Chapter 18

The IBM System/360 Model 91: Machine Philosophy and Instruction-Handling[1]

D. W. Anderson / F. J. Sparacio / F. M. Tomasulo

Abstract The System/360 Model 91 central processing unit provides internal computational performance one to two orders of magnitude greater than that of the IBM 7090 Data Processing System through a combination of advancements in machine organization, circuit design, and hardware packaging. The circuits employed will switch at speeds of less than 3 nsec, and the circuit environment is such that delay is approximately 5 nsec per circuit level. Organizationally, primary emphasis is placed on (1) alleviating the disparity between storage time and circuit speed, and (2) the development of high speed floating-point arithmetic algorithms.

This paper deals mainly with item (1) of the organization. A design is described which improves the ratio of storage bandwidth and access time to cycle time through the use of storage interleaving and CPU buffer registers. It is shown that history recording (the retention of complete instruction loops in the CPU) reduces the need to exercise storage, and that sophisticated employment of buffering techniques has reduced the effective access time. The system is organized so that execution hardware is separated from the instruction unit; the resulting smaller, semiautonomous "packages" improve intra-area communication.

Introduction

This paper presents the organizational philosophy utilized in IBM's highest performance computer, the System/360 [Amdahl, Blaauw, and Brooks, 1964] Model 91. The first section of the paper deals with the development of the assembly-line processing approach adopted for the Model 91. The organizational techniques of storage interleaving, buffering, and arithmetic execution concurrency required to support the approach are discussed. The final topic of this section deals with design refinements which have been added to the basic organization. Special attention is given to minimizing the time lost due to conditional branches, and the basic interrupt problem is covered.

The second section is comprised of a treatment of the instruction unit of the Model 91. It is in this unit that the basic control is exercised which leads to attainment of the performance objectives. The first topic is the fetching of instructions from storage. Branching and interrupting are discussed next. Special handling of branching, such that storage accessing by instructions is sometimes eliminated, is also treated. The final section discusses the interlocks required among instructions as they are issued to the execution units, the initiation of operand fetches from storage, status switching operations, and I/O handling.

CPU Organization

The objective of the Model 91 is to attain a performance greater by one to two orders of magnitude than that of the IBM 7090. Technology (that is, circuitry and hardware) advances[2] alone provide only a four-fold performance increase, so it is necessary to turn to organizational techniques for the remaining improvement. The appropriate selection of existing techniques and the development of new organizational approaches were the objectives of the Model 91 CPU design.

The primary organizational objective for a high performance CPU is concurrency—the parallel execution of different instructions. A consideration of the sequence of functions involved in handling a typical processor instruction makes the need for this approach evident. This sequence—instruction fetching, instruction decoding, operand address generating, operand fetching, and instruction execution—is illustrated in Fig. 1. Clearly, a primary goal of the organization must be to avoid the conventional concatenation of the illustrated functions for successive instructions. Parallelism accomplishes this, and, short of simultaneously performing identical tasks for adjacent instructions, it is desired to "overlay" the separate instruction functions to the greatest possible degree. Doing this requires separation of the CPU into loosely coupled sets of hardware, much like an assembly line, so that each hardware set, similar to its assembly line station counterpart, performs a single specific task. It then becomes

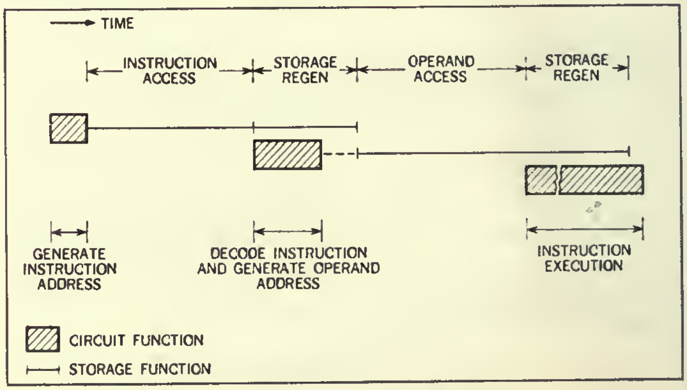


Fig. 1. Typical instruction function time sequence.

[1] *IBM Journal*, vol. 11, January 1967, pp. 8–24.

[2] Circuits employed are from the IBM ASLT family and provide an in-environment switching time in the 5 nsec range.

possible to enter instructions into the hardware sets at shortly spaced time intervals. Then, following the delay caused by the initial filling of the line, the execution results will begin emerging at a rate of one for each time interval. Figure 2 illustrates the objective of the technique.

Defining the time interval (basic CPU clock rate) around which the hardware sets will be designed requires the resolution of a number of conflicting requirements. At first glance it might appear that the shorter the time interval (i.e., the time allocated to successive assembly line stations), the faster the execution rate will be for a series of instructions. Upon investigation, however, several parameters become apparent which frustrate this seemingly simple pattern for high performance design. The parameters of most importance are:

1 An assembly-line station platform (hardware "trigger") is necessary within each time interval, and it generally adds a circuit level to the time interval. The platform "overhead" can add appreciably to the total execution time of any one instruction since a shorter interval implies more stations for any pre-specified function. A longer instruction time is significant when sequential instructions are logically dependent. That is, instruction n cannot proceed until instruction $n + 1$ is completed. The dependency factor, therefore, indicates that the execution time of any individual instruction should not be penalized unnecessarily by overhead time delay.

2 The amount of control hardware—and control complexity—required to handle architectural and machine organization interlocks increases enormously as the number of assembly line stations is increased. This can lead to a situation for which the control paths determining the gating between stations contain more circuit levels than the data paths being controlled.

Parameters of less importance which influence the determination of the basic clock rate include:

1 The number of levels needed to implement certain basic data paths, e.g., address adders, instruction decoders, etc.

2 Effective storage access time, especially when this time is relatively short. Unless the station-to-station time interval of the CPU is a sub-multiple of storage access time the synchronization of storage and CPU functions will involve overhead time.

Judgment, rather than algorithms, gave the method by which the relative weights of the above parameters were evaluated to determine the basic station-to-station time interval.[1] The interval selected led to a splitting of the instruction handling functions as illustrated in Fig. 3.[2]

It can be seen in Fig. 3 that the basic time interval accommodates the assembly line handling of most of the basic hardware functions. However, the storage and many execution operations

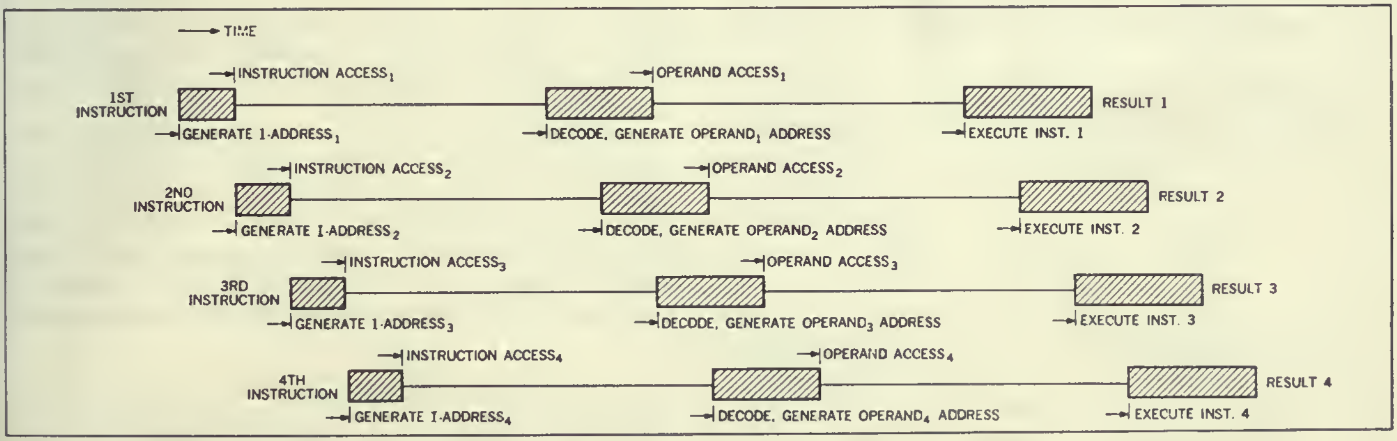


Fig. 2. Illustration of concurrency among successive instructions.

[1] The design objective calls for a 60 nsec basic machine clock interval. The judgment exercised in this selection was tempered by a careful analysis of the number of circuit levels, fan in, fan out, and wiring lengths required to perform some of the basic data path and control functions. The analysis indicated that 11 or 12 circuit levels of 5–6 nsec delay per level were required for the worst-case situations.

[2] Figure 3 also illustrates that the hardware sets are grouped into larger units—instruction unit, main storage control element, fixed-point execution unit, floating-point execution unit. The grouping is primarily caused by packaging restrictions, but a secondary objective is to provide separately designable entities having minimum interfacing. The total hardware required to implement the required CPU functions demands three physical frames, each having dimensions 66″ L × 15″ D × 78″ H. The units are allocated to the frames in such a way as to minimize the effects of interframe transmission delays.

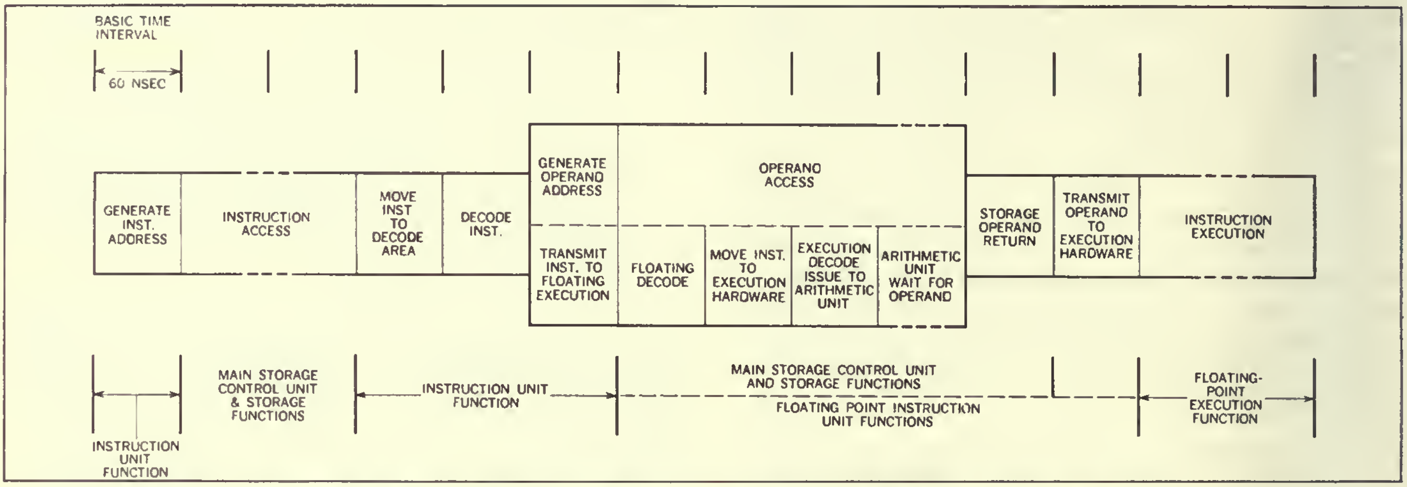

BASIC TIME INTERVAL
60 NSEC
GENERATE INST. ADDRESS
INSTRUCTION ACCESS
MOVE INST TO DECODE AREA
DECODE INST.
GENERATE OPERAND ADDRESS
OPERAND ACCESS
TRANSMIT INST. TO FLOATING EXECUTION
FLOATING DECODE
MOVE INST. TO EXECUTION HARDWARE
EXECUTION DECODE ISSUE TO ARITHMETIC UNIT
ARITHMETIC UNIT WAIT FOR OPERAND
STORAGE OPERAND RETURN
TRANSMIT OPERAND TO EXECUTION HARDWARE
INSTRUCTION EXECUTION
INSTRUCTION UNIT FUNCTION
MAIN STORAGE CONTROL UNIT & STORAGE FUNCTIONS
INSTRUCTION UNIT FUNCTION
MAIN STORAGE CONTROL UNIT AND STORAGE FUNCTIONS
FLOATING POINT INSTRUCTION UNIT FUNCTIONS
FLOATING-POINT EXECUTION FUNCTION

Fig. 3. CPU assembly-line stations required to accommodate a typical floating-point storage-to-register instruction.

require a number of basic intervals. In order to exploit the assembly line processing approach despite these time disparities, the organizational techiques of storage interleaving [Buchholz, 1962], arithmetic execution concurrency, and buffering are utilized.

