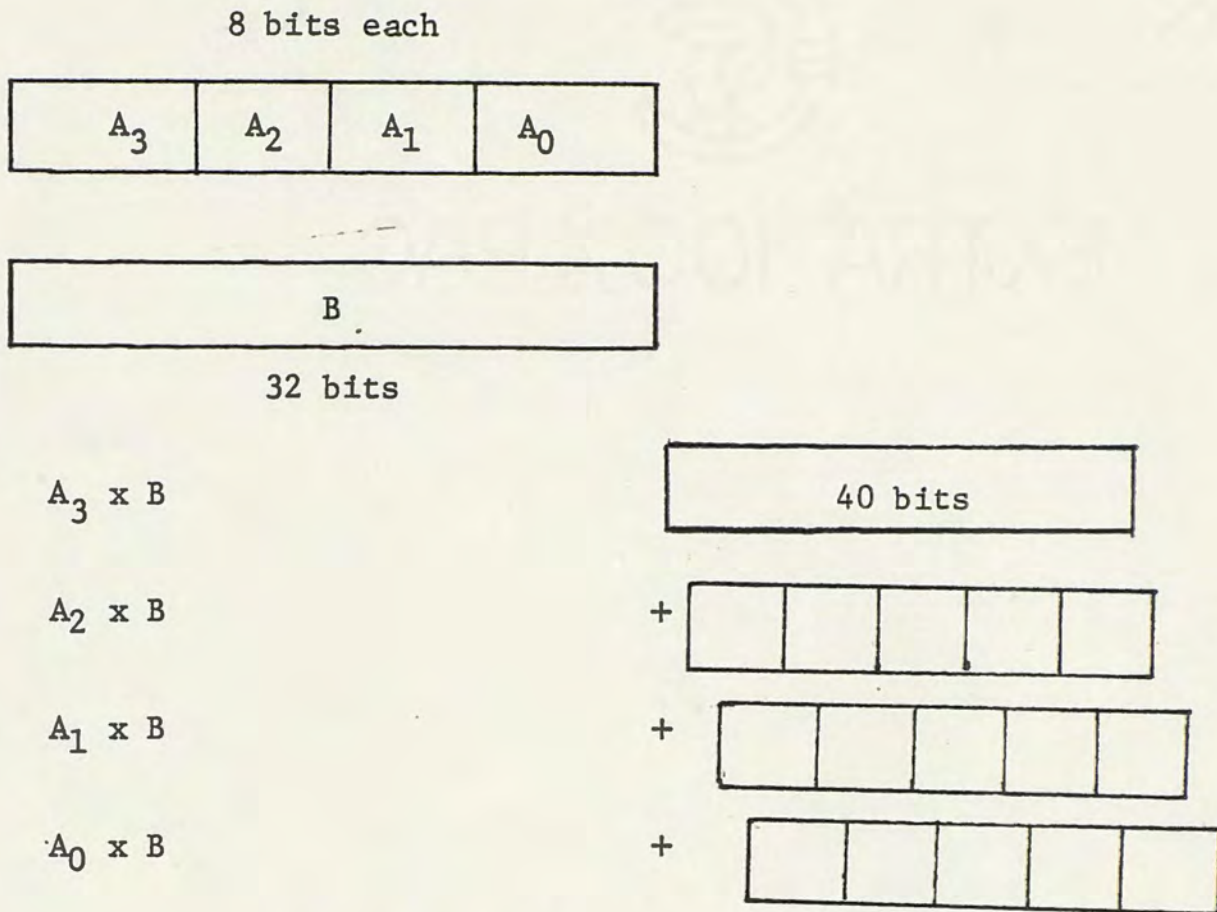


Fig. 8. Pseudo-Parallel multiplication also allows effective execution of double-precision multiplication by reconfiguring the shift registers and adders.

## 2. ARITHMETIC-LOGIC-UNIT OPERATIONS

### (INTEGER ARITHMETIC)

The ALU, which provides one-step 32 bit operations such as add, subtract, OR, AND, COMPLEMENT AND SHIFT, can execute its operations in well under 100 usec for all but multiple shifts.

By executing these operations from the two processor buses, the additional time delay of synchronously clocking the operands into holding registers is avoided. Since the operand access time is 75 nsec and the transfer time is 100 usec, with (for example)



the maximum 32 bit add time of 60 nsec, the operations can be executed in 2 cycles of the 10 MHz clock instead of 3.

Figure 9 illustrates this execution time reduction, particularly valuable when linked with instruction lookahead.

