the secondary computer can be made either as a result of explicit register numbers, etc., written in the secondary instructions, or as a result of the fact that these locations are currently being specified in the currently-waiting primary instruction. In consequence, the secondary computer can be used to monitor or to interpret the program of the primary computer in a highly flexible fashion.

Conclusion

The foregoing discussion outlined some of the paths along which digital systems investigations are proceeding at the National Bureau of Standards. It is evident that the progress made thus far represents only a small start on a series of system problems of wide scope. These problems of network organization will become increasingly important as time goes on, because the large-scale deadline-meeting applications of the future (to the extent that they are limited by the basic component rates) will force designers more and more in the direction of widely extended network systems. In these extended systems, large numbers of independent machines will participate cooperatively in the solution of a common problem. Such systems will inevitably be more loosely organized with respect to centralized control than the relatively compact and logically self-contained machines of today. The dynamics of such loosely organized systems can thus be more effectively conceived of in terms of the group behavior of large numbers of randomly-interacting independent units rather than as a completely prescheduled, unified-machine process. Although the over-all performance of such systems can be best analyzed in probabilistic or statistical terms, the actual elementary building blocks in the system will still have to operate in a rigid, precisely defined way. The problem of system design then becomes one of contriving methods for combining the particular building blocks at our disposal so that fullest advantage is taken of the ways in which the laws of chance act to combine their individual, statistical-performance parameters.

A Program-Controlled Program Interruption System

F. P. BROOKS, JR[†]

Objectives

IN a computer complex now under development at IBM for the Los Alamos Scientific Laboratory,[1] a major objective is to improve performance by eliminating unnecessary waiting. A fundamental concept is that of multiple data processing units sharing a common memory and operating simultaneously and asynchronously. This complex must be capable of immediate and coordinated response to external signals. These two concepts demand special methods of switching any single unit from one program to another. The system by which a computer unit responds to arbitrarily timed signals with programs pertinent to each signal will be called a program interruption system.

There are two quite distinct purposes for which a program interruption system is necessary. The first of these is to provide a means by which a computer can make very rapid response to extra-program circumstances which occur at arbitrary times, performing useful work while waiting for such circumstances. These circumstances will most often be signals from an input-output exchange that some interrogation has been received or that an input-output operation is complete. For efficiency in real-time operation, the computer must respond to these forthwith. This demands a system by which such signals cause a transfer of control to a suitable special program.

The second purpose is to permit the computer to make rapid and facile selection of alternate instructions when program-activated indicators signal that special circumstances have occurred. For example, it is clearly desirable to have such a system for arithmetic overflow, since the alternatives are tedious and wasteful programmed testing or a costly machine stop when the condition arises. As another example, it is desirable to have a special routine seize control and to take corrective steps whenever the regular program attempts a division by zero.

These two purposes—response to asynchronously occurring external signals, and monitoring of exceptional conditions generated by the program itself—are quite distinct, and it would be conceivable to have systems for handling each independently. However, a single system serves both purposes equally well, and provision of a single uniform system permits more powerful operating techniques.

The program interruption system adopted must obey several constraints. The most important is that programming must be straightforward, efficient, and as simple as the inherent conceptual complexities allow. Secondly, the special circuitry must be reasonably modest, for mainte-

[†] IBM Corp., Poughkeepsie, N.Y.
[1] S. W. Dunwell, "Design objectives for the IBM stretch computer," *1956 Proc. EJCC;* p. 20.

nance economy as well as low first cost. Thirdly, the computer must not be retarded by the interruption system, except when interruptions do in fact occur. Finally, since there is little experience in the use of multiprogrammed systems, the system should be as flexible as possible. It is important to avoid inflexibility based upon assuming certain methods of use.

In the next section we shall consider the several parts of the system individually, examining the problem solved by each. A final section will show examples of the use of the system.

System Components

The first question to be answered in a program interruption system is: When is the time to interrupt? This requires a signal when there is a reason for interruption. It also requires a designation as to when interruptions are to be permitted. The solution to the first is straightforward. For each condition which may require attention, there is a flip-flop, called an *indicator,* which may be interrogated by the control mechanism. When the condition arises, the flip-flop is set on, and it may be turned off when the condition disappears or when the program has cared for it. In the system under development there are sixty-four such indicators, and they are grouped together into a single register. This indicator register has an address and can be treated as a data word for ordinary program operations.