Storage interleaving increases the storage bandwidth by enabling multiple accesses to proceed concurrently, which in turn enhances the assembly line handling of the storage function. Briefly, interleaving involves the splitting of storage into independent modules (each containing address decoding, core driving, data read-out sense hardware, and a data register) and arranging the address structure so that adjacent words—or small groups of adjacent words—reside in different modules. Figure 4 illustrates the technique.

The depth of interleaving required to support a desired concurrency level is a function of the storage cycle time, the CPU storage request rate, and the desired effective access time. The effective access time is defined as the sum of the actual storage access time, the average time spent waiting for an available storage, and the communication time between the processor and storage.[1]

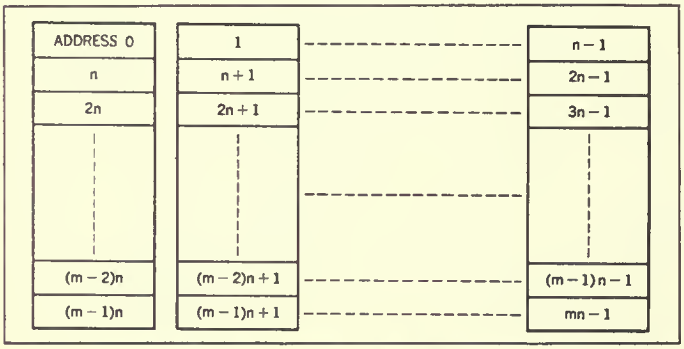


Fig. 4. Arrangement of addresses in *n* storage modules of *m* words per module.

Execution concurrency is facilitated first by the division of this function into separate units for fixed-point execution and floating-point execution. This permits instructions of the two classes to be executed in parallel; in fact, as long as no cross-unit dependencies exist, the execution does not necessarily follow the sequence in which the instructions are programmed.

Within the fixed-point unit, processing proceeds serially, one instruction at a time. However, many of the operations required only one basic time interval to execute, and special emphasis is placed on the storage-to-storage instructions to speed up their execution. These instructions (storage-to-storage) enable the Model 91 to achieve a performance rate of up to 7 times that of the System/360 Model 75 for the "translate-and-test" instruction. A number of new concepts and sequences [Litwiller and Adler] were developed to achieve this performance for normally storage access-dependent instructions.

The floating-point unit is given particular emphasis to provide additional concurrency. Multiple arithmetic execution units, employing fast algorithms for the multiply and divide operations and carry look-ahead adders, are utilized [Anderson, Earle, Goldschmidt, and Powers, 1967]. An internal bus has been designed [Tomasulo, 1967] to link the multiple floating-point execution units. The bus control correctly sequences dependent

[1] Effective access times ranging from 180–600 nsec are anticipated, although the design of the Model 91 is optimized around 360 nsec. Interleaving 400 nsec/cycle storage modules to a depth of 16 satisfies the 360 nsec effective access design point.

"strings" of instructions, but permits those which are independent to be executed out of order.

The organizational techniques described above provide balance between the number of instructions that can be prepared for arithmetic execution and those that can actually be executed in a given period, thereby preventing the arithmetic execution function from creating a "bottleneck" in the assembly line process.

Buffering of various types plays a major role in the Model 91 organization. Some types are required to implement the assembly line concept, while others are, in light of the performance objectives, architecturally imposed. In all cases the buffers provide queueing which smooths the total instruction flow by allowing the initiating assembly line stations to proceed despite unpredictable delays down the line. Instruction fetch, operand fetch, operand store, operation, and address buffering are utilized among the major CPU units as illustrated in Fig. 5.[1]

Instruction fetch buffering provides return data "sinks" for previously initiated instruction storage requests. This prefetching hides the instruction access time for straight-line (no branching) programs, thereby providing a steady flow of instructions to the decoding hardware. The buffering is expanded beyond this need to provide the capacity to hold program loops of meaningful size. Upon encountering a loop which fits, the buffer locks onto the loop and subsequent branching requires less time, since it is to the buffers rather than to storage. The discussion of branching given later in this paper gives a detailed treatment of the loop action.

Operand fetch buffers effectively provide a queue into which storage can "dump" operands and from which execution units can obtain operands. The queue allows the isolation of operand fetching from operand usage for the storage-to-register and storage-to-storage instruction types. The required depth[2] of the queue is a function of the number of basic time intervals required for storage accessing, the instruction "mix" of the operating program, and the relative time and frequency of execution bottlenecks. Operand store buffering provides the same function as fetch buffering, except that the roles of storage and execution

[1] Eight 64-bit double words comprise the array of instruction buffers. Six 32-bit operand buffers are provided in the fixed-point execution unit, while six 64-bit buffers reside in the floating-point execution unit. Three 64-bit store operand buffers along with three store address and four conflict address buffers are provided in the main storage control element. Also, there are six fixed-point and eight floating-point operand buffers.

[2] To show precise algorithms defining these and other buffering requirements is impractical, since different program environments have different needs. The factors considered in selecting specific numbers are cited instead.

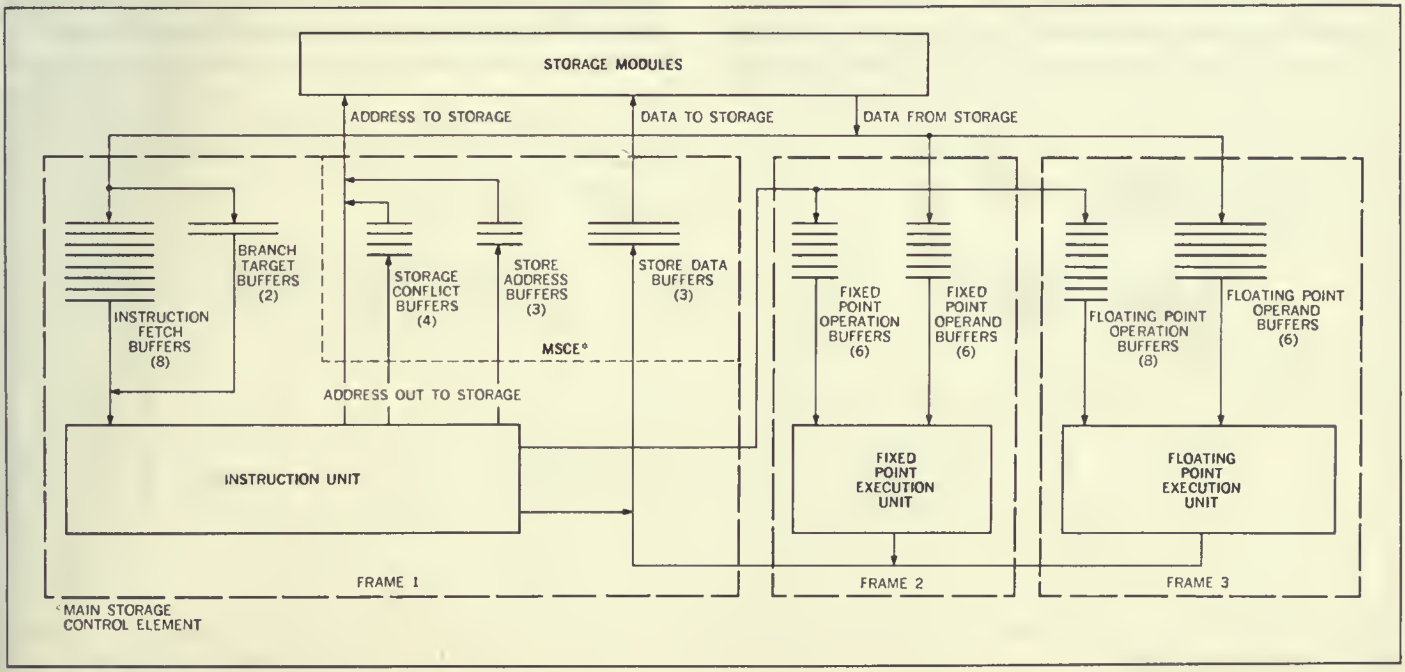

STORAGE MODULES
ADDRESS TO STORAGE
DATA TO STORAGE
DATA FROM STORAGE
BRANCH TARGET BUFFERS (2)
INSTRUCTION FETCH BUFFERS (8)
STORAGE CONFLICT BUFFERS (4)
STORE ADDRESS BUFFERS (3)
STORE DATA BUFFERS (3)
MSCE*
ADDRESS OUT TO STORAGE
INSTRUCTION UNIT
FIXED POINT OPERATION BUFFERS (6)
FIXED POINT OPERAND BUFFERS (6)
FIXED POINT EXECUTION UNIT
FLOATING POINT OPERATION BUFFERS (8)
FLOATING POINT OPERAND BUFFERS (6)
FLOATING POINT EXECUTION UNIT
FRAME 1
FRAME 2
FRAME 3
*MAIN STORAGE CONTROL ELEMENT

Fig. 5. Buffer allocation and function separation.

are reversed. The number of store buffers required is a function of the average waiting time encountered when the desired storage module is busy and the time required for the storage, when available, to utilize the operand.

Operation buffers in the fixed-point and floating-point execution units allow the instruction unit to proceed with its decoding and storage-initiating functions while the execution units wait for storage operands or execution hardware. The depth of the operation buffering is related to the amount of operand buffering provided and the "mix" of register-to-register and storage-to-register instruction types.

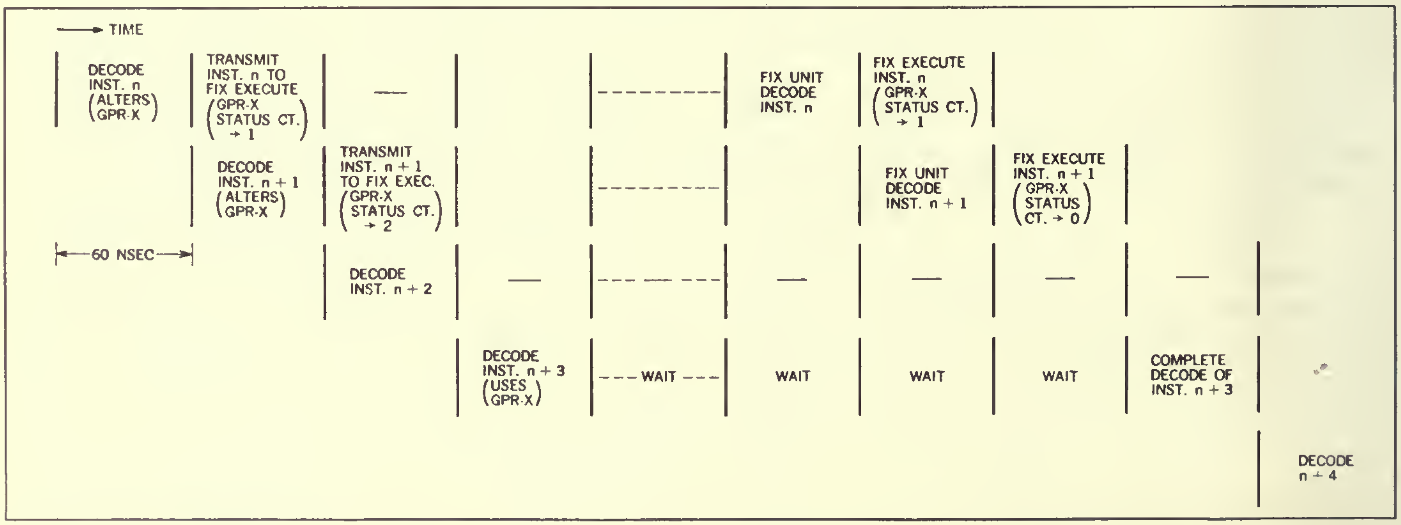
Address buffering is used to queue addresses to busy storage modules and to contain store addresses during the interval between decoding and execution of store instructions. The instruction unit is thereby allowed to proceed to subsequent instructions despite storage conflicts or the encountering of store operations. These buffers have comparators associated with them to establish logical precedence when conflicting program references arise. The number of necessary store address buffers is a function of the average delay between decode and execution, while the depth of the queue caused by storage conflicts is related to the probable length of time a request will be held up by a busy storage module [Boland, Granito, Marcotte, Messina, and Smith, 1967].