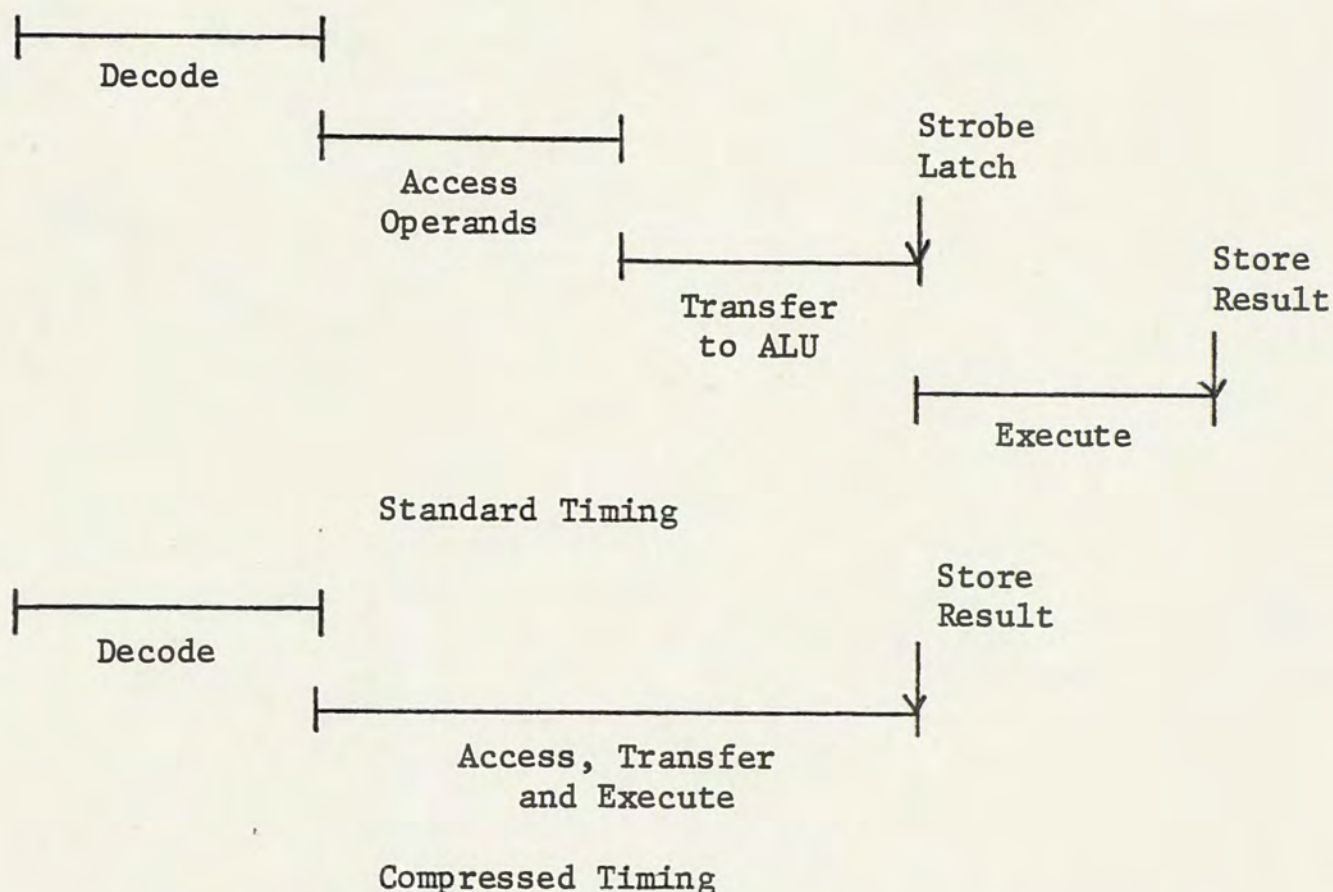


Fig. 9. Execution from Buses speeds One-Step Operations

### 3. COMPOUND OPERATIONS OF ALU

As has been repeatedly emphasized, one of the techniques used to enhance Throughput is to move data as little as possible, mainly by keeping data near where it is used, not out in memory. By making available compound instructions such as Add-and Branch-if-Zero, the processor can avoid having to set-up the operands for two instructions.

For example, to add two numbers and change program flow



based upon the sum by using the PDP-11/20 instruction set requires two instructions:

```
ADD (A + B → B)
```

```
BEQ B,J (branch if equal to J)
```

while the Nova 1200 allows the following

```
ADDZ      A,B,  SZC
```

(skip next instruction if  $A \neq B$ )

Granted that the Nova instruction cannot reference memory or I/O devices nor can the skip action directly yield large changes in the addresses (although Program Counter relative addressing could be used) but the intent of this report is to have the operands in registers and the instructions in cache memory so there is no need for large addressing fields. Thus if the instruction is executed in one continuous flow, there is no need to load the intermediate results in temporary registers and even this slight delay can be avoided.

To summarize, if we take advantage of the operands being in registers and use compound instructions as permissible, then the ADDZ,A,B and BEQ B,J execute times with the operands stored in main memory (needing 7 read or write operations or ~ 3 usec) can be reduced to (75 nsec get instruction, 100 nsec decode, 25 nsec get A and B, 75 nsec add A to B, 50 nsec compare sum, 75 nsec get J, 75 nsec add J to Program Counter and load in PC) a total of 475 nsec, or a Throughput improvement of 6 times.