The designation of times when interruption is permitted can be done in several ways. It is possible to organize a system so that any condition arising at any time can cause interruption. Alternatively one can provide a bit in each instruction which designates whether interruptions shall be permitted at the end of that instruction or not. These methods make no distinction among the interrupting conditions. It is highly desirable to permit selective control of interruptions, so that at any time one class of conditions might be permitted to cause interruptions, and another class might be prevented from causing interruptions. Therefore, each of the sixty-four conditions is provided with a program-set *mask* flip-flop, which allows that condition to cause interruption when it is on. When the mask bit is off, interruption cannot be caused by that condition. As with the indicators, the mask bits are assembled into a single register with an address, so that they can all be loaded and stored as a unit, as well as individually. The indicator register and mask register give the programmer full control as to which conditions are to be permitted to interrupt at any time.

A second major question that a program interruption system must answer is: What is to be done when an interruption occurs? In the simplest systems, the program transfers to some fixed location, where a fix-up routine proceeds to determine which condition caused the interruption and what is to be done. This is rather slow. In order to save time, we provide a branch to a different location for each of the conditions which can cause interruption. The particular location is selected by a leftmost-one identifier. This device generates a number giving the position of the bit within the indicator register which signals the condition causing the interruption. This bit number is used to generate an instruction address for selecting the appropriate operation to be performed. Since it is anticipated that such a computer system will often be operated in a multiprogrammed manner, the bit address is not used directly as the instruction address, for this would require the whole table of fix-up instructions to be changed each time the computer switched to a different program. Instead, the bit address is added to a base address held in an interruption base address register. The sum is used as the next instruction address. One can easily select among several interruption instruction tables by setting the base address.

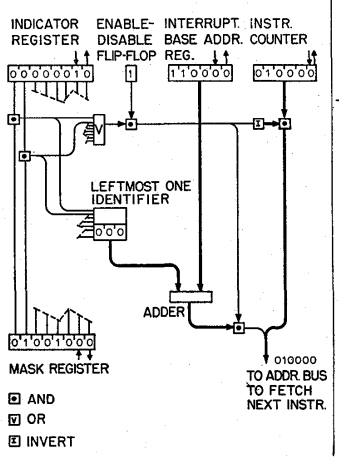
A third major question facing any interruption system is: How does control return to the main program when the fix-up routine is complete? One solution is to employ several instruction counters. A more economical solution is to have the instruction counter contents automatically stored in a fixed location upon interruption. We chose, however, to make no provision for automatic storage. Instead, the address of the instruction executed immediately after interruption is generated without disturbing the contents of the instruction counter. That instruction can store and alter the undisturbed instruction counter contents, if desired. A typical operation for such use would be Store Instruction Counter and Branch.

If the instruction counter is not altered by the interrupting instruction, the program automatically returns to the interrupted program and proceeds. This permits exceptionally rapid and simple treatment of the conditions that can be handled with a single instruction. More complex conditions are handled by a Store Counter and Branch instruction that enters a suitable subroutine just as any other subroutine would be entered. Program control of counter storage has two advantages: the storage location can be selected at will, and it saves time to perform counter storage and change only when needed.

A fourth question faced by any program interruption system is: How are the contents of the accumulator, index registers, etc., to be preserved upon interruption? Automatic storage of these is both time consuming and inflexible. It is better to use the standard subroutine philosophy —the fix-up routine is responsible for preserving and restoring any of the central registers it uses. Special operations simplify storage and restoration of central registers, but full flexibility is left with the subroutine programmer. He need not store and retrieve anything more than he intends to corrupt.

The fifth question that must be answered by a program interruption system is: How are priorities established among interrupting conditions and what provision is made for multiple interruptions? Provision of the full masking facility answers this problem, since any subset of the conditions may be permitted to cause interruption. Each fix-up

DIAGRAM

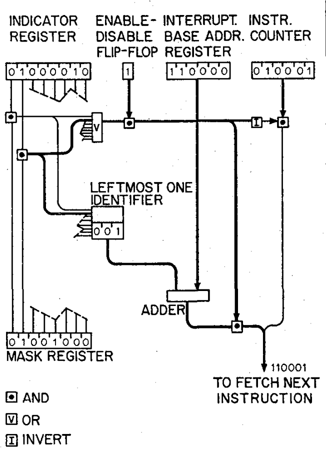


PROGRAM SEQUENCE

INSTR. LOCN.		OPER.	ADDR.	ADDR.
BIN.	DEC.			
001111	(15)	ADD	X	
→ 010000	(16)			

Fig. 1.