Concurrency Limitations

The assembly line processing approach, using the techniques of storage interleaving, arithmetic concurrency, and buffering, provides a solid high-performance base. The orientation is toward smooth-flowing instruction streams for which the assembly line can be kept full. That is, as long as station n need only communicate with station $n + 1$ of the line, highest performance is achieved. For example, floating-point problems which fit this criterion can be executed internally on the Model 91 at up to 100 times the internal speed of the 7090 [Flynn and Low, 1967].

There are, however, cases where simple communication between adjacent assembly line stations is inadequate, e.g., list processing applications, branching, and interrupts. The storage access time and the execution time are necessarily sequential between adjacent instructions. The organization cannot completely circumvent component delay in such instances, and the internal performance gain diminishes to about one order of magnitude greater than that of the 7090.

The list processing application is exemplified by sequentialism in addressing, which produces a major interlock situation in the Model 91. The architecturally specified usage of the general purpose registers (GRP's) for both address quantities and fixed-point data, coupled with the assembly line delay between address generation and fixed-point execution, leads to the performance slowdown. Figure 6 illustrates the interlock and the resulting delay. Instructions n and $n + 1$ set up the interlock on GPR X since they will alter the contents of X. The decode of $n + 3$ finds that the contents of X are to be used as an address parameter, and since the proper contents are not available $n + 3$ must wait until $n + 1$ is executed. The interlock technique involves assigning the decode area a status count for each GPR. A zero status count indicates availability. As fixed-point instructions pass through the decode, they increment the appropriate counter(s). A decode



→ TIME
DECODE INST. n (ALTERS GPR·X)
TRANSMIT INST. n TO FIX EXECUTE (GPR·X STATUS CT. → 1)
FIX UNIT DECODE INST. n
FIX EXECUTE INST. n (GPR·X STATUS CT. → 1)
DECODE INST. n + 1 (ALTERS GPR·X)
TRANSMIT INST. n + 1 TO FIX EXEC. (GPR·X STATUS CT. → 2)
FIX UNIT DECODE INST. n + 1
FIX EXECUTE INST. n + 1 (GPR·X STATUS CT. → 0)
60 NSEC
DECODE INST. n + 2
DECODE INST. n + 3 (USES GPR·X)
WAIT
WAIT
WAIT
WAIT
COMPLETE DECODE OF INST. n + 3
DECODE n + 4

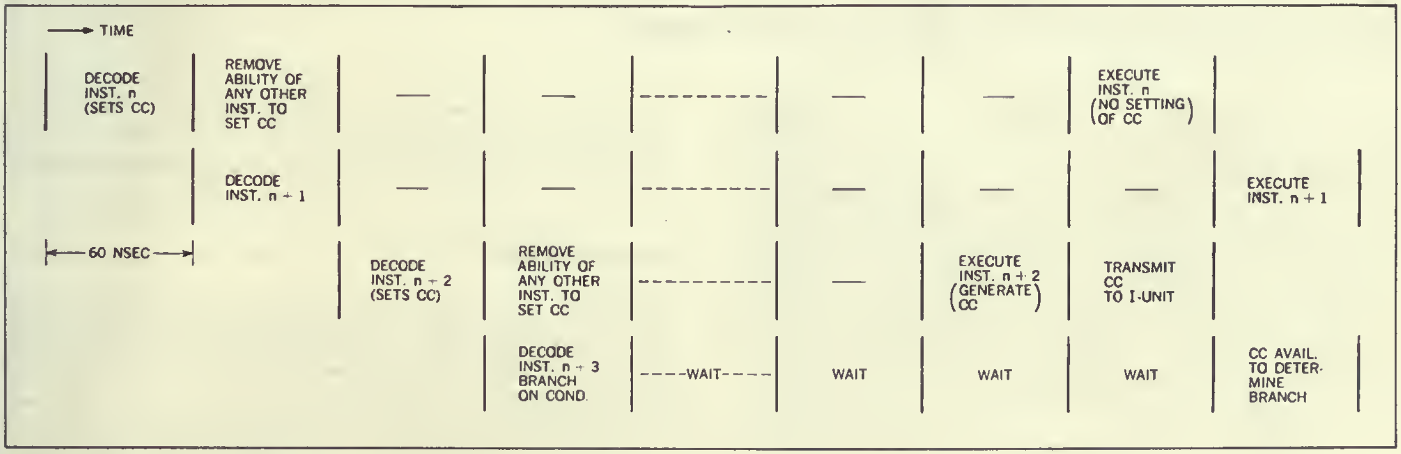
Fig. 6. GPR address interlock.

requiring an unavailable (non-zero status count) GRP cannot be completed. As the fixed-point execution unit completes instructions it decrements the appropriate counter(s), thus eventually freeing the register.

Branching leads to another sequential situation, since a disruption in the instruction supply is created. (Techniques employed to minimize or circumvent the storage access delay involved in obtaining the new instructions are discussed under *Instruction Supplying* in the following section of this paper.) Conditional branching poses an additional delay in that the branch decision depends on the outcome of arithmetic operations in the execution units. The Model 91 has a relatively lower performance in cases for which a large percentage of conditional branch instructions lead to the branch being taken. The discontinuity is minimized, when the branch is not taken, through special handling of the condition code (CC) and the conditional branch instruction (BC). The condition code is a two-bit indicator, set according to the outcome of a variety of instructions, and can subsequently be interrogated for branching through the BC instruction. Since the code is to represent the outcome of the last decoded CC-affecting instruction, and since execution can be out of sequence, interlocks must be established to ensure this. This is accomplished, as illustrated in Fig. 7, by tagging each instruction at decode time if it is to set the CC. Simultaneously, a signal is communicated throughout the CPU to remove all tags from previously decoded but not executed instructions. Allowing only the execution of the tagged instruction to alter the code ensures that the correct CC will be set. The decode hardware monitors the CPU for outstanding tags; only when none exists is the condition code considered valid for interrogation.

The organization assumes that, for a conditional branch, the CC will not be valid when the "branch-on-condition" (BC) is decoded (a most likely situation, considering that most arithmetic and logical operations set the code). Rather than wait for a valid CC, fetches are initiated for two instruction double-words as a hedge against a successful branch. Following this, it is assumed that the branch will fall, and a "conditional mode" is established. In conditional mode, shown in Fig. 8, instructions are decoded and conditionally forwarded to the execution units, and concomitant operand fetches are initiated. The execution units are inhibited from completing conditional instructions. When a valid condition code appears, the appropriate branching action is detected and activates or cancels the conditional instructions. Should the no-branch guess prove correct, a substantial head start is provided by activating the conditionally issued and initiated operand fetches for a number of instructions. If the branch is successful, the previously fetched target words are activated and provide work while the instruction fetching is diverted to the new stream. (Additional optimizing techniques are covered under the discussion of branching in a subsequent section of this paper.)

Interrupts, as architecturally constrained, are a major bottleneck to performance in the assembly line organization. Strict adherence to a specification which states that an interrupt on instruction n should logically precede and inhibit any action from being taken on instruction $n + 1$ leaves two alternatives. The first would be to force sequentialism between instructions which may lead to an interrupt. In view of the variety of interrupt possibilities defined, this course would totally thwart high performance and is necessarily discarded. The second is to set aside sufficient information to permit recovery from any interrupt which might arise. In view of the pipeline and execution concurrency which allows the Model 91 to advance many instructions beyond n prior to its execution, and to execute independent instructions out of sequence ($n + m$ before n), the recovery problem becomes



DECODE INST. n (SETS CC)	REMOVE ABILITY OF ANY OTHER INST. TO SET CC	—	—	---------	—	—	EXECUTE INST. n (NO SETTING OF CC)	
	DECODE INST. n + 1	—	—	---------	—	—	—	EXECUTE INST. n + 1
		DECODE INST. n + 2 (SETS CC)	REMOVE ABILITY OF ANY OTHER INST. TO SET CC	---------	—	EXECUTE INST. n + 2 (GENERATE CC)	TRANSMIT CC TO I-UNIT	
			DECODE INST. n + 3 BRANCH ON COND.	----WAIT----	WAIT	WAIT	WAIT	CC AVAIL. TO DETERMINE BRANCH

Fig. 7. Condition code interlock.

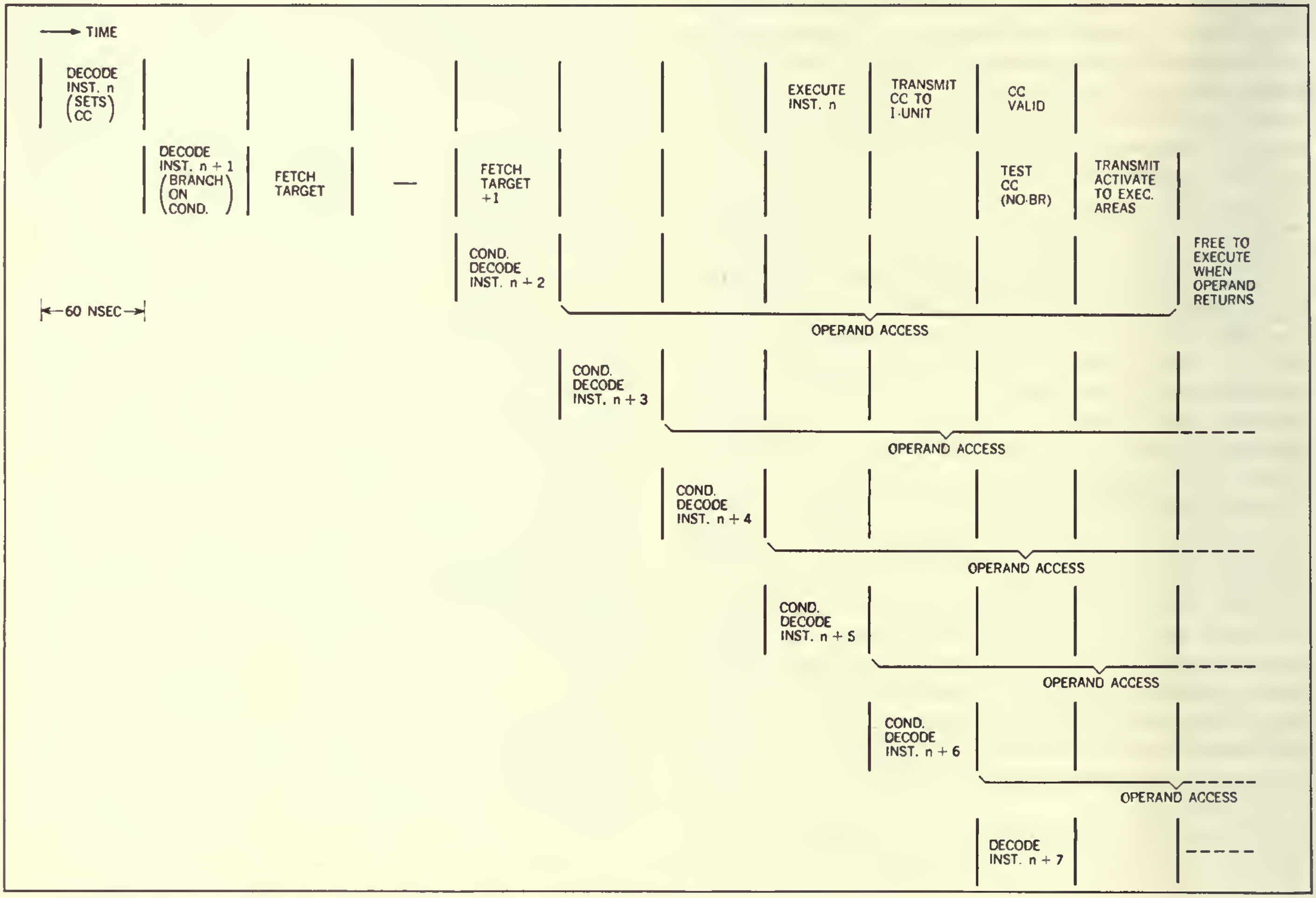


Fig. 8. Conditional instruction issuing: the branch-on-condition philosophy.

extremely complex and costly. Taking this approach would entail hardware additions to the extent that it would severely degrade the performance one is seeking to enhance. The impracticality of both alternatives by which the interrupt specifications could be met made it mandatory that the specifications themselves be altered. The architecture was compromised by removing the above-mentioned "precedence" and "inhibit" requirements. The specification change led to what is termed the "imprecise interrupt" philosophy of the Model 91 and reduced the interrupt bottleneck to an instruction supply discontinuity. The imprecise interrupt, and the manner in which the instruction discontinuity is minimized, are covered in the next section of the paper.