### III. SOFTWARE CONSIDERATIONS

In this portion of the report, section IIIA determines how the software can best utilize the capabilities of the available hardware, presumably that suggested under Hardware Considerations. In Section IIIB and C we look for special contributions to Throughput that certain other software features, mainly variable instruction length, can provide.

#### A. SYSTEM BUSES

The software will not be directly concerned with the BUS(es) and processor buses. The BUS assignments will be handled by the I/O controller, with short processor requirements, such as a 4 memory-words-in-parallel-transfer, given priority. This reversal of the usual priority hierarchy is possible because of the buffering capabilities of the I/O controller.

Nor will the software be concerned about the processor buses, as their usage will be handled by processor control logic, which will probably be either conventional logic or a highly-parallel control word microcontroller, so as to support system speed requirements.

#### 1. REGISTERS

It is intended that all processor working registers be accessible by the same instruction type, while the Supervisory Mode



registers use still another instruction which is restricted to being used by the Operating System (OS). By using only one instruction to access a number of registers, although in separate register files of possibly different sizes, the assembler and compiler are simplified and the logic circuitry needed to select the different files is not increased over that needed by separate instructions.

Also all of the I/O controller registers and data files should be accessible by the OS, so that they may be transferred to or from memory in response to a Power Fail Shutdown or Restart.

## 2. CACHE MEMORY

The processor's cache memory is supported by three special instructions. The first is Multiple Fast Transfer, which guides the block transfer of data from one point in the system to another, not just to cache memory. The second is Conditional Control Transfer, used to transfer control of the processor instruction decode logic from the main memory PC to the cache PC or vice versa, to switch to and from Supervisory mode and to force the processor to operate in the fixed-length instruction mode instead of the variable-length mode.

The third instruction is the type of main memory reference instruction which occurs when the processor is executing instructions from cache memory and suddenly needs to go outside cache memory boundaries. The uniqueness comes by the address of the actual main memory location being computed from the base register for that program, the index register for the particular page of the



program, and the sum of the CPC and the memory address displacement supplied by the memory reference instruction. The capability must exist for this type of addressing.

If the cache is large enough to hold several program segments, with the execute time of any particular segment being long enough to load the cache with the next program segment, then the flow of execution will keep rolling around the cache boundaries; this continual flow of execution can only be implemented by using base and index registers, and the associated "memory" referencing instructions.

### 3. I/O CONTROLLER

The intent of the I/O controller is to free the processor from having to guide I/O activity, and to add certain hardware features which software is too slow to handle anyway, such as disc and tape error detection and correction, and the buffering of high speed data block transfers.

The OS needs to be able to guide the I/O controller, either by direct communication on the system BUS or by presenting commands at a special parallel controller port. Instructions need to be able to handle the following demands:

1. Modify priorities of peripherals as their importance to a program or different programs changes, by a command from OS
2. Be able to acknowledge or ignore peripheral interrupts during preventive maintenance or equipment failure, so that the system is not paralyzed by uncompleted data transfers



3. Be able to handle the discovery of a parity error, or worse, resulting from an I/O transfer or a file read, so far as initiating a retransmit or a reread, or by recording the device and data address where the fault occurred so as to facilitate repair

The last requirement implies that the I/O controller should handle I/O error checking and system error record-keeping in error status registers. Since hardware logic can be more cost effective in finding/correcting I/O and memory errors than can the OS, the only error checking done by the processor should be monitoring for processor errors, but again with hardware. The OS may periodically monitor the error status registers.

#### 4. MAIN MEMORY PARTITIONING

Physical partitioning of main memory was presented as a technique for obtaining rapid transfer of blocks of data. It is also useful for maintaining separation of tasks in a time-shared environment where it is advantageous to keep at least part of the OS in memory as well as user programs awaiting data from mass memory or from special devices such as Fast Fourier Transform modules, where disc swapping would be ineffectual. There is a need for the OS to be able to reconfigure the memory for a better task fit. This falls under the domain of memory management, and should be linked with what is actually resident in the cache memory.

A similar situation occurs when a separate Task-Scheduling processor is concerned with keeping the scientific processor fully occupied with number crunching while it handles the execution of the



OS, as does the B6500 of the ILLIAC IV system (11).

## 5. FLOATING POINT PROCESSOR

There are two basic types of instructions which guide the FPP. The first, as may be expected, are those which specify the various floating point operations and the registers wherein the operands are located. The operations are:

1. add
2. subtract
3. multiply
4. divide
5. invert

The invert operation is included because it provides a useful function, which is often used in matrix operations, without requiring the initialization of a register with 0001 to serve as a dividend.