DIAGRAM



PROGRAM SEQUENCE

INSTR. LOC.		OPER.	ADDR.	ADDR.
BIN.	DEC.			
001111	(15)	ADD	X	
010000	(16)	MULT.	Y	
→ 110001	(49)	CLEAR ADD	L(0)	

Fig. 2.

subroutine can use a mask of its own, thereby defining the conditions which are allowed to cause interruption during that routine. There is also provided a means to disable the whole interruption mechanism for those short intervals when an interruption would be awkward. One such interval is that between the time when a subroutine restores the interruption base address appropriate for the main program and the time when it effects the return to the main program. The mechanism is automatically disabled by certain operations and can be optionally enabled or disabled by others.

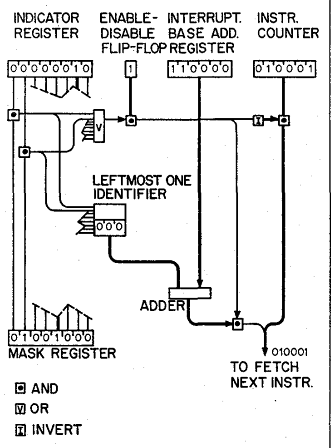
Simultaneous conditions are taken care of by the leftmost-one identifier, which selects that condition with the lowest bit address in the indicator register for first treatment. This is satisfactory because the fix-up routines for the several conditions are largely independent of one another. The positioning of conditions within the indicator register defines a built-in priority, but this priority can readily be overridden by suitable masking when the programmer desires. In fact, it might be said that the leftmost-one identifier solves the problem of simultaneity while the selectivity provided by the mask solves the problem of over-all and longer-term priorities.

Examples

Fig. 1 shows the system organization of an interruption system with only eight conditions. The indicator register has only one condition, number six, on. The mask register is set up to allow only conditions one and four to cause interruption. Instruction 15 has just been executed, and the instruction counter has been stepped up to 16. There is no interruption so the next instruction is taken from location 16 in the normal manner.

In Fig. 2 the execution of instruction 16 is accompanied by the occurrence of condition one. The leftmost-one identifier generates a binary one which is added to the 48 contained in the interruption base address register. The result, 49, is used for the address of the next instruction rather than the 17 contained in the instruction counter, which is unchanged.

DIAGRAM



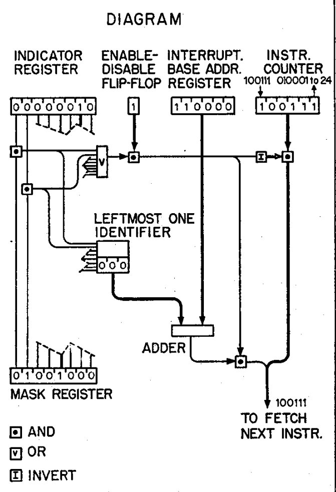
PROGRAM SEQUENCE

INSTR. LOCN.		OPER.	ADDR.	ADDR.
BIN.	DEC.			
001111	(15)	ADD	X	
010000	(16)	MULT.	Y	
110001	(49)	CLEAR ADD	L(0)	
→ 010001	(17)	STORE	Z	

Fig. 3.

In Fig. 3 is shown the case when the instruction at location 49 does not change the instruction counter. The interruption mechanism has turned off condition one which caused the interruption. No other condition and mask bits coincide. After the instruction at location 49 is complete, the next instruction is taken from the location specified by the instruction counter, which still contains 17. This one-instruction fix-up routine might be used to clear the accumulator after a floating point underflow.