The bottlenecks discussed above gave rise to the major interlocks among the separate CPU areas. Within each of the areas, however, additional considerations hold. These are discussed as appropriate in the next section or in following papers.

Instruction Unit

The central control functions for the Model 91 CPU are performed in the instruction unit. The objective here is to discuss these functions in terms of how they are performed and to include the reasons for selecting the present design. However, before proceeding with this discussion it will be useful to examine some over-all design considerations and decisions which directly affect the instruction unit functions. In approaching the design of the instruction unit, many program situations were examined, and it was found that while many short instruction sequences are nicely ordered, the trend is toward frequent branching. Such things as performing short work loops, taking new action based on data results, and calling subroutines are the bases upon which programs are built and, in many instances, these factors play a larger role in the use of available time than does execution. Consequently, emphasis on branch sequencing is required. A second finding was that, even with sophisticated execution algorithms, very few programs can cause answers actually to flow from the assembly line at an average rate in excess of one every two cycles. Inherent inter-instruction dependencies, storage and other hardware conflicts, and the frequency of operations requiring multi-cycle execution all combine to prevent it.

Consideration of branching and execution times indicates that, for overall balance, the instruction unit should be able to surge ahead of the execution units by issuing instructions at a faster-than-execution rate. Then, when a branch is encountered, a significant part of the instruction unit slowdown will be overlapped with execution catch-up. With this objective in mind it becomes necessary to consider what constitutes a fast issue rate and what "trade-offs" would be required to achieve it. It is easily shown that issuing at a rate in excess of one instruction per cycle leads to a rapid expansion of hardware and complexity. (Variable-length instructions, adjacent instruction interdependencies, and storage requirements are prime factors involved.) A one-cycle maximum rate is thereby established, but it too presents difficulties. The assembly line process requires that both instruction fetching and instruction issuing proceed concurrently in order to hide storage delays. It is found through program analysis that slightly more than two instructions will be obtained per 64-bit instruction fetch[1] and that approximately 80% of all instructions require an operand reference to storage. From this it is concluded that issuing the average instruction entails approximately 1.25 storage accesses: 0.45 (instruction fetches) + 0.80 (operand fetches). This figure, with the one-per-cycle issue rate goal, clearly indicates a need for either two address paths to storage and associated return capabilities, or for multiple words returned per fetch. In considering these options, the initial tendency is to separate instruction and operand storage access paths. However, multiple paths to storage give rise to substantial hardware additions and lead to severe control problems, particularly in establishing storage priorities and interlocks due to address dependencies. With a one-at-a-time approach these can be established on each new address as it appears, whereas simultaneous requests involve doing considerably more testing in a shorter time interval. Multiple address paths to storage were considered impractical because of the unfavorable compromise between hardware and performance.

The multiple-words-returned-per-fetch option was considered in conjunction with instruction fetching since the instruction stream is comprised of sequential words. To prevent excessive storage "busying" this approach requires a multiple word readout at the storage unit along with a wider data return path. Also, the interleaving factor is altered from *sequential* to *multi-sequential*, i.e., rather than having sequential double words in different storage modules, groups of sequential words reside in the same module. The interlock problems created by this technique are modest, the change in interleaving technique has little performance effect,[2] and storage can be (is, in some cases) organized to read out multiple words, all of which make this approach feasible. However, packaging density (more hardware required for wide data paths), storage organization constraints, and scheduling were such that this approach was also discarded. As a consequence, the single-port storage bus, which allows sequential accessing of double words, was adopted. This fact, in conjunction with the 1.25 storage accesses required per instruction, leads to a lowering of the average maximum issue rate to 0.8 instructions per machine cycle. The instruction unit achieves the issue rate through an organization which allows concurrency by separating the instruction supplying from the instruction issuing function.

Instruction Supplying

Instruction supplying includes the provision of an instruction stream which will support the desired issue rate in a sequential (non-branch) environment, and the ability to switch readily to a new instruction stream when required because of branching or interrupts.

[1] Storage-to-storage (SS) instructions are not considered here. They can be viewed as macro-operations and are treated as such by the hardware. The macro-operations are equivalent to basic instructions, and the number of micro-instructions involved in performing an SS function indicates that many instruction fetches would be required to perform the same function using other System/360 instructions.

[2] This is more intuitive than analytical. Certainly for strictly random addressing, the interleave technique is irrelevant. However, in real applications, programs are generally localized with (1) the instructions sequential and (2) branches jumping tens or hundreds rather than thousands of words. Data is more random because, even though it is often ordered in arrays, quite frequently many arrays are utilized concurrently. Also, various data constants are used which tend to randomize the total use. A proper analysis must consider all these factors and so becomes complex. In any event, as long as the interleave factor remains fixed the interference appears little affected by small changes in the interleaving pattern.

Sequential Instruction Fetching. Provision of a sequential string of instructions has two fundamental aspects, an initiation or start-up transient, and a steady-state function. The initial transient entails filling the assembly line ahead of the decode station with instructions. In hardware terms, this means initiating sufficient instruction fetches so that, following a wait of one access time, a continuous flow of instruction words will return from storage. Three double-word fetches are the minimum required to fill the assembly line, since approximately two instructions are contained within a double word, and the design point access time is six machine cycles. The actual design exceeds the minimum for several reasons, the first being that during start-up no operand requests are being generated (there are no instructions), and consequently the single address port to storage is totally available for instruction fetching. Second, the start-up delay provides otherwise idle time during which to initiate more fetches, and the eight double words of instruction buffering provide space into which the words can return. A third point is that, should storage requiring more than six cycles of access time be utilized, more fetching-ahead will be required. Finally, establishing an excess queue of instructions during the transient time will allow temporary maintenance of a full assembly line without any further instruction fetching. The significance of this action is that it allows the issuing of a short burst of instructions at a one-per-cycle rate. This follows from the fact that the single, normally shared storage address port becomes exclusively available to the issue function. A start-up fetching burst of five double instruction words was the design point which resulted when all of these factors had been considered.[1]

Steady-state instruction supplying serves the function of maintaining a full assembly line by initiating instruction fetches at appropriate intervals. The address port to storage is multiplexed between instruction fetches and operand fetches, with instructions receiving priority in conflict situations. An additional optimization technique allows the instruction fetching to re-advance to the start-up level of five double words ahead if storage address time "slots" become available. A flow chart of the basic instruction fetch control algorithm is shown in Fig. 9,[2] while Fig. 10 is a schematic of the data paths provided for the total instruction supplying function. Some of the decision blocks contained in the flow chart result from the effects of branch instructions; their function will be clarified in the subsequent discussion of branching. There are two fundamental reasons for checking buffer availability in the algorithm. First, the instruction buffer array is a modulo-eight map of storage that is interleaved by sixteen. Second, fetches can return out of order because storage may be busy or of varying performance. For example, when a branch is encountered, point one above implies that the target may overlay a fetch which has not yet returned from storage. In view of the second point, it is necessary to ensure that the unreturned fetch is ignored, as it would be possible for a new fetch to return ahead of it. Proper sequencing is accomplished by "tagging" the buffers assigned to outstanding fetches, and preventing the initiation of a new fetch to a buffer so tagged.

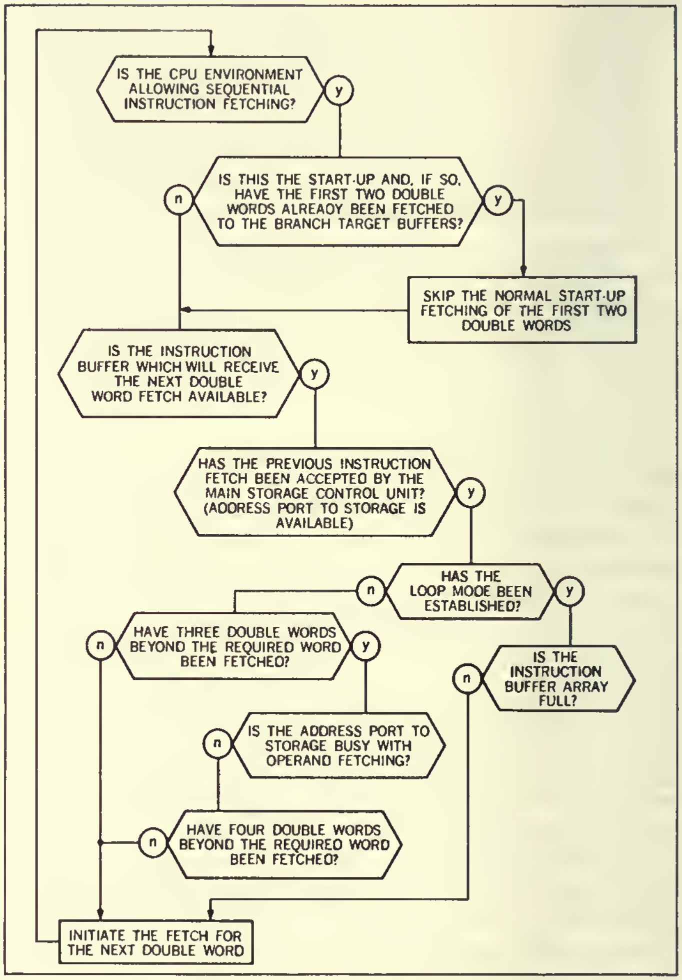

IS THE CPU ENVIRONMENT ALLOWING SEQUENTIAL INSTRUCTION FETCHING?
y
IS THIS THE START-UP AND, IF SO, HAVE THE FIRST TWO DOUBLE WORDS ALREADY BEEN FETCHED TO THE BRANCH TARGET BUFFERS?
n
y
SKIP THE NORMAL START-UP FETCHING OF THE FIRST TWO DOUBLE WORDS
IS THE INSTRUCTION BUFFER WHICH WILL RECEIVE THE NEXT DOUBLE WORD FETCH AVAILABLE?
y
HAS THE PREVIOUS INSTRUCTION FETCH BEEN ACCEPTED BY THE MAIN STORAGE CONTROL UNIT? (ADDRESS PORT TO STORAGE IS AVAILABLE)
y
HAS THE LOOP MODE BEEN ESTABLISHED?
n
y
HAVE THREE DOUBLE WORDS BEYOND THE REQUIRED WORD BEEN FETCHED?
n
y
IS THE INSTRUCTION BUFFER ARRAY FULL?
n
IS THE ADDRESS PORT TO STORAGE BUSY WITH OPERAND FETCHING?
n
HAVE FOUR DOUBLE WORDS BEYOND THE REQUIRED WORD BEEN FETCHED?
n
INITIATE THE FETCH FOR THE NEXT DOUBLE WORD

Fig. 9. Flow chart of the sequential instruction-supply function.

[1] The one disadvantage to over-fetching instructions is that the extra fetches may lead to storage conflicts, delaying the subsequently initiated operand fetches. This is a second-order effect, however, first because it is desirable for the instruction fetches to win conflicts unless these fetches are rendered unnecessary by an intervening branch instruction, and second because the sixteen-deep interleaving of storage significantly lowers the probability of the conflict situation.

[2] In this flow chart, unlabeled exits from decision blocks imply that a "wait" state will exist until the required condition has been satisfied.