The second type of instruction is concerned with the expandability of the FPP. The actual technique used to expand the characteristic size from 32 bits to 48 or 64 may be selected from Section II.B.1. With expanded precision, the 48 bit registers will not hold all of an operand, thus it will be necessary to specify six 48 bit registers (4 operand, 2 for the result) instead of only 3.

The expanded precision instruction should also indicate whether 48 or 64 bits (or other) is being used, as each extra bit of precision requires an extra 100 nsec. One field of the instruc-



tion could contain a binary count of the precision, which is loaded into a down-counter in the FPP, where a Borrow output from the counter halts the computation.

## 6. MEMORY REFERENCE CAPABILITY OF COMPOUND INSTRUCTIONS

One area of software support required by compound instructions comes from the need to be able to execute these operations with the operands contained in either processor files or main memory. Unlike the FPP instructions, which take from  $\approx 500$  nsec for an addition with no decimal point alignment needed, to as long as  $\approx 30$  usec for an extended-precision 64 bit multiplication, and where the instruction lookahead has time to access the operands for the next operation and move them from main memory if needed, the compound operations are so short (<100 nsec execution time, using instruction lookahead) that using separate instructions, to access the operands and store the result back in main memory, is a considerable waste of processor time and memory space. This is illustrated in Figure 10, where the different parts of an instruction execution sequence are assigned typical operate times.

Another advantage occurs where the cache memory branches to main memory for some flag status check or update; if the necessary activity can be pulled from memory in the form of one long instruction, then the system can avoid the Throughput penalizing need of multiple memory accesses.







A third advantage is that if the processor is given a few general-purpose registers which the user programs cannot directly access, then execution of these status monitor functions (or whatever) can proceed without the need for the programmer to move data from registers to cache or memory to make temporary working space.

These three advantages also apply for the other operations of the ALU.

#### B. SPECIAL SOFTWARE CONTRIBUTIONS

The bulk of the Throughput-improving factors presented by this report have been in the hardware. There is, however, one software factor which can significantly affect Throughput. This is the availability and proper application of variable-length instructions.

The benefit arises by not having to force the processor control statements (instructions) into fixed word lengths. It has been shown in Section II.A.3 that a MFT instruction has wide applicability, even though it will need 45 bits of the available 48 allocated as follows:

1. op code--6 bits.
2. cache starting address--10
3. cache or registers--3
4. Number of words to be transferred--10
5. Starting location of memory block--16

However, the bulk of operations, particularly when executing instructions from cache memory, do not need to be 48 bits long.



By using cache PC relative addressing, the address displacement can be limited to 10 bits. Register specification can be limited to 2 or 3 fields of 5 bits or less, so 16 or 24 bit instructions are certainly reasonable and thus justify double-or-triple packing in a 48 bit word.

The following section presents an even denser packing of instructions, coupled with a highly structured operand movement technique; the intent is to minimize both operand movement and instruction access and decode time, mainly by employing very simple instruction formats.

### C. POLISH NOTATION EXECUTION

This paper has repeatedly emphasized that a computer should be judged primarily by its Throughput. A previously mentioned approach to improving Throughput is that of reducing the instruction execution time by storing the program in cache memory. Here we examine another approach of simplifying the instruction format to permit packing two or more instructions per memory word. Obviously it will be difficult to implement memory referencing in small instructions (8 to 24 bits long); indeed, it is even difficult to specify different registers. Perhaps this new approach may be best described as having the operands automatically moved into position--no explicit operand selection. This technique of implicit operand selection corresponds to the technique of Polish Notation--PN.

An example of conventional algebraic notation, requiring



explicit operand selection/location, is

$$(a+b) * (c-d)/f$$

This expression could be evaluated as follows:

1. Evaluate  $a+b$  and store in  $g$
2. Evaluate  $c-d$
3. Multiply  $g$  times  $c-d$  and
4. Divide product by  $f$

PN would rearrange the previous expression as

$$ab+cd-*f/$$

which would be evaluated as previously done, with the difference being that the storage location  $g$  is not required. This assumes that a subtract sign means  $c-d$ , not  $d-c$  subtraction sign/opcode would also be useful.

It is recognized that the following operations are needed:

1. Addition of two numbers             $a+b$
2. Subtraction of two numbers         $a-b$  or  $b-a$
3. Multiplication                       $a \times b$
4. Division                              $a/b$  or  $b/a$
5. End of PN execution list

The processor will be responsible for the actual data operations; it must manage the operand movement and the arithmetic operations as required by the PN op code (at this point the size of the PN op code is undefined).

The previous operations 1 through 5 are actually 7 distinct instructions. It may be argued that the order-dependent operations



of subtraction and division do not have to be bipolar; the Algebraic-to-PN conversion program could be written so that only order-independent operations need be available. However, it is felt that the provision for order-dependent execution will cause little if any time penalty but will permit a simplification of the Algebraic-to-PN conversion program and a considerably easier task of manual conversion.