Fig. 4 shows a different sequence that might have followed Fig. 2. Suppose indicator 1 represented an end-of-file condition on a tape and several instructions are needed to take care of the condition. In this case, the instruction at location 49 disables the interruption mechanism, stores the instruction counter contents (17) in location 24, and causes an unconditional branch to location 39. The fix-up routine proper consists of the three instructions from 39 to 41. It might be any length, and might include testing and scheduling of further input-output operations. During the routine no further interruptions can occur. Instruction 42 is a Branch and Enable instruction, which causes a



PROGRAM SEQUENCE

INSTR. LOCN. BIN.	DEC.	OPER.	ADDR.	ADDR.
001111	(15)	ADD	X	
010000	(16)	MULT.	Y	
110001	(49)	BRANCH & STORE COUNTER, DISABLE	39	24
⟶				
100111	(39)	LOAD	W	
•	•	•	•	
•	•	•	•	
•	•	•	•	
•	•	•	•	
•	•	•	•	
101010	(42)	BRANCH,E	C(24)	
010001	(17)	STORE	Z	

Fig. 4.

branch to the location specified in 24. This returns control to the interrupted program at location 17 and re-enables the mechanism so that further interruptions are possible.

The program in Fig. 4 assumes that it is desired to prevent further interruptions during the fix-up routine. If further interruptions were to be allowed during the routine, and the same mask still applies, one would use a simple Store Counter and Branch instruction at location 49, and a simple branch instruction at location 42. This procedure is appropriate when, and only when, the programmer is certain the condition one cannot arise during the fix-up routine or during any that might interrupt it.

In the most sophisticated use, where it is desired to use a long fix-up routine which is to be interrupted under a different set of conditions, the program shown in Table I is suitable. The mechanism is disabled at the time of the first instruction after interruption. The new mask is loaded and the old preserved. The mechanism is then enabled. At the end of the routine the mechanism is disabled, the old mask restored, and the mechanism is re-enabled as control is transferred to the originally interrupted routine at location 17.

This procedure is clearly suitable for any number of levels of interruptions upon interruptions, each of which may have a different set of causing conditions. Each level of routine is under only the usual subroutine constraint of preserving the contents of the registers it uses. Full program control simplifies programming and multiprogramming, as does the refusal to assign special functions to fixed memory locations. The task of the programmer of fix-up routines is simplified by the provision of special operations and by the adoption of the same conventions and requirements for interruption routines as for ordinary subroutines.

An especially important feature of the program interruption system just described is that it makes almost no demands upon the writer of the lowest level program. He need only set up the interruption base address register and the mask register. He need not even understand what

TABLE I

Inst. Location Binary	Decimal	Operation	Address	Address
00111	(15)	ADD	X	
010000	(16)	MULT	Y	
110001	(49)	BRANCH, STORE COUNTER, DISABLE	39	24
100111	(39)	SWAP	Mask	23
101000	(40)	ENABLE		
101001	(41)	LOAD	W	
.	.	.	.	
.	.	.	.	
101101	(45)	DISABLE		
101110	(46)	SWAP	Mask	23
101111	(47)	BRANCH, ENABLE	c(24)	
010001	(17)	STORE	Z	

he puts there or why, but may follow the local ground rules of his installation. Priorities, preservation of data, and other programming considerations that are inherent in program interruption concern only the author of the fix-up routines. In open-shop installations it is important that any programming burden inherent in such sophisticated operation fall upon the full-time utility programmer rather than upon the general user.

In summary, the program interruption system includes an indicator register in which are assembled flip-flops for the conditions that may cause interruption. These conditions may be program-generated, as by invalid instructions, or external, as by remote interrogations. Corresponding to each condition is a mask bit, and these are assembled into a mask register. When any condition occurs whose mask bit is on, the bit address of that indicator is added to a base address to determine the next instruction to be executed. This instruction is essentially inserted into the normal instruction sequence. If the inserted instruction is not a branch, the program executes it and proceeds exactly as if the extra operation were a part of the main program itself. If the inserted instruction is a branch, it provides for entry to a fix-up subroutine in the same manner any other subroutine would be entered. When the fix-up subroutine is complete, return to the main program is effected just as it would be from any other subroutine. Programming is facilitated by uniformity with ordinary subroutine control, by special operations, and by full programmable selection of interrupting conditions, storage locations, and fix-up routine locations. The governing philosophy of design has been that flexibility and power of program control is of more value to the sophisticated user than would be pseudoconvenience of an inflexible automatic interruption system, but that the power and flexibility made available to the sophisticated user must not be gained at the expense of convenience and simplicity for the casual user.

Acknowledgment

The author wishes to acknowledge the contributions of several colleagues: D. W. Sweeney, who proposed many of the general concepts; Dr. G. A. Blaauw, who offered many suggestions; and Dr. W. Buchholz, under whose guidance this system was designed.