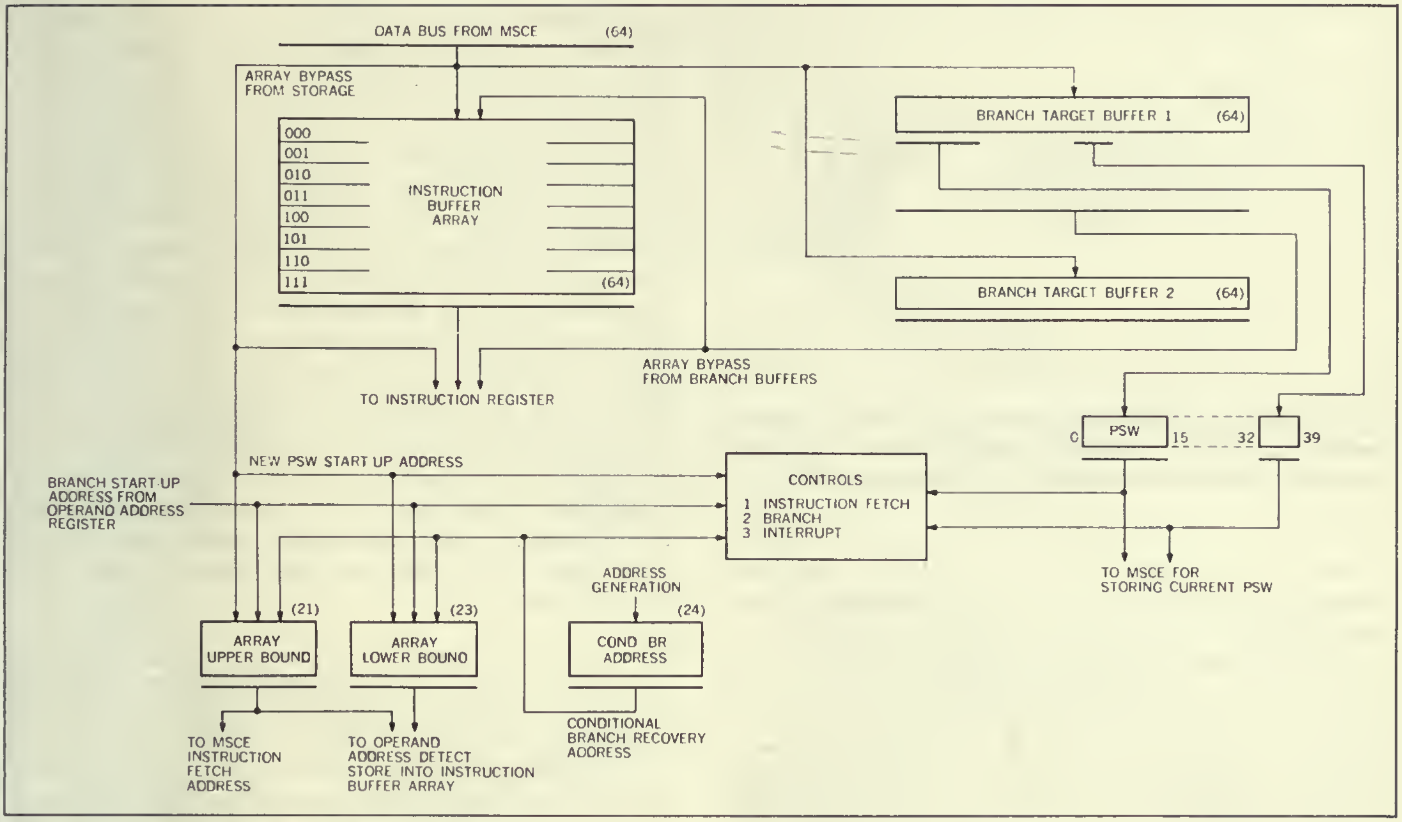

DATA BUS FROM MSCE (64)
ARRAY BYPASS FROM STORAGE
000
001
010
011
100
101
110
111
INSTRUCTION BUFFER ARRAY
(64)
BRANCH TARGET BUFFER 1 (64)
BRANCH TARGET BUFFER 2 (64)
TO INSTRUCTION REGISTER
ARRAY BYPASS FROM BRANCH BUFFERS
0
PSW
15
32
39
NEW PSW START UP ADDRESS
BRANCH START-UP ADDRESS FROM OPERAND ADDRESS REGISTER
CONTROLS
1 INSTRUCTION FETCH
2 BRANCH
3 INTERRUPT
TO MSCE FOR STORING CURRENT PSW
ADDRESS GENERATION
(21)
(23)
(24)
ARRAY UPPER BOUND
ARRAY LOWER BOUND
COND BR ADDRESS
TO MSCE INSTRUCTION FETCH ADDRESS
TO OPERAND ADDRESS DETECT STORE INTO INSTRUCTION BUFFER ARRAY
CONDITIONAL BRANCH RECOVERY ADDRESS

Fig. 10. Data paths for the basic instruction supply.

Branch Handling. Branching adds to the complexity of the instruction supplying function because attempts are made to minimize discontinuities caused by the branching and the consequent adverse effects on the issue rate. The discontinuities result because for each branch the supply of instructions is disrupted for a time roughly equivalent to the greater of the storage access period (start-up transient previously mentioned), or the internal testing and "housekeeping" time required to make and carry out the branch decision. This time can severely limit the total CPU performance in short program loops. It has a somewhat less pronounced effect in longer loops because the branch time becomes a smaller percentage of the total problem loop time and, more important, the instruction unit has greater opportunity to run ahead of the execution units (see Fig. 11). This last makes more time available in which to overlap the branch time with execution catch-up.

The detrimental performance effect which stems from short loops led to a dual branch philosophy. The first aspect deals with branches which are either forward into the instruction stream,[1] beyond the prefetched instructions, or if backward from the branch instruction, greater than eight double-words back. In these situations, the branch storage-delay is unavoidable. As a hedge against such a branch being taken, the branch sequencing (Fig. 12) initiates fetches for the first two double words down the target path. Two branch buffers are provided (Fig. 10—the instruction supply data flow) to receive these words, in order that the instruction buffer array will be unaffected if the result is a no-branch decision. The branch housekeeping and decision making are carried on in parallel with the access time of the target fetches. If a branch decision is reached before the access has been completed, additional optimizing hardware routes the target fetch around the buffer and directly to the instruction register, from which it will be decoded. Minor disadvantages of the technique

[1] In the actual program the branch instruction would precede the target for this case.

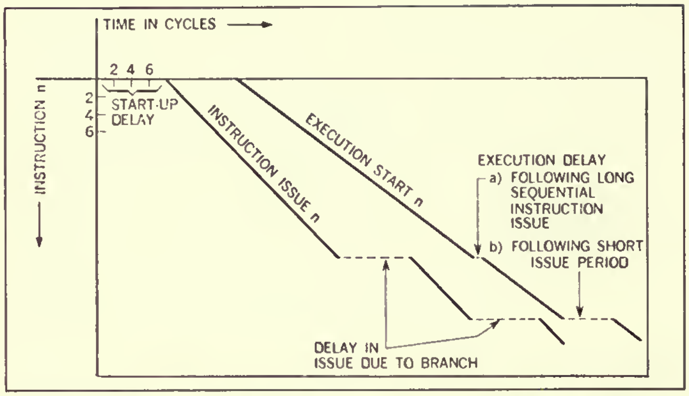

TIME IN CYCLES
2 4 6
INSTRUCTION n
2
4
6
START-UP DELAY
INSTRUCTION ISSUE n
EXECUTION START n
EXECUTION DELAY
a) FOLLOWING LONG SEQUENTIAL INSTRUCTION ISSUE
b) FOLLOWING SHORT ISSUE PERIOD
DELAY IN ISSUE DUE TO BRANCH

Fig. 11. Schematic representation of execution delays caused by (branch) discontinuities in the instruction issuing rate, for the case in which the issuing rate is faster than the execution rate.

are that the "hedge" fetching results in a delay of the no-branch decision and may lead to storage conflicts. Consequently, a small amount of time is lost for a branch which "falls through."

The second aspect of the branch philosophy treats the case for which the target is backward within eight double words of the branch instruction. A separation of eight double words or less defines a "short" loop—this number being chosen as a hardware/performance compromise. Part of the housekeeping required in the branch sequencing is a "back eight" test. If this test is satisfied the instruction unit enters what is termed "loop mode." Two beneficial results derive from loop mode. First, the complete loop is fetched into the instruction buffer array, after which instruction fetching ceases. Consequently, the address port to storage is totally available for operand fetching and a one instruction per cycle issue rate is possible. The second advantage gained by loop mode is a reduction by a factor of two to three in the time required to sequence the loop-establishing branch instruction. (For example, the "branch on index" instruction normally requires eight cycles for a successful branch, while in loop mode three cycles are sufficient.) In many significant programs it is estimated that the CPU will be in loop mode up to 30% of the time.

Loop mode may be established by all branch instructions except "branch and link." It was judged highly improbable that this instruction would be used to establish the type of short repetitious program loops to which loop mode is oriented. A conditional branch instruction, because it is data dependent and therefore less predictable in its outcome than other branch instructions, requires special consideration in setting up loop mode. Initial planning was to prevent looping with this instruction, but consultation with programmers has indicated that loops are frequently closed conditionally, since this allows a convenient means for loop breaking when exception conditions arise.

Furthermore, in these situations the most likely outcome is often known and can be utilized to bias the branch decision whichever way is desirable. For such reasons, the "back eight" test is made during the sequencing of a conditional branch instruction, and the status is saved through conditional mode. Should it subsequently be determined that the branch is to be taken, and the "saved" status indicates "back eight," loop mode is established. Thereafter the role of conditional mode is reversed, i.e., when the conditional branch is next encountered, it will be assumed that the branch will be taken. The conditionally issued instructions are from the target path rather than from the no-branch path as is the case when not in loop mode. A cancel requires recovery from the branch guess. Figure 12 is a flow chart of this action. In retrospect, the conditional philosophy and its effects on loop mode, although significant to the performance of the CPU and conceptually simple, were found to require numerous interlocks throughout the CPU. The complications of conditional mode, coupled with the fact that it is primarily aimed at circumventing storage access delays, indicate that a careful re-examination of its usefulness will be called for as the access time decreases.

Interrupts. Interrupts, like branching, are another disruption to a smooth instruction supply. In the interrupt situation the instruction discontinuity is worsened because, following the recognition of the interrupt, two sequential storage access delays are encountered prior to receiving the next instruction.[1] Fortunately, and this is unlike branches, interrupts are relatively infrequent. In defining the interrupt function it was decided that architectural "imprecise" compromise mentioned in the previous section would be invoked only where necessary to achieve the required performance. In terms of the assembly line concept, this means that interrupts associated with an instruction unit decode time interval will conform with the specifications. Consequently, only interrupts which result from address, storage, and execution functions are imprecise.

One advantage of this dual treatment is that System/360 compatibility is retained to a useful degree. For example, a programming strategy sometimes employed to call special subroutines involves using a selected invalid instruction code. The ensuing interrupt provides a convenient subroutine entry technique. Retaining the compatible interrupt philosophy through the decoding time interval in the Model 91 allows it to operate

[1] This arises from the architectural technique of indirectly entering the interrupt subroutines. In System/360 the interrupts are divided into classes. Each class is assigned a different, fixed low storage address which contains the status to which the CPU shall be set should an interrupt of the associated class occur. Part of this status is a new program address. Consequently, interrupting requires obtaining a new supply of instructions from storage indirectly, through the new status word.