Another arguable point is the need for inclusion of logic operations. To "resolve" both arguments, it has been decided to set the PN op code at 4 bits, thus allowing a considerable expansion of the set of 7 previously discussed.

A third consideration is "why has not PN become popular?" One answer is provided by the article "Microprogramming, Stack Architecture Ease Minicomputer Programmer's Burden" in the February 15, 1973 issue of Electronics (12). To quote,

"In addition, the stack concept is convenient for writing the compiler. Proof is that compiler writers using conventional computers create stack environments in software. Thus, from the standpoint of any user the availability of a minicomputer with a stack architecture makes it cheaper to obtain a compiler for the particular high-level language that suits his application."

And the answer is--stacks are popular (with enhanced PN execution a main reason) but a stack which operates without software assistance does require a considerable amount of hardware--an amount comparable to a small computer of several years ago. Figure 11, excerpted from the Burroughs B5000 manual (13), presents the stack components.



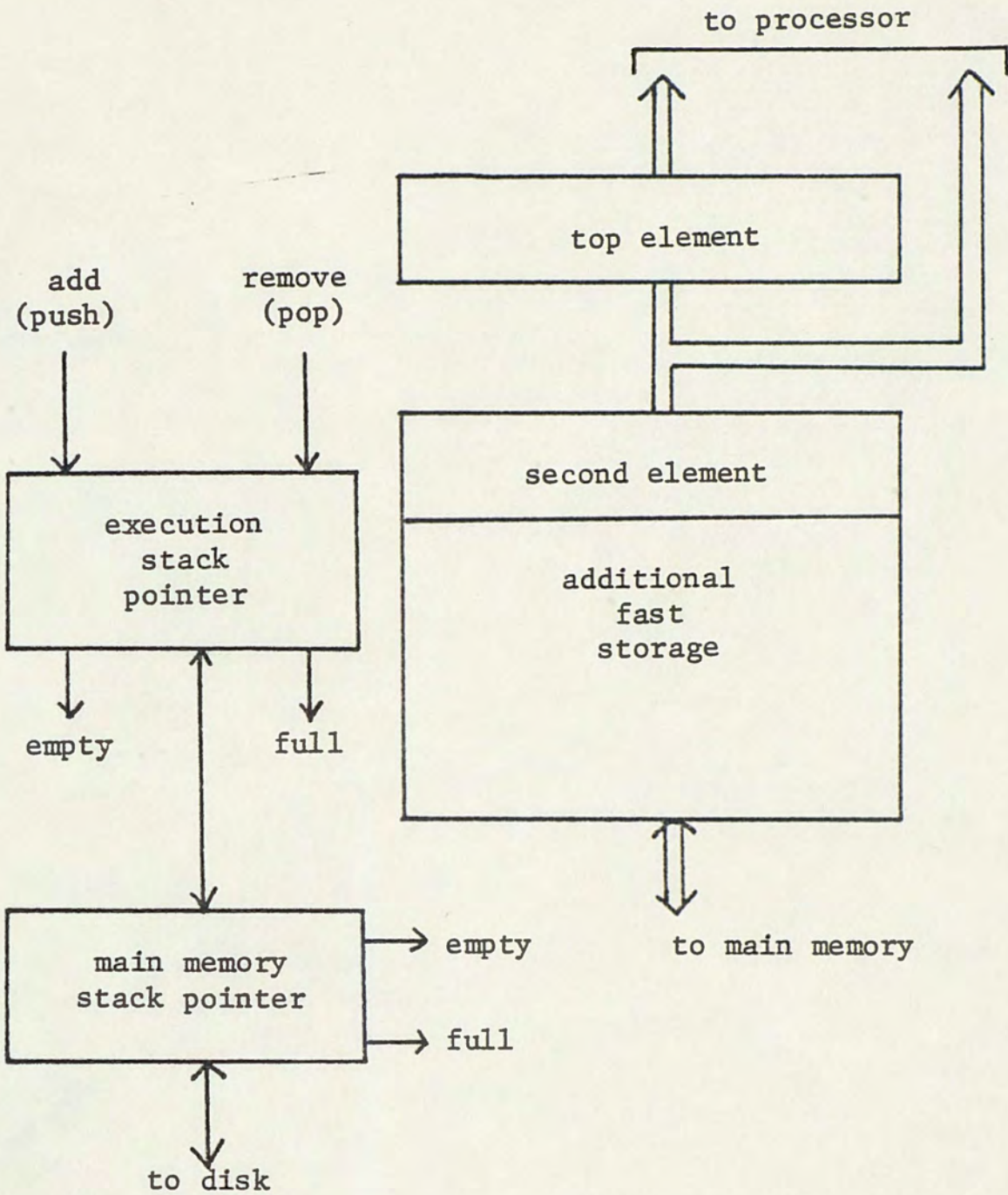


Fig. 11. Hardware Elements of a Stack

Perhaps the most straight forward stack implementation would be a shift register with the properties:

1. N-bits wide (N is the size of the operands)
2. Very long or deep
3. Left and right shiftable



4. At least the top 2 elements visible to the processor. This hardware element does not exist. Indeed, the author is not aware if even a finite length by m-bits wide shift register exists. However, such a stack could be implemented with large quantities of 8-bit-long shift registers (e.g. SN74198).

A slightly different approach uses IC RAMs such as the SN7489, a 16 words of 4 bits memory. At current per-bit prices, the 7489 is ~ 75% cheaper than the 74198 but is slower in that a Read/Write cycle is required rather than a simple shift. Figure 12 presents the operation of a RAM implemented stack, which is executing the function

$$A - \left( \frac{F \times E \times D}{C} + D \right)$$

The action codes are as follows:

- N. PUSH operand onto stack
- F. POP operand from stack
- A. Add top to second element
- M. Multiply top and second element
- D. Divide top element by second element
- E. Divide second element by top element
- S. Subtract top element from second element
- T. Subtract second element from top element
- H. Execute next cache word as a conventional instruction



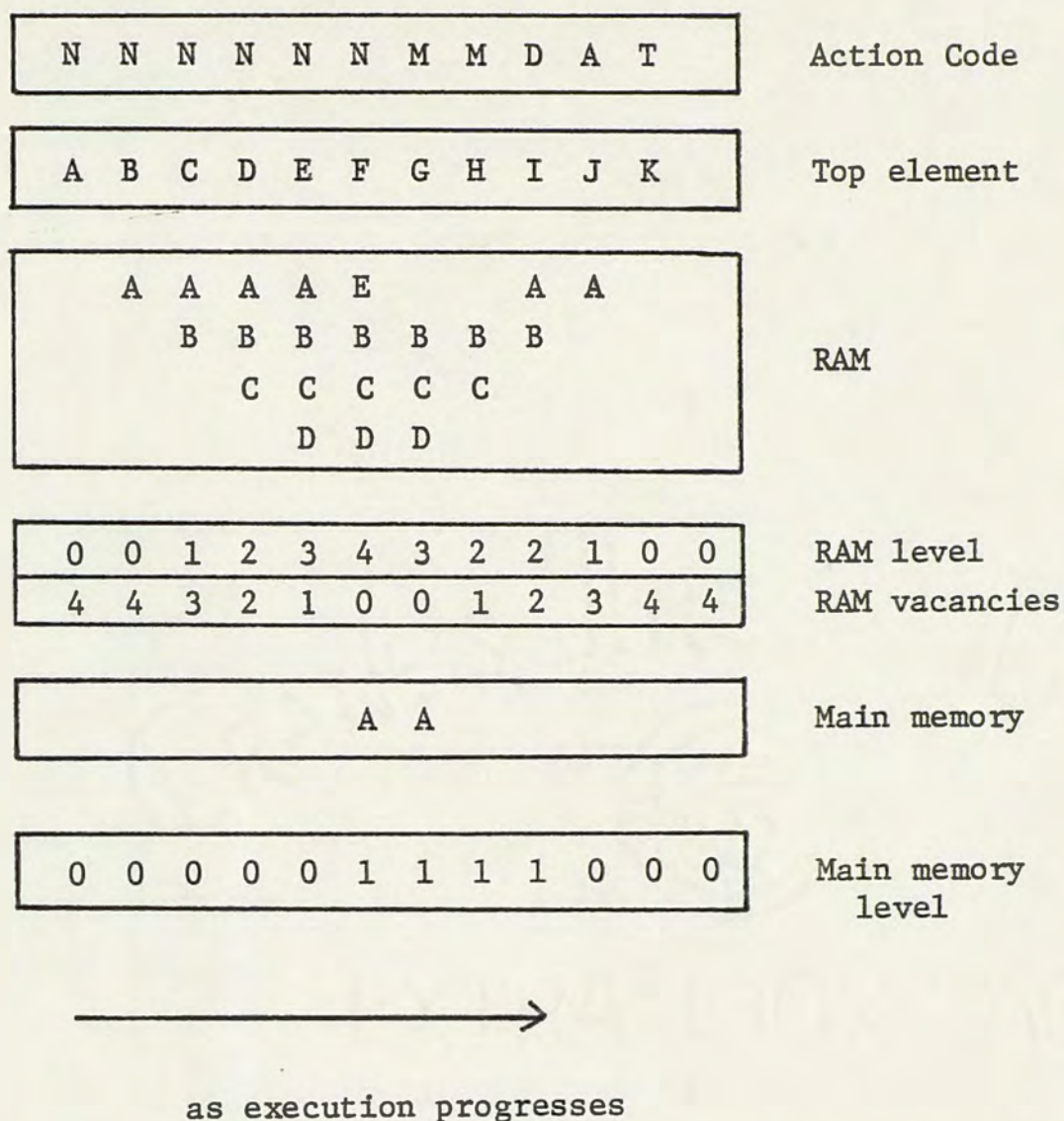


Fig. 12. Typical Stack PUSH, POP, and Execute Activity

The general philosophy is one of (1) be hesitant to move words from RAM to main memory (move only when RAM is full, or when a PN op code says to) and (2) be quick to move words from main memory to RAM (do so whenever the RAM is less than half full, unless inhibited by a PN op code) but do not fill up the RAM.