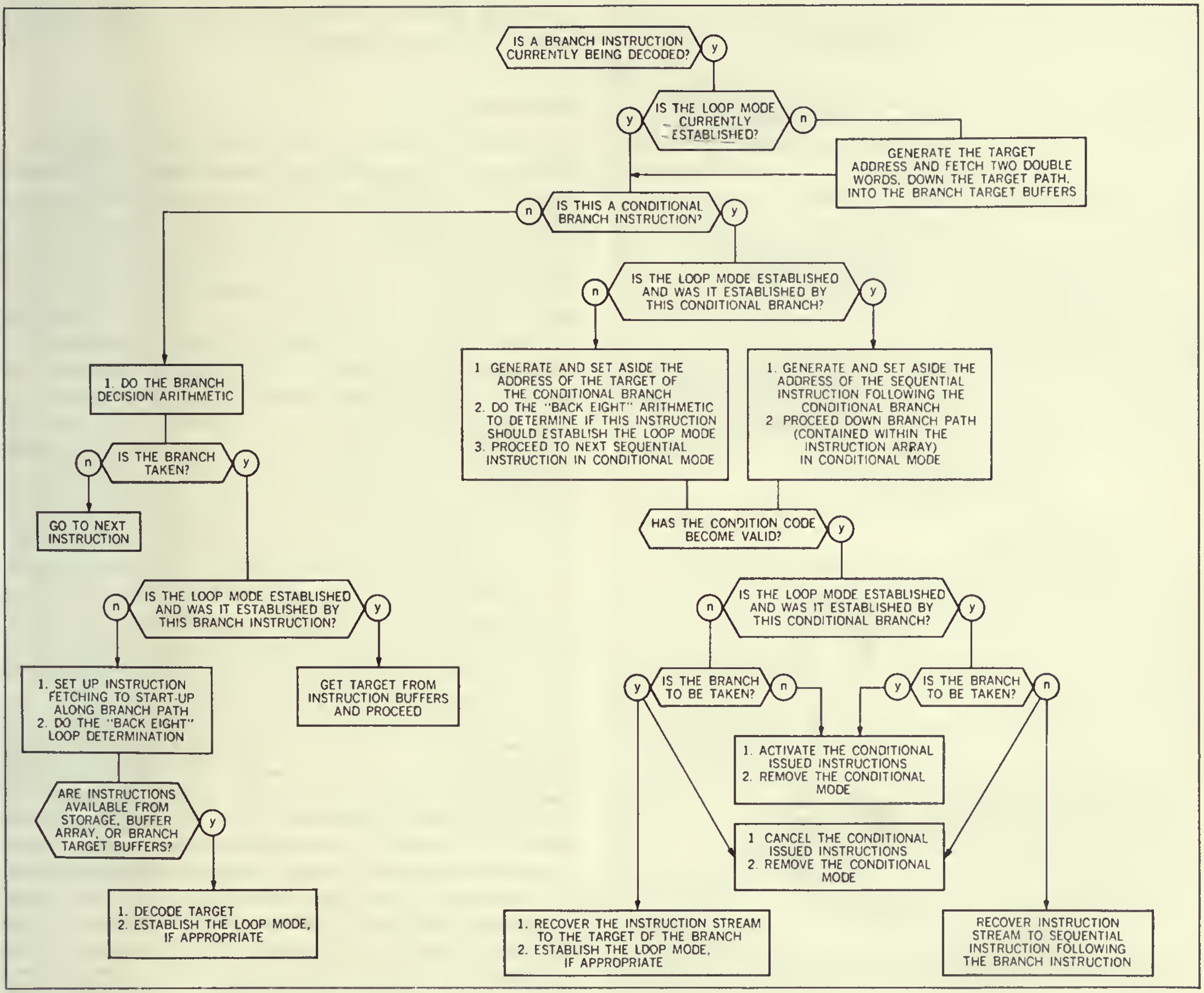

IS A BRANCH INSTRUCTION CURRENTLY BEING DECODED?
y
IS THE LOOP MODE CURRENTLY ESTABLISHED?
y
n
GENERATE THE TARGET ADDRESS AND FETCH TWO DOUBLE WORDS, DOWN THE TARGET PATH, INTO THE BRANCH TARGET BUFFERS
IS THIS A CONDITIONAL BRANCH INSTRUCTION?
n
y
IS THE LOOP MODE ESTABLISHED AND WAS IT ESTABLISHED BY THIS CONDITIONAL BRANCH?
n
y
1. DO THE BRANCH DECISION ARITHMETIC
1 GENERATE AND SET ASIDE THE ADDRESS OF THE TARGET OF THE CONDITIONAL BRANCH
2. DO THE "BACK EIGHT" ARITHMETIC TO DETERMINE IF THIS INSTRUCTION SHOULD ESTABLISH THE LOOP MODE
3. PROCEED TO NEXT SEQUENTIAL INSTRUCTION IN CONDITIONAL MODE
1. GENERATE AND SET ASIDE THE ADDRESS OF THE SEQUENTIAL INSTRUCTION FOLLOWING THE CONDITIONAL BRANCH
2 PROCEED DOWN BRANCH PATH (CONTAINED WITHIN THE INSTRUCTION ARRAY) IN CONDITIONAL MODE
IS THE BRANCH TAKEN?
n
y
GO TO NEXT INSTRUCTION
HAS THE CONDITION CODE BECOME VALID?
y
IS THE LOOP MODE ESTABLISHED AND WAS IT ESTABLISHED BY THIS BRANCH INSTRUCTION?
n
y
IS THE LOOP MODE ESTABLISHED AND WAS IT ESTABLISHED BY THIS CONDITIONAL BRANCH?
n
y
1. SET UP INSTRUCTION FETCHING TO START-UP ALONG BRANCH PATH
2. DO THE "BACK EIGHT" LOOP DETERMINATION
GET TARGET FROM INSTRUCTION BUFFERS AND PROCEED
y
IS THE BRANCH TO BE TAKEN?
n
y
IS THE BRANCH TO BE TAKEN?
n
1. ACTIVATE THE CONDITIONAL ISSUED INSTRUCTIONS
2. REMOVE THE CONDITIONAL MODE
ARE INSTRUCTIONS AVAILABLE FROM STORAGE, BUFFER ARRAY, OR BRANCH TARGET BUFFERS?
y
1 CANCEL THE CONDITIONAL ISSUED INSTRUCTIONS
2. REMOVE THE CONDITIONAL MODE
1. DECODE TARGET
2. ESTABLISH THE LOOP MODE, IF APPROPRIATE
1. RECOVER THE INSTRUCTION STREAM TO THE TARGET OF THE BRANCH
2. ESTABLISH THE LOOP MODE, IF APPROPRIATE
RECOVER INSTRUCTION STREAM TO SEQUENTIAL INSTRUCTION FOLLOWING THE BRANCH INSTRUCTION

Fig. 12. Flow chart of the branching sequence.

programs employing such techniques. The manifestation of this approach is illustrated in the flow chart of Fig. 13. In accordance with System/360 specifications, no further decoding is allowed once either a precise or an imprecise interrupt has been signalled. With the assembly line organization, it is highly probable that at the time of the interrupt there will be instructions still in the pipeline which should be executed prior to changing the CPU status to that of the interrupt routine. However, it is also desirable to minimize the effect of the interrupt on the instruction supply, so the new status word is fetched to the existing branch target buffer in parallel with the execution completion. After the return from storage of the new status word, if execution is still incomplete, further optimizing allows the fetching of instructions for the interrupt routine. Before proceeding, it becomes necessary to consider an implication resulting from the dual interrupt philosophy. Should a precise interrupt have initiated the action, it is

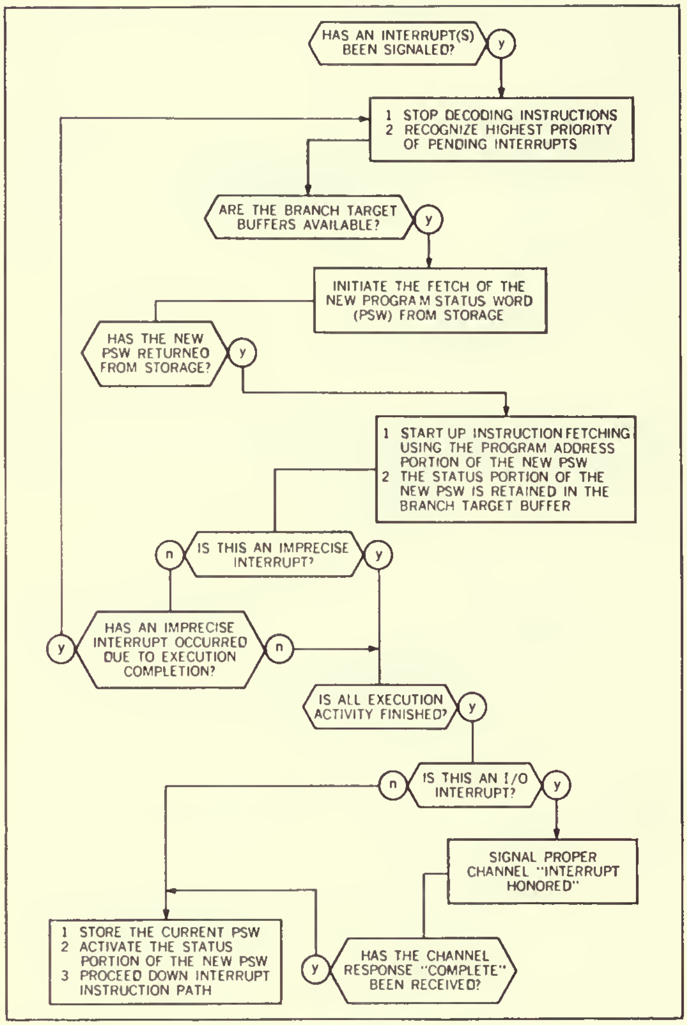

HAS AN INTERRUPT(S) BEEN SIGNALED?
y
1 STOP DECODING INSTRUCTIONS
2 RECOGNIZE HIGHEST PRIORITY OF PENDING INTERRUPTS
ARE THE BRANCH TARGET BUFFERS AVAILABLE?
y
INITIATE THE FETCH OF THE NEW PROGRAM STATUS WORD (PSW) FROM STORAGE
HAS THE NEW PSW RETURNED FROM STORAGE?
y
1 START UP INSTRUCTION FETCHING USING THE PROGRAM ADDRESS PORTION OF THE NEW PSW
2 THE STATUS PORTION OF THE NEW PSW IS RETAINED IN THE BRANCH TARGET BUFFER
n
IS THIS AN IMPRECISE INTERRUPT?
y
y
HAS AN IMPRECISE INTERRUPT OCCURRED DUE TO EXECUTION COMPLETION?
n
IS ALL EXECUTION ACTIVITY FINISHED?
y
n
IS THIS AN I/O INTERRUPT?
y
SIGNAL PROPER CHANNEL "INTERRUPT HONORED"
1 STORE THE CURRENT PSW
2 ACTIVATE THE STATUS PORTION OF THE NEW PSW
3 PROCEED DOWN INTERRUPT INSTRUCTION PATH
y
HAS THE CHANNEL RESPONSE "COMPLETE" BEEN RECEIVED?

Fig. 13. Flow chart of the interrupt sequence.

possible that the execution "cleanup" will lead to an imprecise condition. In this event, and in view of the desire to maintain compatibility for precise cases, the logically preceding imprecise signal should cancel all previous precise action. The flow chart (Fig. 13) illustrates this cancel-recovery action. Should no cancel action occur (the more likely situation), the completion of all execution functions results, with one exception, in the release of the new status word and instruction supply. The I/O interrupts require special consideration because of certain peculiarities in the channel hardware (the System 360/Model 60–75 channel hardware is used). Because of them, the CPU-channel communication cannot be carried out in parallel with the execution completion. However, the relative infrequency of I/O interrupts renders negligible the degradation caused by this.

Instruction Issuing

The instruction-issuing hardware initiates and controls orderly concurrency in the assembly line process leading to instruction execution. It accomplishes this by scanning each instruction, in the order presented by the program, and clearing all necessary interlocks before releasing the instruction. In addition, should a storage reference be required by the operation, the issuing mechanism performs the necessary address calculations, initiates the storage action, and establishes the routing by which the operand and operation will ultimately be merged for execution. In addition, certain essential inter-instruction dependencies are maintained while the issue functions proceed concurrently.

In terms of the assembly line of Fig. 3, the moving of instructions to the decode area, the decode, and the operand address generation comprise the issue stations. The moving of instructions to the decode area entails the taking of 64-bit double-words, as provided by the instruction supply, and extracting from them the proper instruction half-words, one instruction at a time. The instruction register is the area through which this is accomplished (Fig. 14). The register efficiently handles variable-length instructions and provides a stable platform from which to decode. All available space in this 64-bit register is kept full of instructions yet to be decoded, provided only that the required new instruction information has returned from storage. The decoder scans across the instruction register, starting at any half-word (16-bit) boundary, with new instructions refilling any space vacated by instruction issuing. The register is treated conceptually as a cylinder; i.e., the end of the register is concatenated with the beginning, since the decode scan must accommodate instructions which cross double-word boundaries.