At this point it is assumed that a stack has been implemented, via the cache memory and cache PC. It is now necessary to develop additional PN instructions; this is done by observing the



PN implementation of a series evaluation.

For real values of X

$$\sin X = \frac{X^1}{1!} - \frac{X^3}{3!} + \frac{X^5}{5!} - \frac{X^7}{7!} + \dots$$

(X in radians)

which may be rewritten as (using the first 4 terms)

$$\sin X = X \left( 1 - \frac{X^2}{3 \times 2} \left( 1 + \frac{X^2}{5 \times 4} \left( 1 - \frac{X^2}{7 \times 6} \right) \right) \right)$$

This could be programmed in the following fashion, beginning with the innermost operations

<u>OPERATION/CODE (8 bit word)</u>	<u>NEW TOP ELEMENT</u>
Push	X
Push	1
Push	1/6
Push	X <sup>2</sup>
Push	1
Push	1/20
Push	X <sup>2</sup>
Push	1
Push	1/42
Push	X <sup>2</sup>
M	X <sup>2</sup> /42
S	1-X <sup>2</sup> /42



M	$X^2(1-X^2/42)$
M	$(X^2/20)(1-X^2/42)$
A	$1+(X^2/20)(1-X^2/42)$
M	$X^2(1+(X^2/20)(...))$
M	$(X^2/6)(1+(X^2/20)(...))$
S	$1-(X^2/6)(.....)$
Pop	top element of previous work

The significant addition is the PUSH command, used to load operands onto the stack, and the POP command which is used to remove the answers from the stack and store them in a register; a conventional Data Move instruction could be used to access the answer (if the Top Element could also be treated as a register) for use elsewhere but would not remove the answer from the stack.

It is not possible to contain PUSH and POP within the previously mentioned 4 bit op code. Both commands must specify a register to be the operand source or sink, respectively. It is possible to have PUSH and POP communicate with only one register, but there is a more effective approach. Notice that the sin X evaluation PUSHes 6 different operands and if only one register were available, the program would have had to End PN Execution, load the register and commence PN execution on 6 different occasions.

The better approach is to initialize a group of registers with the necessary operands--for example  $X^2$ ,  $1/42$ ,  $1$ ,  $1/20$ ,  $1/6$ ,  $X$ --



and then PUSH the operands onto the Stack when needed, without having to exit and return PN execution. Of course several bits will be needed to indicate which register holds the operand (or is to receive the operand, if POP is commanded). How many bits will suffice for most PN programs? Another example will help.

Another example is that of Matrix Inversion, with the matrices stored in cache.

$$A^{-1} A = I = A A^{-1} ; A , A^{-1} , I \text{ of order } \underline{n}$$

$$\begin{bmatrix} A_{11}^I & A_{12}^I \\ A_{21}^I & A_{22}^I \end{bmatrix} \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

By Gaussian Reduction of the A matrix we have

$$\begin{bmatrix} A_{11}^I & A_{12}^I \\ A_{21}^I & A_{22}^I \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} C & D \\ E & F \end{bmatrix}$$

requiring  $n$  divisions and  $1/2 n(n+1)$  multiplications and additions. For matrices containable in the available processor files, autoincrementing index registers could be used to point to the operand-holding register (with the incrementing triggered by the index register being used to compute a cache address).

For matrices which must remain in main memory, autoincrementing index registers will point to the memory word. The key idea is the use of autoincrementing index registers.

Remember the point of discussion is how many registers to make available to the PUSH and POP instructions; or how big is the



register specification field of the PN instruction? A consideration of the data base addressing requirements shows that 16 registers, or a 4 bit field, should be adequate. However, since there is 8 bits in the instruction, the # of accessible registers is increased to 32, or a 5 bit register select field, to allow referencing most, if not all, of the processor registers. This requires that the PUSH and POP instructions be recognized by 3 bit op codes. How should these 32 available registers be used.

Several should be autoincrementing index registers. The others will be conventional working registers, used to hold constants--such as 1, or to hold pointers which are used to reinitialize the autoincrementing index registers after a matrix has been inverted and a new matrix is ready for inversion.