The decoding station is the time interval during which instruction scanning and interlock clearing take place. Instruction-independent functions (interval timer update, wait state, certain interrupts and manual intervention) are subject to entry interlocks during this interval. Instruction-associated functions also have interlocks which check for such things as the validity of the scanned portion of the instruction register, whether or not the instruction starts on a half-word boundary, whether the instruction is a valid operation, whether an address is to be generated for the instruction (and if so, whether the address adder is available), and where the instruction is to be executed. In conjunction with this last point, should the fixed- or floating-point execution units be involved, availability of operation buffering is checked. Inter-instruction dependencies are the final class of interlocks which can occur during the decoding interval. These arise because of decision predictions which, if proven wrong, require that decod-

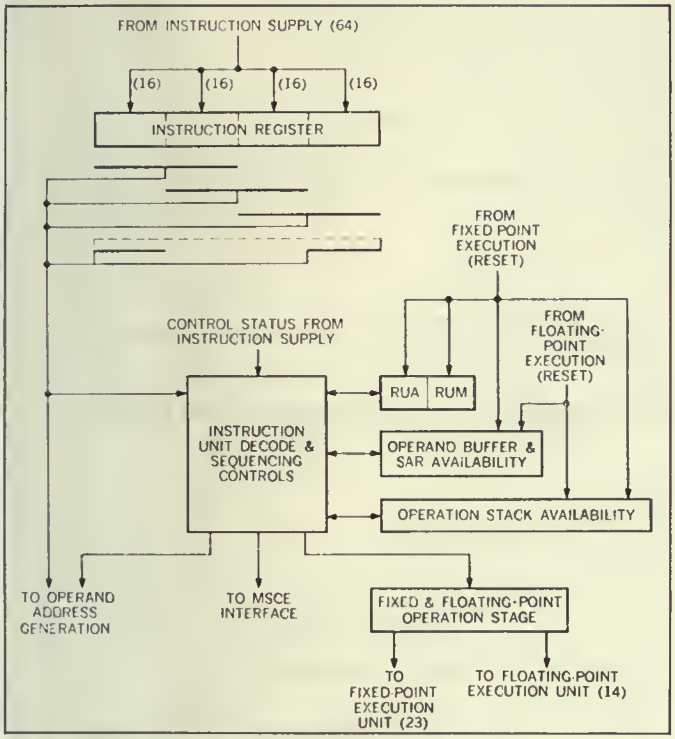

FROM INSTRUCTION SUPPLY (64)
(16)
(16)
(16)
(16)
INSTRUCTION REGISTER
FROM FIXED POINT EXECUTION (RESET)
CONTROL STATUS FROM INSTRUCTION SUPPLY
FROM FLOATING-POINT EXECUTION (RESET)
RUA
RUM
INSTRUCTION UNIT DECODE & SEQUENCING CONTROLS
OPERAND BUFFER & SAR AVAILABILITY
OPERATION STACK AVAILABILITY
TO OPERAND ADDRESS GENERATION
TO MSCE INTERFACE
FIXED & FLOATING-POINT OPERATION STAGE
TO FIXED-POINT EXECUTION UNIT (23)
TO FLOATING-POINT EXECUTION UNIT (14)

Fig. 14. Data flow for instruction decoding and instruction issuing.

ing cease immediately so that recovery can be initiated with a minimum of backup facilities.

Such occurrences as the discovery of a branch wrong guess or a store instruction which may alter the prefetched instruction stream generate these inter-instruction interlocks. Figure 15 illustrates the interlock function. The placement of a store instruction in the instruction stream, in particular, warrants further discussion because it presents a serious time problem in the instruction unit. The dilemma stems both from the concurrency philosophy and from the architectural specification that a store operation may alter the subsequent instruction. Recall that, through the pipeline concept, decoding can occur on successive cycles, with one instruction being decoded at the same time the address for the previous instruction is being generated. Therefore, for a decode which follows a store instruction, a test between the instruction counter and the storage address is required to detect whether or not the subsequent decode is affected by the store. Unless rather extensive recovery hardware is used, the decode, if affected, must be suppressed. However, the assembly line basic time interval is too short to both complete the detection and block the decode. The simplest solution would require a null decode time following each store issue. However, the frequency of store instructions is high enough that the performance degradation would be objectionable. The compromise solution which was adopted reduces the number of decoding delays by utilizing a truncated-address compare. The time requirements prohibit anything more than a compare of the low-order six bits of the storage address currently being generated, using the algorithm illustrated in Fig. 16.

The algorithm attaches relatively little significance to the low-order three adder bits (dealing with byte, half-word and full-word addresses) since the primary performance concern is with stores of double-words. It is seen, for example, that for the full-word case the probability of a carry into the double-word address is approximately 1/4, while for double-word handling it is negligible. The double-word address three-bit compare will occur with 1/8 probability while the word boundary crossover term has a probability of 1/16 (probability that instruction can cross boundary, 1/2, × probability that the crossover is into the store-affected word, 1/8). The two cases thus have the probabilities:

Full word $1/4 + 1/8 + 1/6 = 7/16$ and
Double-word $1/8 + 1/16 = 3/16$.

These figures indicate the likelihood of a decode time-interval delay following the issue of a store instruction. When such a decode delay is encountered, the following cycle is used to complete the test, that is, to check the total address to determine whether an instruction word has in fact been altered. To this effect, the generated storage address is compared with the upper and lower bounds of the instruction array (Fig. 16). A between-the-bounds indication results in a decode halt, a re-fetch of the affected instruction double-word, then resumption of normal processing. This second portion of the interlock is only slightly less critical in timing than the first. Figure 17 illustrates the re-fetch timing sequence. One difficulty with the store interlock is that in blocking the decode, it must inhibit action over a significant portion of the instruction unit. This implies both heavy loading and lengthy wire, each of which seriously hampers circuit performance. It was therefore important that the unit be as small as possible and that the layout of the hardware constantly consider the interlock.

For each instruction, following the clearing of all interlocks, the decode decision determines whether to issue the instruction to an execution unit and initiate address generation, or to retain the instruction for sequencing within the instruction unit. The issuing to an execution unit and the operand fetching for storage-to-register (RX) instructions constitutes a controlled splitting operation; sufficient information is forwarded along both paths to effect a proper execution unit merge. For example, buffer assignment is carried in both paths so that the main storage control element will return the operand to the buffer which will be accessed by the

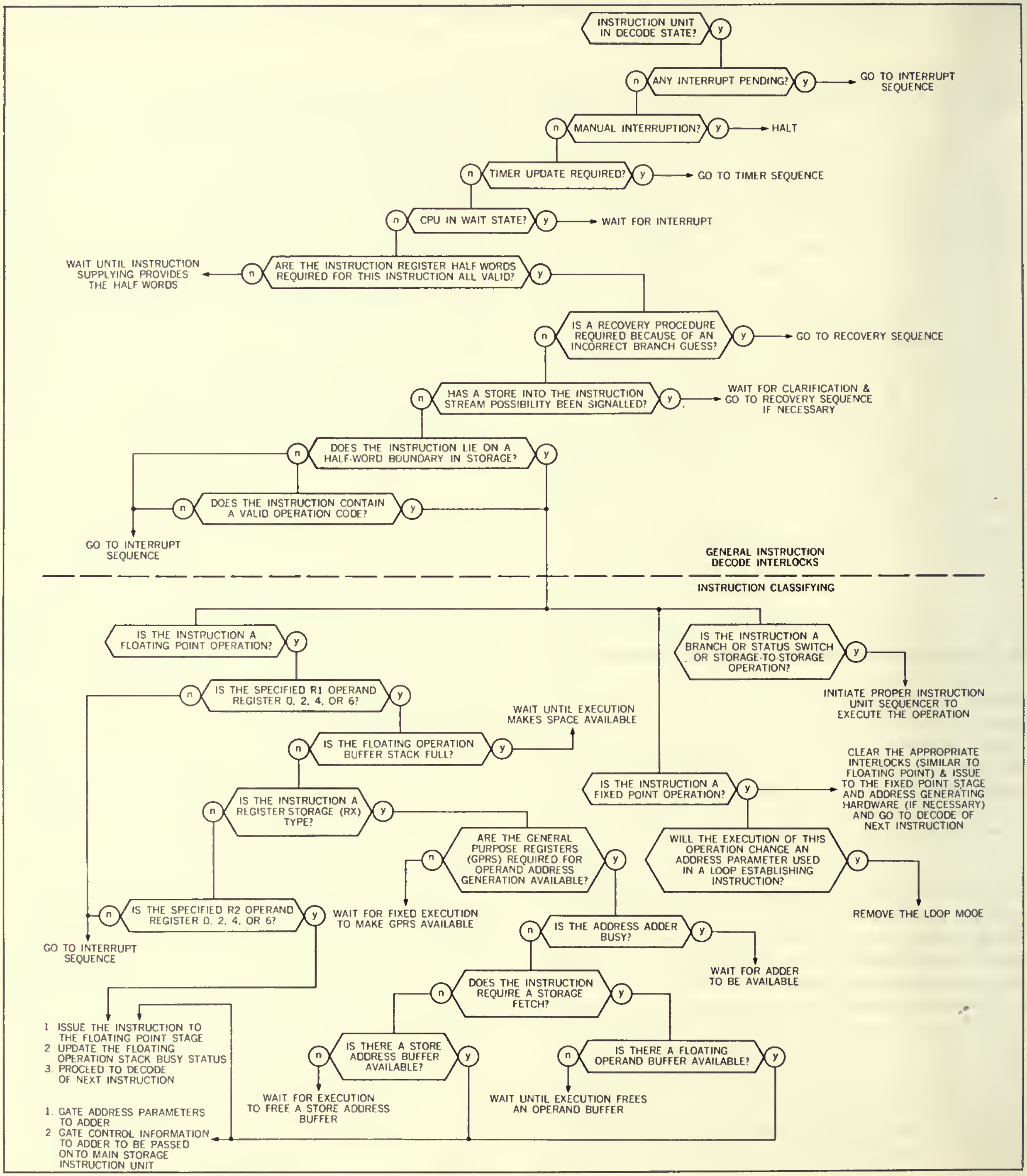

INSTRUCTION UNIT IN DECODE STATE?
y
n
ANY INTERRUPT PENDING?
y
GO TO INTERRUPT SEQUENCE
n
MANUAL INTERRUPTION?
y
HALT
n
TIMER UPDATE REQUIRED?
y
GO TO TIMER SEQUENCE
n
CPU IN WAIT STATE?
y
WAIT FOR INTERRUPT
WAIT UNTIL INSTRUCTION SUPPLYING PROVIDES THE HALF WORDS
n
ARE THE INSTRUCTION REGISTER HALF WORDS REQUIRED FOR THIS INSTRUCTION ALL VALID?
y
n
IS A RECOVERY PROCEDURE REQUIRED BECAUSE OF AN INCORRECT BRANCH GUESS?
y
GO TO RECOVERY SEQUENCE
n
HAS A STORE INTO THE INSTRUCTION STREAM POSSIBILITY BEEN SIGNALLED?
y
WAIT FOR CLARIFICATION & GO TO RECOVERY SEQUENCE IF NECESSARY
n
DOES THE INSTRUCTION LIE ON A HALF-WORD BOUNDARY IN STORAGE?
y
n
DOES THE INSTRUCTION CONTAIN A VALID OPERATION CODE?
y
GO TO INTERRUPT SEQUENCE
GENERAL INSTRUCTION DECODE INTERLOCKS
INSTRUCTION CLASSIFYING
IS THE INSTRUCTION A FLOATING POINT OPERATION?
y
n
IS THE SPECIFIED R1 OPERAND REGISTER 0, 2, 4, OR 6?
y
WAIT UNTIL EXECUTION MAKES SPACE AVAILABLE
n
IS THE FLOATING OPERATION BUFFER STACK FULL?
y
n
IS THE INSTRUCTION A REGISTER STORAGE (RX) TYPE?
y
n
ARE THE GENERAL PURPOSE REGISTERS (GPRS) REQUIRED FOR OPERAND ADDRESS GENERATION AVAILABLE?
y
n
IS THE SPECIFIED R2 OPERAND REGISTER 0, 2, 4, OR 6?
y
WAIT FOR FIXED EXECUTION TO MAKE GPRS AVAILABLE
GO TO INTERRUPT SEQUENCE
n
IS THE ADDRESS ADDER BUSY?
y
WAIT FOR ADDER TO BE AVAILABLE
n
DOES THE INSTRUCTION REQUIRE A STORAGE FETCH?
y
1 ISSUE THE INSTRUCTION TO THE FLOATING POINT STAGE
2 UPDATE THE FLOATING OPERATION STACK BUSY STATUS
3. PROCEED TO DECODE OF NEXT INSTRUCTION
n
IS THERE A STORE ADDRESS BUFFER AVAILABLE?
y
n
IS THERE A FLOATING OPERAND BUFFER AVAILABLE?
y
WAIT FOR EXECUTION TO FREE A STORE ADDRESS BUFFER
WAIT UNTIL EXECUTION FREES AN OPERAND BUFFER
1. GATE ADDRESS PARAMETERS TO ADDER
2 GATE CONTROL INFORMATION TO ADDER TO BE PASSED ON TO MAIN STORAGE INSTRUCTION UNIT
IS THE INSTRUCTION A BRANCH OR STATUS SWITCH OR STORAGE-TO-STORAGE OPERATION?
y
INITIATE PROPER INSTRUCTION UNIT SEQUENCER TO EXECUTE THE OPERATION
IS THE INSTRUCTION A FIXED POINT OPERATION?
y
CLEAR THE APPROPRIATE INTERLOCKS (SIMILAR TO FLOATING POINT) & ISSUE TO THE FIXED POINT STAGE AND ADDRESS GENERATING HARDWARE (IF NECESSARY) AND GO TO DECODE OF NEXT INSTRUCTION
WILL THE EXECUTION OF THIS OPERATION CHANGE AN ADDRESS PARAMETER USED IN A LOOP ESTABLISHING INSTRUCTION?
y
REMOVE THE LOOP MODE

Fig. 15. Decision sequence for instruction decoding and instruction issuing.

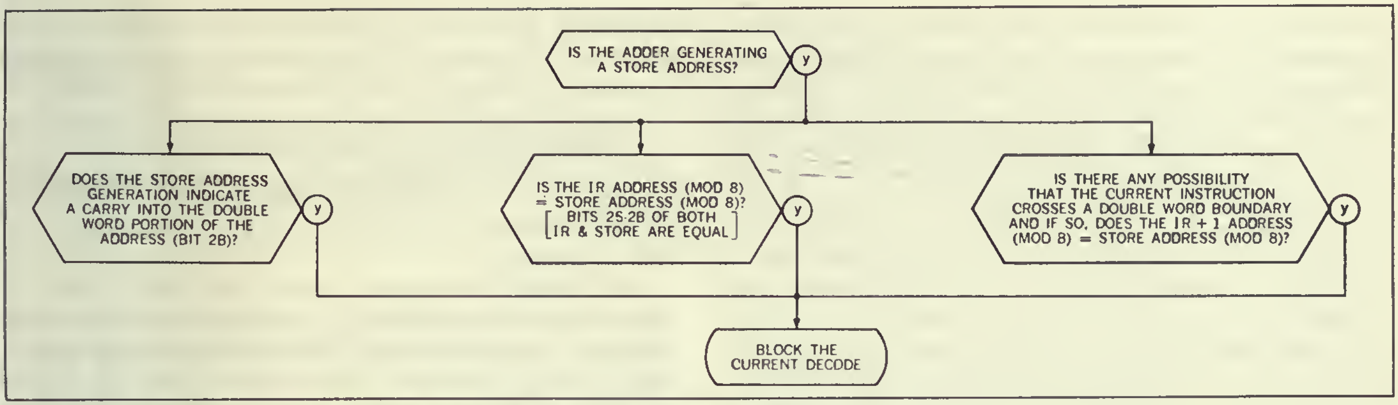

IS THE ADDER GENERATING A STORE ADDRESS?
y
DOES THE STORE ADDRESS GENERATION INDICATE A CARRY INTO THE DOUBLE WORD PORTION OF THE ADDRESS (BIT 28)?
y
IS THE IR ADDRESS (MOD 8) = STORE ADDRESS (MOD 8)? [BITS 25-28 OF BOTH IR & STORE ARE EQUAL]
y
IS THERE ANY POSSIBILITY THAT THE CURRENT INSTRUCTION CROSSES A DOUBLE WORD BOUNDARY AND IF SO, DOES THE IR + 1 ADDRESS (MOD 8) = STORE ADDRESS (MOD 8)?
y
BLOCK THE CURRENT DECODE

Fig. 16. Decode interlock (established following the issue of a store instruction).

execution unit when it prepares to execute the instruction. With this technique the execution units are isolated from storage and can be designed to treat all operations as involving only registers.

A final decoding function is mentioned here, to exemplify the sort of design considerations and hardware additions that are caused by performance-optimizing techniques. The branch sequencing is optimized so that no address generation is required when a branch which established the loop mode is reencountered. This is done by saving the location, within the instruction array, of the target. It is possible, even if unlikely, that one of the instructions contained in a loop may alter the parameter originally used to generate the target address which is now being assumed. This possibility, although rare, does require hardware to detect the occurrence and terminate the loop mode. This hardware includes two 4-bit registers, required to preserve the address of the general purpose registers (X and B) utilized in the target address generation, and comparators which check these addresses against the sink address (RI) of the fixed-point instructions. Detection of a compare and termination of loop mode are necessary during the decoding interval to ensure that subsequent branch sequencing will be correct.

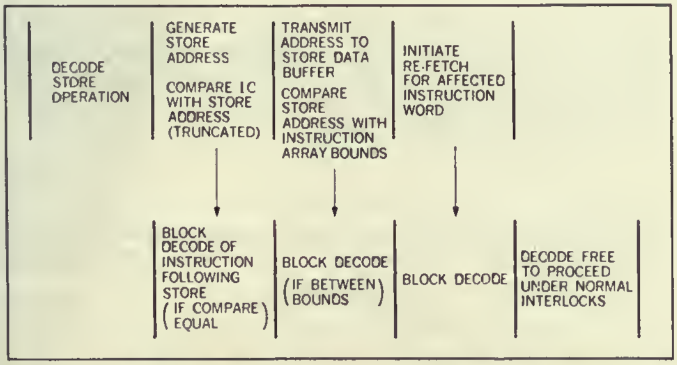

DECODE STORE OPERATION
GENERATE STORE ADDRESS
COMPARE IC WITH STORE ADDRESS (TRUNCATED)
TRANSMIT ADDRESS TO STORE DATA BUFFER
COMPARE STORE ADDRESS WITH INSTRUCTION ARRAY BOUNDS
INITIATE RE-FETCH FOR AFFECTED INSTRUCTION WORD
BLOCK DECODE OF INSTRUCTION FOLLOWING STORE (IF COMPARE EQUAL)
BLOCK DECODE (IF BETWEEN BOUNDS)
BLOCK DECODE
DECODE FREE TO PROCEED UNDER NORMAL INTERLOCKS

Fig. 17. Effect of the decode interlock on pre-fetched instructions.

The address-generating time interval provides for the combining of proper address parameters and for the forwarding of the associated operation (fetch or store) control to the main storage control element through an interface register. A major concern, associated with the address parameters, was to decide where the physical location of the general purpose registers should be. This concern arises since the fixed-point execution unit, as well as the instruction unit, makes demands on the GPR's, while the packaging split will cause the registers to be relatively far from one of the units. It was decided to place them in the execution unit since, first, execution tends to change the registers while address generation merely examines their contents, and second, it was desired that a fixed-point execution unit be able to iteratively use any particular register on successive time intervals. In order to circumvent the resulting time delay (long wire separation) between the general purpose registers and the address adder, each register is fed via "hot" lines to the instruction unit. The gating of a particular GPR to the adder can thereby be implemented locally within the instruction unit, and no transmission delay is incurred unless the register contents have just been changed.

Placing the GPR's outside the instruction unit creates a delay of two basic time intervals before a change initiated by the instruction unit is reflected at the address parameter inputs from the GPRs. This delay is particularly evident when it is realized that the address generated immediately following such a GPR change generally requires the contents of the affected register as a parameter. For example, branch on index, branch on count, branch and link, and load address are instruction unit operations which change the contents of a GPR. Further, in loop situations the target of the branch frequently uses the changed register as an index quantity in its address. Performance demands led to the incorporating of controls which recognize the above situation and effect a by-pass of the GPR. This entails substituting the content of the adder output register (which contains the new GPR data) for the content of the affected GPR. One performance cycle was saved by this technique.

In addition to address generation, the address adder serves to accomplish branch decision arithmetic, loop mode testing, and instruction counter value generation for various situations. In order to perform all of these functions, it was required that the adder have two 32-bit inputs and one input of 12 bits. One of the 32-bit inputs is complementable and a variety of fixed, single-bit inputs is provided for miscellaneous sequences. The data path is illustrated in Fig. 18.

Status Switching and Input/Output. The philosophy associated with status switching instructions is primarily one of design expediency. Basic existing hardware paths are exercised wherever possible, and an attempt is made to adhere to the architectural interrupt specifications. When status switching instructions are encountered in conditional mode the instruction unit is halted and no action is taken until the condition is cleared.

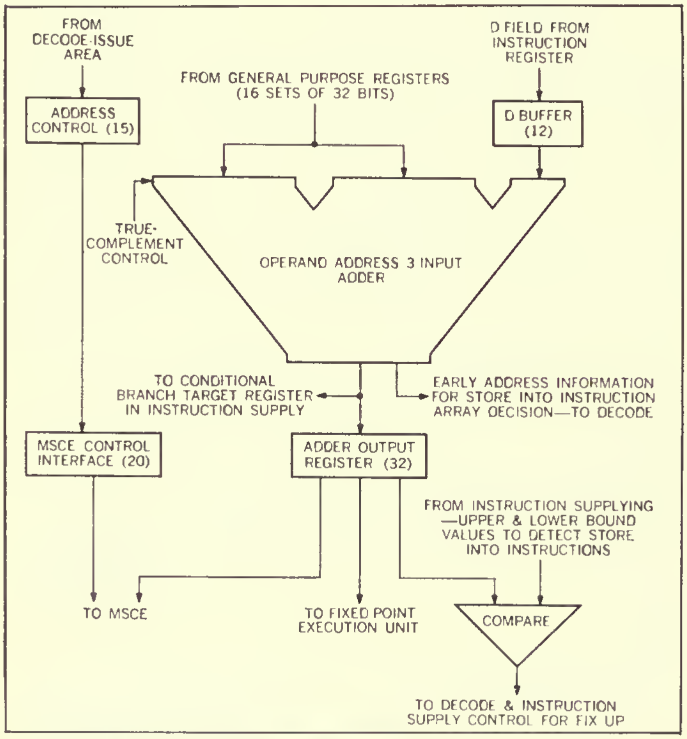

FROM DECODE-ISSUE AREA
D FIELD FROM INSTRUCTION REGISTER
FROM GENERAL PURPOSE REGISTERS (16 SETS OF 32 BITS)
ADDRESS CONTROL (15)
D BUFFER (12)
TRUE-COMPLEMENT CONTROL
OPERAND ADDRESS 3 INPUT ADDER
TO CONDITIONAL BRANCH TARGET REGISTER IN INSTRUCTION SUPPLY
EARLY ADDRESS INFORMATION FOR STORE INTO INSTRUCTION ARRAY DECISION—TO DECODE
MSCE CONTROL INTERFACE (20)
ADDER OUTPUT REGISTER (32)
FROM INSTRUCTION SUPPLYING —UPPER & LOWER BOUND VALUES TO DETECT STORE INTO INSTRUCTIONS
TO MSCE
TO FIXED POINT EXECUTION UNIT
COMPARE
TO DECODE & INSTRUCTION SUPPLY CONTROL FOR FIX UP

Fig. 18. Data flow for address generation.

The supervisor call (SVC) instruction is treated by the interrupt hardware as a precise interrupt. The same new status word pre-fetch philosophy is utilized in the load program status word (LPSW) operation.

One difficulty encountered in conjunction with the start-up fetching of instructions following a status switch (or interrupt) is that a new storage protect key[1] is likely to obtain. Consequently, a period exists during which two protect keys are active, the first for previously delayed, still outstanding accesses associated with the current execution clean-up, and the second for the fetching of instructions. This situation is handled by sending both keys to the main storage control element and attaching proper control information to the instruction fetches.

The set program mask (SPM) implementation has a minor optimization: Whenever the new mask equals the current mask, the instruction completes immediately. Otherwise an execution clean-up is effected before setting the new mask to make certain that outstanding operations are executed in the proper mask environment.

I/O instructions, and I/O interrupts, require a wait for channel communications. The independent channel and CPU paths to storage demand that the CPU be finished setting up the I/O controls in storage before the channel can be notified to proceed. Once notified, the channel must interrogate the instruction-addressed device prior to setting the condition code in the CPU. This is accomplished by lower-speed circuitry and involves units some distances away; consequently, I/O initiation times are of the order of 5–10 microseconds.

References

Anderson, Sparacio, and Tomasulo [1967]; Amdahl, Blaauw, and Brooks [1964]; Anderson, Earle, Goldschmidt, and Powers [1967]; Boland, Granito, Marcotte, Messina, and Smith [1967]; Buchholz et al. [1962]; Flynn and Low [1967]; Litwiller and Adler [private communication]; Tomasulo [1967].

[1] The storage protect key is contained in the program status word (PSW). It is a tag which accompanies all storage requests, and from it the storage can determine when a protect violation occurs.