This analysis results in the following PN instructions, 8 bits wide:

1. ADD             $A + B \rightarrow A$
2. SUBTRACT      $A - B \rightarrow A$
3. SUBTRACT      $B - A \rightarrow A$
4. MULTIPLY      $A \times B \rightarrow A$
5. DIVIDE         $A / B \rightarrow A$
6. DIVIDE         $B / A \rightarrow A$
7. PUSH    cache (reg xxxxx)
8. POP     cache (reg xxxxx)
9. End of PN Execution (next 48 bit word should be interpreted as a conventional instruction)



#### IV. CONCLUSIONS

Although the bulk of the proposed Throughput enhancements have been hardware oriented, they result in dramatic software changes as well. To make best use of the new hardware resources, the software needs to be equally carefully considered.

We have shown that the most important feature, hardware or software, is the inclusion of a Floating Point Processor; by using a pseudo-parallel approach, as much as a 200X Throughput improvement can be gained, as compared to a software floating point package.

The next most significant position should be shared by the cache memory, which can provide at least a 5X improvement in effective memory access times with a lesser Throughput improvement, and by the Polish Notation technique of structuring data and permitting simple instructions. The other features all together boost the Throughput by smaller amounts as estimated below:

- (a) wide buses -- 1.25X
- (b) numerous registers -- 2X
- (c) instruction lookahead -- 1.25X
- (d) I/O controller -- 1.5X
- (e) partitioned memory -- 1.25X

The product of these factors is 5.9X, a very nice Throughput enhancement for any computer, but especially effective when coupled



with the three previously mentioned features.

The result is a computer with a maximum of 3 or 4 Mega Instruction per Second execution rate, and capable of 600,000 floating point multiplications per second.



## APPENDIX

## How Large Should The Cache Memory Be?

The following example is excerpted from a Fortran program that was written to compute and graph the spectral content of various waveforms, which the program also generated. The excerpt is the Fourier Transform computation routine.

<u>Fortran Statements</u>	<u>Memory Words per Statement</u>
DO 115 N=1,NW (NW=100)	8
H=N	2
W=H*DW	4
RT=.0	2
GT=.0	2
DO 112 K=1,NT (NT=200)	8
H=K	2
T=AT+H*DT	5
R=A(K)*DT*CPS(W*T)	43
RT=RT+R	4
G=A(K)*DT*SIN(W*T)	43
GT=GT+G	4
112 CONTINUE	



AMP=SQRT(RT*RT+GT*GT)	30
DATA(N)=AMP	1
115 CONTINUE	_____
TOTAL	158

CALL GRAFTU(DATA,NW)

The outer loop (loop counter N) computes 100 spectral points, while the inner loop (loop counter K) uses the 200 time function points to compute each spectral point.

In estimating the number of conventional machine level instructions to equal this Fortran excerpt, we assume that  $A*B+C$  (for example) requires 4 instructions (A to (P)rocessor, B to (P),  $A*B$ , C to (P)  $+A*B$ ) and that a  $\sin(X)$  or  $\cos(X)$  function with .0000001% accurate result needs  $X^{17}/17!$  as the last term, with 3 instructions ( $1/N!$  to (P),  $(1/N!)*X^2$ ,  $1+X^2/N!$ , with  $X^2$  and 1 contained in registers per term, needs about 35 instructions including setup operations.

The total number of instructions is 158, with approximately 50 operand storage locations plus an array of 100 locations and another of 200, requiring 408 words of high speed storage.

To allow for even bigger computation loops (the example is admittedly simplistic) and to minimize the need for swapping arrays from cache to main memory as various sections of the arrays are needed by the program, at least 1K word of cache memory should be available.



## LIST OF REFERENCES

1. Foster, C. Computer Architecture. New York: Van Nostrand Reinhold, 1970.
2. How to Use the NOVA Computers. Southboro, MA: Data General Corporation, 1971.
3. Processor Handbook PDP11/20-15-r20. Maynard, MA: Digital Equipment Corporation, 1971.
4. Programmers Reference Manual for the Model 960A Computer. Dallas, TX: Texas Instruments Incorporated, 1971.
5. The TTL Data Book for Design Engineers. Dallas, TX: Texas Instruments Incorporated, 1973.
6. Hellerman, H. Digital Computer System Principles. New York: McGraw-Hill, 1967.
7. The Value of Power. Anaheim, CA: General Automation Inc., 1973.
8. Lorin, H. Parallelism in Hardware and Software (Real and Apparent Concurrency). Englewood Cliffs, N.J.: Prentice Ha-1, 1972.
9. TTL Data Book. Palo Alto, CA: Fairchild Semiconductor Inc., 1972.
10. Mick, J., ed. Digital Signal Processing Handbook. Sunnyvale, CA: Advanced Micro Devices, Inc., 1976.
11. Bell, G. and Newell, L. Computer Structures: Reading and Examples. New York: McGraw-Hill, 1971.
12. "Microprogramming, Stack Architecture Ease Minicomputer Programmers Burden." Electronics 46 (February 15, 1973): 95-101.
13. Burroughs B6700 Information Processing Systems Reference Manual. Detroit: Burroughs Corporation, 1972.